```
**********************************************************
    1938 Mon Aug 17 21:08:01 2015
new/usr/src/cmd/cmd-inet/usr.bin/netstat/Makefile
XXXX adding PID information to netstat output
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# Copyright (c) 1990 Mentat Inc.
#
# cmd/cmd-inet/usr.bin/netstat/Makefile

PROG=           netstat

LOCALOBJS=      netstat.o
LOCALOBJS=      netstat.o unix.o
COMMONOBJS=     compat.o

include ../../../Makefile.cmd
include ../../Makefile.cmd-inet

LOCALSRCS=      $(LOCALOBJS:%.o=%.c)
COMMONSRCS=     $(CMDINETCOMMONDIR)/$(COMMONOBJS:%.o=%.c)

STATCOMMONDIR = $(SRC)/cmd/stat/common

STAT_COMMON_OBJS = timestamp.o
STAT_COMMON_SRCS = $(STAT_COMMON_OBJS:%.o=$(STATCOMMONDIR)/%.c)

OBJS=           $(LOCALOBJS) $(COMMONOBJS) $(STAT_COMMON_OBJS)
SRCS=           $(LOCALSRCS) $(COMMONSRCS) $(STAT_COMMON_SRCS)

CPPFLAGS += -DNDEBUG -I$(CMDINETCOMMONDIR) -I$(STATCOMMONDIR)
CERRWARN += -_gcc=-Wno-uninitialized
CERRWARN += -_gcc=-Wno-parentheses
LDLIBS += -ldhcpagent -lsocket -lnsl -lkstat -ltsnet -ltsol

.KEEP_STATE:

all: $(PROG) $(NPROG)

ROOTPROG=   $(PROG:%=$(ROOTBIN)/%)

$(PROG): $(OBJS)
        $(LINK.c) $(OBJS) -o $@ $(LDLIBS)
```



```
        $(POST_PROCESS)

%.o : $(STATCOMMONDIR)/%.c
        $(COMPILE.c) -o $@ $<
        $(POST_PROCESS_O)

install: all $(ROOTPROG)

clean:
        $(RM) $(OBJS)

lint:   lint_SRCS

include ../../../Makefile.targ
```

```
**********************************************************
  203800 Mon Aug 17 21:08:02 2015
new/usr/src/cmd/cmd-inet/usr.bin/netstat/netstat.c
XXXX adding PID information to netstat output
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 1990  Mentat Inc.
  24  * netstat.c 2.2, last change 9/9/91
  25  * MROUTING Revision 3.5
  26  */

  28 /*
  29  * simple netstat based on snmp/mib-2 interface to the TCP/IP stack
  30  *
  31  * NOTES:
  32  * 1. A comment "LINTED: (note 1)" appears before certain lines where
  33  *    lint would have complained, "pointer cast may result in improper
  34  *    alignment". These are lines where lint had suspected potential
  35  *    improper alignment of a data structure; in each such situation
  36  *    we have relied on the kernel guaranteeing proper alignment.
  37  * 2. Some 'for' loops have been commented as "'for' loop 1", etc
  38  *    because they have 'continue' or 'break' statements in their
  39  *    bodies. 'continue' statements have been used inside some loops
  40  *    where avoiding them would have led to deep levels of indentation.
  41  *
  42  * TODO:
  43  *      Add ability to request subsets from kernel (with level = MIB2_IP;
  44  *      name = 0 meaning everything for compatibility)
  45  */

  47 #include <stdio.h>
  48 #include <stdlib.h>
  49 #include <stdarg.h>
  50 #include <unistd.h>
  51 #include <strings.h>
  52 #include <string.h>
  53 #include <errno.h>
  54 #include <ctype.h>
  55 #include <kstat.h>
  56 #include <assert.h>
  57 #include <locale.h>
  58 #include <pwd.h>
  59 #include <limits.h>
  60 #endif /* ! codereview */
```



```
  62 #include <sys/types.h>
  63 #include <sys/stat.h>
  64 #endif /* ! codereview */
  65 #include <sys/stream.h>
  66 #include <stropts.h>
  67 #include <sys/strstat.h>
  68 #include <sys/tihdr.h>
  69 #include <procfs.h>
  70 #endif /* ! codereview */

  72 #include <sys/socket.h>
  73 #include <sys/socketvar.h>
  74 #endif /* ! codereview */
  75 #include <sys/sockio.h>
  76 #include <netinet/in.h>
  77 #include <net/if.h>
  78 #include <net/route.h>

  80 #include <inet/mib2.h>
  81 #include <inet/ip.h>
  82 #include <inet/arp.h>
  83 #include <inet/tcp.h>
  84 #include <netinet/igmp_var.h>
  85 #include <netinet/ip_mroute.h>

  87 #include <arpa/inet.h>
  88 #include <netdb.h>
  89 #include <fcntl.h>
  90 #include <sys/systeminfo.h>
  91 #include <arpa/inet.h>

  93 #include <netinet/dhcp.h>
  94 #include <dhcpagent_ipc.h>
  95 #include <dhcpagent_util.h>
  96 #include <compat.h>

  98 #include <libtsnet.h>
  99 #include <tsol/label.h>

 101 #include "statcommon.h"

  58 extern void     unixpr(kstat_ctl_t *kc);

 104 #define STR_EXPAND      4

 106 #define V4MASK_TO_V6(v4, v6)    ((v6)._S6_un._S6_u32[0] = 0xffffffff, \
 107                                  (v6)._S6_un._S6_u32[1] = 0xffffffff, \
 108                                  (v6)._S6_un._S6_u32[2] = 0xffffffff, \
 109                                  (v6)._S6_un._S6_u32[3] = (v4))

 111 #define IN6_IS_V4MASK(v6)       ((v6)._S6_un._S6_u32[0] == 0xffffffff && \
 112                                  (v6)._S6_un._S6_u32[1] == 0xffffffff && \
 113                                  (v6)._S6_un._S6_u32[2] == 0xffffffff)

 115 /*
 116  * This is used as a cushion in the buffer allocation directed by SIOCGLIFNUM.
 117  * Because there's no locking between SIOCGLIFNUM and SIOCGLIFCONF, it's
 118  * possible for an administrator to plumb new interfaces between those two
 119  * calls, resulting in the failure of the latter.  This addition makes that
 120  * less likely.
 121  */
 122 #define LIFN_GUARD_VALUE        10

 124 typedef struct mib_item_s {
 125         struct mib_item_s       *next_item;
 126         int                     group;
```

```
        int                     mib_id;
        int                     length;
        void                    *valp;
} mib_item_t;
```

*unchanged_portion_omitted_*

```
typedef struct proc_info {
        char *pr_user;
        char *pr_fname;
        char *pr_psargs;
} proc_info_t;

#endif /* ! codereview */
static  mib_item_t      *mibget(int sd);
static  void            mibfree(mib_item_t *firstitem);
static  int             mibopen(void);
static void             mib_get_constants(mib_item_t *item);
static mib_item_t       *mib_item_dup(mib_item_t *item);
static mib_item_t       *mib_item_diff(mib_item_t *item1,
    mib_item_t *item2);
static void             mib_item_destroy(mib_item_t **item);

static boolean_t        octetstrmatch(const Octet_t *a, const Octet_t *b);
static char             *octetstr(const Octet_t *op, int code,
                            char *dst, uint_t dstlen);
static char             *pr_addr(uint_t addr,
                            char *dst, uint_t dstlen);
static char             *pr_addrnz(ipaddr_t addr, char *dst, uint_t dstlen);
static char             *pr_addr6(const in6_addr_t *addr,
                            char *dst, uint_t dstlen);
static char             *pr_mask(uint_t addr,
                            char *dst, uint_t dstlen);
static char             *pr_prefix6(const struct in6_addr *addr,
                            uint_t prefixlen, char *dst, uint_t dstlen);
static char             *pr_ap(uint_t addr, uint_t port,
                            char *proto, char *dst, uint_t dstlen);
static char             *pr_ap6(const in6_addr_t *addr, uint_t port,
                            char *proto, char *dst, uint_t dstlen);
static char             *pr_net(uint_t addr, uint_t mask,
                            char *dst, uint_t dstlen);
static char             *pr_netaddr(uint_t addr, uint_t mask,
                            char *dst, uint_t dstlen);
static char             *fmodestr(uint_t fmode);
static char             *portname(uint_t port, char *proto,
                            char *dst, uint_t dstlen);

static const char       *mitcp_state(int code,
                            const mib2_transportMLPEntry_t *attr);
static const char       *miudp_state(int code,
                            const mib2_transportMLPEntry_t *attr);

static void             stat_report(mib_item_t *item);
static void             mrt_stat_report(mib_item_t *item);
static void             arp_report(mib_item_t *item);
static void             ndp_report(mib_item_t *item);
static void             mrt_report(mib_item_t *item);
static void             if_stat_total(struct ifstat *oldstats,
                            struct ifstat *newstats, struct ifstat *sumstats);
static void             if_report(mib_item_t *item, char *ifname,
                            int Iflag_only, boolean_t once_only);
static void             if_report_ip4(mib2_ipAddrEntry_t *ap,
                            char ifname[], char logintname[],
                            struct ifstat *statptr, boolean_t ksp_not_null);
static void             if_report_ip6(mib2_ipv6AddrEntry_t *ap6,
                            char ifname[], char logintname[],
                            struct ifstat *statptr, boolean_t ksp_not_null);
```



```
static void             ire_report(const mib_item_t *item);
static void             tcp_report(const mib_item_t *item);
static void             udp_report(const mib_item_t *item);
static void             uds_report(kstat_ctl_t *);
#endif /* ! codereview */
static void             group_report(mib_item_t *item);
static void             dce_report(mib_item_t *item);
static void             print_ip_stats(mib2_ip_t *ip);
static void             print_icmp_stats(mib2_icmp_t *icmp);
static void             print_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6);
static void             print_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6);
static void             print_sctp_stats(mib2_sctp_t *tcp);
static void             print_tcp_stats(mib2_tcp_t *tcp);
static void             print_udp_stats(mib2_udp_t *udp);
static void             print_rawip_stats(mib2_rawip_t *rawip);
static void             print_igmp_stats(struct igmpstat *igps);
static void             print_mrt_stats(struct mrtstat *mrts);
static void             sctp_report(const mib_item_t *item);
static void             sum_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6,
                            mib2_ipv6IfStatsEntry_t *sum6);
static void             sum_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6,
                            mib2_ipv6IfIcmpEntry_t *sum6);
static void             m_report(void);
static void             dhcp_report(char *);

static  uint64_t        kstat_named_value(kstat_t *, char *);
static  kid_t           safe_kstat_read(kstat_ctl_t *, kstat_t *, void *);
static int              isnum(char *);
static char             *plural(int n);
static char             *pluraly(int n);
static char             *plurales(int n);
static void             process_filter(char *arg);
static char             *ifindex2str(uint_t, char *);
static boolean_t        family_selected(int family);

static void             usage(char *);
static char             *get_username(uid_t);
proc_info_t             *get_proc_info(pid_t);
#endif /* ! codereview */
static void             fatal(int errcode, char *str1, ...);

#define PLURAL(n) plural((int)n)
#define PLURALY(n) pluraly((int)n)
#define PLURALES(n) plurales((int)n)
#define IFLAGMOD(flg, val1, val2)       if (flg == val1) flg = val2
#define MDIFF(diff, elem2, elem1, member)       (diff)->member = \
        (elem2)->member - (elem1)->member


static  boolean_t       Aflag = B_FALSE;        /* All sockets/ifs/rtng-tbls */
static  boolean_t       Dflag = B_FALSE;        /* DCE info */
static  boolean_t       Iflag = B_FALSE;        /* IP Traffic Interfaces */
static  boolean_t       Mflag = B_FALSE;        /* STREAMS Memory Statistics */
static  boolean_t       Nflag = B_FALSE;        /* Numeric Network Addresses */
static  boolean_t       Rflag = B_FALSE;        /* Routing Tables */
static  boolean_t       RSECflag = B_FALSE;     /* Security attributes */
static  boolean_t       Sflag = B_FALSE;        /* Per-protocol Statistics */
static  boolean_t       Vflag = B_FALSE;        /* Verbose */
static  boolean_t       Uflag = B_FALSE;        /* Show PID and UID info. */
#endif /* ! codereview */
static  boolean_t       Pflag = B_FALSE;        /* Net to Media Tables */
static  boolean_t       Gflag = B_FALSE;        /* Multicast group membership */
static  boolean_t       MMflag = B_FALSE;       /* Multicast routing table */
static  boolean_t       DHCPflag = B_FALSE;     /* DHCP statistics */
static  boolean_t       Xflag = B_FALSE;        /* Debug Info */
```

```
272 static  int     v4compat = 0;   /* Compatible printing format for status */

274 static int      proto = IPPROTO_MAX;    /* all protocols */
275 kstat_ctl_t     *kc = NULL;

277 /*
278  * Sizes of data structures extracted from the base mib.
279  * This allows the size of the tables entries to grow while preserving
280  * binary compatibility.
281  */
282 static int ipAddrEntrySize;
283 static int ipRouteEntrySize;
284 static int ipNetToMediaEntrySize;
285 static int ipMemberEntrySize;
286 static int ipGroupSourceEntrySize;
287 static int ipRouteAttributeSize;
288 static int vifctlSize;
289 static int mfcctlSize;

291 static int ipv6IfStatsEntrySize;
292 static int ipv6IfIcmpEntrySize;
293 static int ipv6AddrEntrySize;
294 static int ipv6RouteEntrySize;
295 static int ipv6NetToMediaEntrySize;
296 static int ipv6MemberEntrySize;
297 static int ipv6GroupSourceEntrySize;

299 static int ipDestEntrySize;

301 static int transportMLPSize;
302 static int tcpConnEntrySize;
303 static int tcp6ConnEntrySize;
304 static int udpEntrySize;
305 static int udp6EntrySize;
306 static int sctpEntrySize;
307 static int sctpLocalEntrySize;
308 static int sctpRemoteEntrySize;

310 #define protocol_selected(p)    (proto == IPPROTO_MAX || proto == (p))

312 /* Machinery used for -f (filter) option */
313 enum { FK_AF = 0, FK_OUTIF, FK_DST, FK_FLAGS, NFILTERKEYS };

315 static const char *filter_keys[NFILTERKEYS] = {
316         "af", "outif", "dst", "flags"
317 };

319 static m_label_t *zone_security_label = NULL;

321 /* Flags on routes */
322 #define FLF_A           0x00000001
323 #define FLF_b           0x00000002
324 #define FLF_D           0x00000004
325 #define FLF_G           0x00000008
326 #define FLF_H           0x00000010
327 #define FLF_L           0x00000020
328 #define FLF_U           0x00000040
329 #define FLF_M           0x00000080
330 #define FLF_S           0x00000100
331 #define FLF_C           0x00000200      /* IRE_IF_CLONE */
332 #define FLF_I           0x00000400      /* RTF_INDIRECT */
333 #define FLF_R           0x00000800      /* RTF_REJECT */
334 #define FLF_B           0x00001000      /* RTF_BLACKHOLE */
335 #define FLF_Z           0x00100000      /* RTF_ZONE */

337 static const char flag_list[] = "AbDGHLUMSCIRBZ";
```



```
339 typedef struct filter_rule filter_t;

341 struct filter_rule {
342         filter_t *f_next;
343         union {
344                 int f_family;
345                 const char *f_ifname;
346                 struct {
347                         struct hostent *f_address;
348                         in6_addr_t f_mask;
349                 } a;
350                 struct {
351                         uint_t f_flagset;
352                         uint_t f_flagclear;
353                 } f;
354         } u;
355 };

357 /*
358  * The user-specified filters are linked into lists separated by
359  * keyword (type of filter).  Thus, the matching algorithm is:
360  *      For each non-empty filter list
361  *              If no filters in the list match
362  *                      then stop here; route doesn't match
363  *      If loop above completes, then route does match and will be
364  *      displayed.
365  */
366 static filter_t *filters[NFILTERKEYS];

368 static uint_t timestamp_fmt = NODATE;

370 #if !defined(TEXT_DOMAIN)               /* Should be defined by cc -D */
371 #define TEXT_DOMAIN "SYS_TEST"          /* Use this only if it isn't */
372 #endif

374 int
375 main(int argc, char **argv)
376 {
377         char            *name;
378         mib_item_t      *item = NULL;
379         mib_item_t      *previtem = NULL;
380         int             sd = -1;
381         char    *ifname = NULL;
382         int     interval = 0;   /* Single time by default */
383         int     count = -1;     /* Forever */
384         int     c;
385         int     d;
386         /*
387          * Possible values of 'Iflag_only':
388          * -1, no feature-flags;
389          *  0, IFlag and other feature-flags enabled
390          *  1, IFlag is the only feature-flag enabled
391          * : trinary variable, modified using IFLAGMOD()
392          */
393         int Iflag_only = -1;
394         boolean_t once_only = B_FALSE; /* '-i' with count > 1 */
395         extern char     *optarg;
396         extern int      optind;
397         char *default_ip_str = NULL;

399         name = argv[0];

401         v4compat = get_compat_flag(&default_ip_str);
402         if (v4compat == DEFAULT_PROT_BAD_VALUE)
403                 fatal(2, "%s: %s: Bad value for %s in %s\n", name,
```

```
404                     default_ip_str, DEFAULT_IP, INET_DEFAULT_FILE);
405         free(default_ip_str);

407         (void) setlocale(LC_ALL, "");
408         (void) textdomain(TEXT_DOMAIN);

410         while ((c = getopt(argc, argv, "adimnrspMgvuxf:P:I:DRT:")) != -1) {
102         while ((c = getopt(argc, argv, "adimnrspMgvxf:P:I:DRT:")) != -1) {
411                 switch ((char)c) {
412                 case 'a':               /* all connections */
413                         Aflag = B_TRUE;
414                         break;

416                 case 'd':               /* DCE info */
417                         Dflag = B_TRUE;
418                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
419                         break;

421                 case 'i':               /* interface (ill/ipif report) */
422                         Iflag = B_TRUE;
423                         IFLAGMOD(Iflag_only, -1, 1); /* '-i' exists */
424                         break;

426                 case 'm':               /* streams msg report */
427                         Mflag = B_TRUE;
428                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
429                         break;

431                 case 'n':               /* numeric format */
432                         Nflag = B_TRUE;
433                         break;

435                 case 'r':               /* route tables */
436                         Rflag = B_TRUE;
437                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
438                         break;

440                 case 'R':               /* security attributes */
441                         RSECflag = B_TRUE;
442                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
443                         break;

445                 case 's':               /* per-protocol statistics */
446                         Sflag = B_TRUE;
447                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
448                         break;

450                 case 'p':               /* arp/ndp table */
451                         Pflag = B_TRUE;
452                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
453                         break;

455                 case 'M':               /* multicast routing tables */
456                         MMflag = B_TRUE;
457                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
458                         break;

460                 case 'g':               /* multicast group membership */
461                         Gflag = B_TRUE;
462                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
463                         break;

465                 case 'v':               /* verbose output format */
466                         Vflag = B_TRUE;
467                         IFLAGMOD(Iflag_only, 1, 0); /* see macro def'n */
468                         break;
```



```
470                 case 'u':               /* show pid and uid information */
471                         Uflag = B_TRUE;
472                         break;

474 #endif /* ! codereview */
475                 case 'x':               /* turn on debugging */
476                         Xflag = B_TRUE;
477                         break;

479                 case 'f':
480                         process_filter(optarg);
481                         break;

483                 case 'P':
484                         if (strcmp(optarg, "ip") == 0) {
485                                 proto = IPPROTO_IP;
486                         } else if (strcmp(optarg, "ipv6") == 0 ||
487                             strcmp(optarg, "ip6") == 0) {
488                                 v4compat = 0;   /* Overridden */
489                                 proto = IPPROTO_IPV6;
490                         } else if (strcmp(optarg, "icmp") == 0) {
491                                 proto = IPPROTO_ICMP;
492                         } else if (strcmp(optarg, "icmpv6") == 0 ||
493                             strcmp(optarg, "icmp6") == 0) {
494                                 v4compat = 0;   /* Overridden */
495                                 proto = IPPROTO_ICMPV6;
496                         } else if (strcmp(optarg, "igmp") == 0) {
497                                 proto = IPPROTO_IGMP;
498                         } else if (strcmp(optarg, "udp") == 0) {
499                                 proto = IPPROTO_UDP;
500                         } else if (strcmp(optarg, "tcp") == 0) {
501                                 proto = IPPROTO_TCP;
502                         } else if (strcmp(optarg, "sctp") == 0) {
503                                 proto = IPPROTO_SCTP;
504                         } else if (strcmp(optarg, "raw") == 0 ||
505                             strcmp(optarg, "rawip") == 0) {
506                                 proto = IPPROTO_RAW;
507                         } else {
508                                 fatal(1, "%s: unknown protocol.\n", optarg);
509                         }
510                         break;

512                 case 'I':
513                         ifname = optarg;
514                         Iflag = B_TRUE;
515                         IFLAGMOD(Iflag_only, -1, 1); /* see macro def'n */
516                         break;

518                 case 'D':
519                         DHCPflag = B_TRUE;
520                         Iflag_only = 0;
521                         break;

523                 case 'T':
524                         if (optarg) {
525                                 if (*optarg == 'u')
526                                         timestamp_fmt = UDATE;
527                                 else if (*optarg == 'd')
528                                         timestamp_fmt = DDATE;
529                                 else
530                                         usage(name);
531                         } else {
532                                 usage(name);
533                         }
534                         break;
```

```
536                 case '?':
537                 default:
538                         usage(name);
539                 }
540         }

542         /*
543          * Make sure -R option is set only on a labeled system.
544          */
545         if (RSECflag && !is_system_labeled()) {
546                 (void) fprintf(stderr, "-R set but labeling is not enabled\n");
547                 usage(name);
548         }

550         /*
551          * Handle other arguments: find interval, count; the
552          * flags that accept 'interval' and 'count' are OR'd
553          * in the outermost 'if'; more flags may be added as
554          * required
555          */
556         if (Iflag || Sflag || Mflag) {
557                 for (d = optind; d < argc; d++) {
558                         if (isnum(argv[d])) {
559                                 interval = atoi(argv[d]);
560                                 if (d + 1 < argc &&
561                                     isnum(argv[d + 1])) {
562                                         count = atoi(argv[d + 1]);
563                                         optind++;
564                                 }
565                                 optind++;
566                                 if (interval == 0 || count == 0)
567                                         usage(name);
568                                 break;
569                         }
570                 }
571         }
572         if (optind < argc) {
573                 if (Iflag && isnum(argv[optind])) {
574                         count = atoi(argv[optind]);
575                         if (count == 0)
576                                 usage(name);
577                         optind++;
578                 }
579         }
580         if (optind < argc) {
581                 (void) fprintf(stderr,
582                     "%s: extra arguments\n", name);
583                 usage(name);
584         }
585         if (interval)
586                 setbuf(stdout, NULL);

588         if (DHCPflag) {
589                 dhcp_report(Iflag ? ifname : NULL);
590                 exit(0);
591         }

593         /*
594          * Get this process's security label if the -R switch is set.
595          * We use this label as the current zone's security label.
596          */
597         if (RSECflag) {
598                 zone_security_label = m_label_alloc(MAC_LABEL);
599                 if (zone_security_label == NULL)
600                         fatal(errno, "m_label_alloc() failed");
```



```
601                 if (getplabel(zone_security_label) < 0)
602                         fatal(errno, "getplabel() failed");
603         }

605         /* Get data structures: priming before iteration */
606         if (family_selected(AF_INET) || family_selected(AF_INET6)) {
607                 sd = mibopen();
608                 if (sd == -1)
609                         fatal(1, "can't open mib stream\n");
610                 if ((item = mibget(sd)) == NULL) {
611                         (void) close(sd);
612                         fatal(1, "mibget() failed\n");
613                 }
614                 /* Extract constant sizes - need do once only */
615                 mib_get_constants(item);
616         }
617         if ((kc = kstat_open()) == NULL) {
618                 mibfree(item);
619                 (void) close(sd);
620                 fail(1, "kstat_open(): can't open /dev/kstat");
621         }

623         if (interval <= 0) {
624                 count = 1;
625                 once_only = B_TRUE;
626         }
627         /* 'for' loop 1: */
628         for (;;) {
629                 mib_item_t *curritem = NULL; /* only for -[M]s */

631                 if (timestamp_fmt != NODATE)
632                         print_timestamp(timestamp_fmt);

634                 /* netstat: AF_INET[6] behaviour */
635                 if (family_selected(AF_INET) || family_selected(AF_INET6)) {
636                         if (Sflag) {
637                                 curritem = mib_item_diff(previtem, item);
638                                 if (curritem == NULL)
639                                         fatal(1, "can't process mib data, "
640                                             "out of memory\n");
641                                 mib_item_destroy(&previtem);
642                         }

644                         if (!(Dflag || Iflag || Rflag || Sflag || Mflag ||
645                             MMflag || Pflag || Gflag || DHCPflag)) {
646                                 if (protocol_selected(IPPROTO_UDP))
647                                         udp_report(item);
648                                 if (protocol_selected(IPPROTO_TCP))
649                                         tcp_report(item);
650                                 if (protocol_selected(IPPROTO_SCTP))
651                                         sctp_report(item);
652                         }
653                         if (Iflag)
654                                 if_report(item, ifname, Iflag_only, once_only);
655                         if (Mflag)
656                                 m_report();
657                         if (Rflag)
658                                 ire_report(item);
659                         if (Sflag && MMflag) {
660                                 mrt_stat_report(curritem);
661                         } else {
662                                 if (Sflag)
663                                         stat_report(curritem);
664                                 if (MMflag)
665                                         mrt_report(item);
666                         }
```

```
                        if (Gflag)
                                group_report(item);
                        if (Pflag) {
                                if (family_selected(AF_INET))
                                        arp_report(item);
                                if (family_selected(AF_INET6))
                                        ndp_report(item);
                        }
                        if (Dflag)
                                dce_report(item);
                        mib_item_destroy(&curritem);
                }

                /* netstat: AF_UNIX behaviour */
                if (family_selected(AF_UNIX) &&
                    (!(Dflag || Iflag || Rflag || Sflag || Mflag ||
                    MMflag || Pflag || Gflag)))
                        uds_report(kc);
                        unixpr(kc);
                (void) kstat_close(kc);

                /* iteration handling code */
                if (count > 0 && --count == 0)
                        break;
                (void) sleep(interval);

                /* re-populating of data structures */
                if (family_selected(AF_INET) || family_selected(AF_INET6)) {
                        if (Sflag) {
                                /* previtem is a cut-down list */
                                previtem = mib_item_dup(item);
                                if (previtem == NULL)
                                        fatal(1, "can't process mib data, "
                                            "out of memory\n");
                        }
                        mibfree(item);
                        (void) close(sd);
                        if ((sd = mibopen()) == -1)
                                fatal(1, "can't open mib stream anymore\n");
                        if ((item = mibget(sd)) == NULL) {
                                (void) close(sd);
                                fatal(1, "mibget() failed\n");
                        }
                }
                if ((kc = kstat_open()) == NULL)
                        fail(1, "kstat_open(): can't open /dev/kstat");

        } /* 'for' loop 1 ends */
        mibfree(item);
        (void) close(sd);
        if (zone_security_label != NULL)
                m_label_free(zone_security_label);

        return (0);
}
```

unchanged_portion_omitted_

```
/* ------------------------------ TCP_REPORT------------------------------ */

static const char tcp_hdr_v4[] =
"\nTCP: IPv4\n";
static const char tcp_hdr_v4_compat[] =
"\nTCP\n";
static const char tcp_hdr_v4_verbose[] =
"Local/Remote Address Swind  Snext     Suna   Rwind  Rnext     Rack   "
" Rto   Mss     State\n"
```



```
"-------------------- ----- -------- -------- ----- -------- -------- "
"----- ----- -----------\n";
static const char tcp_hdr_v4_normal[] =
"   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q "
"   State\n"
"-------------------- -------------------- ----- ------ ----- ------ "
"-----------\n";
static const char tcp_hdr_v4_pid[] =
"   Local Address        Remote Address      User     Pid     Command     Swind"
"   Send-Q  Rwind  Recv-Q    State\n"
"-------------------- -------------------- -------- ------ ------------ ------"
"- ------ ------- ------ -----------\n";
static const char tcp_hdr_v4_pid_verbose[] =
"Local/Remote Address  Swind   Snext     Suna    Rwind   Rnext     Rack    Rto "
"  Mss     State      User    Pid      Command\n"
"-------------------- ------- -------- -------- ------- -------- -------- -----"
" ----- ----------- -------- ------ --------------\n";
#endif /* ! codereview */

static const char tcp_hdr_v6[] =
"\nTCP: IPv6\n";
static const char tcp_hdr_v6_verbose[] =
"Local/Remote Address               Swind  Snext     Suna   Rwind  Rnext   "
"  Rack    Rto   Mss    State      If\n"
"--------------------------------- ----- -------- -------- ----- -------- "
"-------- ----- ----- ----------- -----\n";
static const char tcp_hdr_v6_normal[] =
"   Local Address                         Remote Address                "
"Swind Send-Q Rwind Recv-Q   State      If\n"
"--------------------------------- ------------------------------- "
"----- ------ ----- ------ ----------- -----\n";
static const char tcp_hdr_v6_pid[] =
"   Local Address                         Remote Address                   User"
"    Pid      Command      Swind  Send-Q  Rwind  Recv-Q   State      If\n"
"--------------------------------- ------------------------------- --------"
" ------ -------------- ------- ------ ------- ------ ----------- -----\n";
static const char tcp_hdr_v6_pid_verbose[] =
"Local/Remote Address               Swind   Snext     Suna    Rwind   Rnext"
"     Rack    Rto   Mss    State      If     User    Pid     Command\n"
"--------------------------------- ------- -------- -------- ------- --------"
" -------- ----- ----- ----------- ----- -------- ------ --------------\n";
#endif /* ! codereview */

static boolean_t tcp_report_item_v4(const mib2_tcpConnEntry_t *,
    conn_pid_info_t *, boolean_t first,
    const mib2_transportMLPEntry_t *);
    boolean_t first, const mib2_transportMLPEntry_t *);
static boolean_t tcp_report_item_v6(const mib2_tcp6ConnEntry_t *,
    conn_pid_info_t *, boolean_t first,
    const mib2_transportMLPEntry_t *);

    boolean_t first, const mib2_transportMLPEntry_t *);

static void
tcp_report(const mib_item_t *item)
{
        int                             jtemp = 0;
        boolean_t                       print_hdr_once_v4 = B_TRUE;
        boolean_t                       print_hdr_once_v6 = B_TRUE;
        mib2_tcpConnEntry_t             *tp;
        mib2_tcp6ConnEntry_t            *tp6;
        mib2_transportMLPEntry_t        **v4_attrs, **v6_attrs;
        mib2_transportMLPEntry_t        **v4a, **v6a;
        mib2_transportMLPEntry_t        *aptr;
        conn_pid_info_t                 *cpi;
#endif /* ! codereview */
```

```
4830         if (!protocol_selected(IPPROTO_TCP))
4831                 return;

4833         /*
4834          * Preparation pass: the kernel returns separate entries for TCP
4835          * connection table entries and Multilevel Port attributes.  We loop
4836          * through the attributes first and set up an array for each address
4837          * family.
4838          */
4839         v4_attrs = family_selected(AF_INET) && RSECflag ?
4840             gather_attrs(item, MIB2_TCP, MIB2_TCP_CONN, tcpConnEntrySize) :
4841             NULL;
4842         v6_attrs = family_selected(AF_INET6) && RSECflag ?
4843             gather_attrs(item, MIB2_TCP6, MIB2_TCP6_CONN, tcp6ConnEntrySize) :
4844             NULL;

4846         /* 'for' loop 1: */
4847         v4a = v4_attrs;
4848         v6a = v6_attrs;
4849         for (; item != NULL; item = item->next_item) {
4850                 if (Xflag) {
4851                         (void) printf("\n--- Entry %d ---\n", ++jtemp);
4852                         (void) printf("Group = %d, mib_id = %d, "
4853                             "length = %d, valp = 0x%p\n",
4854                             item->group, item->mib_id,
4855                             item->length, item->valp);
4856                 }

4858                 if (!((item->group == MIB2_TCP &&
4859                     item->mib_id == MIB2_TCP_CONN) ||
4860                     (item->group == MIB2_TCP6 &&
4861                     item->mib_id == MIB2_TCP6_CONN) ||
4862                     (item->group == MIB2_TCP &&
4863                     item->mib_id == EXPER_XPORT_PROC_INFO) ||
4864                     (item->group == MIB2_TCP6 &&
4865                     item->mib_id == EXPER_XPORT_PROC_INFO)))
4265                     item->mib_id == MIB2_TCP6_CONN)))
4866                         continue; /* 'for' loop 1 */

4868                 if (item->group == MIB2_TCP && !family_selected(AF_INET))
4869                         continue; /* 'for' loop 1 */
4870                 else if (item->group == MIB2_TCP6 && !family_selected(AF_INET6))
4871                         continue; /* 'for' loop 1 */

4873                 if ((!Uflag) && item->group == MIB2_TCP &&
4874                     item->mib_id == MIB2_TCP_CONN) {
4273                 if (item->group == MIB2_TCP) {
4875                         for (tp = (mib2_tcpConnEntry_t *)item->valp;
4876                             (char *)tp < (char *)item->valp + item->length;
4877                             /* LINTED: (note 1) */
4878                             tp = (mib2_tcpConnEntry_t *)((char *)tp +
4879                             tcpConnEntrySize)) {
4880                                 aptr = v4a == NULL ? NULL : *v4a++;
4881                                 print_hdr_once_v4 = tcp_report_item_v4(tp,
4882                                     NULL, print_hdr_once_v4, aptr);
4281                                     print_hdr_once_v4, aptr);
4883                         }
4884                 } else if ((!Uflag) && item->group == MIB2_TCP6 &&
4885                     item->mib_id == MIB2_TCP6_CONN) {
4283                 } else {
4886                         for (tp6 = (mib2_tcp6ConnEntry_t *)item->valp;
4887                             (char *)tp6 < (char *)item->valp + item->length;
4888                             /* LINTED: (note 1) */
4889                             tp6 = (mib2_tcp6ConnEntry_t *)((char *)tp6 +
4890                             tcp6ConnEntrySize)) {
```



```
4891                                 aptr = v6a == NULL ? NULL : *v6a++;
4892                                 print_hdr_once_v6 = tcp_report_item_v6(tp6,
4893                                     NULL, print_hdr_once_v6, aptr);
4894                         }
4895                 } else if ((Uflag) && item->group == MIB2_TCP &&
4896                     item->mib_id == EXPER_XPORT_PROC_INFO) {
4897                         for (tp = (mib2_tcpConnEntry_t *)item->valp;
4898                             (char *)tp < (char *)item->valp + item->length;
4899                             /* LINTED: (note 1) */
4900                             tp = (mib2_tcpConnEntry_t *)((char *)cpi +
4901                             cpi->cpi_tot_size)) {
4902                                 aptr = v4a == NULL ? NULL : *v4a++;
4903                                 /* LINTED: (note 1) */
4904                                 cpi = (conn_pid_info_t *) ((char *)tp +
4905                                     tcpConnEntrySize);
4906                                 print_hdr_once_v4 = tcp_report_item_v4(tp,
4907                                     cpi, print_hdr_once_v4, aptr);
4908                         }
4909                 } else if ((Uflag) && item->group == MIB2_TCP6 &&
4910                     item->mib_id == EXPER_XPORT_PROC_INFO) {
4911                         for (tp6 = (mib2_tcp6ConnEntry_t *)item->valp;
4912                             (char *)tp6 < (char *)item->valp + item->length;
4913                             /* LINTED: (note 1) */
4914                             tp6 = (mib2_tcp6ConnEntry_t *)((char *)cpi +
4915                             cpi->cpi_tot_size)) {
4916                                 aptr = v6a == NULL ? NULL : *v6a++;
4917                                 /* LINTED: (note 1) */
4918                                 cpi = (conn_pid_info_t *) ((char *)tp6 +
4919                                     tcp6ConnEntrySize);
4920                                 print_hdr_once_v6 = tcp_report_item_v6(tp6,
4921                                     cpi, print_hdr_once_v6, aptr);
4291                                     print_hdr_once_v6, aptr);
4922                         }
4923                 }

4925 #endif /* ! codereview */
4926         } /* 'for' loop 1 ends */
4927         (void) fflush(stdout);

4929         if (v4_attrs != NULL)
4930                 free(v4_attrs);
4931         if (v6_attrs != NULL)
4932                 free(v6_attrs);
4933 }

4935 static boolean_t
4936 tcp_report_item_v4(const mib2_tcpConnEntry_t *tp, conn_pid_info_t * cpi,
4937     boolean_t first, const mib2_transportMLPEntry_t *attr)
4294 tcp_report_item_v4(const mib2_tcpConnEntry_t *tp, boolean_t first,
4295     const mib2_transportMLPEntry_t *attr)
4938 {
4939         /*
4940          * lname and fname below are for the hostname as well as the portname
4941          * There is no limit on portname length so we assume MAXHOSTNAMELEN
4942          * as the limit
4943          */
4944         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4945         char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];


4948 #endif /* ! codereview */
4949         if (!(Aflag || tp->tcpConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
4950                 return (first); /* Nothing to print */

4952         if (first) {
4953                 (void) printf(v4compat ? tcp_hdr_v4_compat : tcp_hdr_v4);
```

```
4954                if (Uflag)
4955                        (void) printf(Vflag ? tcp_hdr_v4_pid_verbose :
4956                                tcp_hdr_v4_pid);
4957                else
4958                        (void) printf(Vflag ? tcp_hdr_v4_verbose :
4959                                tcp_hdr_v4_normal);
4305                (void) printf(Vflag ? tcp_hdr_v4_verbose : tcp_hdr_v4_normal);
4960        }

4962        if ((!Uflag) && Vflag) {
4308        if (Vflag) {
4963                (void) printf("%-20s\n%-20s %5u %08x %08x %5u %08x %08x "
4964                    "%5u %5u %s\n",
4965                    pr_ap(tp->tcpConnLocalAddress,
4966                    tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4967                    pr_ap(tp->tcpConnRemAddress,
4968                    tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4969                    tp->tcpConnEntryInfo.ce_swnd,
4970                    tp->tcpConnEntryInfo.ce_snxt,
4971                    tp->tcpConnEntryInfo.ce_suna,
4972                    tp->tcpConnEntryInfo.ce_rwnd,
4973                    tp->tcpConnEntryInfo.ce_rnxt,
4974                    tp->tcpConnEntryInfo.ce_rack,
4975                    tp->tcpConnEntryInfo.ce_rto,
4976                    tp->tcpConnEntryInfo.ce_mss,
4977                    mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4978        } else if ((!Uflag) && (!Vflag)) {
4324        } else {
4979                int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
4980                    (int)tp->tcpConnEntryInfo.ce_suna - 1;
4981                int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
4982                    (int)tp->tcpConnEntryInfo.ce_rack;

4984                (void) printf("%-20s %-20s %5u %6d %5u %6d %s\n",
4985                    pr_ap(tp->tcpConnLocalAddress,
4986                    tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4987                    pr_ap(tp->tcpConnRemAddress,
4988                    tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4989                    tp->tcpConnEntryInfo.ce_swnd,
4990                    (sq >= 0) ? sq : 0,
4991                    tp->tcpConnEntryInfo.ce_rwnd,
4992                    (rq >= 0) ? rq : 0,
4993                    mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4994        } else if (Uflag && Vflag) {
4995                int i = 0;
4996                pid_t *pids = cpi->cpi_pids;
4997                proc_info_t *pinfo;
4998                do {
4999                        pinfo = get_proc_info(*pids);
5000                        (void) printf("%-20s\n%-20s %7u %08x %08x %7u %08x %08x
5001                            "%5u %5u %-11s %-8.8s %6u %s\n",
5002                            pr_ap(tp->tcpConnLocalAddress,
5003                            tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
5004                            pr_ap(tp->tcpConnRemAddress,
5005                            tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
5006                            tp->tcpConnEntryInfo.ce_swnd,
5007                            tp->tcpConnEntryInfo.ce_snxt,
5008                            tp->tcpConnEntryInfo.ce_suna,
5009                            tp->tcpConnEntryInfo.ce_rwnd,
5010                            tp->tcpConnEntryInfo.ce_rnxt,
5011                            tp->tcpConnEntryInfo.ce_rack,
5012                            tp->tcpConnEntryInfo.ce_rto,
5013                            tp->tcpConnEntryInfo.ce_mss,
5014                            mitcp_state(tp->tcpConnEntryInfo.ce_state, attr),
5015                            pinfo->pr_user, (int)*pids, pinfo->pr_psargs);
5016                        i++; pids++;
```



```
5017                } while (i < cpi->cpi_pids_cnt);
5018        } else if (Uflag && (!Vflag)) {
5019                int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
5020                    (int)tp->tcpConnEntryInfo.ce_suna - 1;
5021                int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
5022                   (int)tp->tcpConnEntryInfo.ce_rack;
5023                int i = 0;
5024                pid_t *pids = cpi->cpi_pids;
5025                proc_info_t *pinfo;
5026                do {
5027                        pinfo = get_proc_info(*pids);
5028                        (void) printf("%-20s %-20s %-8.8s %6u %-13.13s %7u %6d %
5029                            pr_ap(tp->tcpConnLocalAddress,
5030                            tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
5031                            pr_ap(tp->tcpConnRemAddress,
5032                            tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
5033                            pinfo->pr_user, (int)*pids, pinfo->pr_fname,
5034                            tp->tcpConnEntryInfo.ce_swnd,
5035                            (sq >= 0) ? sq : 0,
5036                            tp->tcpConnEntryInfo.ce_rwnd,
5037                            (rq >= 0) ? rq : 0,
5038                            mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
5039                        i++; pids++;
5040                } while (i < cpi->cpi_pids_cnt);
5041 #endif /* ! codereview */
5042        }

5044        print_transport_label(attr);

5046        return (B_FALSE);
5047 }

5049 static boolean_t
5050 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6, conn_pid_info_t *cpi,
5051     boolean_t first, const mib2_transportMLPEntry_t *attr)
4340 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6, boolean_t first,
4341     const mib2_transportMLPEntry_t *attr)
5052 {
5053        /*
5054         * lname and fname below are for the hostname as well as the portname
5055         * There is no limit on portname length so we assume MAXHOSTNAMELEN
5056         * as the limit
5057         */
5058        char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5059        char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5060        char    ifname[LIFNAMSIZ + 1];
5061        char    *ifnamep;

5063        if (!(Aflag || tp6->tcp6ConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
5064                return (first); /* Nothing to print */

5066        if (first) {
5067                (void) printf(tcp_hdr_v6);
5068                if (Uflag)
5069                        (void) printf(Vflag ? tcp_hdr_v6_pid_verbose :
5070                                tcp_hdr_v6_pid);
5071                else
5072                        (void) printf(Vflag ? tcp_hdr_v6_verbose :
5073                                tcp_hdr_v6_normal);
4358                (void) printf(Vflag ? tcp_hdr_v6_verbose : tcp_hdr_v6_normal);
5074        }

5076        ifnamep = (tp6->tcp6ConnIfIndex != 0) ?
5077            if_indextoname(tp6->tcp6ConnIfIndex, ifname) : NULL;
5078        if (ifnamep == NULL)
5079                ifnamep = "";
```

```
5081          if ((!Uflag) && Vflag) {
4366          if (Vflag) {
5082                  (void) printf("%-33s\n%-33s %5u %08x %08x %5u %08x %08x "
5083                      "%5u %5u %-11s %s\n",
5084                      pr_ap6(&tp6->tcp6ConnLocalAddress,
5085                      tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
5086                      pr_ap6(&tp6->tcp6ConnRemAddress,
5087                      tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5088                      tp6->tcp6ConnEntryInfo.ce_swnd,
5089                      tp6->tcp6ConnEntryInfo.ce_snxt,
5090                      tp6->tcp6ConnEntryInfo.ce_suna,
5091                      tp6->tcp6ConnEntryInfo.ce_rwnd,
5092                      tp6->tcp6ConnEntryInfo.ce_rnxt,
5093                      tp6->tcp6ConnEntryInfo.ce_rack,
5094                      tp6->tcp6ConnEntryInfo.ce_rto,
5095                      tp6->tcp6ConnEntryInfo.ce_mss,
5096                      mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5097                      ifnamep);
5098          } else if ((!Uflag) && (!Vflag)) {
4383          } else {
5099                  int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
5100                      (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
5101                  int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
5102                      (int)tp6->tcp6ConnEntryInfo.ce_rack;

5104                  (void) printf("%-33s %-33s %5u %6d %5u %6d %-11s %s\n",
5105                      pr_ap6(&tp6->tcp6ConnLocalAddress,
5106                      tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
5107                      pr_ap6(&tp6->tcp6ConnRemAddress,
5108                      tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5109                      tp6->tcp6ConnEntryInfo.ce_swnd,
5110                      (sq >= 0) ? sq : 0,
5111                      tp6->tcp6ConnEntryInfo.ce_rwnd,
5112                      (rq >= 0) ? rq : 0,
5113                      mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5114                      ifnamep);
5115          } else if (Uflag && Vflag) {
5116                  int i = 0;
5117                  pid_t *pids = cpi->cpi_pids;
5118                  proc_info_t *pinfo;
5119                  do {
5120                          pinfo = get_proc_info(*pids);
5121                          (void) printf("%-33s\n%-33s %7u %08x %08x %7u %08x %08x
5122                              "%5u %5u %-11s %-5.5s %-8.8s %6u %s\n",
5123                              pr_ap6(&tp6->tcp6ConnLocalAddress,
5124                              tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)
5125                              pr_ap6(&tp6->tcp6ConnRemAddress,
5126                              tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5127                              tp6->tcp6ConnEntryInfo.ce_swnd,
5128                              tp6->tcp6ConnEntryInfo.ce_snxt,
5129                              tp6->tcp6ConnEntryInfo.ce_suna,
5130                              tp6->tcp6ConnEntryInfo.ce_rwnd,
5131                              tp6->tcp6ConnEntryInfo.ce_rnxt,
5132                              tp6->tcp6ConnEntryInfo.ce_rack,
5133                              tp6->tcp6ConnEntryInfo.ce_rto,
5134                              tp6->tcp6ConnEntryInfo.ce_mss,
5135                              mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5136                              ifnamep, pinfo->pr_user, (int)*pids, pinfo->pr_psarg
5137                          i++; pids++;
5138                  } while (i < cpi->cpi_pids_cnt);
5139          } else if (Uflag && (!Vflag)) {
5140                  int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
5141                      (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
5142                  int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
5143                      (int)tp6->tcp6ConnEntryInfo.ce_rack;
```



```
5144                  int i = 0;
5145                  pid_t *pids = cpi->cpi_pids;
5146                  proc_info_t *pinfo;
5147                  do {
5148                          pinfo = get_proc_info(*pids);
5149                          (void) printf("%-33s %-33s %-8.8s %6u %-14.14s %7d %6u %
5150                              pr_ap6(&tp6->tcp6ConnLocalAddress,
5151                              tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)
5152                              pr_ap6(&tp6->tcp6ConnRemAddress,
5153                              tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5154                              pinfo->pr_user, (int)*pids, pinfo->pr_fname,
5155                              tp6->tcp6ConnEntryInfo.ce_swnd,
5156                              (sq >= 0) ? sq : 0,
5157                              tp6->tcp6ConnEntryInfo.ce_rwnd,
5158                              (rq >= 0) ? rq : 0,
5159                              mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5160                              ifnamep);
5161                          i++; pids++;
5162                  } while (i < cpi->cpi_pids_cnt);
5163  #endif /* ! codereview */
5164          }

5166          print_transport_label(attr);

5168          return (B_FALSE);
5169  }

5171  /* ------------------------------ UDP_REPORT------------------------------ */

5173  static boolean_t udp_report_item_v4(const mib2_udpEntry_t *ude,
5174      conn_pid_info_t *cpi, boolean_t first,
5175      const mib2_transportMLPEntry_t *attr);
4400      boolean_t first, const mib2_transportMLPEntry_t *attr);
5176  static boolean_t udp_report_item_v6(const mib2_udp6Entry_t *ude6,
5177      conn_pid_info_t *cpi, boolean_t first,
5178      const mib2_transportMLPEntry_t *attr);
4402      boolean_t first, const mib2_transportMLPEntry_t *attr);

5180  static const char udp_hdr_v4[] =
5181  "   Local Address        Remote Address      State\n"
5182  "-------------------- -------------------- ----------\n";
5183  static const char udp_hdr_v4_pid[] =
5184  "   Local Address        Remote Address      User    Pid   "
5185  "   Command       State\n"
5186  "-------------------- -------------------- -------- ------ "
5187  "-------------- ----------\n";
5188  static const char udp_hdr_v4_pid_verbose[] =
5189  "   Local Address        Remote Address      User    Pid     State    "
5190  "   Command\n"
5191  "-------------------- -------------------- -------- ------ ---------- "
5192  "---------------\n";
5193  #endif /* ! codereview */

5195  static const char udp_hdr_v6[] =
5196  "   Local Address                       Remote Address                "
5197  "  State      If\n"
5198  "--------------------------------- ------------------------------- "
5199  "---------- -----\n";
5200  static const char udp_hdr_v6_pid[] =
5201  "   Local Address                       Remote Address                "
5202  "  User    Pid      Command       State      If\n"
5203  "--------------------------------- ------------------------------- "
5204  "-------- ------ -------------- ---------- -----\n";
5205  static const char udp_hdr_v6_pid_verbose[] =
5206  "   Local Address                       Remote Address                "
5207  "  User    Pid     State      If     Command\n"
```

```
"-------------------------------- -------------------------------- "
"-------- ------ ---------- ----- ----------------\n";

#endif /* ! codereview */

static void
udp_report(const mib_item_t *item)
{
        int                             jtemp = 0;
        boolean_t                       print_hdr_once_v4 = B_TRUE;
        boolean_t                       print_hdr_once_v6 = B_TRUE;
        mib2_udpEntry_t                 *ude;
        mib2_udp6Entry_t                *ude6;
        mib2_transportMLPEntry_t        **v4_attrs, **v6_attrs;
        mib2_transportMLPEntry_t        **v4a, **v6a;
        mib2_transportMLPEntry_t        *aptr;
        conn_pid_info_t                 *cpi;
#endif /* ! codereview */

        if (!protocol_selected(IPPROTO_UDP))
                return;

        /*
         * Preparation pass: the kernel returns separate entries for UDP
         * connection table entries and Multilevel Port attributes.  We loop
         * through the attributes first and set up an array for each address
         * family.
         */
        v4_attrs = family_selected(AF_INET) && RSECflag ?
            gather_attrs(item, MIB2_UDP, MIB2_UDP_ENTRY, udpEntrySize) : NULL;
        v6_attrs = family_selected(AF_INET6) && RSECflag ?
            gather_attrs(item, MIB2_UDP6, MIB2_UDP6_ENTRY, udp6EntrySize) :
            NULL;

        v4a = v4_attrs;
        v6a = v6_attrs;
        /* 'for' loop 1: */
        for (; item; item = item->next_item) {
                if (Xflag) {
                        (void) printf("\n--- Entry %d ---\n", ++jtemp);
                        (void) printf("Group = %d, mib_id = %d, "
                            "length = %d, valp = 0x%p\n",
                            item->group, item->mib_id,
                            item->length, item->valp);
                }
                if (!((item->group == MIB2_UDP &&
                    item->mib_id == MIB2_UDP_ENTRY) ||
                    (item->group == MIB2_UDP6 &&
                    item->mib_id == MIB2_UDP6_ENTRY) ||
                    (item->group == MIB2_UDP &&
                    item->mib_id == EXPER_XPORT_PROC_INFO) ||
                    (item->group == MIB2_UDP6 &&
                    item->mib_id == EXPER_XPORT_PROC_INFO)))
                    item->mib_id == MIB2_UDP6_ENTRY)))
                        continue; /* 'for' loop 1 */

                if (item->group == MIB2_UDP && !family_selected(AF_INET))
                        continue; /* 'for' loop 1 */
                else if (item->group == MIB2_UDP6 && !family_selected(AF_INET6))
                        continue; /* 'for' loop 1 */

                /*      xxx.xxx.xxx.xxx,pppp  sss... */
                if ((!Uflag) && item->group == MIB2_UDP &&
                    item->mib_id == MIB2_UDP_ENTRY) {
                if (item->group == MIB2_UDP) {
                        for (ude = (mib2_udpEntry_t *)item->valp;
```



```
                            (char *)ude < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            ude = (mib2_udpEntry_t *)((char *)ude +
                            udpEntrySize)) {
                                aptr = v4a == NULL ? NULL : *v4a++;
                                print_hdr_once_v4 = udp_report_item_v4(ude,
                                    NULL, print_hdr_once_v4, aptr);
                                    print_hdr_once_v4, aptr);
                        }
                } else if ((!Uflag) && item->group == MIB2_UDP6 &&
                     item->mib_id == MIB2_UDP6_ENTRY) {
                } else {
                        for (ude6 = (mib2_udp6Entry_t *)item->valp;
                            (char *)ude6 < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            ude6 = (mib2_udp6Entry_t *)((char *)ude6 +
                            udp6EntrySize)) {
                                aptr = v6a == NULL ? NULL : *v6a++;
                                print_hdr_once_v6 = udp_report_item_v6(ude6,
                                    NULL, print_hdr_once_v6, aptr);
                        }
                } else if ((Uflag) && item->group == MIB2_UDP &&
                    item->mib_id == EXPER_XPORT_PROC_INFO) {
                        for (ude = (mib2_udpEntry_t *)item->valp;
                            (char *)ude < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            ude = (mib2_udpEntry_t *)((char *)cpi +
                            cpi->cpi_tot_size)) {
                                aptr = v4a == NULL ? NULL : *v4a++;
                                /* LINTED: (note 1) */
                                cpi = (conn_pid_info_t *) ((char *)ude +
                                    udpEntrySize);
                                print_hdr_once_v4 = udp_report_item_v4(ude,
                                    cpi, print_hdr_once_v4, aptr);
                        }
                } else if ((Uflag) && item->group == MIB2_UDP6 &&
                    item->mib_id == EXPER_XPORT_PROC_INFO) {
                        for (ude6 = (mib2_udp6Entry_t *)item->valp;
                            (char *)ude6 < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            ude6 = (mib2_udp6Entry_t *)((char *)cpi +
                            cpi->cpi_tot_size)) {
                                aptr = v6a == NULL ? NULL : *v6a++;
                                /* LINTED: (note 1) */
                                cpi = (conn_pid_info_t *) ((char *)ude6 +
                                    udp6EntrySize);
                                print_hdr_once_v6 = udp_report_item_v6(ude6,
                                    cpi, print_hdr_once_v6, aptr);
                                    print_hdr_once_v6, aptr);
                        }
                }
        } /* 'for' loop 1 ends */
        (void) fflush(stdout);

        if (v4_attrs != NULL)
                free(v4_attrs);
        if (v6_attrs != NULL)
                free(v6_attrs);
}

static boolean_t
udp_report_item_v4(const mib2_udpEntry_t *ude, conn_pid_info_t *cpi,
    boolean_t first, const mib2_transportMLPEntry_t *attr)
udp_report_item_v4(const mib2_udpEntry_t *ude, boolean_t first,
    const mib2_transportMLPEntry_t *attr)
{
```

```
5333         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5334                         /* hostname + portname */

5336         if (!(Aflag || ude->udpEntryInfo.ue_state >= MIB2_UDP_connected))
5337                 return (first); /* Nothing to print */

5339         if (first) {
5340                 (void) printf(v4compat ? "\nUDP\n" : "\nUDP: IPv4\n");

5342                 if (Uflag)
5343                         (void) printf(Vflag ? udp_hdr_v4_pid_verbose :
5344                             udp_hdr_v4_pid);
5345                 else
5346 #endif /* ! codereview */
5347                         (void) printf(udp_hdr_v4);

5349 #endif /* ! codereview */
5350                 first = B_FALSE;
5351         }

5353         (void) printf("%-20s %-20s ",
4458         (void) printf("%-20s ",
5354             pr_ap(ude->udpLocalAddress, ude->udpLocalPort, "udp",
5355             lname, sizeof (lname)),
4460             lname, sizeof (lname)));
4461         (void) printf("%-20s %s\n",
5356             ude->udpEntryInfo.ue_state == MIB2_UDP_connected ?
5357             pr_ap(ude->udpEntryInfo.ue_RemoteAddress,
5358             ude->udpEntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5359             "");
5360         if (!Uflag) {
5361                 (void) printf("%s\n",
4465             "",
5362                     miudp_state(ude->udpEntryInfo.ue_state, attr));
5363         } else {
5364                 int i = 0;
5365                 pid_t *pids = cpi->cpi_pids;
5366                 proc_info_t *pinfo;
5367                 do {
5368                         pinfo = get_proc_info(*pids);
5369                         (void) printf("%-8.8s %6u ", pinfo->pr_user,
5370                             (int)*pids);
5371                         if (Vflag) {
5372                                 (void) printf("%-10.10s %s\n",
5373                                     miudp_state(ude->udpEntryInfo.ue_state,
5374                                     attr),
5375                                     pinfo->pr_psargs);
5376                         } else {
5377                                 (void) printf("%-14.14s %s\n", pinfo->pr_fname,
5378                                   miudp_state(ude->udpEntryInfo.ue_state,
5379                                   attr));
5380                         }
5381                         i++; pids++;
5382                 } while (i < cpi->cpi_pids_cnt);
5383         }
5384 #endif /* ! codereview */

5386         print_transport_label(attr);

5388         return (first);
5389 }

5391 static boolean_t
5392 udp_report_item_v6(const mib2_udp6Entry_t *ude6, conn_pid_info_t *cpi,
5393     boolean_t first, const mib2_transportMLPEntry_t *attr)
4467 udp_report_item_v6(const mib2_udp6Entry_t *ude6, boolean_t first,
```



```
4468     const mib2_transportMLPEntry_t *attr)
5394 {
5395         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5396                         /* hostname + portname */
5397         char    ifname[LIFNAMSIZ + 1];
5398         const char *ifnamep;

5400         if (!(Aflag || ude6->udp6EntryInfo.ue_state >= MIB2_UDP_connected))
5401                 return (first); /* Nothing to print */

5403         if (first) {
5404                 (void) printf("\nUDP: IPv6\n");

5406                 if (Uflag)
5407                         (void) printf(Vflag ? udp_hdr_v6_pid_verbose :
5408                             udp_hdr_v6_pid);
5409                 else
5410 #endif /* ! codereview */
5411                         (void) printf(udp_hdr_v6);

5413 #endif /* ! codereview */
5414                 first = B_FALSE;
5415         }

5417         ifnamep = (ude6->udp6IfIndex != 0) ?
5418             if_indextoname(ude6->udp6IfIndex, ifname) : NULL;

5420         (void) printf("%-33s %-33s ",
4480         (void) printf("%-33s ",
5421             pr_ap6(&ude6->udp6LocalAddress,
5422             ude6->udp6LocalPort, "udp", lname, sizeof (lname)),
4482             ude6->udp6LocalPort, "udp", lname, sizeof (lname)));
4483         (void) printf("%-33s %-10s %s\n",
5423             ude6->udp6EntryInfo.ue_state == MIB2_UDP_connected ?
5424             pr_ap6(&ude6->udp6EntryInfo.ue_RemoteAddress,
5425             ude6->udp6EntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5426             "");
5427         if (!Uflag) {
5428                 (void) printf("%-10s %s\n",
4487             "",
5429                     miudp_state(ude6->udp6EntryInfo.ue_state, attr),
5430                     ifnamep == NULL ? "" : ifnamep);
5431         } else {
5432                 int i = 0;
5433                 pid_t *pids = cpi->cpi_pids;
5434                 proc_info_t *pinfo;
5435                 do {
5436                         pinfo = get_proc_info(*pids);
5437                         (void) printf("%-8.8s %6u ", pinfo->pr_user,
5438                             (int)*pids);
5439                         if (Vflag) {
5440                                 (void) printf("%-10.10s %-5.5s %s\n",
5441                                     miudp_state(ude6->udp6EntryInfo.ue_state,
5442                                     attr),
5443                                     ifnamep == NULL ? "" : ifnamep,
5444                                     pinfo->pr_psargs);
5445                         } else {
5446                                 (void) printf("%-14.14s %-10.10s %s\n",
5447                                     pinfo->pr_fname,
5448                                     miudp_state(ude6->udp6EntryInfo.ue_state,
5449                                     attr),
5450                                     ifnamep == NULL ? "" : ifnamep);
5451                         }
5452                         i++; pids++;
5453                 } while (i < cpi->cpi_pids_cnt);
5454         }
```

```
#endif /* ! codereview */

        print_transport_label(attr);

        return (first);
}

/* ---------------------------- SCTP_REPORT----------------------------- */

static const char sctp_hdr[] =
"\nSCTP:";
static const char sctp_hdr_normal[] =
"        Local Address                 Remote Address          "
"Swind  Send-Q Rwind  Recv-Q StrsI/O  State\n"
"------------------------------ ------------------------------ "
"------ ------ ------ ------ ------- -----------";
static const char sctp_hdr_pid[] =
"        Local Address                 Remote Address          "
"Swind  Send-Q Rwind  Recv-Q StrsI/O   User    Pid      Command      State\n"
"------------------------------ ------------------------------ ------ "
"------ ------ ------ ------ -------- ------ -------------- -----------";
static const char sctp_hdr_pid_verbose[] =
"        Local Address                 Remote Address          "
"Swind  Send-Q Rwind  Recv-Q StrsI/O   User    Pid    State         Command\n"
"------------------------------ ------------------------------ ------ "
"------ ------ ------ ------ -------- ------ ----------- --------------";
#endif /* ! codereview */

static const char *
nssctp_state(int state, const mib2_transportMLPEntry_t *attr)
{
        static char sctpsbuf[50];
        const char *cp;

        switch (state) {
        case MIB2_SCTP_closed:
                cp = "CLOSED";
                break;
        case MIB2_SCTP_cookieWait:
                cp = "COOKIE_WAIT";
                break;
        case MIB2_SCTP_cookieEchoed:
                cp = "COOKIE_ECHOED";
                break;
        case MIB2_SCTP_established:
                cp = "ESTABLISHED";
                break;
        case MIB2_SCTP_shutdownPending:
                cp = "SHUTDOWN_PENDING";
                break;
        case MIB2_SCTP_shutdownSent:
                cp = "SHUTDOWN_SENT";
                break;
        case MIB2_SCTP_shutdownReceived:
                cp = "SHUTDOWN_RECEIVED";
                break;
        case MIB2_SCTP_shutdownAckSent:
                cp = "SHUTDOWN_ACK_SENT";
                break;
        case MIB2_SCTP_listen:
                cp = "LISTEN";
                break;
        default:
                (void) snprintf(sctpsbuf, sizeof (sctpsbuf),
                    "UNKNOWN STATE(%d)", state);
                cp = sctpsbuf;
```



```
                break;
        }

        if (RSECflag && attr != NULL && attr->tme_flags != 0) {
                if (cp != sctpsbuf) {
                        (void) strlcpy(sctpsbuf, cp, sizeof (sctpsbuf));
                        cp = sctpsbuf;
                }
                if (attr->tme_flags & MIB2_TMEF_PRIVATE)
                        (void) strlcat(sctpsbuf, " P", sizeof (sctpsbuf));
                if (attr->tme_flags & MIB2_TMEF_SHARED)
                        (void) strlcat(sctpsbuf, " S", sizeof (sctpsbuf));
        }

        return (cp);
}

static const mib2_sctpConnRemoteEntry_t *
sctp_getnext_rem(const mib_item_t **itemp,
    const mib2_sctpConnRemoteEntry_t *current, uint32_t associd)
{
        const mib_item_t *item = *itemp;
        const mib2_sctpConnRemoteEntry_t        *sre;

        for (; item != NULL; item = item->next_item, current = NULL) {
                if (!(item->group == MIB2_SCTP &&
                    item->mib_id == MIB2_SCTP_CONN_REMOTE)) {
                        continue;
                }

                if (current != NULL) {
                        /* LINTED: (note 1) */
                        sre = (const mib2_sctpConnRemoteEntry_t *)
                            ((const char *)current + sctpRemoteEntrySize);
                } else {
                        sre = item->valp;
                }
                for (; (char *)sre < (char *)item->valp + item->length;
                    /* LINTED: (note 1) */
                    sre = (const mib2_sctpConnRemoteEntry_t *)
                    ((const char *)sre + sctpRemoteEntrySize)) {
                        if (sre->sctpAssocId != associd) {
                                continue;
                        }
                        *itemp = item;
                        return (sre);
                }
        }
        *itemp = NULL;
        return (NULL);
}

static const mib2_sctpConnLocalEntry_t *
sctp_getnext_local(const mib_item_t **itemp,
    const mib2_sctpConnLocalEntry_t *current, uint32_t associd)
{
        const mib_item_t *item = *itemp;
        const mib2_sctpConnLocalEntry_t *sle;

        for (; item != NULL; item = item->next_item, current = NULL) {
                if (!(item->group == MIB2_SCTP &&
                    item->mib_id == MIB2_SCTP_CONN_LOCAL)) {
                        continue;
                }

                if (current != NULL) {
```

```
5587                        /* LINTED: (note 1) */
5588                        sle = (const mib2_sctpConnLocalEntry_t *)
5589                            ((const char *)current + sctpLocalEntrySize);
5590                } else {
5591                        sle = item->valp;
5592                }
5593                for (; (char *)sle < (char *)item->valp + item->length;
5594                    /* LINTED: (note 1) */
5595                    sle = (const mib2_sctpConnLocalEntry_t *)
5596                    ((const char *)sle + sctpLocalEntrySize)) {
5597                        if (sle->sctpAssocId != associd) {
5598                                continue;
5599                        }
5600                        *itemp = item;
5601                        return (sle);
5602                }
5603        }
5604        *itemp = NULL;
5605        return (NULL);
5606 }

5608 static void
5609 sctp_pr_addr(int type, char *name, int namelen, const in6_addr_t *addr,
5610     int port)
5611 {
5612        ipaddr_t        v4addr;
5613        in6_addr_t      v6addr;

5615        /*
5616         * Address is either a v4 mapped or v6 addr. If
5617         * it's a v4 mapped, convert to v4 before
5618         * displaying.
5619         */
5620        switch (type) {
5621        case MIB2_SCTP_ADDR_V4:
5622                /* v4 */
5623                v6addr = *addr;

5625                IN6_V4MAPPED_TO_IPADDR(&v6addr, v4addr);
5626                if (port > 0) {
5627                        (void) pr_ap(v4addr, port, "sctp", name, namelen);
5628                } else {
5629                        (void) pr_addr(v4addr, name, namelen);
5630                }
5631                break;

5633        case MIB2_SCTP_ADDR_V6:
5634                /* v6 */
5635                if (port > 0) {
5636                        (void) pr_ap6(addr, port, "sctp", name, namelen);
5637                } else {
5638                        (void) pr_addr6(addr, name, namelen);
5639                }
5640                break;

5642        default:
5643                (void) snprintf(name, namelen, "<unknown addr type>");
5644                break;
5645        }
5646 }

5648 static boolean_t
5649 sctp_conn_report_item(const mib_item_t *head, conn_pid_info_t * cpi,
5650     boolean_t print_sctp_hdr, const mib2_sctpConnEntry_t *sp,
4490 static void
4491 sctp_conn_report_item(const mib_item_t *head, const mib2_sctpConnEntry_t *sp,
```



```
5651     const mib2_transportMLPEntry_t *attr)
5652 {
5653        char            lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5654        char            fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5655        const mib2_sctpConnRemoteEntry_t        *sre = NULL;
5656        const mib2_sctpConnLocalEntry_t *sle = NULL;
5657        const mib_item_t *local = head;
5658        const mib_item_t *remote = head;
5659        uint32_t        id = sp->sctpAssocId;
5660        boolean_t       printfirst = B_TRUE;

5662        if (print_sctp_hdr == B_TRUE) {
5663                (void) puts(sctp_hdr);
5664                if (Uflag)
5665                        (void) puts(Vflag? sctp_hdr_pid_verbose: sctp_hdr_pid);
5666                else
5667                        (void) puts(sctp_hdr_normal);

5669                print_sctp_hdr = B_FALSE;
5670        }

5672 #endif /* ! codereview */
5673        sctp_pr_addr(sp->sctpAssocRemPrimAddrType, fname, sizeof (fname),
5674            &sp->sctpAssocRemPrimAddr, sp->sctpAssocRemPort);
5675        sctp_pr_addr(sp->sctpAssocRemPrimAddrType, lname, sizeof (lname),
5676            &sp->sctpAssocLocPrimAddr, sp->sctpAssocLocalPort);

5678        if (Uflag) {
5679                int i = 0;
5680                pid_t *pids = cpi->cpi_pids;
5681                proc_info_t *pinfo;
5682                do {
5683                        pinfo = get_proc_info(*pids);
5684                        (void) printf("%-31s %-31s %6u %6d %6u %6d %3d/%-3d %-8.
5685                            lname, fname,
5686                            sp->sctpConnEntryInfo.ce_swnd,
5687                            sp->sctpConnEntryInfo.ce_sendq,
5688                            sp->sctpConnEntryInfo.ce_rwnd,
5689                            sp->sctpConnEntryInfo.ce_recvq,
5690                            sp->sctpAssocInStreams,
5691                            sp->sctpAssocOutStreams,
5692                            pinfo->pr_user, (int)*pids);
5693                        if (Vflag) {
5694                                (void) printf("%-11.11s %s\n",
5695                                    nssctp_state(sp->sctpAssocState, attr),
5696                                    pinfo->pr_psargs);
5697                        } else {
5698                                (void) printf("%-14.14s %s\n",
5699                                    pinfo->pr_fname,
5700                                    nssctp_state(sp->sctpAssocState, attr));
5701                        }
5702                        i++; pids++;
5703                } while (i < cpi->cpi_pids_cnt);

5705        } else {

5707 #endif /* ! codereview */
5708                (void) printf("%-31s %-31s %6u %6d %6u %6d %3d/%-3d %s\n",
5709                    lname, fname,
5710                    sp->sctpConnEntryInfo.ce_swnd,
5711                    sp->sctpConnEntryInfo.ce_sendq,
5712                    sp->sctpConnEntryInfo.ce_rwnd,
5713                    sp->sctpConnEntryInfo.ce_recvq,
5714                    sp->sctpAssocInStreams, sp->sctpAssocOutStreams,
5715                    nssctp_state(sp->sctpAssocState, attr));
5716        }
```

```
#endif /* ! codereview */

        print_transport_label(attr);

        if (!Vflag) {
                return (print_sctp_hdr);
                return;
        }

        /* Print remote addresses/local addresses on following lines */
        while ((sre = sctp_getnext_rem(&remote, sre, id)) != NULL) {
                if (!IN6_ARE_ADDR_EQUAL(&sre->sctpAssocRemAddr,
                    &sp->sctpAssocRemPrimAddr)) {
                        if (printfirst == B_TRUE) {
                                (void) fputs("\t<Remote: ", stdout);
                                printfirst = B_FALSE;
                        } else {
                                (void) fputs(", ", stdout);
                        }
                        sctp_pr_addr(sre->sctpAssocRemAddrType, fname,
                            sizeof (fname), &sre->sctpAssocRemAddr, -1);
                        if (sre->sctpAssocRemAddrActive == MIB2_SCTP_ACTIVE) {
                                (void) fputs(fname, stdout);
                        } else {
                                (void) printf("(%s)", fname);
                        }
                }
        }
        if (printfirst == B_FALSE) {
                (void) puts(">");
                printfirst = B_TRUE;
        }
        while ((sle = sctp_getnext_local(&local, sle, id)) != NULL) {
                if (!IN6_ARE_ADDR_EQUAL(&sle->sctpAssocLocalAddr,
                    &sp->sctpAssocLocPrimAddr)) {
                        if (printfirst == B_TRUE) {
                                (void) fputs("\t<Local: ", stdout);
                                printfirst = B_FALSE;
                        } else {
                                (void) fputs(", ", stdout);
                        }
                        sctp_pr_addr(sle->sctpAssocLocalAddrType, lname,
                            sizeof (lname), &sle->sctpAssocLocalAddr, -1);
                        (void) fputs(lname, stdout);
                }
        }
        if (printfirst == B_FALSE) {
                (void) puts(">");
        }

        return (print_sctp_hdr);
#endif /* ! codereview */
}

static void
sctp_report(const mib_item_t *item)
{
        const mib_item_t                *head;
        const mib2_sctpConnEntry_t      *sp;
        boolean_t                       print_sctp_hdr_once = B_TRUE;
        boolean_t               first = B_TRUE;
        mib2_transportMLPEntry_t        **attrs, **aptr;
        mib2_transportMLPEntry_t        *attr;
        conn_pid_info_t                 *cpi;
#endif /* ! codereview */
```



```
        /*
         * Preparation pass: the kernel returns separate entries for SCTP
         * connection table entries and Multilevel Port attributes.  We loop
         * through the attributes first and set up an array for each address
         * family.
         */
        attrs = RSECflag ?
            gather_attrs(item, MIB2_SCTP, MIB2_SCTP_CONN, sctpEntrySize) :
            NULL;

        aptr = attrs;
        head = item;
        for (; item != NULL; item = item->next_item) {

                if (!((item->group == MIB2_SCTP &&
                    item->mib_id == MIB2_SCTP_CONN) ||
                    (item->group == MIB2_SCTP &&
                    item->mib_id == EXPER_XPORT_PROC_INFO)))
                if (!(item->group == MIB2_SCTP &&
                    item->mib_id == MIB2_SCTP_CONN))
                        continue;

                if ((!Uflag) && item->group == MIB2_SCTP
                    && item->mib_id == MIB2_SCTP_CONN) {
#endif /* ! codereview */
                        for (sp = item->valp;
                            (char *)sp < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            sp = (mib2_sctpConnEntry_t *)((char *)sp + sctpEntry
                                if (!(Aflag ||
                                  sp->sctpAssocState >= MIB2_SCTP_established))
                                        continue;
#endif /* ! codereview */
                                attr = aptr == NULL ? NULL : *aptr++;
                                print_sctp_hdr_once = sctp_conn_report_item(head
                                                        print_sctp_hdr_once, sp,
                                                        attr);
                        }
                } else if ((Uflag) && item->group == MIB2_SCTP &&
                    item->mib_id == EXPER_XPORT_PROC_INFO) {
                        for (sp = (mib2_sctpConnEntry_t *)item->valp;
                            (char *)sp < (char *)item->valp + item->length;
                            /* LINTED: (note 1) */
                            sp = (mib2_sctpConnEntry_t *)((char *)cpi +
                            cpi->cpi_tot_size)) {
                                /* LINTED: (note 1) */
                                cpi = (conn_pid_info_t *) ((char *)sp +
                                  sctpEntrySize);
                                if (!(Aflag ||
                                  sp->sctpAssocState >= MIB2_SCTP_established))
                                        continue;
                                attr = aptr == NULL ? NULL : *aptr++;
                                print_sctp_hdr_once =
                                    sctp_conn_report_item(head, cpi,
                                        print_sctp_hdr_once, sp, attr);
                        if (Aflag ||
                            sp->sctpAssocState >= MIB2_SCTP_established) {
                                if (first == B_TRUE) {
                                        (void) puts(sctp_hdr);
                                        (void) puts(sctp_hdr_normal);
                                        first = B_FALSE;
                                }
                                sctp_conn_report_item(head, sp, attr);
                        }
                }
        }
```

```
5837         if (attrs != NULL)
5838                 free(attrs);
5839 }
```

*unchanged_portion_omitted_*

```
6820 /*
6821  * Gets proc info in (proc_info_t) given pid. It doesn't return NULL.
6822  */
6823 proc_info_t *
6824 get_proc_info(pid_t pid)
6825 {
6826         static pid_t saved_pid = 0;
6827         static proc_info_t saved_proc_info;
6828         static proc_info_t unknown_proc_info = {"<unknown>","",""};
6829         static psinfo_t pinfo;
6830         char path[128];
6831         int fd;

6833         /* hardcode pid = 0 */
6834         if (pid == 0) {
6835                 saved_proc_info.pr_user = "root";
6836                 saved_proc_info.pr_fname = "sched";
6837                 saved_proc_info.pr_psargs = "sched";
6838                 saved_pid = 0;
6839                 return &saved_proc_info;
6840         }

6842         if (pid == saved_pid)
6843                 return &saved_proc_info;
6844         if ((snprintf(path, 128, "/proc/%u/psinfo",(int)pid) > 0) &&
6845             ((fd = open(path, O_RDONLY)) != -1)) {
6846                 if (read(fd, &pinfo, sizeof(pinfo)) == sizeof(pinfo)){
6847                         saved_proc_info.pr_user = get_username(pinfo.pr_uid);
6848                         saved_proc_info.pr_fname = pinfo.pr_fname;
6849                         saved_proc_info.pr_psargs = pinfo.pr_psargs;
6850                         saved_pid = pid;
6851                         (void) close(fd);
6852                         return &saved_proc_info;
6853                 } else {
6854                         (void) close(fd);
6855                 }
6856         }

6858         return (&unknown_proc_info);
6859 }

6861 /*
6862  * Gets username given uid. It doesn't return NULL.
6863  */
6864 static char *
6865 get_username(uid_t u)
6866 {
6867         static uid_t saved_uid = UINT_MAX;
6868         static char  saved_username[128];
6869         struct passwd *pw = NULL;
6870         if (u == UINT_MAX)
6871                 return "<unknown>";
6872         if (u == saved_uid && saved_username[0] != '\0')
6873                 return  (saved_username);
6874         setpwent();
6875         if ((pw = getpwuid(u)) != NULL)
6876                 (void) strlcpy(saved_username, pw->pw_name, 128);
6877         else
6878                 (void) snprintf(saved_username, 128, "%u", u);
6879         saved_uid = u;
6880         return saved_username;
```



```
6881 }

6883 /*
6884 #endif /* ! codereview */
6885  * print the usage line
6886  */
6887 static void
6888 usage(char *cmdname)
6889 {
6890         (void) fprintf(stderr, "usage: %s [-anuv] [-f address_family] "
5548         (void) fprintf(stderr, "usage: %s [-anv] [-f address_family] "
6891             "[-T d|u]\n", cmdname);
6892         (void) fprintf(stderr, "       %s [-n] [-f address_family] "
6893             "[-P protocol] [-T d|u] [-g | -p | -s [interval [count]]]\n",
6894             cmdname);
6895         (void) fprintf(stderr, "       %s -m [-v] [-T d|u] "
6896             "[interval [count]]\n", cmdname);
6897         (void) fprintf(stderr, "       %s -i [-I interface] [-an] "
6898             "[-f address_family] [-T d|u] [interval [count]]\n", cmdname);
6899         (void) fprintf(stderr, "       %s -r [-anv] "
6900             "[-f address_family|filter] [-T d|u]\n", cmdname);
6901         (void) fprintf(stderr, "       %s -M [-ns] [-f address_family] "
6902             "[-T d|u]\n", cmdname);
6903         (void) fprintf(stderr, "       %s -D [-I interface] "
6904             "[-f address_family] [-T d|u]\n", cmdname);
6905         exit(EXIT_FAILURE);
6906 }

6908 /*
6909  * fatal: print error message to stderr and
6910  * call exit(errcode)
6911  */
6912 /*PRINTFLIKE2*/
6913 static void
6914 fatal(int errcode, char *format, ...)
6915 {
6916         va_list argp;

6918         if (format == NULL)
6919                 return;

6921         va_start(argp, format);
6922         (void) vfprintf(stderr, format, argp);
6923         va_end(argp);

6925         exit(errcode);
6926 }


6929 /* ------------------UNIX Domain Sockets Report--------------------------- */


6932 #define NO_ADDR         "                                       "
6933 #define SO_PAIR         " (socketpair)                          "

6935 static char             *typetoname(t_scalar_t);
6936 static boolean_t        uds_report_item(struct sockinfo *, boolean_t);


6939 static char uds_hdr[] = "\nActive UNIX domain sockets\n";

6941 static char uds_hdr_normal[] =
6942 " Type       Local Adress                          "
6943 " Remote Address\n"
6944 "---------- ------------------------------------- "
6945 "-------------------------------------\n";
```

```
6947 static char uds_hdr_pid[] =
6948 " Type       User     Pid     Command       "
6949 " Local Address                          "
6950 " Remote Address\n"
6951 "---------- -------- ------ -------------- "
6952 "------------------------------------ "
6953 "------------------------------------\n";
6954 static char uds_hdr_pid_verbose[] =
6955 " Type       User     Pid    Local Address                          "
6956 " Remote Address                        Command\n"
6957 "---------- -------- ------ -------------------------------------- "
6958 "------------------------------------ --------------\n";

6960 /*
6961  * Print a summary of connections related to unix protocols.
6962  */
6963 static void
6964 uds_report(kstat_ctl_t  *kc)
6965 {
6966         int             i;
6967         kstat_t         *ksp;
6968         struct sockinfo *psi;
6969         boolean_t       print_uds_hdr_once = B_TRUE;

6971         if (kc == NULL) {
6972                 fail(0, "uds_report: No kstat");
6973                 exit(3);
6974         }

6976         if ((ksp = kstat_lookup(kc, "sockfs", 0, "sock_unix_list")) ==
6977             (kstat_t *)NULL) {
6978                 fail(0, "kstat_data_lookup failed\n");
6979         }

6981         if (kstat_read(kc, ksp, NULL) == -1) {
6982                 fail(0, "kstat_read failed for sock_unix_list\n");
6983         }

6985         if (ksp->ks_ndata == 0) {
6986                 return;                 /* no AF_UNIX sockets found     */
6987         }

6989         /*
6990          * Having ks_data set with ks_data == NULL shouldn't happen;
6991          * If it does, the sockfs kstat is seriously broken.
6992          */
6993         if ((psi = ksp->ks_data) == NULL) {
6994                 fail(0, "uds_report: no kstat data\n");
6995         }

6997         for (i = 0; i < ksp->ks_ndata; i++) {

6999                 print_uds_hdr_once = uds_report_item(psi, print_uds_hdr_once);

7001                 /* if si_size didn't get filled in, then we're done     */
7002                 if (psi->si_size == 0 ||
7003                     !IS_P2ALIGNED(psi->si_size, sizeof (psi))) {
7004                         break;
7005                 }

7007                 /* point to the next sockinfo in the array */
7008                 /* LINTED: (note 1) */
7009                 psi = (struct sockinfo *)(((char *)psi) + psi->si_size);
7010         }
7011 }
```



```
7013 static boolean_t
7014 uds_report_item(struct sockinfo *psi, boolean_t first)
7015 {
7016         int             i = 0;
7017         pid_t           *pids;
7018         proc_info_t     *pinfo;
7019         char            *laddr, *raddr;

7021         if(first) {
7022                 (void) printf("%s", uds_hdr);
7023                 if (Uflag)
7024                         (void) printf("%s", Vflag?uds_hdr_pid_verbose:
7025                             uds_hdr_pid);
7026                 else
7027                         (void) printf("%s", uds_hdr_normal);

7029                 first = B_FALSE;
7030         }

7032         pids = psi->si_pids;

7034         do {
7035                 pinfo = get_proc_info(*pids);
7036                 raddr = laddr = NO_ADDR;

7038                 /* Try to fill laddr */
7039                 if ((psi->si_state & SS_ISBOUND) &&
7040                     strlen(psi->si_laddr_sun_path) != 0 &&
7041                     psi->si_laddr_soa_len != 0) {
7042                         if (psi->si_faddr_noxlate) {
7043                                 laddr = SO_PAIR;
7044                         } else {
7045                                 if (psi->si_laddr_soa_len >
7046                                     sizeof (psi->si_laddr_family))
7047                                         laddr = psi->si_laddr_sun_path;
7048                         }
7049                 }

7051                 /* Try to fill raddr */
7052                 if ((psi->si_state & SS_ISCONNECTED) &&
7053                     strlen(psi->si_faddr_sun_path) != 0 &&
7054                     psi->si_faddr_soa_len != 0) {

7056                         if (psi->si_faddr_noxlate) {
7057                                 raddr = SO_PAIR;
7058                         } else {
7059                                 if (psi->si_faddr_soa_len >
7060                                     sizeof (psi->si_faddr_family))
7061                                         raddr = psi->si_faddr_sun_path;
7062                         }
7063                 }

7065                 if (Uflag && Vflag) {
7066                         (void) printf("%-10.10s %-8.8s %6u "
7067                                 "%-39.39s %-39.39s %s\n",
7068                                 typetoname(psi->si_serv_type), pinfo->pr_user,
7069                                 (int)*pids, laddr, raddr, pinfo->pr_psargs);
7070                 } else if (Uflag && (!Vflag)) {
7071                         (void) printf("%-10.10s %-8.8s %6u %-14.14s"
7072                                 "%-39.39s %-39.39s\n",
7073                                 typetoname(psi->si_serv_type), pinfo->pr_user,
7074                                 (int)*pids, pinfo->pr_fname, laddr, raddr);
7075                 } else {
7076                         (void) printf("%-10.10s %s %s\n",
7077                                 typetoname(psi->si_serv_type), laddr, raddr);
```

```
7078                 }
7080                 i++; pids++;
7081         } while (i < psi->si_pn_cnt);
7083         return (first);
7084 }
7086 static char *
7087 typetoname(t_scalar_t type)
7088 {
7089         switch (type) {
7090         case T_CLTS:
7091                 return ("dgram");
7093         case T_COTS:
7094                 return ("stream");
7096         case T_COTS_ORD:
7097                 return ("stream-ord");
7099         default:
7100                 return ("");
7101         }
7102 #endif /* ! codereview */
7103 }
```

```
**********************************************************
   48122 Mon Aug 17 21:08:02 2015
new/usr/src/cmd/perl/contrib/Sun/Solaris/Kstat/Kstat.xs
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2014 Racktop Systems.
 */

/*
 * Kstat.xs is a Perl XS (eXStension module) that makes the Solaris
 * kstat(3KSTAT) facility available to Perl scripts.  Kstat is a general-purpose
 * mechanism  for  providing kernel statistics to users.  The Solaris API is
 * function-based (see the manpage for details), but for ease of use in Perl
 * scripts this module presents the information as a nested hash data structure.
 * It would be too inefficient to read every kstat in the system, so this module
 * uses the Perl TIEHASH mechanism to implement a read-on-demand semantic, which
 * only reads and updates kstats as and when they are actually accessed.
 */

/*
 * Ignored raw kstats.
 *
 * Some raw kstats are ignored by this module, these are listed below.  The
 * most common reason is that the kstats are stored as arrays and the ks_ndata
 * and/or ks_data_size fields are invalid.  In this case it is impossible to
 * know how many records are in the array, so they can't be read.
 *
 * unix:*:sfmmu_percpu_stat
 * This is stored as an array with one entry per cpu.  Each element is of type
 * struct sfmmu_percpu_stat.  The ks_ndata and ks_data_size fields are bogus.
 *
 * ufs directio:*:UFS DirectIO Stats
 * The structure definition used for these kstats (ufs_directio_kstats) is in a
 * C file (uts/common/fs/ufs/ufs_directio.c) rather than a header file, so it
 * isn't accessible.
 *
 * qlc:*:statistics
 * This is a third-party driver for which we don't have source.
 *
 * mm:*:phys_installed
 * This is stored as an array of uint64_t, with each pair of values being the
 * (address, size) of a memory segment.  The ks_ndata and ks_data_size fields
 * are both zero.
```



```
 *
 * sockfs:*:sock_unix_list
 * This is stored as an array with one entry per active socket.  Each element
 * is of type struct sockinfo.  ks_ndata s the number of elements of that array
 * and ks_data_size is the total size of the array.
 * is of type struct k_sockinfo.  The ks_ndata and ks_data_size fields are both
 * zero.
 *
 * Note that the ks_ndata and ks_data_size of many non-array raw kstats are
 * also incorrect.  The relevant assertions are therefore commented out in the
 * appropriate raw kstat read routines.
 */

/* Kstat related includes */
#include <libgen.h>
#include <kstat.h>
#include <sys/var.h>
#include <sys/utsname.h>
#include <sys/sysinfo.h>
#include <sys/flock.h>
#include <sys/dnlc.h>
#include <nfs/nfs.h>
#include <nfs/nfs_clnt.h>

/* Ultra-specific kstat includes */
#ifdef __sparc
#include <vm/hat_sfmmu.h>       /* from /usr/platform/sun4u/include */
#include <sys/simmstat.h>       /* from /usr/platform/sun4u/include */
#include <sys/sysctrl.h>        /* from /usr/platform/sun4u/include */
#include <sys/fhc.h>            /* from /usr/include */
#endif

/*
 * Solaris #defines SP, which conflicts with the perl definition of SP
 * We don't need the Solaris one, so get rid of it to avoid warnings
 */
#undef SP

/* Perl XS includes */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* Debug macros */
#define DEBUG_ID "Sun::Solaris::Kstat"
#ifdef KSTAT_DEBUG
#define PERL_ASSERT(EXP) \
    ((void)((EXP) || (croak("%s: assertion failed at %s:%d: %s", \
    DEBUG_ID, __FILE__, __LINE__, #EXP), 0), 0))
#define PERL_ASSERTMSG(EXP, MSG) \
    ((void)((EXP) || (croak(DEBUG_ID ": " MSG), 0), 0))
#else
#define PERL_ASSERT(EXP)                ((void)0)
#define PERL_ASSERTMSG(EXP, MSG)        ((void)0)
#endif

/* Macros for saving the contents of KSTAT_RAW structures */
#if defined(HAS_QUAD) && defined(USE_64_BIT_INT)
#define NEW_IV(V) \
    (newSViv((IVTYPE) V))
#define NEW_UV(V) \
    (newSVuv((UVTYPE) V))
#else
#define NEW_IV(V) \
    (V >= IV_MIN && V <= IV_MAX ? newSViv((IVTYPE) V) : newSVnv((NVTYPE) V))
#if defined(UVTYPE)
```

```
 126 #define NEW_UV(V) \
 127     (V <= UV_MAX ? newSVuv((UVTYPE) V) : newSVnv((NVTYPE) V))
 128 # else
 129 #define NEW_UV(V) \
 130     (V <= IV_MAX ? newSViv((IVTYPE) V) : newSVnv((NVTYPE) V))
 131 #endif
 132 #endif
 133 #define NEW_HRTIME(V) \
 134     newSVnv((NVTYPE) (V / 1000000000.0))

 136 #define SAVE_FNP(H, F, K) \
 137     hv_store(H, K, sizeof (K) - 1, newSViv((IVTYPE)(uintptr_t)&F), 0)
 138 #define SAVE_STRING(H, S, K, SS) \
 139     hv_store(H, #K, sizeof (#K) - 1, \
 140     newSVpvn(S->K, SS ? strlen(S->K) : sizeof(S->K)), 0)
 141 #define SAVE_INT32(H, S, K) \
 142     hv_store(H, #K, sizeof (#K) - 1, NEW_IV(S->K), 0)
 143 #define SAVE_UINT32(H, S, K) \
 144     hv_store(H, #K, sizeof (#K) - 1, NEW_UV(S->K), 0)
 145 #define SAVE_INT64(H, S, K) \
 146     hv_store(H, #K, sizeof (#K) - 1, NEW_IV(S->K), 0)
 147 #define SAVE_UINT64(H, S, K) \
 148     hv_store(H, #K, sizeof (#K) - 1, NEW_UV(S->K), 0)
 149 #define SAVE_HRTIME(H, S, K) \
 150     hv_store(H, #K, sizeof (#K) - 1, NEW_HRTIME(S->K), 0)

 152 /* Private structure used for saving kstat info in the tied hashes */
 153 typedef struct {
 154         char            read;           /* Kstat block has been read before */
 155         char            valid;          /* Kstat still exists in kstat chain */
 156         char            strip_str;      /* Strip KSTAT_DATA_CHAR fields */
 157         kstat_ctl_t     *kstat_ctl;     /* Handle returned by kstat_open */
 158         kstat_t         *kstat;         /* Handle used by kstat_read */
 159 } KstatInfo_t;
```

*_____unchanged_portion_omitted_*

```
**********************************************************
    6287 Mon Aug 17 21:08:02 2015
new/usr/src/common/list/list.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

#define list_remove_node(node)                                  \
        (node)->list_prev->list_next = (node)->list_next;       \
        (node)->list_next->list_prev = (node)->list_prev;       \
        (node)->list_next = (node)->list_prev = NULL

void
list_create(list_t *list, size_t size, size_t offset)
{
        ASSERT(list);
        ASSERT(size > 0);
        ASSERT(size >= offset + sizeof (list_node_t));

        list->list_size = size;
        list->list_offset = offset;
        list->list_numnodes = 0;
#endif /* ! codereview */
        list->list_head.list_next = list->list_head.list_prev =
            &list->list_head;
}

void
list_destroy(list_t *list)
{
        list_node_t *node = &list->list_head;

        ASSERT(list);
        ASSERT(list->list_head.list_next == node);
        ASSERT(list->list_head.list_prev == node);

        list->list_numnodes = 0;
#endif /* ! codereview */
        node->list_next = node->list_prev = NULL;
}

void
list_insert_after(list_t *list, void *object, void *nobject)
{
        if (object == NULL) {
                list_insert_head(list, nobject);
        } else {
                list_node_t *lold = list_d2l(list, object);
                list_insert_after_node(list, lold, nobject);
        }
}

void
list_insert_before(list_t *list, void *object, void *nobject)
{
        if (object == NULL) {
                list_insert_tail(list, nobject);
        } else {
                list_node_t *lold = list_d2l(list, object);
                list_insert_before_node(list, lold, nobject);
        }
}

void
list_insert_head(list_t *list, void *object)
{
```



```
        list_node_t *lold = &list->list_head;
        list->list_numnodes++;
#endif /* ! codereview */
        list_insert_after_node(list, lold, object);
}

void
list_insert_tail(list_t *list, void *object)
{
        list_node_t *lold = &list->list_head;
        list->list_numnodes++;
#endif /* ! codereview */
        list_insert_before_node(list, lold, object);
}

void
list_remove(list_t *list, void *object)
{
        list_node_t *lold = list_d2l(list, object);
        ASSERT(!list_empty(list));
        ASSERT(lold->list_next != NULL);
        list->list_numnodes--;
#endif /* ! codereview */
        list_remove_node(lold);
}

void *
list_remove_head(list_t *list)
{
        list_node_t *head = list->list_head.list_next;
        if (head == &list->list_head)
                return (NULL);
        list->list_numnodes--;
#endif /* ! codereview */
        list_remove_node(head);
        return (list_object(list, head));
}

void *
list_remove_tail(list_t *list)
{
        list_node_t *tail = list->list_head.list_prev;
        if (tail == &list->list_head)
                return (NULL);
        list->list_numnodes--;
#endif /* ! codereview */
        list_remove_node(tail);
        return (list_object(list, tail));
}

void *
list_head(list_t *list)
{
        if (list_empty(list))
                return (NULL);
        return (list_object(list, list->list_head.list_next));
}

void *
list_tail(list_t *list)
{
        if (list_empty(list))
                return (NULL);
        return (list_object(list, list->list_head.list_prev));
}
```

```
189 void *
190 list_next(list_t *list, void *object)
191 {
192         list_node_t *node = list_d2l(list, object);
193
194         if (node->list_next != &list->list_head)
195                 return (list_object(list, node->list_next));
196
197         return (NULL);
198 }
199
200 void *
201 list_prev(list_t *list, void *object)
202 {
203         list_node_t *node = list_d2l(list, object);
204
205         if (node->list_prev != &list->list_head)
206                 return (list_object(list, node->list_prev));
207
208         return (NULL);
209 }
210
211 /*
212  *  Insert src list after dst list. Empty src list thereafter.
213  */
214 void
215 list_move_tail(list_t *dst, list_t *src)
216 {
217         list_node_t *dstnode = &dst->list_head;
218         list_node_t *srcnode = &src->list_head;
219
220         ASSERT(dst->list_size == src->list_size);
221         ASSERT(dst->list_offset == src->list_offset);
222
223         if (list_empty(src))
224                 return;
225
226         dst->list_numnodes += src->list_numnodes;
227 #endif /* ! codereview */
228         dstnode->list_prev->list_next = srcnode->list_next;
229         srcnode->list_next->list_prev = dstnode->list_prev;
230         dstnode->list_prev = srcnode->list_prev;
231         srcnode->list_prev->list_next = dstnode;
232
233         /* empty src list */
234         src->list_numnodes = 0;
235 #endif /* ! codereview */
236         srcnode->list_next = srcnode->list_prev = srcnode;
237 }
238
239 void
240 list_link_replace(list_node_t *lold, list_node_t *lnew)
241 {
242         ASSERT(list_link_active(lold));
243         ASSERT(!list_link_active(lnew));
244
245         lnew->list_next = lold->list_next;
246         lnew->list_prev = lold->list_prev;
247         lold->list_prev->list_next = lnew;
248         lold->list_next->list_prev = lnew;
249         lold->list_next = lold->list_prev = NULL;
250 }
251
252 void
253 list_link_init(list_node_t *link)
254 {
```



```
255         link->list_next = NULL;
256         link->list_prev = NULL;
257 }
258
259 int
260 list_link_active(list_node_t *link)
261 {
262         return (link->list_next != NULL);
263 }
264
265 int
266 list_is_empty(list_t *list)
267 {
268         return (list_empty(list));
269 }
270
271 ulong_t
272 list_numnodes(list_t *list)
273 {
274         /*
275         size_t sz = 0;
276         list_node_t *node;
277
278         node = &list->list_head;
279         while (node->list_next != &list->list_head) {
280                 sz++;
281                 node = node->list_next;
282         }
283         return (sz);
284         */
285         return (list->list_numnodes);
286 #endif /* ! codereview */
287 }
```

```
**********************************************************
   90098 Mon Aug 17 21:08:03 2015
new/usr/src/pkg/manifests/system-header.mf
XXXX adding PID information to netstat output
**********************************************************
```

```
   1 #
   2 # CDDL HEADER START
   3 #
   4 # The contents of this file are subject to the terms of the
   5 # Common Development and Distribution License (the "License").
   6 # You may not use this file except in compliance with the License.
   7 #
   8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9 # or http://www.opensolaris.org/os/licensing.
  10 # See the License for the specific language governing permissions
  11 # and limitations under the License.
  12 #
  13 # When distributing Covered Code, include this CDDL HEADER in each
  14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15 # If applicable, add the following below this CDDL HEADER, with the
  16 # fields enclosed by brackets "[]" replaced with your own identifying
  17 # information: Portions Copyright [yyyy] [name of copyright owner]
  18 #
  19 # CDDL HEADER END
  20 #

  22 #
  23 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
  24 # Copyright (c) 2012 by Delphix. All rights reserved.
  25 # Copyright 2012 Nexenta Systems, Inc. All rights reserved.
  26 # Copyright 2014 Garrett D'Amore <garrett@damore.org>
  27 #

  29 set name=pkg.fmri value=pkg:/system/header@$(PKGVERS)
  30 set name=pkg.description \
  31     value="SunOS C/C++ header files for general development of software"
  32 set name=pkg.summary value="SunOS Header Files"
  33 set name=info.classification value=org.opensolaris.category.2008:System/Core
  34 set name=variant.arch value=$(ARCH)
  35 dir path=usr group=sys
  36 dir path=usr/include
  37 $(i386_ONLY)dir path=usr/include/$(ARCH64)
  38 $(i386_ONLY)dir path=usr/include/$(ARCH64)/sys
  39 dir path=usr/include/arpa
  40 dir path=usr/include/asm
  41 dir path=usr/include/ast
  42 dir path=usr/include/bsm
  43 dir path=usr/include/dat
  44 dir path=usr/include/des
  45 dir path=usr/include/gssapi
  46 dir path=usr/include/hal
  47 $(i386_ONLY)dir path=usr/include/ia32
  48 $(i386_ONLY)dir path=usr/include/ia32/sys
  49 dir path=usr/include/inet
  50 dir path=usr/include/inet/kssl
  51 dir path=usr/include/ipp
  52 dir path=usr/include/ipp/ipgpc
  53 dir path=usr/include/iso
  54 dir path=usr/include/kerberosv5
  55 dir path=usr/include/libpolkit
  56 dir path=usr/include/net
  57 dir path=usr/include/netinet
  58 dir path=usr/include/nfs
  59 dir path=usr/include/protocols
  60 dir path=usr/include/rpc
  61 dir path=usr/include/rpcsvc
```



```
  62 dir path=usr/include/sasl
  63 dir path=usr/include/scsi
  64 dir path=usr/include/scsi/plugins
  65 dir path=usr/include/scsi/plugins/ses
  66 dir path=usr/include/scsi/plugins/ses/framework
  67 dir path=usr/include/scsi/plugins/ses/vendor
  68 dir path=usr/include/scsi/plugins/smp
  69 dir path=usr/include/scsi/plugins/smp/engine
  70 dir path=usr/include/scsi/plugins/smp/framework
  71 dir path=usr/include/security
  72 dir path=usr/include/sharefs
  73 dir path=usr/include/sys
  74 dir path=usr/include/sys/av
  75 dir path=usr/include/sys/contract
  76 dir path=usr/include/sys/crypto
  77 dir path=usr/include/sys/dktp
  78 dir path=usr/include/sys/fc4
  79 dir path=usr/include/sys/fm
  80 dir path=usr/include/sys/fm/cpu
  81 dir path=usr/include/sys/fm/fs
  82 dir path=usr/include/sys/fm/io
  83 $(sparc_ONLY)dir path=usr/include/sys/fpu
  84 dir path=usr/include/sys/fs
  85 dir path=usr/include/sys/hotplug
  86 dir path=usr/include/sys/hotplug/pci
  87 dir path=usr/include/sys/ib
  88 dir path=usr/include/sys/ib/adapters
  89 dir path=usr/include/sys/ib/adapters/hermon
  90 dir path=usr/include/sys/ib/adapters/tavor
  91 dir path=usr/include/sys/ib/clients
  92 dir path=usr/include/sys/ib/clients/ibd
  93 dir path=usr/include/sys/ib/clients/of
  94 dir path=usr/include/sys/ib/clients/of/rdma
  95 dir path=usr/include/sys/ib/clients/of/sol_ofs
  96 dir path=usr/include/sys/ib/clients/of/sol_ucma
  97 dir path=usr/include/sys/ib/clients/of/sol_umad
  98 dir path=usr/include/sys/ib/clients/of/sol_uverbs
  99 dir path=usr/include/sys/ib/ibnex
 100 dir path=usr/include/sys/ib/ibtl
 101 dir path=usr/include/sys/ib/ibtl/impl
 102 dir path=usr/include/sys/ib/mgt
 103 dir path=usr/include/sys/ib/mgt/ibmf
 104 dir path=usr/include/sys/iso
 105 dir path=usr/include/sys/lvm
 106 dir path=usr/include/sys/proc
 107 dir path=usr/include/sys/rsm
 108 $(i386_ONLY)dir path=usr/include/sys/sata group=sys
 109 dir path=usr/include/sys/scsi
 110 dir path=usr/include/sys/scsi/adapters
 111 dir path=usr/include/sys/scsi/conf
 112 dir path=usr/include/sys/scsi/generic
 113 dir path=usr/include/sys/scsi/impl
 114 dir path=usr/include/sys/scsi/targets
 115 dir path=usr/include/sys/sysevent
 116 dir path=usr/include/sys/tsol
 117 dir path=usr/include/tsol
 118 dir path=usr/include/uuid
 119 $(sparc_ONLY)dir path=usr/include/v7
 120 $(sparc_ONLY)dir path=usr/include/v7/sys
 121 $(sparc_ONLY)dir path=usr/include/v9
 122 $(sparc_ONLY)dir path=usr/include/v9/sys
 123 dir path=usr/include/vm
 124 dir path=usr/platform group=sys
 125 $(sparc_ONLY)dir path=usr/platform/SUNW,A70 group=sys
 126 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300 group=sys
 127 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300/include
```

```
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010/include
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T12 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T4 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,SPARC-Enterprise group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Serverblade1 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-100 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1000 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1500 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-2500 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-15000 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-280R group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-480R group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-880 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V215 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V240 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V250 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V440 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V445 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V490 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V890 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-2 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-250 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-4 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise-10000 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIi-Netract group=sys
$(i386_ONLY)dir path=usr/platform/i86pc group=sys
$(i386_ONLY)dir path=usr/platform/i86pc/include
$(i386_ONLY)dir path=usr/platform/i86pc/include/sys
$(i386_ONLY)dir path=usr/platform/i86pc/include/vm
$(i386_ONLY)dir path=usr/platform/i86xpv group=sys
$(i386_ONLY)dir path=usr/platform/i86xpv/include
$(i386_ONLY)dir path=usr/platform/i86xpv/include/sys
$(i386_ONLY)dir path=usr/platform/i86xpv/include/vm
$(sparc_ONLY)dir path=usr/platform/sun4u group=sys
$(sparc_ONLY)dir path=usr/platform/sun4u/include
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/clients
$(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/misc
$(sparc_ONLY)dir path=usr/platform/sun4u/include/vm
$(sparc_ONLY)dir path=usr/platform/sun4v group=sys
$(sparc_ONLY)dir path=usr/platform/sun4v/include
$(sparc_ONLY)dir path=usr/platform/sun4v/include/sys
$(sparc_ONLY)dir path=usr/platform/sun4v/include/vm
dir path=usr/share
dir path=usr/share/man
dir path=usr/share/man/man3head
dir path=usr/share/man/man4
dir path=usr/share/man/man5
dir path=usr/share/man/man7i
dir path=usr/share/src group=sys
dir path=usr/share/src/uts
$(i386_ONLY)dir path=usr/share/src/uts/i86pc
$(i386_ONLY)dir path=usr/share/src/uts/i86xpv
$(sparc_ONLY)dir path=usr/share/src/uts/sun4u
$(sparc_ONLY)dir path=usr/share/src/uts/sun4v
dir path=usr/xpg4
dir path=usr/xpg4/include
$(i386_ONLY)file path=usr/include/$(ARCH64)/sys/kdi_regs.h
$(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privmregs.h
$(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privregs.h
```



```
file path=usr/include/aio.h
file path=usr/include/alloca.h
file path=usr/include/apptrace.h
file path=usr/include/apptrace_impl.h
file path=usr/include/ar.h
file path=usr/include/archives.h
file path=usr/include/arpa/ftp.h
file path=usr/include/arpa/inet.h
file path=usr/include/arpa/nameser.h
file path=usr/include/arpa/nameser_compat.h
file path=usr/include/arpa/telnet.h
file path=usr/include/arpa/tftp.h
$(i386_ONLY)file path=usr/include/asm/atomic.h
$(i386_ONLY)file path=usr/include/asm/bitmap.h
$(i386_ONLY)file path=usr/include/asm/byteorder.h
$(i386_ONLY)file path=usr/include/asm/clock.h
$(i386_ONLY)file path=usr/include/asm/cpu.h
$(i386_ONLY)file path=usr/include/asm/cpuvar.h
$(sparc_ONLY)file path=usr/include/asm/flush.h
$(i386_ONLY)file path=usr/include/asm/htable.h
$(i386_ONLY)file path=usr/include/asm/mmu.h
file path=usr/include/asm/sunddi.h
file path=usr/include/asm/thread.h
file path=usr/include/assert.h
file path=usr/include/ast/align.h
file path=usr/include/ast/ast.h
file path=usr/include/ast/ast_botch.h
file path=usr/include/ast/ast_ccode.h
file path=usr/include/ast/ast_common.h
file path=usr/include/ast/ast_dir.h
file path=usr/include/ast/ast_dirent.h
file path=usr/include/ast/ast_fcntl.h
file path=usr/include/ast/ast_float.h
file path=usr/include/ast/ast_fs.h
file path=usr/include/ast/ast_getopt.h
file path=usr/include/ast/ast_iconv.h
file path=usr/include/ast/ast_lib.h
file path=usr/include/ast/ast_limits.h
file path=usr/include/ast/ast_map.h
file path=usr/include/ast/ast_mmap.h
file path=usr/include/ast/ast_mode.h
file path=usr/include/ast/ast_namval.h
file path=usr/include/ast/ast_ndbm.h
file path=usr/include/ast/ast_nl_types.h
file path=usr/include/ast/ast_param.h
file path=usr/include/ast/ast_standards.h
file path=usr/include/ast/ast_std.h
file path=usr/include/ast/ast_stdio.h
file path=usr/include/ast/ast_sys.h
file path=usr/include/ast/ast_time.h
file path=usr/include/ast/ast_tty.h
file path=usr/include/ast/ast_version.h
file path=usr/include/ast/ast_vfork.h
file path=usr/include/ast/ast_wait.h
file path=usr/include/ast/ast_wchar.h
file path=usr/include/ast/ast_windows.h
file path=usr/include/ast/bytesex.h
file path=usr/include/ast/ccode.h
file path=usr/include/ast/cdt.h
file path=usr/include/ast/cmd.h
file path=usr/include/ast/cmdext.h
file path=usr/include/ast/debug.h
file path=usr/include/ast/dirent.h
file path=usr/include/ast/dlldefs.h
file path=usr/include/ast/dt.h
file path=usr/include/ast/endian.h
```

```
260 file path=usr/include/ast/error.h
261 file path=usr/include/ast/find.h
262 file path=usr/include/ast/fnmatch.h
263 file path=usr/include/ast/fnv.h
264 file path=usr/include/ast/fs3d.h
265 file path=usr/include/ast/fts.h
266 file path=usr/include/ast/ftw.h
267 file path=usr/include/ast/ftwalk.h
268 file path=usr/include/ast/getopt.h
269 file path=usr/include/ast/glob.h
270 file path=usr/include/ast/hash.h
271 file path=usr/include/ast/hashkey.h
272 file path=usr/include/ast/hashpart.h
273 file path=usr/include/ast/history.h
274 file path=usr/include/ast/iconv.h
275 file path=usr/include/ast/ip6.h
276 file path=usr/include/ast/lc.h
277 file path=usr/include/ast/ls.h
278 file path=usr/include/ast/magic.h
279 file path=usr/include/ast/magicid.h
280 file path=usr/include/ast/mc.h
281 file path=usr/include/ast/mime.h
282 file path=usr/include/ast/mnt.h
283 file path=usr/include/ast/modecanon.h
284 file path=usr/include/ast/modex.h
285 file path=usr/include/ast/namval.h
286 file path=usr/include/ast/nl_types.h
287 file path=usr/include/ast/nval.h
288 file path=usr/include/ast/option.h
289 file path=usr/include/ast/preroot.h
290 file path=usr/include/ast/proc.h
291 file path=usr/include/ast/prototyped.h
292 file path=usr/include/ast/re_comp.h
293 file path=usr/include/ast/recfmt.h
294 file path=usr/include/ast/regex.h
295 file path=usr/include/ast/regexp.h
296 file path=usr/include/ast/sfdisc.h
297 file path=usr/include/ast/sfio.h
298 file path=usr/include/ast/sfio_s.h
299 file path=usr/include/ast/sfio_t.h
300 file path=usr/include/ast/shcmd.h
301 file path=usr/include/ast/shell.h
302 file path=usr/include/ast/sig.h
303 file path=usr/include/ast/stack.h
304 file path=usr/include/ast/stak.h
305 file path=usr/include/ast/stdio.h
306 file path=usr/include/ast/stk.h
307 file path=usr/include/ast/sum.h
308 file path=usr/include/ast/swap.h
309 file path=usr/include/ast/tar.h
310 file path=usr/include/ast/times.h
311 file path=usr/include/ast/tm.h
312 file path=usr/include/ast/tmx.h
313 file path=usr/include/ast/tok.h
314 file path=usr/include/ast/tv.h
315 file path=usr/include/ast/usage.h
316 file path=usr/include/ast/vdb.h
317 file path=usr/include/ast/vecargs.h
318 file path=usr/include/ast/vmalloc.h
319 file path=usr/include/ast/wait.h
320 file path=usr/include/ast/wchar.h
321 file path=usr/include/ast/wordexp.h
322 file path=usr/include/atomic.h
323 file path=usr/include/attr.h
324 file path=usr/include/auth_attr.h
325 file path=usr/include/bsm/adt.h
```



```
326 file path=usr/include/bsm/adt_event.h
327 file path=usr/include/bsm/audit.h
328 file path=usr/include/bsm/audit_kernel.h
329 file path=usr/include/bsm/audit_kevents.h
330 file path=usr/include/bsm/audit_record.h
331 file path=usr/include/bsm/audit_uevents.h
332 file path=usr/include/bsm/devices.h
333 file path=usr/include/bsm/libbsm.h
334 file path=usr/include/config_admin.h
335 file path=usr/include/cpio.h
336 file path=usr/include/crypt.h
337 file path=usr/include/cryptoutil.h
338 file path=usr/include/ctype.h
339 file path=usr/include/curses.h
340 file path=usr/include/dat/dat.h
341 file path=usr/include/dat/dat_error.h
342 file path=usr/include/dat/dat_platform_specific.h
343 file path=usr/include/dat/dat_redirection.h
344 file path=usr/include/dat/dat_registry.h
345 file path=usr/include/dat/dat_vendor_specific.h
346 file path=usr/include/dat/udat.h
347 file path=usr/include/dat/udat_config.h
348 file path=usr/include/dat/udat_redirection.h
349 file path=usr/include/dat/udat_vendor_specific.h
350 file path=usr/include/deflt.h
351 file path=usr/include/des/des.h
352 file path=usr/include/des/desdata.h
353 file path=usr/include/des/softdes.h
354 file path=usr/include/device_info.h
355 file path=usr/include/devid.h
356 file path=usr/include/devmgmt.h
357 file path=usr/include/devpoll.h
358 file path=usr/include/dial.h
359 file path=usr/include/dirent.h
360 file path=usr/include/dlfcn.h
361 file path=usr/include/door.h
362 file path=usr/include/elf.h
363 file path=usr/include/err.h
364 file path=usr/include/errno.h
365 file path=usr/include/eti.h
366 file path=usr/include/euc.h
367 file path=usr/include/exacct.h
368 file path=usr/include/exacct_impl.h
369 file path=usr/include/exec_attr.h
370 file path=usr/include/execinfo.h
371 file path=usr/include/fatal.h
372 file path=usr/include/fcntl.h
373 file path=usr/include/float.h
374 file path=usr/include/fmtmsg.h
375 file path=usr/include/fnmatch.h
376 file path=usr/include/form.h
377 file path=usr/include/ftw.h
378 file path=usr/include/gelf.h
379 file path=usr/include/getopt.h
380 file path=usr/include/getwidth.h
381 file path=usr/include/glob.h
382 file path=usr/include/grp.h
383 file path=usr/include/gssapi/gssapi.h
384 file path=usr/include/gssapi/gssapi_ext.h
385 file path=usr/include/hal/libhal-storage.h
386 file path=usr/include/hal/libhal.h
387 $(i386_ONLY)file path=usr/include/ia32/sys/asm_linkage.h
388 $(i386_ONLY)file path=usr/include/ia32/sys/kdi_regs.h
389 $(i386_ONLY)file path=usr/include/ia32/sys/machtypes.h
390 $(i386_ONLY)file path=usr/include/ia32/sys/privmregs.h
391 $(i386_ONLY)file path=usr/include/ia32/sys/privregs.h
```

```
392 $(i386_ONLY)file path=usr/include/ia32/sys/psw.h
393 $(i386_ONLY)file path=usr/include/ia32/sys/pte.h
394 $(i386_ONLY)file path=usr/include/ia32/sys/reg.h
395 $(i386_ONLY)file path=usr/include/ia32/sys/stack.h
396 $(i386_ONLY)file path=usr/include/ia32/sys/trap.h
397 $(i386_ONLY)file path=usr/include/ia32/sys/traptrace.h
398 file path=usr/include/iconv.h
399 file path=usr/include/idmap.h
400 file path=usr/include/ieeefp.h
401 file path=usr/include/ifaddrs.h
402 file path=usr/include/inet/arp.h
403 file path=usr/include/inet/common.h
404 file path=usr/include/inet/ip.h
405 file path=usr/include/inet/ip6.h
406 file path=usr/include/inet/ip6_asp.h
407 file path=usr/include/inet/ip_arp.h
408 file path=usr/include/inet/ip_ftable.h
409 file path=usr/include/inet/ip_if.h
410 file path=usr/include/inet/ip_ire.h
411 file path=usr/include/inet/ip_multi.h
412 file path=usr/include/inet/ip_netinfo.h
413 file path=usr/include/inet/ip_rts.h
414 file path=usr/include/inet/ip_stack.h
415 file path=usr/include/inet/ipclassifier.h
416 file path=usr/include/inet/ipdrop.h
417 file path=usr/include/inet/ipnet.h
418 file path=usr/include/inet/ipp_common.h
419 file path=usr/include/inet/kssl/ksslapi.h
420 file path=usr/include/inet/led.h
421 file path=usr/include/inet/mi.h
422 file path=usr/include/inet/mib2.h
423 file path=usr/include/inet/nd.h
424 file path=usr/include/inet/optcom.h
425 file path=usr/include/inet/sctp_itf.h
426 file path=usr/include/inet/snmpcom.h
427 file path=usr/include/inet/tcp.h
428 file path=usr/include/inet/tcp_sack.h
429 file path=usr/include/inet/tcp_stack.h
430 file path=usr/include/inet/tcp_stats.h
431 file path=usr/include/inet/tunables.h
432 file path=usr/include/inet/wifi_ioctl.h
433 file path=usr/include/inttypes.h
434 file path=usr/include/ipmp.h
435 file path=usr/include/ipmp_admin.h
436 file path=usr/include/ipmp_mpathd.h
437 file path=usr/include/ipmp_query.h
438 file path=usr/include/ipp/ipgpc/ipgpc.h
439 file path=usr/include/ipp/ipp.h
440 file path=usr/include/ipp/ipp_config.h
441 file path=usr/include/ipp/ipp_impl.h
442 file path=usr/include/ipp/ippctl.h
443 file path=usr/include/iso/ctype_iso.h
444 file path=usr/include/iso/limits_iso.h
445 file path=usr/include/iso/locale_iso.h
446 file path=usr/include/iso/setjmp_iso.h
447 file path=usr/include/iso/signal_iso.h
448 file path=usr/include/iso/stdarg_c99.h
449 file path=usr/include/iso/stdarg_iso.h
450 file path=usr/include/iso/stddef_iso.h
451 file path=usr/include/iso/stdio_c99.h
452 file path=usr/include/iso/stdio_iso.h
453 file path=usr/include/iso/stdlib_c99.h
454 file path=usr/include/iso/stdlib_iso.h
455 file path=usr/include/iso/string_iso.h
456 file path=usr/include/iso/time_iso.h
457 file path=usr/include/iso/wchar_c99.h
```



```
458 file path=usr/include/iso/wchar_iso.h
459 file path=usr/include/iso/wctype_iso.h
460 file path=usr/include/iso646.h
461 file path=usr/include/kerberosv5/com_err.h
462 file path=usr/include/kerberosv5/krb5.h
463 file path=usr/include/kerberosv5/mit-sipb-copyright.h
464 file path=usr/include/kerberosv5/mit_copyright.h
465 file path=usr/include/klpd.h
466 file path=usr/include/kmfapi.h
467 file path=usr/include/kmftypes.h
468 file path=usr/include/kstat.h
469 file path=usr/include/kvm.h
470 file path=usr/include/langinfo.h
471 file path=usr/include/lastlog.h
472 file path=usr/include/lber.h
473 file path=usr/include/ldap.h
474 file path=usr/include/libcontract.h
475 file path=usr/include/libctf.h
476 file path=usr/include/libdevice.h
477 file path=usr/include/libdevinfo.h
478 file path=usr/include/libdladm.h
479 file path=usr/include/libdlbridge.h
480 file path=usr/include/libdlib.h
481 file path=usr/include/libdllink.h
482 file path=usr/include/libdlpi.h
483 file path=usr/include/libdlvlan.h
484 file path=usr/include/libelf.h
485 $(i386_ONLY)file path=usr/include/libfdisk.h
486 file path=usr/include/libfstyp.h
487 file path=usr/include/libfstyp_module.h
488 file path=usr/include/libgen.h
489 file path=usr/include/libgrubmgmt.h
490 file path=usr/include/libintl.h
491 file path=usr/include/libipmi.h
492 file path=usr/include/libipp.h
493 file path=usr/include/libnvpair.h
494 file path=usr/include/libnwam.h
495 file path=usr/include/libpolkit/libpolkit.h
496 file path=usr/include/librcm.h
497 file path=usr/include/libscf.h
498 file path=usr/include/libscf_priv.h
499 file path=usr/include/libshare.h
500 file path=usr/include/libsvm.h
501 file path=usr/include/libsysevent.h
502 file path=usr/include/libsysevent_impl.h
503 file path=usr/include/libtsnet.h
504 $(sparc_ONLY)file path=usr/include/libv12n.h
505 file path=usr/include/libw.h
506 file path=usr/include/libzfs.h
507 file path=usr/include/libzfs_core.h
508 file path=usr/include/libzoneinfo.h
509 file path=usr/include/limits.h
510 file path=usr/include/linenum.h
511 file path=usr/include/link.h
512 file path=usr/include/listen.h
513 file path=usr/include/locale.h
514 file path=usr/include/macros.h
515 file path=usr/include/maillock.h
516 file path=usr/include/malloc.h
517 file path=usr/include/md4.h
518 file path=usr/include/md5.h
519 file path=usr/include/mdiox.h
520 file path=usr/include/mdmn_changelog.h
521 file path=usr/include/memory.h
522 file path=usr/include/menu.h
523 file path=usr/include/meta.h
```

```
524 file path=usr/include/meta_basic.h
525 file path=usr/include/meta_runtime.h
526 file path=usr/include/metacl.h
527 file path=usr/include/metad.h
528 file path=usr/include/metadyn.h
529 file path=usr/include/metamed.h
530 file path=usr/include/metamhd.h
531 file path=usr/include/mhdx.h
532 file path=usr/include/mon.h
533 file path=usr/include/monetary.h
534 file path=usr/include/mp.h
535 file path=usr/include/mqueue.h
536 file path=usr/include/mtmalloc.h
537 file path=usr/include/nan.h
538 file path=usr/include/ndbm.h
539 file path=usr/include/ndpd.h
540 file path=usr/include/net/af.h
541 file path=usr/include/net/bridge.h
542 file path=usr/include/net/if.h
543 file path=usr/include/net/if_arp.h
544 file path=usr/include/net/if_dl.h
545 file path=usr/include/net/if_types.h
546 file path=usr/include/net/pfkeyv2.h
547 file path=usr/include/net/pfpolicy.h
548 file path=usr/include/net/ppp-comp.h
549 file path=usr/include/net/ppp_defs.h
550 file path=usr/include/net/pppio.h
551 file path=usr/include/net/radix.h
552 file path=usr/include/net/route.h
553 file path=usr/include/net/trill.h
554 file path=usr/include/net/vjcompress.h
555 file path=usr/include/netconfig.h
556 file path=usr/include/netdb.h
557 file path=usr/include/netdir.h
558 file path=usr/include/netinet/arp.h
559 file path=usr/include/netinet/dhcp.h
560 file path=usr/include/netinet/dhcp6.h
561 file path=usr/include/netinet/icmp6.h
562 file path=usr/include/netinet/icmp_var.h
563 file path=usr/include/netinet/if_ether.h
564 file path=usr/include/netinet/igmp.h
565 file path=usr/include/netinet/igmp_var.h
566 file path=usr/include/netinet/in.h
567 file path=usr/include/netinet/in_pcb.h
568 file path=usr/include/netinet/in_systm.h
569 file path=usr/include/netinet/in_var.h
570 file path=usr/include/netinet/ip.h
571 file path=usr/include/netinet/ip6.h
572 file path=usr/include/netinet/ip_icmp.h
573 file path=usr/include/netinet/ip_mroute.h
574 file path=usr/include/netinet/ip_var.h
575 file path=usr/include/netinet/pim.h
576 file path=usr/include/netinet/sctp.h
577 file path=usr/include/netinet/tcp.h
578 file path=usr/include/netinet/tcp_debug.h
579 file path=usr/include/netinet/tcp_fsm.h
580 file path=usr/include/netinet/tcp_seq.h
581 file path=usr/include/netinet/tcp_timer.h
582 file path=usr/include/netinet/tcp_var.h
583 file path=usr/include/netinet/tcpip.h
584 file path=usr/include/netinet/udp.h
585 file path=usr/include/netinet/udp_var.h
586 file path=usr/include/netinet/vrrp.h
587 file path=usr/include/nfs/auth.h
588 file path=usr/include/nfs/export.h
589 file path=usr/include/nfs/lm.h
```



```
590 file path=usr/include/nfs/mapid.h
591 file path=usr/include/nfs/mount.h
592 file path=usr/include/nfs/nfs.h
593 file path=usr/include/nfs/nfs4.h
594 file path=usr/include/nfs/nfs4_attr.h
595 file path=usr/include/nfs/nfs4_clnt.h
596 file path=usr/include/nfs/nfs4_db_impl.h
597 file path=usr/include/nfs/nfs4_idmap_impl.h
598 file path=usr/include/nfs/nfs4_kprot.h
599 file path=usr/include/nfs/nfs_acl.h
600 file path=usr/include/nfs/nfs_clnt.h
601 file path=usr/include/nfs/nfs_cmd.h
602 file path=usr/include/nfs/nfs_log.h
603 file path=usr/include/nfs/nfs_sec.h
604 file path=usr/include/nfs/nfsid_map.h
605 file path=usr/include/nfs/nfssys.h
606 file path=usr/include/nfs/rnode.h
607 file path=usr/include/nfs/rnode4.h
608 file path=usr/include/nl_types.h
609 file path=usr/include/nlist.h
610 file path=usr/include/note.h
611 file path=usr/include/nss_common.h
612 file path=usr/include/nss_dbdefs.h
613 file path=usr/include/nss_netdir.h
614 file path=usr/include/nsswitch.h
615 file path=usr/include/panel.h
616 file path=usr/include/paths.h
617 file path=usr/include/pcsample.h
618 file path=usr/include/pfmt.h
619 file path=usr/include/pkgdev.h
620 file path=usr/include/pkginfo.h
621 file path=usr/include/pkglocs.h
622 file path=usr/include/pkgstrct.h
623 file path=usr/include/pkgtrans.h
624 file path=usr/include/poll.h
625 file path=usr/include/port.h
626 file path=usr/include/priv.h
627 file path=usr/include/proc_service.h
628 file path=usr/include/procfs.h
629 file path=usr/include/prof.h
630 file path=usr/include/prof_attr.h
631 file path=usr/include/project.h
632 file path=usr/include/protocols/dumprestore.h
633 file path=usr/include/protocols/routed.h
634 file path=usr/include/protocols/rwhod.h
635 file path=usr/include/protocols/timed.h
636 file path=usr/include/pthread.h
637 file path=usr/include/pw.h
638 file path=usr/include/pwd.h
639 file path=usr/include/rcm_module.h
640 file path=usr/include/rctl.h
641 file path=usr/include/re_comp.h
642 file path=usr/include/regex.h
643 file path=usr/include/regexp.h
644 file path=usr/include/regexpr.h
645 file path=usr/include/resolv.h
646 file path=usr/include/rje.h
647 file path=usr/include/rp_plugin.h
648 file path=usr/include/rpc/auth.h
649 file path=usr/include/rpc/auth_des.h
650 file path=usr/include/rpc/auth_sys.h
651 file path=usr/include/rpc/auth_unix.h
652 file path=usr/include/rpc/bootparam.h
653 file path=usr/include/rpc/clnt.h
654 file path=usr/include/rpc/clnt_soc.h
655 file path=usr/include/rpc/clnt_stat.h
```

```
656 file path=usr/include/rpc/des_crypt.h
657 $(sparc_ONLY)file path=usr/include/rpc/ib.h
658 file path=usr/include/rpc/key_prot.h
659 file path=usr/include/rpc/nettype.h
660 file path=usr/include/rpc/pmap_clnt.h
661 file path=usr/include/rpc/pmap_prot.h
662 file path=usr/include/rpc/pmap_prot.x
663 file path=usr/include/rpc/pmap_rmt.h
664 file path=usr/include/rpc/raw.h
665 file path=usr/include/rpc/rpc.h
666 file path=usr/include/rpc/rpc_com.h
667 file path=usr/include/rpc/rpc_msg.h
668 file path=usr/include/rpc/rpc_rdma.h
669 file path=usr/include/rpc/rpc_sztypes.h
670 file path=usr/include/rpc/rpcb_clnt.h
671 file path=usr/include/rpc/rpcb_prot.h
672 file path=usr/include/rpc/rpcb_prot.x
673 file path=usr/include/rpc/rpcent.h
674 file path=usr/include/rpc/rpcsec_gss.h
675 file path=usr/include/rpc/rpcsys.h
676 file path=usr/include/rpc/svc.h
677 file path=usr/include/rpc/svc_auth.h
678 file path=usr/include/rpc/svc_mt.h
679 file path=usr/include/rpc/svc_soc.h
680 file path=usr/include/rpc/types.h
681 file path=usr/include/rpc/xdr.h
682 file path=usr/include/rpcsvc/autofs_prot.h
683 file path=usr/include/rpcsvc/autofs_prot.x
684 file path=usr/include/rpcsvc/bootparam.h
685 file path=usr/include/rpcsvc/bootparam_prot.h
686 file path=usr/include/rpcsvc/bootparam_prot.x
687 file path=usr/include/rpcsvc/dbm.h
688 file path=usr/include/rpcsvc/key_prot.x
689 file path=usr/include/rpcsvc/mount.h
690 file path=usr/include/rpcsvc/mount.x
691 file path=usr/include/rpcsvc/nfs4_prot.h
692 file path=usr/include/rpcsvc/nfs4_prot.x
693 file path=usr/include/rpcsvc/nfs_acl.h
694 file path=usr/include/rpcsvc/nfs_acl.x
695 file path=usr/include/rpcsvc/nfs_prot.h
696 file path=usr/include/rpcsvc/nfs_prot.x
697 file path=usr/include/rpcsvc/nis.h
698 file path=usr/include/rpcsvc/nis.x
699 file path=usr/include/rpcsvc/nis_db.h
700 file path=usr/include/rpcsvc/nis_object.x
701 file path=usr/include/rpcsvc/nislib.h
702 file path=usr/include/rpcsvc/nlm_prot.h
703 file path=usr/include/rpcsvc/nlm_prot.x
704 file path=usr/include/rpcsvc/nsm_addr.h
705 file path=usr/include/rpcsvc/nsm_addr.x
706 file path=usr/include/rpcsvc/rex.h
707 file path=usr/include/rpcsvc/rex.x
708 file path=usr/include/rpcsvc/rpc_sztypes.h
709 file path=usr/include/rpcsvc/rpc_sztypes.x
710 file path=usr/include/rpcsvc/rquota.h
711 file path=usr/include/rpcsvc/rquota.x
712 file path=usr/include/rpcsvc/rstat.h
713 file path=usr/include/rpcsvc/rstat.x
714 file path=usr/include/rpcsvc/rusers.h
715 file path=usr/include/rpcsvc/rusers.x
716 file path=usr/include/rpcsvc/rwall.h
717 file path=usr/include/rpcsvc/rwall.x
718 file path=usr/include/rpcsvc/sm_inter.h
719 file path=usr/include/rpcsvc/sm_inter.x
720 file path=usr/include/rpcsvc/spray.h
721 file path=usr/include/rpcsvc/spray.x
```



```
722 file path=usr/include/rpcsvc/ufs_prot.h
723 file path=usr/include/rpcsvc/ufs_prot.x
724 file path=usr/include/rpcsvc/yp.x
725 file path=usr/include/rpcsvc/yp_prot.h
726 file path=usr/include/rpcsvc/ypclnt.h
727 file path=usr/include/rpcsvc/yppasswd.h
728 file path=usr/include/rpcsvc/ypupd.h
729 file path=usr/include/rsmapi.h
730 file path=usr/include/rtld_db.h
731 file path=usr/include/sac.h
732 file path=usr/include/sasl/prop.h
733 file path=usr/include/sasl/sasl.h
734 file path=usr/include/sasl/saslplug.h
735 file path=usr/include/sasl/saslutil.h
736 file path=usr/include/sched.h
737 file path=usr/include/schedctl.h
738 file path=usr/include/scsi/libscsi.h
739 file path=usr/include/scsi/libses.h
740 file path=usr/include/scsi/libses_plugin.h
741 file path=usr/include/scsi/libsmp.h
742 file path=usr/include/scsi/libsmp_plugin.h
743 file path=usr/include/scsi/plugins/ses/framework/libses.h
744 file path=usr/include/scsi/plugins/ses/framework/ses2.h
745 file path=usr/include/scsi/plugins/ses/framework/ses2_impl.h
746 file path=usr/include/scsi/plugins/ses/vendor/sun.h
747 file path=usr/include/sdp.h
748 file path=usr/include/search.h
749 file path=usr/include/secdb.h
750 file path=usr/include/security/auditd.h
751 file path=usr/include/security/cryptoki.h
752 file path=usr/include/security/pam_appl.h
753 file path=usr/include/security/pam_modules.h
754 file path=usr/include/security/pkcs11.h
755 file path=usr/include/security/pkcs11f.h
756 file path=usr/include/security/pkcs11t.h
757 file path=usr/include/semaphore.h
758 file path=usr/include/setjmp.h
759 file path=usr/include/sgtty.h
760 file path=usr/include/sha1.h
761 file path=usr/include/sha2.h
762 file path=usr/include/shadow.h
763 file path=usr/include/sharefs/share.h
764 file path=usr/include/sharefs/sharefs.h
765 file path=usr/include/sharefs/sharetab.h
766 file path=usr/include/siginfo.h
767 file path=usr/include/signal.h
768 file path=usr/include/sip.h
769 file path=usr/include/smbios.h
770 file path=usr/include/spawn.h
771 $(i386_ONLY)file path=usr/include/stack_unwind.h
772 file path=usr/include/stdarg.h
773 file path=usr/include/stdbool.h
774 file path=usr/include/stddef.h
775 file path=usr/include/stdint.h
776 file path=usr/include/stdio.h
777 file path=usr/include/stdio_ext.h
778 file path=usr/include/stdio_impl.h
779 file path=usr/include/stdio_tag.h
780 file path=usr/include/stdlib.h
781 file path=usr/include/storclass.h
782 file path=usr/include/string.h
783 file path=usr/include/strings.h
784 file path=usr/include/stropts.h
785 file path=usr/include/syms.h
786 file path=usr/include/synch.h
787 file path=usr/include/sys/acct.h
```

```
 788 file path=usr/include/sys/acctctl.h
 789 file path=usr/include/sys/acl.h
 790 file path=usr/include/sys/acl_impl.h
 791 file path=usr/include/sys/acpi_drv.h
 792 file path=usr/include/sys/aio.h
 793 file path=usr/include/sys/aio_impl.h
 794 file path=usr/include/sys/aio_req.h
 795 file path=usr/include/sys/aiocb.h
 796 file path=usr/include/sys/archsystm.h
 797 file path=usr/include/sys/ascii.h
 798 file path=usr/include/sys/asm_linkage.h
 799 file path=usr/include/sys/asynch.h
 800 file path=usr/include/sys/atomic.h
 801 file path=usr/include/sys/attr.h
 802 file path=usr/include/sys/autoconf.h
 803 file path=usr/include/sys/auxv.h
 804 file path=usr/include/sys/auxv_386.h
 805 file path=usr/include/sys/auxv_SPARC.h
 806 file path=usr/include/sys/av/iec61883.h
 807 file path=usr/include/sys/avintr.h
 808 file path=usr/include/sys/avl.h
 809 file path=usr/include/sys/avl_impl.h
 810 file path=usr/include/sys/bitmap.h
 811 file path=usr/include/sys/bitset.h
 812 file path=usr/include/sys/bl.h
 813 file path=usr/include/sys/blkdev.h
 814 file path=usr/include/sys/bofi.h
 815 file path=usr/include/sys/bofi_impl.h
 816 file path=usr/include/sys/bootconf.h
 817 $(i386_ONLY)file path=usr/include/sys/bootregs.h
 818 file path=usr/include/sys/bootstat.h
 819 $(i386_ONLY)file path=usr/include/sys/bootsvcs.h
 820 file path=usr/include/sys/bpp_io.h
 821 file path=usr/include/sys/brand.h
 822 file path=usr/include/sys/buf.h
 823 file path=usr/include/sys/bufmod.h
 824 file path=usr/include/sys/bustypes.h
 825 file path=usr/include/sys/byteorder.h
 826 file path=usr/include/sys/callb.h
 827 file path=usr/include/sys/callo.h
 828 file path=usr/include/sys/cap_util.h
 829 file path=usr/include/sys/ccompile.h
 830 file path=usr/include/sys/cdio.h
 831 file path=usr/include/sys/cis.h
 832 file path=usr/include/sys/cis_handlers.h
 833 file path=usr/include/sys/cis_protos.h
 834 file path=usr/include/sys/cladm.h
 835 file path=usr/include/sys/class.h
 836 file path=usr/include/sys/clconf.h
 837 file path=usr/include/sys/cmlb.h
 838 file path=usr/include/sys/cmn_err.h
 839 $(sparc_ONLY)file path=usr/include/sys/cmpregs.h
 840 file path=usr/include/sys/compress.h
 841 file path=usr/include/sys/condvar.h
 842 file path=usr/include/sys/condvar_impl.h
 843 file path=usr/include/sys/conf.h
 844 file path=usr/include/sys/consdev.h
 845 file path=usr/include/sys/console.h
 846 file path=usr/include/sys/consplat.h
 847 file path=usr/include/sys/contract.h
 848 file path=usr/include/sys/contract/device.h
 849 file path=usr/include/sys/contract/device_impl.h
 850 file path=usr/include/sys/contract/process.h
 851 file path=usr/include/sys/contract/process_impl.h
 852 file path=usr/include/sys/contract_impl.h
 853 $(i386_ONLY)file path=usr/include/sys/controlregs.h
```



```
 854 file path=usr/include/sys/copyops.h
 855 file path=usr/include/sys/core.h
 856 file path=usr/include/sys/corectl.h
 857 file path=usr/include/sys/cpc_impl.h
 858 file path=usr/include/sys/cpc_pcbe.h
 859 file path=usr/include/sys/cpr.h
 860 file path=usr/include/sys/cpu.h
 861 file path=usr/include/sys/cpucaps.h
 862 file path=usr/include/sys/cpucaps_impl.h
 863 file path=usr/include/sys/cpupart.h
 864 file path=usr/include/sys/cpuvar.h
 865 file path=usr/include/sys/crc32.h
 866 file path=usr/include/sys/cred.h
 867 file path=usr/include/sys/cred_impl.h
 868 file path=usr/include/sys/crtctl.h
 869 file path=usr/include/sys/crypto/api.h
 870 file path=usr/include/sys/crypto/common.h
 871 file path=usr/include/sys/crypto/ioctl.h
 872 file path=usr/include/sys/crypto/ioctladmin.h
 873 file path=usr/include/sys/crypto/spi.h
 874 file path=usr/include/sys/cs.h
 875 file path=usr/include/sys/cs_priv.h
 876 file path=usr/include/sys/cs_strings.h
 877 file path=usr/include/sys/cs_stubs.h
 878 file path=usr/include/sys/cs_types.h
 879 file path=usr/include/sys/csiioctl.h
 880 file path=usr/include/sys/ctf.h
 881 file path=usr/include/sys/ctf_api.h
 882 file path=usr/include/sys/ctfs.h
 883 file path=usr/include/sys/ctfs_impl.h
 884 file path=usr/include/sys/ctype.h
 885 file path=usr/include/sys/cyclic.h
 886 file path=usr/include/sys/cyclic_impl.h
 887 file path=usr/include/sys/dacf.h
 888 file path=usr/include/sys/dacf_impl.h
 889 file path=usr/include/sys/damap.h
 890 file path=usr/include/sys/damap_impl.h
 891 file path=usr/include/sys/dc_ki.h
 892 file path=usr/include/sys/ddi.h
 893 file path=usr/include/sys/ddi_hp.h
 894 file path=usr/include/sys/ddi_hp_impl.h
 895 file path=usr/include/sys/ddi_impldefs.h
 896 file path=usr/include/sys/ddi_implfuncs.h
 897 file path=usr/include/sys/ddi_intr.h
 898 file path=usr/include/sys/ddi_intr_impl.h
 899 file path=usr/include/sys/ddi_isa.h
 900 file path=usr/include/sys/ddi_obsolete.h
 901 file path=usr/include/sys/ddi_periodic.h
 902 file path=usr/include/sys/ddidevmap.h
 903 file path=usr/include/sys/ddidmareq.h
 904 file path=usr/include/sys/ddifm.h
 905 file path=usr/include/sys/ddifm_impl.h
 906 file path=usr/include/sys/ddimapreq.h
 907 file path=usr/include/sys/ddipropdefs.h
 908 file path=usr/include/sys/dditypes.h
 909 file path=usr/include/sys/debug.h
 910 $(i386_ONLY)file path=usr/include/sys/debugreg.h
 911 file path=usr/include/sys/des.h
 912 file path=usr/include/sys/devcache.h
 913 file path=usr/include/sys/devcache_impl.h
 914 file path=usr/include/sys/devctl.h
 915 file path=usr/include/sys/devfm.h
 916 file path=usr/include/sys/devid_cache.h
 917 file path=usr/include/sys/devinfo_impl.h
 918 file path=usr/include/sys/devops.h
 919 file path=usr/include/sys/devpolicy.h
```

```
 920 file path=usr/include/sys/devpoll.h
 921 file path=usr/include/sys/dirent.h
 922 file path=usr/include/sys/disp.h
 923 file path=usr/include/sys/dkbad.h
 924 file path=usr/include/sys/dkio.h
 925 file path=usr/include/sys/dklabel.h
 926 $(sparc_ONLY)file path=usr/include/sys/dkmpio.h
 927 $(i386_ONLY)file path=usr/include/sys/dktp/altsctr.h
 928 $(i386_ONLY)file path=usr/include/sys/dktp/cmpkt.h
 929 file path=usr/include/sys/dktp/dadkio.h
 930 file path=usr/include/sys/dktp/fdisk.h
 931 file path=usr/include/sys/dl.h
 932 file path=usr/include/sys/dld.h
 933 file path=usr/include/sys/dlpi.h
 934 file path=usr/include/sys/dls_mgmt.h
 935 $(i386_ONLY)file path=usr/include/sys/dma_engine.h
 936 file path=usr/include/sys/dma_i8237A.h
 937 file path=usr/include/sys/dnlc.h
 938 file path=usr/include/sys/door.h
 939 file path=usr/include/sys/door_data.h
 940 file path=usr/include/sys/door_impl.h
 941 file path=usr/include/sys/dumphdr.h
 942 file path=usr/include/sys/ecppio.h
 943 file path=usr/include/sys/ecppreg.h
 944 file path=usr/include/sys/ecppsys.h
 945 file path=usr/include/sys/ecppvar.h
 946 file path=usr/include/sys/efi_partition.h
 947 file path=usr/include/sys/elf.h
 948 file path=usr/include/sys/elf_386.h
 949 file path=usr/include/sys/elf_SPARC.h
 950 file path=usr/include/sys/elf_amd64.h
 951 file path=usr/include/sys/elf_notes.h
 952 file path=usr/include/sys/elftypes.h
 953 file path=usr/include/sys/epm.h
 954 file path=usr/include/sys/errno.h
 955 file path=usr/include/sys/errorq.h
 956 file path=usr/include/sys/errorq_impl.h
 957 file path=usr/include/sys/esunddi.h
 958 file path=usr/include/sys/ethernet.h
 959 file path=usr/include/sys/euc.h
 960 file path=usr/include/sys/eucioctl.h
 961 file path=usr/include/sys/exacct.h
 962 file path=usr/include/sys/exacct_catalog.h
 963 file path=usr/include/sys/exacct_impl.h
 964 file path=usr/include/sys/exec.h
 965 file path=usr/include/sys/exechdr.h
 966 file path=usr/include/sys/fault.h
 967 file path=usr/include/sys/fbio.h
 968 file path=usr/include/sys/fbuf.h
 969 file path=usr/include/sys/fc4/fc.h
 970 file path=usr/include/sys/fc4/fc_transport.h
 971 file path=usr/include/sys/fc4/fcal.h
 972 file path=usr/include/sys/fc4/fcal_linkapp.h
 973 file path=usr/include/sys/fc4/fcal_transport.h
 974 file path=usr/include/sys/fc4/fcio.h
 975 file path=usr/include/sys/fc4/fcp.h
 976 file path=usr/include/sys/fc4/linkapp.h
 977 file path=usr/include/sys/fcntl.h
 978 file path=usr/include/sys/fdbuffer.h
 979 file path=usr/include/sys/fdio.h
 980 $(sparc_ONLY)file path=usr/include/sys/fdreg.h
 981 $(sparc_ONLY)file path=usr/include/sys/fdvar.h
 982 file path=usr/include/sys/feature_tests.h
 983 file path=usr/include/sys/fem.h
 984 file path=usr/include/sys/file.h
 985 file path=usr/include/sys/filio.h
```



```
 986 file path=usr/include/sys/flock.h
 987 file path=usr/include/sys/flock_impl.h
 988 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/SPARC64-VI.h
 989 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-II.h
 990 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-III.h
 991 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-T1.h
 992 file path=usr/include/sys/fm/fs/zfs.h
 993 file path=usr/include/sys/fm/io/ddi.h
 994 file path=usr/include/sys/fm/io/disk.h
 995 file path=usr/include/sys/fm/io/opl_mc_fm.h
 996 file path=usr/include/sys/fm/io/pci.h
 997 file path=usr/include/sys/fm/io/scsi.h
 998 file path=usr/include/sys/fm/io/sun4upci.h
 999 file path=usr/include/sys/fm/protocol.h
1000 file path=usr/include/sys/fm/util.h
1001 file path=usr/include/sys/fork.h
1002 $(i386_ONLY)file path=usr/include/sys/fp.h
1003 $(sparc_ONLY)file path=usr/include/sys/fpu/fpu_simulator.h
1004 $(sparc_ONLY)file path=usr/include/sys/fpu/fpusystm.h
1005 $(sparc_ONLY)file path=usr/include/sys/fpu/globals.h
1006 $(sparc_ONLY)file path=usr/include/sys/fpu/ieee.h
1007 file path=usr/include/sys/frame.h
1008 file path=usr/include/sys/fs/autofs.h
1009 file path=usr/include/sys/fs/cachefs_dir.h
1010 file path=usr/include/sys/fs/cachefs_dlog.h
1011 file path=usr/include/sys/fs/cachefs_filegrp.h
1012 file path=usr/include/sys/fs/cachefs_fs.h
1013 file path=usr/include/sys/fs/cachefs_fscache.h
1014 file path=usr/include/sys/fs/cachefs_ioctl.h
1015 file path=usr/include/sys/fs/cachefs_log.h
1016 file path=usr/include/sys/fs/decomp.h
1017 file path=usr/include/sys/fs/dv_node.h
1018 file path=usr/include/sys/fs/fifonode.h
1019 file path=usr/include/sys/fs/hsfs_isospec.h
1020 file path=usr/include/sys/fs/hsfs_node.h
1021 file path=usr/include/sys/fs/hsfs_rrip.h
1022 file path=usr/include/sys/fs/hsfs_spec.h
1023 file path=usr/include/sys/fs/hsfs_susp.h
1024 file path=usr/include/sys/fs/lofs_info.h
1025 file path=usr/include/sys/fs/lofs_node.h
1026 file path=usr/include/sys/fs/mntdata.h
1027 file path=usr/include/sys/fs/namenode.h
1028 file path=usr/include/sys/fs/pc_dir.h
1029 file path=usr/include/sys/fs/pc_fs.h
1030 file path=usr/include/sys/fs/pc_label.h
1031 file path=usr/include/sys/fs/pc_node.h
1032 file path=usr/include/sys/fs/pxfs_ki.h
1033 file path=usr/include/sys/fs/sdev_impl.h
1034 file path=usr/include/sys/fs/snode.h
1035 file path=usr/include/sys/fs/swapnode.h
1036 file path=usr/include/sys/fs/tmp.h
1037 file path=usr/include/sys/fs/tmpnode.h
1038 file path=usr/include/sys/fs/udf_inode.h
1039 file path=usr/include/sys/fs/udf_volume.h
1040 file path=usr/include/sys/fs/ufs_acl.h
1041 file path=usr/include/sys/fs/ufs_bio.h
1042 file path=usr/include/sys/fs/ufs_filio.h
1043 file path=usr/include/sys/fs/ufs_fs.h
1044 file path=usr/include/sys/fs/ufs_fsdir.h
1045 file path=usr/include/sys/fs/ufs_inode.h
1046 file path=usr/include/sys/fs/ufs_lockfs.h
1047 file path=usr/include/sys/fs/ufs_log.h
1048 file path=usr/include/sys/fs/ufs_mount.h
1049 file path=usr/include/sys/fs/ufs_panic.h
1050 file path=usr/include/sys/fs/ufs_prot.h
1051 file path=usr/include/sys/fs/ufs_quota.h
```

```
1052 file path=usr/include/sys/fs/ufs_snap.h
1053 file path=usr/include/sys/fs/ufs_trans.h
1054 file path=usr/include/sys/fs/zfs.h
1055 file path=usr/include/sys/fs_reparse.h
1056 file path=usr/include/sys/fs_subr.h
1057 file path=usr/include/sys/fsid.h
1058 $(sparc_ONLY)file path=usr/include/sys/fsr.h
1059 file path=usr/include/sys/fss.h
1060 file path=usr/include/sys/fssnap.h
1061 file path=usr/include/sys/fssnap_if.h
1062 file path=usr/include/sys/fsspriocntl.h
1063 file path=usr/include/sys/fstyp.h
1064 file path=usr/include/sys/ftrace.h
1065 file path=usr/include/sys/fx.h
1066 file path=usr/include/sys/fxpriocntl.h
1067 file path=usr/include/sys/gfs.h
1068 file path=usr/include/sys/gld.h
1069 file path=usr/include/sys/gldpriv.h
1070 file path=usr/include/sys/group.h
1071 file path=usr/include/sys/hdio.h
1072 file path=usr/include/sys/hook.h
1073 file path=usr/include/sys/hook_event.h
1074 file path=usr/include/sys/hook_impl.h
1075 file path=usr/include/sys/hotplug/hpcsvc.h
1076 file path=usr/include/sys/hotplug/hpctrl.h
1077 file path=usr/include/sys/hotplug/pci/pcicfg.h
1078 file path=usr/include/sys/hotplug/pci/pcihp.h
1079 file path=usr/include/sys/hwconf.h
1080 $(i386_ONLY)file path=usr/include/sys/hypervisor.h
1081 $(i386_ONLY)file path=usr/include/sys/i8272A.h
1082 file path=usr/include/sys/ia.h
1083 file path=usr/include/sys/iapriocntl.h
1084 file path=usr/include/sys/ib/adapters/hermon/hermon_ioctl.h
1085 file path=usr/include/sys/ib/adapters/mlnx_umap.h
1086 file path=usr/include/sys/ib/adapters/tavor/tavor_ioctl.h
1087 file path=usr/include/sys/ib/clients/ibd/ibd.h
1088 file path=usr/include/sys/ib/clients/of/ofa_solaris.h
1089 file path=usr/include/sys/ib/clients/of/ofed_kernel.h
1090 file path=usr/include/sys/ib/clients/of/rdma/ib_addr.h
1091 file path=usr/include/sys/ib/clients/of/rdma/ib_user_mad.h
1092 file path=usr/include/sys/ib/clients/of/rdma/ib_user_sa.h
1093 file path=usr/include/sys/ib/clients/of/rdma/ib_user_verbs.h
1094 file path=usr/include/sys/ib/clients/of/rdma/ib_verbs.h
1095 file path=usr/include/sys/ib/clients/of/rdma/rdma_cm.h
1096 file path=usr/include/sys/ib/clients/of/rdma/rdma_user_cm.h
1097 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_cma.h
1098 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ib_cma.h
1099 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_kverb_impl.h
1100 file path=usr/include/sys/ib/clients/of/sol_ofs/sol_ofs_common.h
1101 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_rdma_user_cm.h
1102 file path=usr/include/sys/ib/clients/of/sol_ucma/sol_ucma.h
1103 file path=usr/include/sys/ib/clients/of/sol_umad/sol_umad.h
1104 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs.h
1105 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs2ucma.h
1106 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_comp.h
1107 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_event.h
1108 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_hca.h
1109 file path=usr/include/sys/ib/clients/of/sol_uverbs/sol_uverbs_qp.h
1110 file path=usr/include/sys/ib/ib_pkt_hdrs.h
1111 file path=usr/include/sys/ib/ib_types.h
1112 file path=usr/include/sys/ib/ibnex/ibnex_devctl.h
1113 file path=usr/include/sys/ib/ibtl/ibci.h
1114 file path=usr/include/sys/ib/ibtl/ibti.h
1115 file path=usr/include/sys/ib/ibtl/ibti_cm.h
1116 file path=usr/include/sys/ib/ibtl/ibti_common.h
1117 file path=usr/include/sys/ib/ibtl/ibtl_ci_types.h
```



```
1118 file path=usr/include/sys/ib/ibtl/ibtl_status.h
1119 file path=usr/include/sys/ib/ibtl/ibtl_types.h
1120 file path=usr/include/sys/ib/ibtl/ibvti.h
1121 file path=usr/include/sys/ib/ibtl/impl/ibtl_util.h
1122 file path=usr/include/sys/ib/mgt/ib_dm_attr.h
1123 file path=usr/include/sys/ib/mgt/ib_mad.h
1124 file path=usr/include/sys/ib/mgt/ibmf/ibmf.h
1125 file path=usr/include/sys/ib/mgt/ibmf/ibmf_msg.h
1126 file path=usr/include/sys/ib/mgt/ibmf/ibmf_saa.h
1127 file path=usr/include/sys/ib/mgt/ibmf/ibmf_utils.h
1128 file path=usr/include/sys/ib/mgt/sa_recs.h
1129 file path=usr/include/sys/ib/mgt/sm_attr.h
1130 file path=usr/include/sys/ibpart.h
1131 file path=usr/include/sys/id32.h
1132 file path=usr/include/sys/id_space.h
1133 file path=usr/include/sys/idmap.h
1134 file path=usr/include/sys/inline.h
1135 file path=usr/include/sys/instance.h
1136 file path=usr/include/sys/int_const.h
1137 file path=usr/include/sys/int_fmtio.h
1138 file path=usr/include/sys/int_limits.h
1139 file path=usr/include/sys/int_types.h
1140 file path=usr/include/sys/inttypes.h
1141 file path=usr/include/sys/ioccom.h
1142 file path=usr/include/sys/ioctl.h
1143 $(i386_ONLY)file path=usr/include/sys/iommulib.h
1144 file path=usr/include/sys/ipc.h
1145 file path=usr/include/sys/ipc_impl.h
1146 file path=usr/include/sys/ipc_rctl.h
1147 file path=usr/include/sys/isa_defs.h
1148 file path=usr/include/sys/iso/signal_iso.h
1149 file path=usr/include/sys/jioctl.h
1150 file path=usr/include/sys/kbd.h
1151 file path=usr/include/sys/kbdreg.h
1152 file path=usr/include/sys/kbio.h
1153 file path=usr/include/sys/kcpc.h
1154 file path=usr/include/sys/kd.h
1155 file path=usr/include/sys/kdi.h
1156 file path=usr/include/sys/kdi_impl.h
1157 file path=usr/include/sys/kdi_machimpl.h
1158 $(i386_ONLY)file path=usr/include/sys/kdi_regs.h
1159 file path=usr/include/sys/kiconv.h
1160 file path=usr/include/sys/kidmap.h
1161 file path=usr/include/sys/klpd.h
1162 file path=usr/include/sys/klwp.h
1163 file path=usr/include/sys/kmem.h
1164 file path=usr/include/sys/kmem_impl.h
1165 file path=usr/include/sys/kobj.h
1166 file path=usr/include/sys/kobj_impl.h
1167 file path=usr/include/sys/ksocket.h
1168 file path=usr/include/sys/kstat.h
1169 file path=usr/include/sys/kstr.h
1170 file path=usr/include/sys/ksyms.h
1171 file path=usr/include/sys/ksynch.h
1172 file path=usr/include/sys/lc_core.h
1173 file path=usr/include/sys/ldterm.h
1174 file path=usr/include/sys/lgrp.h
1175 file path=usr/include/sys/lgrp_user.h
1176 file path=usr/include/sys/link.h
1177 file path=usr/include/sys/list.h
1178 file path=usr/include/sys/list_impl.h
1179 file path=usr/include/sys/llc1.h
1180 file path=usr/include/sys/loadavg.h
1181 file path=usr/include/sys/localedef.h
1182 file path=usr/include/sys/lock.h
1183 file path=usr/include/sys/lockfs.h
```

```
1184 file path=usr/include/sys/lofi.h
1185 file path=usr/include/sys/log.h
1186 file path=usr/include/sys/logindmux.h
1187 file path=usr/include/sys/lvm/md_basic.h
1188 file path=usr/include/sys/lvm/md_convert.h
1189 file path=usr/include/sys/lvm/md_crc.h
1190 file path=usr/include/sys/lvm/md_hotspares.h
1191 file path=usr/include/sys/lvm/md_mddb.h
1192 file path=usr/include/sys/lvm/md_mdiox.h
1193 file path=usr/include/sys/lvm/md_mhdx.h
1194 file path=usr/include/sys/lvm/md_mirror.h
1195 file path=usr/include/sys/lvm/md_mirror_shared.h
1196 file path=usr/include/sys/lvm/md_names.h
1197 file path=usr/include/sys/lvm/md_notify.h
1198 file path=usr/include/sys/lvm/md_raid.h
1199 file path=usr/include/sys/lvm/md_rename.h
1200 file path=usr/include/sys/lvm/md_sp.h
1201 file path=usr/include/sys/lvm/md_stripe.h
1202 file path=usr/include/sys/lvm/md_trans.h
1203 file path=usr/include/sys/lvm/mdio.h
1204 file path=usr/include/sys/lvm/mdmed.h
1205 file path=usr/include/sys/lvm/mdmn_commd.h
1206 file path=usr/include/sys/lvm/mdvar.h
1207 file path=usr/include/sys/lwp.h
1208 file path=usr/include/sys/lwp_timer_impl.h
1209 file path=usr/include/sys/lwp_upimutex_impl.h
1210 file path=usr/include/sys/mac.h
1211 file path=usr/include/sys/mac_ether.h
1212 file path=usr/include/sys/mac_flow.h
1213 file path=usr/include/sys/mac_provider.h
1214 file path=usr/include/sys/machelf.h
1215 file path=usr/include/sys/machlock.h
1216 file path=usr/include/sys/machsig.h
1217 file path=usr/include/sys/machtypes.h
1218 file path=usr/include/sys/map.h
1219 $(i386_ONLY)file path=usr/include/sys/mc.h
1220 $(i386_ONLY)file path=usr/include/sys/mc_amd.h
1221 $(i386_ONLY)file path=usr/include/sys/mc_intel.h
1222 $(i386_ONLY)file path=usr/include/sys/mca_amd.h
1223 $(i386_ONLY)file path=usr/include/sys/mca_x86.h
1224 file path=usr/include/sys/md4.h
1225 file path=usr/include/sys/md5.h
1226 file path=usr/include/sys/md5_consts.h
1227 file path=usr/include/sys/mdi_impldefs.h
1228 file path=usr/include/sys/mem.h
1229 file path=usr/include/sys/mem_config.h
1230 file path=usr/include/sys/memlist.h
1231 file path=usr/include/sys/mhd.h
1232 file path=usr/include/sys/mii.h
1233 file path=usr/include/sys/miiregs.h
1234 file path=usr/include/sys/mkdev.h
1235 file path=usr/include/sys/mman.h
1236 file path=usr/include/sys/mmapobj.h
1237 file path=usr/include/sys/mntent.h
1238 file path=usr/include/sys/mntio.h
1239 file path=usr/include/sys/mnttab.h
1240 file path=usr/include/sys/modctl.h
1241 file path=usr/include/sys/mode.h
1242 file path=usr/include/sys/model.h
1243 file path=usr/include/sys/modhash.h
1244 file path=usr/include/sys/modhash_impl.h
1245 file path=usr/include/sys/mount.h
1246 file path=usr/include/sys/mouse.h
1247 file path=usr/include/sys/msacct.h
1248 file path=usr/include/sys/msg.h
1249 file path=usr/include/sys/msg_impl.h
```



```
1250 file path=usr/include/sys/msio.h
1251 file path=usr/include/sys/msreg.h
1252 file path=usr/include/sys/mtio.h
1253 file path=usr/include/sys/multidata.h
1254 file path=usr/include/sys/mutex.h
1255 $(i386_ONLY)file path=usr/include/sys/mutex_impl.h
1256 file path=usr/include/sys/nbmlock.h
1257 file path=usr/include/sys/ndi_impldefs.h
1258 file path=usr/include/sys/ndifm.h
1259 file path=usr/include/sys/netconfig.h
1260 file path=usr/include/sys/neti.h
1261 file path=usr/include/sys/netstack.h
1262 file path=usr/include/sys/nexusdefs.h
1263 file path=usr/include/sys/note.h
1264 file path=usr/include/sys/nvpair.h
1265 file path=usr/include/sys/nvpair_impl.h
1266 file path=usr/include/sys/objfs.h
1267 file path=usr/include/sys/objfs_impl.h
1268 file path=usr/include/sys/obpdefs.h
1269 file path=usr/include/sys/old_procfs.h
1270 file path=usr/include/sys/open.h
1271 file path=usr/include/sys/openpromio.h
1272 file path=usr/include/sys/panic.h
1273 file path=usr/include/sys/param.h
1274 file path=usr/include/sys/pathconf.h
1275 file path=usr/include/sys/pathname.h
1276 file path=usr/include/sys/pattr.h
1277 file path=usr/include/sys/pbio.h
1278 file path=usr/include/sys/pcb.h
1279 file path=usr/include/sys/pccard.h
1280 file path=usr/include/sys/pci.h
1281 $(i386_ONLY)file path=usr/include/sys/pcic_reg.h
1282 $(i386_ONLY)file path=usr/include/sys/pcic_var.h
1283 file path=usr/include/sys/pcie.h
1284 file path=usr/include/sys/pcmcia.h
1285 file path=usr/include/sys/pctypes.h
1286 file path=usr/include/sys/pfmod.h
1287 file path=usr/include/sys/pg.h
1288 file path=usr/include/sys/pghw.h
1289 file path=usr/include/sys/physmem.h
1290 $(i386_ONLY)file path=usr/include/sys/pic.h
1291 file path=usr/include/sys/pidnode.h
1292 #endif /* ! codereview */
1293 $(i386_ONLY)file path=usr/include/sys/pit.h
1294 file path=usr/include/sys/pkp_hash.h
1295 file path=usr/include/sys/pm.h
1296 $(i386_ONLY)file path=usr/include/sys/pmem.h
1297 file path=usr/include/sys/policy.h
1298 file path=usr/include/sys/poll.h
1299 file path=usr/include/sys/poll_impl.h
1300 file path=usr/include/sys/pool.h
1301 file path=usr/include/sys/pool_impl.h
1302 file path=usr/include/sys/pool_pset.h
1303 file path=usr/include/sys/port.h
1304 file path=usr/include/sys/port_impl.h
1305 file path=usr/include/sys/port_kernel.h
1306 file path=usr/include/sys/ppmio.h
1307 file path=usr/include/sys/priocntl.h
1308 file path=usr/include/sys/priv.h
1309 file path=usr/include/sys/priv_const.h
1310 file path=usr/include/sys/priv_impl.h
1311 file path=usr/include/sys/priv_names.h
1312 $(i386_ONLY)file path=usr/include/sys/privmregs.h
1313 $(i386_ONLY)file path=usr/include/sys/privregs.h
1314 file path=usr/include/sys/prnio.h
1315 file path=usr/include/sys/proc.h
```

```
1316 file path=usr/include/sys/proc/prdata.h
1317 file path=usr/include/sys/processor.h
1318 file path=usr/include/sys/procfs.h
1319 file path=usr/include/sys/procfs_isa.h
1320 file path=usr/include/sys/procset.h
1321 file path=usr/include/sys/project.h
1322 $(i386_ONLY)file path=usr/include/sys/prom_emul.h
1323 $(i386_ONLY)file path=usr/include/sys/prom_isa.h
1324 $(i386_ONLY)file path=usr/include/sys/prom_plat.h
1325 file path=usr/include/sys/promif.h
1326 file path=usr/include/sys/promimpl.h
1327 file path=usr/include/sys/protosw.h
1328 file path=usr/include/sys/prsystm.h
1329 file path=usr/include/sys/pset.h
1330 file path=usr/include/sys/psw.h
1331 $(i386_ONLY)file path=usr/include/sys/pte.h
1332 file path=usr/include/sys/ptem.h
1333 file path=usr/include/sys/ptms.h
1334 file path=usr/include/sys/ptyvar.h
1335 file path=usr/include/sys/queue.h
1336 file path=usr/include/sys/raidioctl.h
1337 file path=usr/include/sys/ramdisk.h
1338 file path=usr/include/sys/random.h
1339 file path=usr/include/sys/rctl.h
1340 file path=usr/include/sys/rctl_impl.h
1341 file path=usr/include/sys/rds.h
1342 file path=usr/include/sys/reboot.h
1343 file path=usr/include/sys/refstr.h
1344 file path=usr/include/sys/refstr_impl.h
1345 file path=usr/include/sys/reg.h
1346 file path=usr/include/sys/regset.h
1347 file path=usr/include/sys/resource.h
1348 file path=usr/include/sys/rlioctl.h
1349 file path=usr/include/sys/rsm/rsm.h
1350 file path=usr/include/sys/rsm/rsm_common.h
1351 file path=usr/include/sys/rsm/rsmapi_common.h
1352 file path=usr/include/sys/rsm/rsmka_path_int.h
1353 file path=usr/include/sys/rsm/rsmndi.h
1354 file path=usr/include/sys/rsm/rsmpi.h
1355 file path=usr/include/sys/rsm/rsmpi_driver.h
1356 file path=usr/include/sys/rt.h
1357 $(i386_ONLY)file path=usr/include/sys/rtc.h
1358 file path=usr/include/sys/rtpriocntl.h
1359 file path=usr/include/sys/rwlock.h
1360 file path=usr/include/sys/rwlock_impl.h
1361 file path=usr/include/sys/rwstlock.h
1362 file path=usr/include/sys/sad.h
1363 $(i386_ONLY)file path=usr/include/sys/sata/sata_defs.h
1364 $(i386_ONLY)file path=usr/include/sys/sata/sata_hba.h
1365 file path=usr/include/sys/schedctl.h
1366 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/ifpio.h
1367 file path=usr/include/sys/scsi/adapters/scsi_vhci.h
1368 $(sparc_ONLY)file path=usr/include/sys/scsi/adapters/sfvar.h
1369 file path=usr/include/sys/scsi/conf/autoconf.h
1370 file path=usr/include/sys/scsi/conf/device.h
1371 file path=usr/include/sys/scsi/generic/commands.h
1372 file path=usr/include/sys/scsi/generic/dad_mode.h
1373 file path=usr/include/sys/scsi/generic/inquiry.h
1374 file path=usr/include/sys/scsi/generic/message.h
1375 file path=usr/include/sys/scsi/generic/mode.h
1376 file path=usr/include/sys/scsi/generic/persist.h
1377 file path=usr/include/sys/scsi/generic/sense.h
1378 file path=usr/include/sys/scsi/generic/sff_frames.h
1379 file path=usr/include/sys/scsi/generic/smp_frames.h
1380 file path=usr/include/sys/scsi/generic/status.h
1381 file path=usr/include/sys/scsi/impl/commands.h
```



```
1382 file path=usr/include/sys/scsi/impl/inquiry.h
1383 file path=usr/include/sys/scsi/impl/mode.h
1384 file path=usr/include/sys/scsi/impl/scsi_reset_notify.h
1385 file path=usr/include/sys/scsi/impl/scsi_sas.h
1386 file path=usr/include/sys/scsi/impl/sense.h
1387 file path=usr/include/sys/scsi/impl/services.h
1388 file path=usr/include/sys/scsi/impl/smp_transport.h
1389 file path=usr/include/sys/scsi/impl/spc3_types.h
1390 file path=usr/include/sys/scsi/impl/status.h
1391 file path=usr/include/sys/scsi/impl/transport.h
1392 file path=usr/include/sys/scsi/impl/types.h
1393 file path=usr/include/sys/scsi/impl/uscsi.h
1394 file path=usr/include/sys/scsi/impl/usmp.h
1395 file path=usr/include/sys/scsi/scsi.h
1396 file path=usr/include/sys/scsi/scsi_address.h
1397 file path=usr/include/sys/scsi/scsi_ctl.h
1398 file path=usr/include/sys/scsi/scsi_fm.h
1399 file path=usr/include/sys/scsi/scsi_params.h
1400 file path=usr/include/sys/scsi/scsi_pkt.h
1401 file path=usr/include/sys/scsi/scsi_resource.h
1402 file path=usr/include/sys/scsi/scsi_types.h
1403 file path=usr/include/sys/scsi/scsi_watch.h
1404 file path=usr/include/sys/scsi/targets/sddef.h
1405 file path=usr/include/sys/scsi/targets/ses.h
1406 file path=usr/include/sys/scsi/targets/sesio.h
1407 file path=usr/include/sys/scsi/targets/sgendef.h
1408 file path=usr/include/sys/scsi/targets/smp.h
1409 $(sparc_ONLY)file path=usr/include/sys/scsi/targets/ssddef.h
1410 file path=usr/include/sys/scsi/targets/stdef.h
1411 $(i386_ONLY)file path=usr/include/sys/segment.h
1412 $(i386_ONLY)file path=usr/include/sys/segments.h
1413 file path=usr/include/sys/select.h
1414 file path=usr/include/sys/sem.h
1415 file path=usr/include/sys/sem_impl.h
1416 file path=usr/include/sys/sema_impl.h
1417 file path=usr/include/sys/semaphore.h
1418 file path=usr/include/sys/sendfile.h
1419 $(sparc_ONLY)file path=usr/include/sys/ser_async.h
1420 file path=usr/include/sys/ser_sync.h
1421 $(sparc_ONLY)file path=usr/include/sys/ser_zscc.h
1422 file path=usr/include/sys/serializer.h
1423 file path=usr/include/sys/session.h
1424 file path=usr/include/sys/sha1.h
1425 file path=usr/include/sys/sha2.h
1426 file path=usr/include/sys/share.h
1427 file path=usr/include/sys/shm.h
1428 file path=usr/include/sys/shm_impl.h
1429 file path=usr/include/sys/sid.h
1430 file path=usr/include/sys/siginfo.h
1431 file path=usr/include/sys/signal.h
1432 file path=usr/include/sys/sleepq.h
1433 file path=usr/include/sys/smbios.h
1434 file path=usr/include/sys/smbios_impl.h
1435 file path=usr/include/sys/smedia.h
1436 file path=usr/include/sys/sobject.h
1437 $(sparc_ONLY)file path=usr/include/sys/socal_cq_defs.h
1438 $(sparc_ONLY)file path=usr/include/sys/socalio.h
1439 $(sparc_ONLY)file path=usr/include/sys/socalmap.h
1440 $(sparc_ONLY)file path=usr/include/sys/socalreg.h
1441 $(sparc_ONLY)file path=usr/include/sys/socalvar.h
1442 file path=usr/include/sys/socket.h
1443 file path=usr/include/sys/socket_impl.h
1444 file path=usr/include/sys/socket_proto.h
1445 file path=usr/include/sys/socketvar.h
1446 file path=usr/include/sys/sockio.h
1447 file path=usr/include/sys/spl.h
```

```
1448 file path=usr/include/sys/squeue.h
1449 file path=usr/include/sys/squeue_impl.h
1450 file path=usr/include/sys/sservice.h
1451 file path=usr/include/sys/stack.h
1452 file path=usr/include/sys/stat.h
1453 file path=usr/include/sys/stat_impl.h
1454 file path=usr/include/sys/statfs.h
1455 file path=usr/include/sys/statvfs.h
1456 file path=usr/include/sys/stdbool.h
1457 file path=usr/include/sys/stdint.h
1458 file path=usr/include/sys/stermio.h
1459 file path=usr/include/sys/stream.h
1460 file path=usr/include/sys/strft.h
1461 file path=usr/include/sys/strlog.h
1462 file path=usr/include/sys/strmdep.h
1463 file path=usr/include/sys/stropts.h
1464 file path=usr/include/sys/strredir.h
1465 file path=usr/include/sys/strstat.h
1466 file path=usr/include/sys/strsubr.h
1467 file path=usr/include/sys/strsun.h
1468 file path=usr/include/sys/strtty.h
1469 file path=usr/include/sys/sunddi.h
1470 file path=usr/include/sys/sunldi.h
1471 file path=usr/include/sys/sunldi_impl.h
1472 file path=usr/include/sys/sunmdi.h
1473 file path=usr/include/sys/sunndi.h
1474 file path=usr/include/sys/sunpm.h
1475 file path=usr/include/sys/suntpi.h
1476 file path=usr/include/sys/suntty.h
1477 file path=usr/include/sys/swap.h
1478 file path=usr/include/sys/synch.h
1479 file path=usr/include/sys/syscall.h
1480 file path=usr/include/sys/sysconf.h
1481 file path=usr/include/sys/sysconfig.h
1482 file path=usr/include/sys/sysconfig_impl.h
1483 file path=usr/include/sys/sysdc.h
1484 file path=usr/include/sys/sysdc_impl.h
1485 file path=usr/include/sys/sysevent.h
1486 file path=usr/include/sys/sysevent/ap_driver.h
1487 file path=usr/include/sys/sysevent/dev.h
1488 file path=usr/include/sys/sysevent/domain.h
1489 file path=usr/include/sys/sysevent/dr.h
1490 file path=usr/include/sys/sysevent/env.h
1491 file path=usr/include/sys/sysevent/eventdefs.h
1492 file path=usr/include/sys/sysevent/ipmp.h
1493 file path=usr/include/sys/sysevent/pwrctl.h
1494 file path=usr/include/sys/sysevent/svm.h
1495 file path=usr/include/sys/sysevent/vrrp.h
1496 file path=usr/include/sys/sysevent_impl.h
1497 $(i386_ONLY)file path=usr/include/sys/sysi86.h
1498 file path=usr/include/sys/sysinfo.h
1499 file path=usr/include/sys/syslog.h
1500 file path=usr/include/sys/sysmacros.h
1501 file path=usr/include/sys/systeminfo.h
1502 file path=usr/include/sys/systm.h
1503 file path=usr/include/sys/t_kuser.h
1504 file path=usr/include/sys/t_lock.h
1505 file path=usr/include/sys/task.h
1506 file path=usr/include/sys/taskq.h
1507 file path=usr/include/sys/taskq_impl.h
1508 file path=usr/include/sys/telioctl.h
1509 file path=usr/include/sys/termio.h
1510 file path=usr/include/sys/termios.h
1511 file path=usr/include/sys/termiox.h
1512 file path=usr/include/sys/thread.h
1513 file path=usr/include/sys/ticlts.h
```



```
1514 file path=usr/include/sys/ticots.h
1515 file path=usr/include/sys/ticotsord.h
1516 file path=usr/include/sys/tihdr.h
1517 file path=usr/include/sys/time.h
1518 file path=usr/include/sys/time_impl.h
1519 file path=usr/include/sys/time_std_impl.h
1520 file path=usr/include/sys/timeb.h
1521 file path=usr/include/sys/timer.h
1522 file path=usr/include/sys/times.h
1523 file path=usr/include/sys/timex.h
1524 file path=usr/include/sys/timod.h
1525 file path=usr/include/sys/tirdwr.h
1526 file path=usr/include/sys/tiuser.h
1527 file path=usr/include/sys/tl.h
1528 file path=usr/include/sys/tnf.h
1529 file path=usr/include/sys/tnf_com.h
1530 file path=usr/include/sys/tnf_probe.h
1531 file path=usr/include/sys/tnf_writer.h
1532 file path=usr/include/sys/todio.h
1533 file path=usr/include/sys/tpicommon.h
1534 file path=usr/include/sys/trap.h
1535 $(i386_ONLY)file path=usr/include/sys/traptrace.h
1536 file path=usr/include/sys/ts.h
1537 file path=usr/include/sys/tsol/label.h
1538 file path=usr/include/sys/tsol/label_macro.h
1539 file path=usr/include/sys/tsol/priv.h
1540 file path=usr/include/sys/tsol/tndb.h
1541 file path=usr/include/sys/tsol/tsyscall.h
1542 file path=usr/include/sys/tspriocntl.h
1543 $(i386_ONLY)file path=usr/include/sys/tss.h
1544 file path=usr/include/sys/ttcompat.h
1545 file path=usr/include/sys/ttold.h
1546 file path=usr/include/sys/tty.h
1547 file path=usr/include/sys/ttychars.h
1548 file path=usr/include/sys/ttydev.h
1549 $(sparc_ONLY)file path=usr/include/sys/ttymux.h
1550 $(sparc_ONLY)file path=usr/include/sys/ttymuxuser.h
1551 file path=usr/include/sys/tuneable.h
1552 file path=usr/include/sys/turnstile.h
1553 file path=usr/include/sys/types.h
1554 file path=usr/include/sys/types32.h
1555 file path=usr/include/sys/tzfile.h
1556 file path=usr/include/sys/u8_textprep.h
1557 file path=usr/include/sys/uadmin.h
1558 $(i386_ONLY)file path=usr/include/sys/ucode.h
1559 file path=usr/include/sys/ucontext.h
1560 file path=usr/include/sys/uio.h
1561 file path=usr/include/sys/ulimit.h
1562 file path=usr/include/sys/un.h
1563 file path=usr/include/sys/unistd.h
1564 file path=usr/include/sys/user.h
1565 file path=usr/include/sys/ustat.h
1566 file path=usr/include/sys/utime.h
1567 file path=usr/include/sys/utrap.h
1568 file path=usr/include/sys/utsname.h
1569 file path=usr/include/sys/utssys.h
1570 file path=usr/include/sys/uuid.h
1571 file path=usr/include/sys/va_impl.h
1572 file path=usr/include/sys/va_list.h
1573 file path=usr/include/sys/var.h
1574 file path=usr/include/sys/varargs.h
1575 file path=usr/include/sys/vfs.h
1576 file path=usr/include/sys/vfs_opreg.h
1577 file path=usr/include/sys/vfstab.h
1578 file path=usr/include/sys/videodev2.h
1579 file path=usr/include/sys/visual_io.h
```

```
1580 file path=usr/include/sys/vm.h
1581 file path=usr/include/sys/vm_usage.h
1582 file path=usr/include/sys/vmem.h
1583 file path=usr/include/sys/vmem_impl.h
1584 file path=usr/include/sys/vmem_impl_user.h
1585 file path=usr/include/sys/vmparam.h
1586 file path=usr/include/sys/vmsystm.h
1587 file path=usr/include/sys/vnode.h
1588 file path=usr/include/sys/vt.h
1589 file path=usr/include/sys/vtdaemon.h
1590 file path=usr/include/sys/vtoc.h
1591 file path=usr/include/sys/vtrace.h
1592 file path=usr/include/sys/vuid_event.h
1593 file path=usr/include/sys/vuid_queue.h
1594 file path=usr/include/sys/vuid_state.h
1595 file path=usr/include/sys/vuid_store.h
1596 file path=usr/include/sys/vuid_wheel.h
1597 file path=usr/include/sys/wait.h
1598 file path=usr/include/sys/waitq.h
1599 file path=usr/include/sys/watchpoint.h
1600 $(i386_ONLY)file path=usr/include/sys/x86_archext.h
1601 $(i386_ONLY)file path=usr/include/sys/xen_errno.h
1602 file path=usr/include/sys/xti_inet.h
1603 file path=usr/include/sys/xti_osi.h
1604 file path=usr/include/sys/xti_xtiopt.h
1605 file path=usr/include/sys/zcons.h
1606 file path=usr/include/sys/zmod.h
1607 file path=usr/include/sys/zone.h
1608 $(sparc_ONLY)file path=usr/include/sys/zsdev.h
1609 file path=usr/include/sysexits.h
1610 file path=usr/include/syslog.h
1611 file path=usr/include/tar.h
1612 file path=usr/include/tcpd.h
1613 file path=usr/include/term.h
1614 file path=usr/include/termcap.h
1615 file path=usr/include/termio.h
1616 file path=usr/include/termios.h
1617 file path=usr/include/thread.h
1618 file path=usr/include/thread_db.h
1619 file path=usr/include/time.h
1620 file path=usr/include/tiuser.h
1621 file path=usr/include/tsol/label.h
1622 file path=usr/include/tzfile.h
1623 file path=usr/include/ucontext.h
1624 file path=usr/include/ucred.h
1625 file path=usr/include/uid_stp.h
1626 file path=usr/include/ulimit.h
1627 file path=usr/include/umem.h
1628 file path=usr/include/umem_impl.h
1629 file path=usr/include/unctrl.h
1630 file path=usr/include/unistd.h
1631 file path=usr/include/user_attr.h
1632 file path=usr/include/userdefs.h
1633 file path=usr/include/ustat.h
1634 file path=usr/include/utility.h
1635 file path=usr/include/utime.h
1636 file path=usr/include/utmp.h
1637 file path=usr/include/utmpx.h
1638 file path=usr/include/uuid/uuid.h
1639 $(sparc_ONLY)file path=usr/include/v7/sys/machpcb.h
1640 $(sparc_ONLY)file path=usr/include/v7/sys/machtrap.h
1641 $(sparc_ONLY)file path=usr/include/v7/sys/mutex_impl.h
1642 $(sparc_ONLY)file path=usr/include/v7/sys/privregs.h
1643 $(sparc_ONLY)file path=usr/include/v7/sys/prom_isa.h
1644 $(sparc_ONLY)file path=usr/include/v7/sys/psr.h
1645 $(sparc_ONLY)file path=usr/include/v7/sys/traptrace.h
```



```
1646 $(sparc_ONLY)file path=usr/include/v9/sys/asi.h
1647 $(sparc_ONLY)file path=usr/include/v9/sys/machpcb.h
1648 $(sparc_ONLY)file path=usr/include/v9/sys/machtrap.h
1649 $(sparc_ONLY)file path=usr/include/v9/sys/membar.h
1650 $(sparc_ONLY)file path=usr/include/v9/sys/mutex_impl.h
1651 $(sparc_ONLY)file path=usr/include/v9/sys/privregs.h
1652 $(sparc_ONLY)file path=usr/include/v9/sys/prom_isa.h
1653 $(sparc_ONLY)file path=usr/include/v9/sys/psr_compat.h
1654 $(sparc_ONLY)file path=usr/include/v9/sys/vis_simulator.h
1655 file path=usr/include/valtools.h
1656 file path=usr/include/values.h
1657 file path=usr/include/varargs.h
1658 file path=usr/include/vm/anon.h
1659 file path=usr/include/vm/as.h
1660 file path=usr/include/vm/faultcode.h
1661 file path=usr/include/vm/hat.h
1662 file path=usr/include/vm/kpm.h
1663 file path=usr/include/vm/page.h
1664 file path=usr/include/vm/pvn.h
1665 file path=usr/include/vm/rm.h
1666 file path=usr/include/vm/seg.h
1667 file path=usr/include/vm/seg_dev.h
1668 file path=usr/include/vm/seg_enum.h
1669 file path=usr/include/vm/seg_kmem.h
1670 file path=usr/include/vm/seg_kp.h
1671 file path=usr/include/vm/seg_kpm.h
1672 file path=usr/include/vm/seg_map.h
1673 file path=usr/include/vm/seg_spt.h
1674 file path=usr/include/vm/seg_vn.h
1675 file path=usr/include/vm/vpage.h
1676 file path=usr/include/vm/vpm.h
1677 file path=usr/include/volmgt.h
1678 file path=usr/include/wait.h
1679 file path=usr/include/wchar.h
1680 file path=usr/include/wchar_impl.h
1681 file path=usr/include/wctype.h
1682 file path=usr/include/widec.h
1683 file path=usr/include/wordexp.h
1684 file path=usr/include/xlocale.h
1685 file path=usr/include/xti.h
1686 file path=usr/include/xti_inet.h
1687 file path=usr/include/zone.h
1688 file path=usr/include/zonestat.h
1689 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/acpidev.h
1690 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/amd_iommu.h
1691 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/asm_misc.h
1692 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/clock.h
1693 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/cram.h
1694 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/ddi_subrdefs.h
1695 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/debug_info.h
1696 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/fastboot.h
1697 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/mach_mmu.h
1698 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machclock.h
1699 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machcpuvar.h
1700 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machparam.h
1701 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machprivregs.h
1702 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machsystm.h
1703 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machthread.h
1704 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/memnode.h
1705 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/pc_mmu.h
1706 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm.h
1707 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_defs.h
1708 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_modctl.h
1709 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_types.h
1710 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/rm_platter.h
1711 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/sbd_ioctl.h
```

```
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/smp_impldefs.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/vm_machparam.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/x_call.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/xc_levels.h
$(i386_ONLY)file path=usr/platform/i86pc/include/sys/xsvc.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_i86.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_pte.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/hment.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/htable.h
$(i386_ONLY)file path=usr/platform/i86pc/include/vm/kboot_mmu.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/balloon.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/machprivregs.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xen_mmu.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xpv_impl.h
$(i386_ONLY)file path=usr/platform/i86xpv/include/vm/seg_mf.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ac.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/async.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cheetahregs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cherrystone.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/clock.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cmp.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpc_ultra.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpr_impl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_impl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_sgnblk_defs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cvc.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/daktari.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ddi_subrdefs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/dvma.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ecc_kstat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/eeprom.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_gen.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue250.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue450.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/environ.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/errclassify.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/fhc.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/gpio_87317.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/hpc3130_events.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/hpc3130.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/i2c_client.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/lm75.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/max1617.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/pcf8591.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/ssc050.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/misc/i2c_svc.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/idprom.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intr.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intreg.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iocache.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iommu.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ivintr.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/lom_io.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machasi.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machclock.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machcpuvar.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machparam.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machsystm.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machthread.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mem_cache.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memlist_plat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memnode.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mmu.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/nexusdebug.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_hwdesc.h
```



```
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_module.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_debug.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_plat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/pte.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sbd_ioctl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scb.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scsb_led.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/simmstat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/spitregs.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sram.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/starfire.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sun4asi.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysctrl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysioerr.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysiosbus.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/tod.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/todmostek.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/trapstat.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/traptrace.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vis.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vm_machparam.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/x_call.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/xc_impl.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/sys/zsmach.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/vm/hat_sfmmu.h
$(sparc_ONLY)file path=usr/platform/sun4u/include/vm/mach_sfmmu.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/clock.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cmp.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpc_ultra.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpu_sgnblk_defs.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ddi_subrdefs.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_pri.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_snmp.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/dvma.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/eeprom.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/fcode.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hsvc.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hypervisor_api.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/idprom.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intr.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intreg.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ivintr.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machasi.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machclock.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machcpuvar.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machintreg.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machparam.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machsystm.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machthread.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memlist_plat.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memnode.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/mmu.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/nexusdebug.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagaraasi.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagararegs.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ntwdt.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pri.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_debug.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_plat.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pte.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/qcn.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/scb.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/soft_state.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/sun4asi.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/tod.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/trapstat.h
```

```
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/traptrace.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vis.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vm_machparam.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/x_call.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/xc_impl.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/sys/zsmach.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/vm/hat_sfmmu.h
$(sparc_ONLY)file path=usr/platform/sun4v/include/vm/mach_sfmmu.h
file path=usr/share/man/man3head/acct.h.3head
file path=usr/share/man/man3head/aio.h.3head
file path=usr/share/man/man3head/ar.h.3head
file path=usr/share/man/man3head/archives.h.3head
file path=usr/share/man/man3head/assert.h.3head
file path=usr/share/man/man3head/complex.h.3head
file path=usr/share/man/man3head/cpio.h.3head
file path=usr/share/man/man3head/dirent.h.3head
file path=usr/share/man/man3head/errno.h.3head
file path=usr/share/man/man3head/fcntl.h.3head
file path=usr/share/man/man3head/fenv.h.3head
file path=usr/share/man/man3head/float.h.3head
file path=usr/share/man/man3head/floatingpoint.h.3head
file path=usr/share/man/man3head/fmtmsg.h.3head
file path=usr/share/man/man3head/fnmatch.h.3head
file path=usr/share/man/man3head/ftw.h.3head
file path=usr/share/man/man3head/glob.h.3head
file path=usr/share/man/man3head/grp.h.3head
file path=usr/share/man/man3head/iconv.h.3head
file path=usr/share/man/man3head/if.h.3head
file path=usr/share/man/man3head/in.h.3head
file path=usr/share/man/man3head/inet.h.3head
file path=usr/share/man/man3head/inttypes.h.3head
file path=usr/share/man/man3head/ipc.h.3head
file path=usr/share/man/man3head/iso646.h.3head
file path=usr/share/man/man3head/langinfo.h.3head
file path=usr/share/man/man3head/libgen.h.3head
file path=usr/share/man/man3head/libintl.h.3head
file path=usr/share/man/man3head/limits.h.3head
file path=usr/share/man/man3head/locale.h.3head
file path=usr/share/man/man3head/math.h.3head
file path=usr/share/man/man3head/mman.h.3head
file path=usr/share/man/man3head/monetary.h.3head
file path=usr/share/man/man3head/mqueue.h.3head
file path=usr/share/man/man3head/msg.h.3head
file path=usr/share/man/man3head/ndbm.h.3head
file path=usr/share/man/man3head/netdb.h.3head
file path=usr/share/man/man3head/nl_types.h.3head
file path=usr/share/man/man3head/poll.h.3head
file path=usr/share/man/man3head/pthread.h.3head
file path=usr/share/man/man3head/pwd.h.3head
file path=usr/share/man/man3head/regex.h.3head
file path=usr/share/man/man3head/resource.h.3head
file path=usr/share/man/man3head/sched.h.3head
file path=usr/share/man/man3head/search.h.3head
file path=usr/share/man/man3head/select.h.3head
file path=usr/share/man/man3head/sem.h.3head
file path=usr/share/man/man3head/semaphore.h.3head
file path=usr/share/man/man3head/setjmp.h.3head
file path=usr/share/man/man3head/shm.h.3head
file path=usr/share/man/man3head/siginfo.h.3head
file path=usr/share/man/man3head/signal.h.3head
file path=usr/share/man/man3head/socket.h.3head
file path=usr/share/man/man3head/spawn.h.3head
file path=usr/share/man/man3head/stat.h.3head
file path=usr/share/man/man3head/statvfs.h.3head
file path=usr/share/man/man3head/stdbool.h.3head
file path=usr/share/man/man3head/stddef.h.3head
```



```
file path=usr/share/man/man3head/stdint.h.3head
file path=usr/share/man/man3head/stdio.h.3head
file path=usr/share/man/man3head/stdlib.h.3head
file path=usr/share/man/man3head/string.h.3head
file path=usr/share/man/man3head/strings.h.3head
file path=usr/share/man/man3head/stropts.h.3head
file path=usr/share/man/man3head/syslog.h.3head
file path=usr/share/man/man3head/tar.h.3head
file path=usr/share/man/man3head/tcp.h.3head
file path=usr/share/man/man3head/termios.h.3head
file path=usr/share/man/man3head/tgmath.h.3head
file path=usr/share/man/man3head/time.h.3head
file path=usr/share/man/man3head/timeb.h.3head
file path=usr/share/man/man3head/times.h.3head
file path=usr/share/man/man3head/types.h.3head
file path=usr/share/man/man3head/types32.h.3head
file path=usr/share/man/man3head/ucontext.h.3head
file path=usr/share/man/man3head/uio.h.3head
file path=usr/share/man/man3head/ulimit.h.3head
file path=usr/share/man/man3head/un.h.3head
file path=usr/share/man/man3head/unistd.h.3head
file path=usr/share/man/man3head/utime.h.3head
file path=usr/share/man/man3head/utmpx.h.3head
file path=usr/share/man/man3head/utsname.h.3head
file path=usr/share/man/man3head/values.h.3head
file path=usr/share/man/man3head/wait.h.3head
file path=usr/share/man/man3head/wchar.h.3head
file path=usr/share/man/man3head/wctype.h.3head
file path=usr/share/man/man3head/wordexp.h.3head
file path=usr/share/man/man3head/xlocale.h.3head
file path=usr/share/man/man4/note.4
file path=usr/share/man/man5/prof.5
file path=usr/share/man/man7i/cdio.7i
file path=usr/share/man/man7i/dkio.7i
file path=usr/share/man/man7i/fbio.7i
file path=usr/share/man/man7i/fdio.7i
file path=usr/share/man/man7i/hdio.7i
file path=usr/share/man/man7i/iec61883.7i
file path=usr/share/man/man7i/mhd.7i
file path=usr/share/man/man7i/mtio.7i
file path=usr/share/man/man7i/prnio.7i
file path=usr/share/man/man7i/quotactl.7i
file path=usr/share/man/man7i/sesio.7i
file path=usr/share/man/man7i/sockio.7i
file path=usr/share/man/man7i/streamio.7i
file path=usr/share/man/man7i/termio.7i
file path=usr/share/man/man7i/termiox.7i
file path=usr/share/man/man7i/uscsi.7i
file path=usr/share/man/man7i/visual_io.7i
file path=usr/share/man/man7i/vt.7i
file path=usr/xpg4/include/curses.h
file path=usr/xpg4/include/term.h
file path=usr/xpg4/include/unctrl.h
legacy pkg=SUNWhea \
    desc="SunOS C/C++ header files for general development of software" \
    name="SunOS Header Files"
license cr_Sun license=cr_Sun
license lic_CDDL license=lic_CDDL
license license_in_headers license=license_in_headers
license usr/src/lib/pkcs11/include/THIRDPARTYLICENSE \
    license=usr/src/lib/pkcs11/include/THIRDPARTYLICENSE
link path=usr/include/iso/assert_iso.h target=../assert.h
link path=usr/include/iso/errno_iso.h target=../errno.h
link path=usr/include/iso/float_iso.h target=../float.h
link path=usr/include/iso/iso646_iso.h target=../iso646.h
$(sparc_ONLY)link path=usr/platform/SUNW,A70/include target=../sun4u/include
```

```
$(sparc_ONLY)link path=usr/platform/SUNW,Netra-T12/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Netra-T4/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,SPARC-Enterprise/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Serverblade1/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-100/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1000/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1500/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-2500/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-15000/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-280R/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-480R/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-880/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V215/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V240/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V250/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V440/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V445/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V490/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V890/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Ultra-2/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Ultra-250/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Ultra-4/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise-10000/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60/include \
    target=../sun4u/include
$(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIi-Netract/include \
    target=../sun4u/include
link path=usr/share/man/man3head/acct.3head target=acct.h.3head
link path=usr/share/man/man3head/aio.3head target=aio.h.3head
link path=usr/share/man/man3head/ar.3head target=ar.h.3head
link path=usr/share/man/man3head/archives.3head target=archives.h.3head
link path=usr/share/man/man3head/assert.3head target=assert.h.3head
link path=usr/share/man/man3head/complex.3head target=complex.h.3head
link path=usr/share/man/man3head/cpio.3head target=cpio.h.3head
link path=usr/share/man/man3head/dirent.3head target=dirent.h.3head
link path=usr/share/man/man3head/errno.3head target=errno.h.3head
link path=usr/share/man/man3head/fcntl.3head target=fcntl.h.3head
```



```
link path=usr/share/man/man3head/fenv.3head target=fenv.h.3head
link path=usr/share/man/man3head/float.3head target=float.h.3head
link path=usr/share/man/man3head/floatingpoint.3head \
    target=floatingpoint.h.3head
link path=usr/share/man/man3head/fmtmsg.3head target=fmtmsg.h.3head
link path=usr/share/man/man3head/fnmatch.3head target=fnmatch.h.3head
link path=usr/share/man/man3head/ftw.3head target=ftw.h.3head
link path=usr/share/man/man3head/glob.3head target=glob.h.3head
link path=usr/share/man/man3head/grp.3head target=grp.h.3head
link path=usr/share/man/man3head/iconv.3head target=iconv.h.3head
link path=usr/share/man/man3head/if.3head target=if.h.3head
link path=usr/share/man/man3head/in.3head target=in.h.3head
link path=usr/share/man/man3head/inet.3head target=inet.h.3head
link path=usr/share/man/man3head/inttypes.3head target=inttypes.h.3head
link path=usr/share/man/man3head/ipc.3head target=ipc.h.3head
link path=usr/share/man/man3head/iso646.3head target=iso646.h.3head
link path=usr/share/man/man3head/langinfo.3head target=langinfo.h.3head
link path=usr/share/man/man3head/libgen.3head target=libgen.h.3head
link path=usr/share/man/man3head/libintl.3head target=libintl.h.3head
link path=usr/share/man/man3head/limits.3head target=limits.h.3head
link path=usr/share/man/man3head/locale.3head target=locale.h.3head
link path=usr/share/man/man3head/math.3head target=math.h.3head
link path=usr/share/man/man3head/mman.3head target=mman.h.3head
link path=usr/share/man/man3head/monetary.3head target=monetary.h.3head
link path=usr/share/man/man3head/mqueue.3head target=mqueue.h.3head
link path=usr/share/man/man3head/msg.3head target=msg.h.3head
link path=usr/share/man/man3head/ndbm.3head target=ndbm.h.3head
link path=usr/share/man/man3head/netdb.3head target=netdb.h.3head
link path=usr/share/man/man3head/nl_types.3head target=nl_types.h.3head
link path=usr/share/man/man3head/poll.3head target=poll.h.3head
link path=usr/share/man/man3head/pthread.3head target=pthread.h.3head
link path=usr/share/man/man3head/pwd.3head target=pwd.h.3head
link path=usr/share/man/man3head/regex.3head target=regex.h.3head
link path=usr/share/man/man3head/resource.3head target=resource.h.3head
link path=usr/share/man/man3head/sched.3head target=sched.h.3head
link path=usr/share/man/man3head/search.3head target=search.h.3head
link path=usr/share/man/man3head/select.3head target=select.h.3head
link path=usr/share/man/man3head/sem.3head target=sem.h.3head
link path=usr/share/man/man3head/semaphore.3head target=semaphore.h.3head
link path=usr/share/man/man3head/setjmp.3head target=setjmp.h.3head
link path=usr/share/man/man3head/shm.3head target=shm.h.3head
link path=usr/share/man/man3head/siginfo.3head target=siginfo.h.3head
link path=usr/share/man/man3head/signal.3head target=signal.h.3head
link path=usr/share/man/man3head/socket.3head target=socket.h.3head
link path=usr/share/man/man3head/spawn.3head target=spawn.h.3head
link path=usr/share/man/man3head/stat.3head target=stat.h.3head
link path=usr/share/man/man3head/statvfs.3head target=statvfs.h.3head
link path=usr/share/man/man3head/stdbool.3head target=stdbool.h.3head
link path=usr/share/man/man3head/stddef.3head target=stddef.h.3head
link path=usr/share/man/man3head/stdint.3head target=stdint.h.3head
link path=usr/share/man/man3head/stdio.3head target=stdio.h.3head
link path=usr/share/man/man3head/stdlib.3head target=stdlib.h.3head
link path=usr/share/man/man3head/string.3head target=string.h.3head
link path=usr/share/man/man3head/strings.3head target=strings.h.3head
link path=usr/share/man/man3head/stropts.3head target=stropts.h.3head
link path=usr/share/man/man3head/syslog.3head target=syslog.h.3head
link path=usr/share/man/man3head/tar.3head target=tar.h.3head
link path=usr/share/man/man3head/tcp.3head target=tcp.h.3head
link path=usr/share/man/man3head/termios.3head target=termios.h.3head
link path=usr/share/man/man3head/tgmath.3head target=tgmath.h.3head
link path=usr/share/man/man3head/time.3head target=time.h.3head
link path=usr/share/man/man3head/timeb.3head target=timeb.h.3head
link path=usr/share/man/man3head/times.3head target=times.h.3head
link path=usr/share/man/man3head/types.3head target=types.h.3head
link path=usr/share/man/man3head/types32.3head target=types32.h.3head
link path=usr/share/man/man3head/ucontext.3head target=ucontext.h.3head
```

```
2108 link path=usr/share/man/man3head/uio.3head target=uio.h.3head
2109 link path=usr/share/man/man3head/ulimit.3head target=ulimit.h.3head
2110 link path=usr/share/man/man3head/un.3head target=un.h.3head
2111 link path=usr/share/man/man3head/unistd.3head target=unistd.h.3head
2112 link path=usr/share/man/man3head/utime.3head target=utime.h.3head
2113 link path=usr/share/man/man3head/utmpx.3head target=utmpx.h.3head
2114 link path=usr/share/man/man3head/utsname.3head target=utsname.h.3head
2115 link path=usr/share/man/man3head/values.3head target=values.h.3head
2116 link path=usr/share/man/man3head/wait.3head target=wait.h.3head
2117 link path=usr/share/man/man3head/wchar.3head target=wchar.h.3head
2118 link path=usr/share/man/man3head/wctype.3head target=wctype.h.3head
2119 link path=usr/share/man/man3head/wordexp.3head target=wordexp.h.3head
2120 link path=usr/share/man/man3head/xlocale.3head target=xlocale.h.3head
2121 $(i386_ONLY)link path=usr/share/src/uts/i86pc/sys \
2122     target=../../../../platform/i86pc/include/sys
2123 $(i386_ONLY)link path=usr/share/src/uts/i86pc/vm \
2124     target=../../../../platform/i86pc/include/vm
2125 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/sys \
2126     target=../../../../platform/i86xpv/include/sys
2127 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/vm \
2128     target=../../../../platform/i86xpv/include/vm
2129 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/sys \
2130     target=../../../../platform/sun4u/include/sys
2131 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/vm \
2132     target=../../../../platform/sun4u/include/vm
2133 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/sys \
2134     target=../../../../platform/sun4v/include/sys
2135 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/vm \
2136     target=../../../../platform/sun4v/include/vm
```

```
**********************************************************
   87235 Mon Aug 17 21:08:03 2015
new/usr/src/uts/common/fs/doorfs/door_sys.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

1768 /*
1769  * Create a descriptor for the associated file and fill in the
1770  * attributes associated with it.
1771  *
1772  * Return 0 for success, -1 otherwise;
1773  */
1774 int
1775 door_insert(struct file *fp, door_desc_t *dp)
1776 {
1777         struct vnode *vp;
1778         int     fd;
1779         door_attr_t attributes = DOOR_DESCRIPTOR;

1781         ASSERT(MUTEX_NOT_HELD(&door_knob));
1782         if ((fd = ufalloc(0)) == -1)
1783                 return (-1);
1784         setf(fd, fp);
1785         dp->d_data.d_desc.d_descriptor = fd;

1787         /* Add pid to the list associated with that descriptor. */
1788         if (fp->f_vnode != NULL)
1789                 (void) VOP_IOCTL(fp->f_vnode, F_ASSOCI_PID,
1790                     (intptr_t)curproc->p_pidp->pid_id, FKIOCTL, kcred, NULL,
1791                     NULL);

1793 #endif /* ! codereview */
1794         /* Fill in the attributes */
1795         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
1796                 vp = fp->f_vnode;
1797         if (vp && vp->v_type == VDOOR) {
1798                 if (VTOD(vp)->door_target == curproc)
1799                         attributes |= DOOR_LOCAL;
1800                 attributes |= VTOD(vp)->door_flags & DOOR_ATTR_MASK;
1801                 dp->d_data.d_desc.d_id = VTOD(vp)->door_index;
1802         }
1803         dp->d_attributes = attributes;
1804         return (0);
1805 }

1807 /*
1808  * Return an available thread for this server.  A NULL return value indicates
1809  * that either:
1810  *      The door has been revoked, or
1811  *      a signal was received.
1812  * The two conditions can be differentiated using DOOR_INVALID(dp).
1813  */
1814 static kthread_t *
1815 door_get_server(door_node_t *dp)
1816 {
1817         kthread_t **ktp;
1818         kthread_t *server_t;
1819         door_pool_t *pool;
1820         door_server_t *st;
1821         int signalled;

1823         disp_lock_t *tlp;
1824         cpu_t *cp;

1826         ASSERT(MUTEX_HELD(&door_knob));
```



```
1828         if (dp->door_flags & DOOR_PRIVATE)
1829                 pool = &dp->door_servers;
1830         else
1831                 pool = &dp->door_target->p_server_threads;

1833         for (;;) {
1834                 /*
1835                  * We search the thread pool, looking for a server thread
1836                  * ready to take an invocation (i.e. one which is still
1837                  * sleeping on a shuttle object).  If none are available,
1838                  * we sleep on the pool's CV, and will be signaled when a
1839                  * thread is added to the pool.
1840                  *
1841                  * This relies on the fact that once a thread in the thread
1842                  * pool wakes up, it *must* remove and add itself to the pool
1843                  * before it can receive door calls.
1844                  */
1845                 if (DOOR_INVALID(dp))
1846                         return (NULL);  /* Target has become invalid */

1848                 for (ktp = &pool->dp_threads;
1849                     (server_t = *ktp) != NULL;
1850                     ktp = &st->d_servers) {
1851                         st = DOOR_SERVER(server_t->t_door);

1853                         thread_lock(server_t);
1854                         if (server_t->t_state == TS_SLEEP &&
1855                             SOBJ_TYPE(server_t->t_sobj_ops) == SOBJ_SHUTTLE)
1856                                 break;
1857                         thread_unlock(server_t);
1858                 }
1859                 if (server_t != NULL)
1860                         break;          /* we've got a live one! */

1862                 if (!cv_wait_sig_swap_core(&pool->dp_cv, &door_knob,
1863                     &signalled)) {
1864                         /*
1865                          * If we were signaled and the door is still
1866                          * valid, pass the signal on to another waiter.
1867                          */
1868                         if (signalled && !DOOR_INVALID(dp))
1869                                 cv_signal(&pool->dp_cv);
1870                         return (NULL);  /* Got a signal */
1871                 }
1872         }

1874         /*
1875          * We've got a thread_lock()ed thread which is still on the
1876          * shuttle.  Take it off the list of available server threads
1877          * and mark it as ONPROC.  We are committed to resuming this
1878          * thread now.
1879          */
1880         tlp = server_t->t_lockp;
1881         cp = CPU;

1883         *ktp = st->d_servers;
1884         st->d_servers = NULL;
1885         /*
1886          * Setting t_disp_queue prevents erroneous preemptions
1887          * if this thread is still in execution on another processor
1888          */
1889         server_t->t_disp_queue = cp->cpu_disp;
1890         CL_ACTIVE(server_t);
1891         /*
1892          * We are calling thread_onproc() instead of
```

```
1893          * THREAD_ONPROC() because compiler can reorder
1894          * the two stores of t_state and t_lockp in
1895          * THREAD_ONPROC().
1896          */
1897         thread_onproc(server_t, cp);
1898         disp_lock_exit(tlp);
1899         return (server_t);
1900 }

1902 /*
1903  * Put a server thread back in the pool.
1904  */
1905 static void
1906 door_release_server(door_node_t *dp, kthread_t *t)
1907 {
1908         door_server_t *st = DOOR_SERVER(t->t_door);
1909         door_pool_t *pool;

1911         ASSERT(MUTEX_HELD(&door_knob));
1912         st->d_active = NULL;
1913         st->d_caller = NULL;
1914         st->d_layout_done = 0;
1915         if (dp && (dp->door_flags & DOOR_PRIVATE)) {
1916                 ASSERT(dp->door_target == NULL ||
1917                     dp->door_target == ttoproc(t));
1918                 pool = &dp->door_servers;
1919         } else {
1920                 pool = &ttoproc(t)->p_server_threads;
1921         }

1923         st->d_servers = pool->dp_threads;
1924         pool->dp_threads = t;

1926         /* If someone is waiting for a server thread, wake him up */
1927         cv_signal(&pool->dp_cv);
1928 }

1930 /*
1931  * Remove a server thread from the pool if present.
1932  */
1933 static void
1934 door_server_exit(proc_t *p, kthread_t *t)
1935 {
1936         door_pool_t *pool;
1937         kthread_t **next;
1938         door_server_t *st = DOOR_SERVER(t->t_door);

1940         ASSERT(MUTEX_HELD(&door_knob));
1941         if (st->d_pool != NULL) {
1942                 ASSERT(st->d_pool->door_flags & DOOR_PRIVATE);
1943                 pool = &st->d_pool->door_servers;
1944         } else {
1945                 pool = &p->p_server_threads;
1946         }

1948         next = &pool->dp_threads;
1949         while (*next != NULL) {
1950                 if (*next == t) {
1951                         *next = DOOR_SERVER(t->t_door)->d_servers;
1952                         return;
1953                 }
1954                 next = &(DOOR_SERVER((*next)->t_door)->d_servers);
1955         }
1956 }

1958 /*
```



```
1959  * Lookup the door descriptor. Caller must call releasef when finished
1960  * with associated door.
1961  */
1962 static door_node_t *
1963 door_lookup(int did, file_t **fpp)
1964 {
1965         vnode_t *vp;
1966         file_t *fp;

1968         ASSERT(MUTEX_NOT_HELD(&door_knob));
1969         if ((fp = getf(did)) == NULL)
1970                 return (NULL);
1971         /*
1972          * Use the underlying vnode (we may be namefs mounted)
1973          */
1974         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
1975                 vp = fp->f_vnode;

1977         if (vp == NULL || vp->v_type != VDOOR) {
1978                 releasef(did);
1979                 return (NULL);
1980         }

1982         if (fpp)
1983                 *fpp = fp;

1985         return (VTOD(vp));
1986 }

1988 /*
1989  * The current thread is exiting, so clean up any pending
1990  * invocation details
1991  */
1992 void
1993 door_slam(void)
1994 {
1995         door_node_t *dp;
1996         door_data_t *dt;
1997         door_client_t *ct;
1998         door_server_t *st;

2000         /*
2001          * If we are an active door server, notify our
2002          * client that we are exiting and revoke our door.
2003          */
2004         if ((dt = door_my_data(0)) == NULL)
2005                 return;
2006         ct = DOOR_CLIENT(dt);
2007         st = DOOR_SERVER(dt);

2009         mutex_enter(&door_knob);
2010         for (;;) {
2011                 if (DOOR_T_HELD(ct))
2012                         cv_wait(&ct->d_cv, &door_knob);
2013                 else if (DOOR_T_HELD(st))
2014                         cv_wait(&st->d_cv, &door_knob);
2015                 else
2016                         break;                  /* neither flag is set */
2017         }
2018         curthread->t_door = NULL;
2019         if ((dp = st->d_active) != NULL) {
2020                 kthread_t *t = st->d_caller;
2021                 proc_t *p = curproc;

2023                 /* Revoke our door if the process is exiting */
2024                 if (dp->door_target == p && (p->p_flag & SEXITING)) {
```

```
2025                        door_list_delete(dp);
2026                        dp->door_target = NULL;
2027                        dp->door_flags |= DOOR_REVOKED;
2028                        if (dp->door_flags & DOOR_PRIVATE)
2029                                cv_broadcast(&dp->door_servers.dp_cv);
2030                        else
2031                                cv_broadcast(&p->p_server_threads.dp_cv);
2032                }

2034                if (t != NULL) {
2035                        /*
2036                         * Let the caller know we are gone
2037                         */
2038                        DOOR_CLIENT(t->t_door)->d_error = DOOR_EXIT;
2039                        thread_lock(t);
2040                        if (t->t_state == TS_SLEEP &&
2041                            SOBJ_TYPE(t->t_sobj_ops) == SOBJ_SHUTTLE)
2042                                setrun_locked(t);
2043                        thread_unlock(t);
2044                }
2045        }
2046        mutex_exit(&door_knob);
2047        if (st->d_pool)
2048                door_unbind_thread(st->d_pool); /* Implicit door_unbind */
2049        kmem_free(dt, sizeof (door_data_t));
2050 }

2052 /*
2053  * Set DOOR_REVOKED for all doors of the current process. This is called
2054  * on exit before all lwp's are being terminated so that door calls will
2055  * return with an error.
2056  */
2057 void
2058 door_revoke_all()
2059 {
2060        door_node_t *dp;
2061        proc_t *p = ttoproc(curthread);

2063        mutex_enter(&door_knob);
2064        for (dp = p->p_door_list; dp != NULL; dp = dp->door_list) {
2065                ASSERT(dp->door_target == p);
2066                dp->door_flags |= DOOR_REVOKED;
2067                if (dp->door_flags & DOOR_PRIVATE)
2068                        cv_broadcast(&dp->door_servers.dp_cv);
2069        }
2070        cv_broadcast(&p->p_server_threads.dp_cv);
2071        mutex_exit(&door_knob);
2072 }

2074 /*
2075  * The process is exiting, and all doors it created need to be revoked.
2076  */
2077 void
2078 door_exit(void)
2079 {
2080        door_node_t *dp;
2081        proc_t *p = ttoproc(curthread);

2083        ASSERT(p->p_lwpcnt == 1);
2084        /*
2085         * Walk the list of active doors created by this process and
2086         * revoke them all.
2087         */
2088        mutex_enter(&door_knob);
2089        for (dp = p->p_door_list; dp != NULL; dp = dp->door_list) {
2090                dp->door_target = NULL;
```



```
2091                dp->door_flags |= DOOR_REVOKED;
2092                if (dp->door_flags & DOOR_PRIVATE)
2093                        cv_broadcast(&dp->door_servers.dp_cv);
2094        }
2095        cv_broadcast(&p->p_server_threads.dp_cv);
2096        /* Clear the list */
2097        p->p_door_list = NULL;

2099        /* Clean up the unref list */
2100        while ((dp = p->p_unref_list) != NULL) {
2101                p->p_unref_list = dp->door_ulist;
2102                dp->door_ulist = NULL;
2103                mutex_exit(&door_knob);
2104                VN_RELE(DTOV(dp));
2105                mutex_enter(&door_knob);
2106        }
2107        mutex_exit(&door_knob);
2108 }


2111 /*
2112  * The process is executing forkall(), and we need to flag threads that
2113  * are bound to a door in the child.  This will make the child threads
2114  * return an error to door_return unless they call door_unbind first.
2115  */
2116 void
2117 door_fork(kthread_t *parent, kthread_t *child)
2118 {
2119        door_data_t *pt = parent->t_door;
2120        door_server_t *st = DOOR_SERVER(pt);
2121        door_data_t *dt;

2123        ASSERT(MUTEX_NOT_HELD(&door_knob));
2124        if (pt != NULL && (st->d_pool != NULL || st->d_invbound)) {
2125                /* parent thread is bound to a door */
2126                dt = child->t_door =
2127                    kmem_zalloc(sizeof (door_data_t), KM_SLEEP);
2128                DOOR_SERVER(dt)->d_invbound = 1;
2129        }
2130 }

2132 /*
2133  * Deliver queued unrefs to appropriate door server.
2134  */
2135 static int
2136 door_unref(void)
2137 {
2138        door_node_t     *dp;
2139        static door_arg_t unref_args = { DOOR_UNREF_DATA, 0, 0, 0, 0, 0 };
2140        proc_t *p = ttoproc(curthread);

2142        /* make sure there's only one unref thread per process */
2143        mutex_enter(&door_knob);
2144        if (p->p_unref_thread) {
2145                mutex_exit(&door_knob);
2146                return (set_errno(EALREADY));
2147        }
2148        p->p_unref_thread = 1;
2149        mutex_exit(&door_knob);

2151        (void) door_my_data(1);                 /* create info, if necessary */

2153        for (;;) {
2154                mutex_enter(&door_knob);

2156                /* Grab a queued request */
```

```
2157                 while ((dp = p->p_unref_list) == NULL) {
2158                         if (!cv_wait_sig(&p->p_unref_cv, &door_knob)) {
2159                                 /*
2160                                  * Interrupted.
2161                                  * Return so we can finish forkall() or exit().
2162                                  */
2163                                 p->p_unref_thread = 0;
2164                                 mutex_exit(&door_knob);
2165                                 return (set_errno(EINTR));
2166                         }
2167                 }
2168                 p->p_unref_list = dp->door_ulist;
2169                 dp->door_ulist = NULL;
2170                 dp->door_flags |= DOOR_UNREF_ACTIVE;
2171                 mutex_exit(&door_knob);

2173                 (void) door_upcall(DTOV(dp), &unref_args, NULL, SIZE_MAX, 0);

2175                 if (unref_args.rbuf != 0) {
2176                         kmem_free(unref_args.rbuf, unref_args.rsize);
2177                         unref_args.rbuf = NULL;
2178                         unref_args.rsize = 0;
2179                 }

2181                 mutex_enter(&door_knob);
2182                 ASSERT(dp->door_flags & DOOR_UNREF_ACTIVE);
2183                 dp->door_flags &= ~DOOR_UNREF_ACTIVE;
2184                 mutex_exit(&door_knob);
2185                 VN_RELE(DTOV(dp));
2186         }
2187 }


2190 /*
2191  * Deliver queued unrefs to kernel door server.
2192  */
2193 /* ARGSUSED */
2194 static void
2195 door_unref_kernel(caddr_t arg)
2196 {
2197         door_node_t     *dp;
2198         static door_arg_t unref_args = { DOOR_UNREF_DATA, 0, 0, 0, 0, 0 };
2199         proc_t *p = ttoproc(curthread);
2200         callb_cpr_t cprinfo;

2202         /* should only be one of these */
2203         mutex_enter(&door_knob);
2204         if (p->p_unref_thread) {
2205                 mutex_exit(&door_knob);
2206                 return;
2207         }
2208         p->p_unref_thread = 1;
2209         mutex_exit(&door_knob);

2211         (void) door_my_data(1);         /* make sure we have a door_data_t */

2213         CALLB_CPR_INIT(&cprinfo, &door_knob, callb_generic_cpr, "door_unref");
2214         for (;;) {
2215                 mutex_enter(&door_knob);
2216                 /* Grab a queued request */
2217                 while ((dp = p->p_unref_list) == NULL) {
2218                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
2219                         cv_wait(&p->p_unref_cv, &door_knob);
2220                         CALLB_CPR_SAFE_END(&cprinfo, &door_knob);
2221                 }
2222                 p->p_unref_list = dp->door_ulist;
```



```
2223                 dp->door_ulist = NULL;
2224                 dp->door_flags |= DOOR_UNREF_ACTIVE;
2225                 mutex_exit(&door_knob);

2227                 (*(dp->door_pc))(dp->door_data, &unref_args, NULL, NULL, NULL);

2229                 mutex_enter(&door_knob);
2230                 ASSERT(dp->door_flags & DOOR_UNREF_ACTIVE);
2231                 dp->door_flags &= ~DOOR_UNREF_ACTIVE;
2232                 mutex_exit(&door_knob);
2233                 VN_RELE(DTOV(dp));
2234         }
2235 }


2238 /*
2239  * Queue an unref invocation for processing for the current process
2240  * The door may or may not be revoked at this point.
2241  */
2242 void
2243 door_deliver_unref(door_node_t *d)
2244 {
2245         struct proc *server = d->door_target;

2247         ASSERT(MUTEX_HELD(&door_knob));
2248         ASSERT(d->door_active == 0);

2250         if (server == NULL)
2251                 return;
2252         /*
2253          * Create a lwp to deliver unref calls if one isn't already running.
2254          *
2255          * A separate thread is used to deliver unrefs since the current
2256          * thread may be holding resources (e.g. locks) in user land that
2257          * may be needed by the unref processing. This would cause a
2258          * deadlock.
2259          */
2260         if (d->door_flags & DOOR_UNREF_MULTI) {
2261                 /* multiple unrefs */
2262                 d->door_flags &= ~DOOR_DELAY;
2263         } else {
2264                 /* Only 1 unref per door */
2265                 d->door_flags &= ~(DOOR_UNREF|DOOR_DELAY);
2266         }
2267         mutex_exit(&door_knob);

2269         /*
2270          * Need to bump the vnode count before putting the door on the
2271          * list so it doesn't get prematurely released by door_unref.
2272          */
2273         VN_HOLD(DTOV(d));

2275         mutex_enter(&door_knob);
2276         /* is this door already on the unref list? */
2277         if (d->door_flags & DOOR_UNREF_MULTI) {
2278                 door_node_t *dp;
2279                 for (dp = server->p_unref_list; dp != NULL;
2280                     dp = dp->door_ulist) {
2281                         if (d == dp) {
2282                                 /* already there, don't need to add another */
2283                                 mutex_exit(&door_knob);
2284                                 VN_RELE(DTOV(d));
2285                                 mutex_enter(&door_knob);
2286                                 return;
2287                         }
2288                 }
```

```
2289         }
2290         ASSERT(d->door_ulist == NULL);
2291         d->door_ulist = server->p_unref_list;
2292         server->p_unref_list = d;
2293         cv_broadcast(&server->p_unref_cv);
2294 }

2296 /*
2297  * The callers buffer isn't big enough for all of the data/fd's. Allocate
2298  * space in the callers address space for the results and copy the data
2299  * there.
2300  *
2301  * For EOVERFLOW, we must clean up the server's door descriptors.
2302  */
2303 static int
2304 door_overflow(
2305         kthread_t       *caller,
2306         caddr_t         data_ptr,       /* data location */
2307         size_t          data_size,      /* data size */
2308         door_desc_t     *desc_ptr,      /* descriptor location */
2309         uint_t          desc_num)       /* descriptor size */
2310 {
2311         proc_t *callerp = ttoproc(caller);
2312         struct as *as = callerp->p_as;
2313         door_client_t *ct = DOOR_CLIENT(caller->t_door);
2314         caddr_t addr;                   /* Resulting address in target */
2315         size_t  rlen;                   /* Rounded len */
2316         size_t  len;
2317         uint_t  i;
2318         size_t  ds = desc_num * sizeof (door_desc_t);

2320         ASSERT(MUTEX_NOT_HELD(&door_knob));
2321         ASSERT(DOOR_T_HELD(ct) || ct->d_kernel);

2323         /* Do initial overflow check */
2324         if (!ufcanalloc(callerp, desc_num))
2325                 return (EMFILE);

2327         /*
2328          * Allocate space for this stuff in the callers address space
2329          */
2330         rlen = roundup(data_size + ds, PAGESIZE);
2331         as_rangelock(as);
2332         map_addr_proc(&addr, rlen, 0, 1, as->a_userlimit, ttoproc(caller), 0);
2333         if (addr == NULL ||
2334             as_map(as, addr, rlen, segvn_create, zfod_argsp) != 0) {
2335                 /* No virtual memory available, or anon mapping failed */
2336                 as_rangeunlock(as);
2337                 if (!ct->d_kernel && desc_num > 0) {
2338                         int error = door_release_fds(desc_ptr, desc_num);
2339                         if (error)
2340                                 return (error);
2341                 }
2342                 return (EOVERFLOW);
2343         }
2344         as_rangeunlock(as);

2346         if (ct->d_kernel)
2347                 goto out;

2349         if (data_size != 0) {
2350                 caddr_t src = data_ptr;
2351                 caddr_t saddr = addr;

2353                 /* Copy any data */
2354                 len = data_size;
```



```
2355                 while (len != 0) {
2356                         int     amount;
2357                         int     error;

2359                         amount = len > PAGESIZE ? PAGESIZE : len;
2360                         if ((error = door_copy(as, src, saddr, amount)) != 0) {
2361                                 (void) as_unmap(as, addr, rlen);
2362                                 return (error);
2363                         }
2364                         saddr += amount;
2365                         src += amount;
2366                         len -= amount;
2367                 }
2368         }
2369         /* Copy any fd's */
2370         if (desc_num != 0) {
2371                 door_desc_t     *didpp, *start;
2372                 struct file     **fpp;
2373                 int             fpp_size;

2375                 start = didpp = kmem_alloc(ds, KM_SLEEP);
2376                 if (copyin_nowatch(desc_ptr, didpp, ds)) {
2377                         kmem_free(start, ds);
2378                         (void) as_unmap(as, addr, rlen);
2379                         return (EFAULT);
2380                 }

2382                 fpp_size = desc_num * sizeof (struct file *);
2383                 if (fpp_size > ct->d_fpp_size) {
2384                         /* make more space */
2385                         if (ct->d_fpp_size)
2386                                 kmem_free(ct->d_fpp, ct->d_fpp_size);
2387                         ct->d_fpp_size = fpp_size;
2388                         ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2389                 }
2390                 fpp = ct->d_fpp;

2392                 for (i = 0; i < desc_num; i++) {
2393                         struct file *fp;
2394                         int fd = didpp->d_data.d_desc.d_descriptor;

2396                         if (!(didpp->d_attributes & DOOR_DESCRIPTOR) ||
2397                             (fp = getf(fd)) == NULL) {
2398                                 /* close translated references */
2399                                 door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2400                                 /* close untranslated references */
2401                                 door_fd_rele(didpp, desc_num - i, 0);
2402                                 kmem_free(start, ds);
2403                                 (void) as_unmap(as, addr, rlen);
2404                                 return (EINVAL);
2405                         }
2406                         mutex_enter(&fp->f_tlock);
2407                         fp->f_count++;
2408                         mutex_exit(&fp->f_tlock);

2410                         *fpp = fp;
2411                         releasef(fd);

2413                         if (didpp->d_attributes & DOOR_RELEASE) {
2414                                 /* release passed reference */
2415                                 (void) closeandsetf(fd, NULL);
2416                         }

2418                         fpp++; didpp++;
2419                 }
2420                 kmem_free(start, ds);
```

```
2421         }

2423 out:
2424         ct->d_overflow = 1;
2425         ct->d_args.rbuf = addr;
2426         ct->d_args.rsize = rlen;
2427         return (0);
2428 }

2430 /*
2431  * Transfer arguments from the client to the server.
2432  */
2433 static int
2434 door_args(kthread_t *server, int is_private)
2435 {
2436         door_server_t *st = DOOR_SERVER(server->t_door);
2437         door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2438         uint_t  ndid;
2439         size_t  dsize;
2440         int     error;

2442         ASSERT(DOOR_T_HELD(st));
2443         ASSERT(MUTEX_NOT_HELD(&door_knob));

2445         ndid = ct->d_args.desc_num;
2446         if (ndid > door_max_desc)
2447                 return (E2BIG);

2449         /*
2450          * Get the stack layout, and fail now if it won't fit.
2451          */
2452         error = door_layout(server, ct->d_args.data_size, ndid, is_private);
2453         if (error != 0)
2454                 return (error);

2456         dsize = ndid * sizeof (door_desc_t);
2457         if (ct->d_args.data_size != 0) {
2458                 if (ct->d_args.data_size <= door_max_arg) {
2459                         /*
2460                          * Use a 2 copy method for small amounts of data
2461                          *
2462                          * Allocate a little more than we need for the
2463                          * args, in the hope that the results will fit
2464                          * without having to reallocate a buffer
2465                          */
2466                         ASSERT(ct->d_buf == NULL);
2467                         ct->d_bufsize = roundup(ct->d_args.data_size,
2468                             DOOR_ROUND);
2469                         ct->d_buf = kmem_alloc(ct->d_bufsize, KM_SLEEP);
2470                         if (copyin_nowatch(ct->d_args.data_ptr,
2471                             ct->d_buf, ct->d_args.data_size) != 0) {
2472                                 kmem_free(ct->d_buf, ct->d_bufsize);
2473                                 ct->d_buf = NULL;
2474                                 ct->d_bufsize = 0;
2475                                 return (EFAULT);
2476                         }
2477                 } else {
2478                         struct as       *as;
2479                         caddr_t         src;
2480                         caddr_t         dest;
2481                         size_t          len = ct->d_args.data_size;
2482                         uintptr_t       base;

2484                         /*
2485                          * Use a 1 copy method
2486                          */
```



```
2487                         as = ttoproc(server)->p_as;
2488                         src = ct->d_args.data_ptr;

2490                         dest = st->d_layout.dl_datap;
2491                         base = (uintptr_t)dest;

2493                         /*
2494                          * Copy data directly into server.  We proceed
2495                          * downward from the top of the stack, to mimic
2496                          * normal stack usage. This allows the guard page
2497                          * to stop us before we corrupt anything.
2498                          */
2499                         while (len != 0) {
2500                                 uintptr_t start;
2501                                 uintptr_t end;
2502                                 uintptr_t offset;
2503                                 size_t  amount;

2505                                 /*
2506                                  * Locate the next part to copy.
2507                                  */
2508                                 end = base + len;
2509                                 start = P2ALIGN(end - 1, PAGESIZE);

2511                                 /*
2512                                  * if we are on the final (first) page, fix
2513                                  * up the start position.
2514                                  */
2515                                 if (P2ALIGN(base, PAGESIZE) == start)
2516                                         start = base;

2518                                 offset = start - base;  /* the copy offset */
2519                                 amount = end - start;   /* # bytes to copy */

2521                                 ASSERT(amount > 0 && amount <= len &&
2522                                     amount <= PAGESIZE);

2524                                 error = door_copy(as, src + offset,
2525                                     dest + offset, amount);
2526                                 if (error != 0)
2527                                         return (error);
2528                                 len -= amount;
2529                         }
2530                 }
2531         }
2532         /*
2533          * Copyin the door args and translate them into files
2534          */
2535         if (ndid != 0) {
2536                 door_desc_t     *didpp;
2537                 door_desc_t     *start;
2538                 struct file     **fpp;

2540                 start = didpp = kmem_alloc(dsize, KM_SLEEP);

2542                 if (copyin_nowatch(ct->d_args.desc_ptr, didpp, dsize)) {
2543                         kmem_free(start, dsize);
2544                         return (EFAULT);
2545                 }
2546                 ct->d_fpp_size = ndid * sizeof (struct file *);
2547                 ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2548                 fpp = ct->d_fpp;
2549                 while (ndid--) {
2550                         struct file *fp;
2551                         int fd = didpp->d_data.d_desc.d_descriptor;
```

```
2553                        /* We only understand file descriptors as passed objs */
2554                        if (!(didpp->d_attributes & DOOR_DESCRIPTOR) ||
2555                            (fp = getf(fd)) == NULL) {
2556                                /* close translated references */
2557                                door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2558                                /* close untranslated references */
2559                                door_fd_rele(didpp, ndid + 1, 0);
2560                                kmem_free(start, dsize);
2561                                kmem_free(ct->d_fpp, ct->d_fpp_size);
2562                                ct->d_fpp = NULL;
2563                                ct->d_fpp_size = 0;
2564                                return (EINVAL);
2565                        }
2566                        /* Hold the fp */
2567                        mutex_enter(&fp->f_tlock);
2568                        fp->f_count++;
2569                        mutex_exit(&fp->f_tlock);

2571                        *fpp = fp;
2572                        releasef(fd);

2574                        if (didpp->d_attributes & DOOR_RELEASE) {
2575                                /* release passed reference */
2576                                (void) closeandsetf(fd, NULL);
2577                        }

2579                        fpp++; didpp++;
2580                }
2581                kmem_free(start, dsize);
2582        }
2583        return (0);
2584 }

2586 /*
2587  * Transfer arguments from a user client to a kernel server.  This copies in
2588  * descriptors and translates them into door handles.  It doesn't touch the
2589  * other data, letting the kernel server deal with that (to avoid needing
2590  * to copy the data twice).
2591  */
2592 static int
2593 door_translate_in(void)
2594 {
2595        door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2596        uint_t  ndid;

2598        ASSERT(MUTEX_NOT_HELD(&door_knob));
2599        ndid = ct->d_args.desc_num;
2600        if (ndid > door_max_desc)
2601                return (E2BIG);
2602        /*
2603         * Copyin the door args and translate them into door handles.
2604         */
2605        if (ndid != 0) {
2606                door_desc_t     *didpp;
2607                door_desc_t     *start;
2608                size_t          dsize = ndid * sizeof (door_desc_t);
2609                struct file     *fp;

2611                start = didpp = kmem_alloc(dsize, KM_SLEEP);

2613                if (copyin_nowatch(ct->d_args.desc_ptr, didpp, dsize)) {
2614                        kmem_free(start, dsize);
2615                        return (EFAULT);
2616                }
2617                while (ndid--) {
2618                        vnode_t *vp;
```



```
2619                        int fd = didpp->d_data.d_desc.d_descriptor;

2621                        /*
2622                         * We only understand file descriptors as passed objs
2623                         */
2624                        if ((didpp->d_attributes & DOOR_DESCRIPTOR) &&
2625                            (fp = getf(fd)) != NULL) {
2626                                didpp->d_data.d_handle = FTODH(fp);
2627                                /* Hold the door */
2628                                door_ki_hold(didpp->d_data.d_handle);

2630                                releasef(fd);

2632                                if (didpp->d_attributes & DOOR_RELEASE) {
2633                                        /* release passed reference */
2634                                        (void) closeandsetf(fd, NULL);
2635                                }

2637                                if (VOP_REALVP(fp->f_vnode, &vp, NULL))
2638                                        vp = fp->f_vnode;

2640                                /* Set attributes */
2641                                didpp->d_attributes = DOOR_HANDLE |
2642                                    (VTOD(vp)->door_flags & DOOR_ATTR_MASK);
2643                        } else {
2644                                /* close translated references */
2645                                door_fd_close(start, didpp - start);
2646                                /* close untranslated references */
2647                                door_fd_rele(didpp, ndid + 1, 0);
2648                                kmem_free(start, dsize);
2649                                return (EINVAL);
2650                        }
2651                        didpp++;
2652                }
2653                ct->d_args.desc_ptr = start;
2654        }
2655        return (0);
2656 }

2658 /*
2659  * Translate door arguments from kernel to user.  This copies the passed
2660  * door handles.  It doesn't touch other data.  It is used by door_upcall,
2661  * and for data returned by a door_call to a kernel server.
2662  */
2663 static int
2664 door_translate_out(void)
2665 {
2666        door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2667        uint_t  ndid;

2669        ASSERT(MUTEX_NOT_HELD(&door_knob));
2670        ndid = ct->d_args.desc_num;
2671        if (ndid > door_max_desc) {
2672                door_fd_rele(ct->d_args.desc_ptr, ndid, 1);
2673                return (E2BIG);
2674        }
2675        /*
2676         * Translate the door args into files
2677         */
2678        if (ndid != 0) {
2679                door_desc_t     *didpp = ct->d_args.desc_ptr;
2680                struct file     **fpp;

2682                ct->d_fpp_size = ndid * sizeof (struct file *);
2683                fpp = ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2684                while (ndid--) {
```

```
2685                        struct file *fp = NULL;
2686                        int fd = -1;

2688                        /*
2689                         * We understand file descriptors and door
2690                         * handles as passed objs.
2691                         */
2692                        if (didpp->d_attributes & DOOR_DESCRIPTOR) {
2693                                fd = didpp->d_data.d_desc.d_descriptor;
2694                                fp = getf(fd);
2695                        } else if (didpp->d_attributes & DOOR_HANDLE)
2696                                fp = DHTOF(didpp->d_data.d_handle);
2697                        if (fp != NULL) {
2698                                /* Hold the fp */
2699                                mutex_enter(&fp->f_tlock);
2700                                fp->f_count++;
2701                                mutex_exit(&fp->f_tlock);

2703                                *fpp = fp;
2704                                if (didpp->d_attributes & DOOR_DESCRIPTOR)
2705                                        releasef(fd);
2706                                if (didpp->d_attributes & DOOR_RELEASE) {
2707                                        /* release passed reference */
2708                                        if (fd >= 0)
2709                                                (void) closeandsetf(fd, NULL);
2710                                        else
2711                                                (void) closef(fp);
2712                                }
2713                        } else {
2714                                /* close translated references */
2715                                door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2716                                /* close untranslated references */
2717                                door_fd_rele(didpp, ndid + 1, 1);
2718                                kmem_free(ct->d_fpp, ct->d_fpp_size);
2719                                ct->d_fpp = NULL;
2720                                ct->d_fpp_size = 0;
2721                                return (EINVAL);
2722                        }
2723                        fpp++; didpp++;
2724                }
2725        }
2726        return (0);
2727 }

2729 /*
2730  * Move the results from the server to the client
2731  */
2732 static int
2733 door_results(kthread_t *caller, caddr_t data_ptr, size_t data_size,
2734                 door_desc_t *desc_ptr, uint_t desc_num)
2735 {
2736        door_client_t   *ct = DOOR_CLIENT(caller->t_door);
2737        door_upcall_t   *dup = ct->d_upcall;
2738        size_t          dsize;
2739        size_t          rlen;
2740        size_t          result_size;

2742        ASSERT(DOOR_T_HELD(ct));
2743        ASSERT(MUTEX_NOT_HELD(&door_knob));

2745        if (ct->d_noresults)
2746                return (E2BIG);         /* No results expected */

2748        if (desc_num > door_max_desc)
2749                return (E2BIG);         /* Too many descriptors */
```



```
2751        dsize = desc_num * sizeof (door_desc_t);
2752        /*
2753         * Check if the results are bigger than the clients buffer
2754         */
2755        if (dsize)
2756                rlen = roundup(data_size, sizeof (door_desc_t));
2757        else
2758                rlen = data_size;
2759        if ((result_size = rlen + dsize) == 0)
2760                return (0);

2762        if (dup != NULL) {
2763                if (desc_num > dup->du_max_descs)
2764                        return (EMFILE);

2766                if (data_size > dup->du_max_data)
2767                        return (E2BIG);

2769                /*
2770                 * Handle upcalls
2771                 */
2772                if (ct->d_args.rbuf == NULL || ct->d_args.rsize < result_size) {
2773                        /*
2774                         * If there's no return buffer or the buffer is too
2775                         * small, allocate a new one.  The old buffer (if it
2776                         * exists) will be freed by the upcall client.
2777                         */
2778                        if (result_size > door_max_upcall_reply)
2779                                return (E2BIG);
2780                        ct->d_args.rsize = result_size;
2781                        ct->d_args.rbuf = kmem_alloc(result_size, KM_SLEEP);
2782                }
2783                ct->d_args.data_ptr = ct->d_args.rbuf;
2784                if (data_size != 0 &&
2785                    copyin_nowatch(data_ptr, ct->d_args.data_ptr,
2786                    data_size) != 0)
2787                        return (EFAULT);
2788        } else if (result_size > ct->d_args.rsize) {
2789                return (door_overflow(caller, data_ptr, data_size,
2790                    desc_ptr, desc_num));
2791        } else if (data_size != 0) {
2792                if (data_size <= door_max_arg) {
2793                        /*
2794                         * Use a 2 copy method for small amounts of data
2795                         */
2796                        if (ct->d_buf == NULL) {
2797                                ct->d_bufsize = data_size;
2798                                ct->d_buf = kmem_alloc(ct->d_bufsize, KM_SLEEP);
2799                        } else if (ct->d_bufsize < data_size) {
2800                                kmem_free(ct->d_buf, ct->d_bufsize);
2801                                ct->d_bufsize = data_size;
2802                                ct->d_buf = kmem_alloc(ct->d_bufsize, KM_SLEEP);
2803                        }
2804                        if (copyin_nowatch(data_ptr, ct->d_buf, data_size) != 0)
2805                                return (EFAULT);
2806                } else {
2807                        struct as *as = ttoproc(caller)->p_as;
2808                        caddr_t dest = ct->d_args.rbuf;
2809                        caddr_t src = data_ptr;
2810                        size_t  len = data_size;

2812                        /* Copy data directly into client */
2813                        while (len != 0) {
2814                                uint_t  amount;
2815                                uint_t  max;
2816                                uint_t  off;
```

```
2817                                int     error;
2819                                off = (uintptr_t)dest & PAGEOFFSET;
2820                                if (off)
2821                                        max = PAGESIZE - off;
2822                                else
2823                                        max = PAGESIZE;
2824                                amount = len > max ? max : len;
2825                                error = door_copy(as, src, dest, amount);
2826                                if (error != 0)
2827                                        return (error);
2828                                dest += amount;
2829                                src += amount;
2830                                len -= amount;
2831                        }
2832                }
2833        }
2835        /*
2836         * Copyin the returned door ids and translate them into door_node_t
2837         */
2838        if (desc_num != 0) {
2839                door_desc_t *start;
2840                door_desc_t *didpp;
2841                struct file **fpp;
2842                size_t  fpp_size;
2843                uint_t  i;
2845                /* First, check if we would overflow client */
2846                if (!ufcanalloc(ttoproc(caller), desc_num))
2847                        return (EMFILE);
2849                start = didpp = kmem_alloc(dsize, KM_SLEEP);
2850                if (copyin_nowatch(desc_ptr, didpp, dsize)) {
2851                        kmem_free(start, dsize);
2852                        return (EFAULT);
2853                }
2854                fpp_size = desc_num * sizeof (struct file *);
2855                if (fpp_size > ct->d_fpp_size) {
2856                        /* make more space */
2857                        if (ct->d_fpp_size)
2858                                kmem_free(ct->d_fpp, ct->d_fpp_size);
2859                        ct->d_fpp_size = fpp_size;
2860                        ct->d_fpp = kmem_alloc(fpp_size, KM_SLEEP);
2861                }
2862                fpp = ct->d_fpp;
2864                for (i = 0; i < desc_num; i++) {
2865                        struct file *fp;
2866                        int fd = didpp->d_data.d_desc.d_descriptor;
2868                        /* Only understand file descriptor results */
2869                        if (!(didpp->d_attributes & DOOR_DESCRIPTOR) ||
2870                            (fp = getf(fd)) == NULL) {
2871                                /* close translated references */
2872                                door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2873                                /* close untranslated references */
2874                                door_fd_rele(didpp, desc_num - i, 0);
2875                                kmem_free(start, dsize);
2876                                return (EINVAL);
2877                        }
2879                        mutex_enter(&fp->f_tlock);
2880                        fp->f_count++;
2881                        mutex_exit(&fp->f_tlock);
```



```
2883                        *fpp = fp;
2884                        releasef(fd);
2886                        if (didpp->d_attributes & DOOR_RELEASE) {
2887                                /* release passed reference */
2888                                (void) closeandsetf(fd, NULL);
2889                        }
2891                        fpp++; didpp++;
2892                }
2893                kmem_free(start, dsize);
2894        }
2895        return (0);
2896 }
2898 /*
2899  * Close all the descriptors.
2900  */
2901 static void
2902 door_fd_close(door_desc_t *d, uint_t n)
2903 {
2904        uint_t  i;
2906        ASSERT(MUTEX_NOT_HELD(&door_knob));
2907        for (i = 0; i < n; i++) {
2908                if (d->d_attributes & DOOR_DESCRIPTOR) {
2909                        (void) closeandsetf(
2910                            d->d_data.d_desc.d_descriptor, NULL);
2911                } else if (d->d_attributes & DOOR_HANDLE) {
2912                        door_ki_rele(d->d_data.d_handle);
2913                }
2914                d++;
2915        }
2916 }
2918 /*
2919  * Close descriptors that have the DOOR_RELEASE attribute set.
2920  */
2921 void
2922 door_fd_rele(door_desc_t *d, uint_t n, int from_kernel)
2923 {
2924        uint_t  i;
2926        ASSERT(MUTEX_NOT_HELD(&door_knob));
2927        for (i = 0; i < n; i++) {
2928                if (d->d_attributes & DOOR_RELEASE) {
2929                        if (d->d_attributes & DOOR_DESCRIPTOR) {
2930                                (void) closeandsetf(
2931                                    d->d_data.d_desc.d_descriptor, NULL);
2932                        } else if (from_kernel &&
2933                            (d->d_attributes & DOOR_HANDLE)) {
2934                                door_ki_rele(d->d_data.d_handle);
2935                        }
2936                }
2937                d++;
2938        }
2939 }
2941 /*
2942  * Copy descriptors into the kernel so we can release any marked
2943  * DOOR_RELEASE.
2944  */
2945 int
2946 door_release_fds(door_desc_t *desc_ptr, uint_t ndesc)
2947 {
2948        size_t dsize;
```

```
2949         door_desc_t *didpp;
2950         uint_t desc_num;
2951 
2952         ASSERT(MUTEX_NOT_HELD(&door_knob));
2953         ASSERT(ndesc != 0);
2954 
2955         desc_num = MIN(ndesc, door_max_desc);
2956 
2957         dsize = desc_num * sizeof (door_desc_t);
2958         didpp = kmem_alloc(dsize, KM_SLEEP);
2959 
2960         while (ndesc > 0) {
2961                 uint_t count = MIN(ndesc, desc_num);
2962 
2963                 if (copyin_nowatch(desc_ptr, didpp,
2964                     count * sizeof (door_desc_t))) {
2965                         kmem_free(didpp, dsize);
2966                         return (EFAULT);
2967                 }
2968                 door_fd_rele(didpp, count, 0);
2969 
2970                 ndesc -= count;
2971                 desc_ptr += count;
2972         }
2973         kmem_free(didpp, dsize);
2974         return (0);
2975 }
2976 
2977 /*
2978  * Decrement ref count on all the files passed
2979  */
2980 static void
2981 door_fp_close(struct file **fp, uint_t n)
2982 {
2983         uint_t  i;
2984 
2985         ASSERT(MUTEX_NOT_HELD(&door_knob));
2986 
2987         for (i = 0; i < n; i++)
2988                 (void) closef(fp[i]);
2989 }
2990 
2991 /*
2992  * Copy data from 'src' in current address space to 'dest' in 'as' for 'len'
2993  * bytes.
2994  *
2995  * Performs this using 1 mapin and 1 copy operation.
2996  *
2997  * We really should do more than 1 page at a time to improve
2998  * performance, but for now this is treated as an anomalous condition.
2999  */
3000 static int
3001 door_copy(struct as *as, caddr_t src, caddr_t dest, uint_t len)
3002 {
3003         caddr_t kaddr;
3004         caddr_t rdest;
3005         uint_t  off;
3006         page_t  **pplist;
3007         page_t  *pp = NULL;
3008         int     error = 0;
3009 
3010         ASSERT(len <= PAGESIZE);
3011         off = (uintptr_t)dest & PAGEOFFSET;     /* offset within the page */
3012         rdest = (caddr_t)((uintptr_t)dest &
3013             (uintptr_t)PAGEMASK);       /* Page boundary */
3014         ASSERT(off + len <= PAGESIZE);
```



```
3016         /*
3017          * Lock down destination page.
3018          */
3019         if (as_pagelock(as, &pplist, rdest, PAGESIZE, S_WRITE))
3020                 return (E2BIG);
3021         /*
3022          * Check if we have a shadow page list from as_pagelock. If not,
3023          * we took the slow path and have to find our page struct the hard
3024          * way.
3025          */
3026         if (pplist == NULL) {
3027                 pfn_t   pfnum;
3028 
3029                 /* MMU mapping is already locked down */
3030                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3031                 pfnum = hat_getpfnum(as->a_hat, rdest);
3032                 AS_LOCK_EXIT(as, &as->a_lock);
3033 
3034                 /*
3035                  * TODO: The pfn step should not be necessary - need
3036                  * a hat_getpp() function.
3037                  */
3038                 if (pf_is_memory(pfnum)) {
3039                         pp = page_numtopp_nolock(pfnum);
3040                         ASSERT(pp == NULL || PAGE_LOCKED(pp));
3041                 } else
3042                         pp = NULL;
3043                 if (pp == NULL) {
3044                         as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3045                         return (E2BIG);
3046                 }
3047         } else {
3048                 pp = *pplist;
3049         }
3050         /*
3051          * Map destination page into kernel address
3052          */
3053         if (kpm_enable)
3054                 kaddr = (caddr_t)hat_kpm_mapin(pp, (struct kpme *)NULL);
3055         else
3056                 kaddr = (caddr_t)ppmapin(pp, PROT_READ | PROT_WRITE,
3057                     (caddr_t)-1);
3058 
3059         /*
3060          * Copy from src to dest
3061          */
3062         if (copyin_nowatch(src, kaddr + off, len) != 0)
3063                 error = EFAULT;
3064         /*
3065          * Unmap destination page from kernel
3066          */
3067         if (kpm_enable)
3068                 hat_kpm_mapout(pp, (struct kpme *)NULL, kaddr);
3069         else
3070                 ppmapout(kaddr);
3071         /*
3072          * Unlock destination page
3073          */
3074         as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3075         return (error);
3076 }
3077 
3078 /*
3079  * General kernel upcall using doors
3080  *      Returns 0 on success, errno for failures.
```

```
 *      Caller must have a hold on the door based vnode, and on any
 *      references passed in desc_ptr.  The references are released
 *      in the event of an error, and passed without duplication
 *      otherwise.  Note that param->rbuf must be 64-bit aligned in
 *      a 64-bit kernel, since it may be used to store door descriptors
 *      if they are returned by the server.  The caller is responsible
 *      for holding a reference to the cred passed in.
 */
int
door_upcall(vnode_t *vp, door_arg_t *param, struct cred *cred,
    size_t max_data, uint_t max_descs)
{
        /* Locals */
        door_upcall_t   *dup;
        door_node_t     *dp;
        kthread_t       *server_thread;
        int             error = 0;
        klwp_t          *lwp;
        door_client_t   *ct;            /* curthread door_data */
        door_server_t   *st;            /* server thread door_data */
        int             gotresults = 0;
        int             cancel_pending;

        if (vp->v_type != VDOOR) {
                if (param->desc_num)
                        door_fd_rele(param->desc_ptr, param->desc_num, 1);
                return (EINVAL);
        }

        lwp = ttolwp(curthread);
        ct = door_my_client(1);
        dp = VTOD(vp);  /* Convert to a door_node_t */

        dup = kmem_zalloc(sizeof (*dup), KM_SLEEP);
        dup->du_cred = (cred != NULL) ? cred : curthread->t_cred;
        dup->du_max_data = max_data;
        dup->du_max_descs = max_descs;

        /*
         * This should be done in shuttle_resume(), just before going to
         * sleep, but we want to avoid overhead while holding door_knob.
         * prstop() is just a no-op if we don't really go to sleep.
         * We test not-kernel-address-space for the sake of clustering code.
         */
        if (lwp && lwp->lwp_nostop == 0 && curproc->p_as != &kas)
                prstop(PR_REQUESTED, 0);

        mutex_enter(&door_knob);
        if (DOOR_INVALID(dp)) {
                mutex_exit(&door_knob);
                if (param->desc_num)
                        door_fd_rele(param->desc_ptr, param->desc_num, 1);
                error = EBADF;
                goto out;
        }

        if (dp->door_target == &p0) {
                /* Can't do an upcall to a kernel server */
                mutex_exit(&door_knob);
                if (param->desc_num)
                        door_fd_rele(param->desc_ptr, param->desc_num, 1);
                error = EINVAL;
                goto out;
        }

        error = door_check_limits(dp, param, 1);
```



```
        if (error != 0) {
                mutex_exit(&door_knob);
                if (param->desc_num)
                        door_fd_rele(param->desc_ptr, param->desc_num, 1);
                goto out;
        }

        /*
         * Get a server thread from the target domain
         */
        if ((server_thread = door_get_server(dp)) == NULL) {
                if (DOOR_INVALID(dp))
                        error = EBADF;
                else
                        error = EAGAIN;
                mutex_exit(&door_knob);
                if (param->desc_num)
                        door_fd_rele(param->desc_ptr, param->desc_num, 1);
                goto out;
        }

        st = DOOR_SERVER(server_thread->t_door);
        ct->d_buf = param->data_ptr;
        ct->d_bufsize = param->data_size;
        ct->d_args = *param;    /* structure assignment */

        if (ct->d_args.desc_num) {
                /*
                 * Move data from client to server
                 */
                DOOR_T_HOLD(st);
                mutex_exit(&door_knob);
                error = door_translate_out();
                mutex_enter(&door_knob);
                DOOR_T_RELEASE(st);
                if (error) {
                        /*
                         * We're not going to resume this thread after all
                         */
                        door_release_server(dp, server_thread);
                        shuttle_sleep(server_thread);
                        mutex_exit(&door_knob);
                        goto out;
                }
        }

        ct->d_upcall = dup;
        if (param->rsize == 0)
                ct->d_noresults = 1;
        else
                ct->d_noresults = 0;

        dp->door_active++;

        ct->d_error = DOOR_WAIT;
        st->d_caller = curthread;
        st->d_active = dp;

        shuttle_resume(server_thread, &door_knob);

        mutex_enter(&door_knob);
shuttle_return:
        if ((error = ct->d_error) < 0) {        /* DOOR_WAIT or DOOR_EXIT */
                /*
                 * Premature wakeup. Find out why (stop, forkall, sig, exit ...)
                 */
```

```
3213                mutex_exit(&door_knob);         /* May block in ISSIG */
3214                cancel_pending = 0;
3215                if (lwp && (ISSIG(curthread, FORREAL) || lwp->lwp_sysabort ||
3216                    MUSTRETURN(curproc, curthread) ||
3217                    (cancel_pending = schedctl_cancel_pending()) != 0)) {
3218                        /* Signal, forkall, ... */
3219                        if (cancel_pending)
3220                                schedctl_cancel_eintr();
3221                        lwp->lwp_sysabort = 0;
3222                        mutex_enter(&door_knob);
3223                        error = EINTR;
3224                        /*
3225                         * If the server has finished processing our call,
3226                         * or exited (calling door_slam()), then d_error
3227                         * will have changed.  If the server hasn't finished
3228                         * yet, d_error will still be DOOR_WAIT, and we
3229                         * let it know we are not interested in any
3230                         * results by sending a SIGCANCEL, unless the door
3231                         * is marked with DOOR_NO_CANCEL.
3232                         */
3233                        if (ct->d_error == DOOR_WAIT &&
3234                            st->d_caller == curthread) {
3235                                proc_t  *p = ttoproc(server_thread);

3237                                st->d_active = NULL;
3238                                st->d_caller = NULL;
3239                                if (!(dp->door_flags & DOOR_NO_CANCEL)) {
3240                                        DOOR_T_HOLD(st);
3241                                        mutex_exit(&door_knob);

3243                                        mutex_enter(&p->p_lock);
3244                                        sigtoproc(p, server_thread, SIGCANCEL);
3245                                        mutex_exit(&p->p_lock);

3247                                        mutex_enter(&door_knob);
3248                                        DOOR_T_RELEASE(st);
3249                                }
3250                        }
3251                } else {
3252                        /*
3253                         * Return from stop(), server exit...
3254                         *
3255                         * Note that the server could have done a
3256                         * door_return while the client was in stop state
3257                         * (ISSIG), in which case the error condition
3258                         * is updated by the server.
3259                         */
3260                        mutex_enter(&door_knob);
3261                        if (ct->d_error == DOOR_WAIT) {
3262                                /* Still waiting for a reply */
3263                                shuttle_swtch(&door_knob);
3264                                mutex_enter(&door_knob);
3265                                if (lwp)
3266                                        lwp->lwp_asleep = 0;
3267                                goto    shuttle_return;
3268                        } else if (ct->d_error == DOOR_EXIT) {
3269                                /* Server exit */
3270                                error = EINTR;
3271                        } else {
3272                                /* Server did a door_return during ISSIG */
3273                                error = ct->d_error;
3274                        }
3275                }
3276                /*
3277                 * Can't exit if the server is currently copying
3278                 * results for me
```



```
3279                 */
3280                while (DOOR_T_HELD(ct))
3281                        cv_wait(&ct->d_cv, &door_knob);

3283                /*
3284                 * Find out if results were successfully copied.
3285                 */
3286                if (ct->d_error == 0)
3287                        gotresults = 1;
3288        }
3289        if (lwp) {
3290                lwp->lwp_asleep = 0;            /* /proc */
3291                lwp->lwp_sysabort = 0;          /* /proc */
3292        }
3293        if (--dp->door_active == 0 && (dp->door_flags & DOOR_DELAY))
3294                door_deliver_unref(dp);
3295        mutex_exit(&door_knob);

3297        /*
3298         * Translate returned doors (if any)
3299         */

3301        if (ct->d_noresults)
3302                goto out;

3304        if (error) {
3305                /*
3306                 * If server returned results successfully, then we've
3307                 * been interrupted and may need to clean up.
3308                 */
3309                if (gotresults) {
3310                        ASSERT(error == EINTR);
3311                        door_fp_close(ct->d_fpp, ct->d_args.desc_num);
3312                }
3313                goto out;
3314        }

3316        if (ct->d_args.desc_num) {
3317                struct file     **fpp;
3318                door_desc_t     *didpp;
3319                vnode_t         *vp;
3320                uint_t          n = ct->d_args.desc_num;

3322                didpp = ct->d_args.desc_ptr = (door_desc_t *)(ct->d_args.rbuf +
3323                    roundup(ct->d_args.data_size, sizeof (door_desc_t)));
3324                fpp = ct->d_fpp;

3326                while (n--) {
3327                        struct file *fp;

3329                        fp = *fpp;
3330                        if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3331                                vp = fp->f_vnode;

3333                        didpp->d_attributes = DOOR_HANDLE |
3334                            (VTOD(vp)->door_flags & DOOR_ATTR_MASK);
3335                        didpp->d_data.d_handle = FTODH(fp);

3337                        fpp++; didpp++;
3338                }
3339        }

3341        /* on return data is in rbuf */
3342        *param = ct->d_args;            /* structure assignment */

3344 out:
```

```
3345         kmem_free(dup, sizeof (*dup));

3347         if (ct->d_fpp) {
3348                 kmem_free(ct->d_fpp, ct->d_fpp_size);
3349                 ct->d_fpp = NULL;
3350                 ct->d_fpp_size = 0;
3351         }

3353         ct->d_upcall = NULL;
3354         ct->d_noresults = 0;
3355         ct->d_buf = NULL;
3356         ct->d_bufsize = 0;
3357         return (error);
3358 }

3360 /*
3361  * Add a door to the per-process list of active doors for which the
3362  * process is a server.
3363  */
3364 static void
3365 door_list_insert(door_node_t *dp)
3366 {
3367         proc_t *p = dp->door_target;

3369         ASSERT(MUTEX_HELD(&door_knob));
3370         dp->door_list = p->p_door_list;
3371         p->p_door_list = dp;
3372 }

3374 /*
3375  * Remove a door from the per-process list of active doors.
3376  */
3377 void
3378 door_list_delete(door_node_t *dp)
3379 {
3380         door_node_t **pp;

3382         ASSERT(MUTEX_HELD(&door_knob));
3383         /*
3384          * Find the door in the list.  If the door belongs to another process,
3385          * it's OK to use p_door_list since that process can't exit until all
3386          * doors have been taken off the list (see door_exit).
3387          */
3388         pp = &(dp->door_target->p_door_list);
3389         while (*pp != dp)
3390                 pp = &((*pp)->door_list);

3392         /* found it, take it off the list */
3393         *pp = dp->door_list;
3394 }


3397 /*
3398  * External kernel interfaces for doors.  These functions are available
3399  * outside the doorfs module for use in creating and using doors from
3400  * within the kernel.
3401  */

3403 /*
3404  * door_ki_upcall invokes a user-level door server from the kernel, with
3405  * the credentials associated with curthread.
3406  */
3407 int
3408 door_ki_upcall(door_handle_t dh, door_arg_t *param)
3409 {
3410         return (door_ki_upcall_limited(dh, param, NULL, SIZE_MAX, UINT_MAX));
```



```
3411 }

3413 /*
3414  * door_ki_upcall_limited invokes a user-level door server from the
3415  * kernel with the given credentials and reply limits.  If the "cred"
3416  * argument is NULL, uses the credentials associated with current
3417  * thread.  max_data limits the maximum length of the returned data (the
3418  * client will get E2BIG if they go over), and max_desc limits the
3419  * number of returned descriptors (the client will get EMFILE if they
3420  * go over).
3421  */
3422 int
3423 door_ki_upcall_limited(door_handle_t dh, door_arg_t *param, struct cred *cred,
3424     size_t max_data, uint_t max_desc)
3425 {
3426         file_t *fp = DHTOF(dh);
3427         vnode_t *realvp;

3429         if (VOP_REALVP(fp->f_vnode, &realvp, NULL))
3430                 realvp = fp->f_vnode;
3431         return (door_upcall(realvp, param, cred, max_data, max_desc));
3432 }

3434 /*
3435  * Function call to create a "kernel" door server.  A kernel door
3436  * server provides a way for a user-level process to invoke a function
3437  * in the kernel through a door_call.  From the caller's point of
3438  * view, a kernel door server looks the same as a user-level one
3439  * (except the server pid is 0).  Unlike normal door calls, the
3440  * kernel door function is invoked via a normal function call in the
3441  * same thread and context as the caller.
3442  */
3443 int
3444 door_ki_create(void (*pc_cookie)(), void *data_cookie, uint_t attributes,
3445     door_handle_t *dhp)
3446 {
3447         int err;
3448         file_t *fp;

3450         /* no DOOR_PRIVATE */
3451         if ((attributes & ~DOOR_KI_CREATE_MASK) ||
3452             (attributes & (DOOR_UNREF | DOOR_UNREF_MULTI)) ==
3453             (DOOR_UNREF | DOOR_UNREF_MULTI))
3454                 return (EINVAL);

3456         err = door_create_common(pc_cookie, data_cookie, attributes,
3457             1, NULL, &fp);
3458         if (err == 0 && (attributes & (DOOR_UNREF | DOOR_UNREF_MULTI)) &&
3459             p0.p_unref_thread == 0) {
3460                 /* need to create unref thread for process 0 */
3461                 (void) thread_create(NULL, 0, door_unref_kernel, NULL, 0, &p0,
3462                     TS_RUN, minclsyspri);
3463         }
3464         if (err == 0) {
3465                 *dhp = FTODH(fp);
3466         }
3467         return (err);
3468 }

3470 void
3471 door_ki_hold(door_handle_t dh)
3472 {
3473         file_t *fp = DHTOF(dh);

3475         mutex_enter(&fp->f_tlock);
3476         fp->f_count++;
```

```
3477         mutex_exit(&fp->f_tlock);
3478 }

3480 void
3481 door_ki_rele(door_handle_t dh)
3482 {
3483         file_t *fp = DHTOF(dh);

3485         (void) closef(fp);
3486 }

3488 int
3489 door_ki_open(char *pathname, door_handle_t *dhp)
3490 {
3491         file_t *fp;
3492         vnode_t *vp;
3493         int err;

3495         if ((err = lookupname(pathname, UIO_SYSSPACE, FOLLOW, NULL, &vp)) != 0)
3496                 return (err);
3497         if (err = VOP_OPEN(&vp, FREAD, kcred, NULL)) {
3498                 VN_RELE(vp);
3499                 return (err);
3500         }
3501         if (vp->v_type != VDOOR) {
3502                 VN_RELE(vp);
3503                 return (EINVAL);
3504         }
3505         if ((err = falloc(vp, FREAD | FWRITE, &fp, NULL)) != 0) {
3506                 VN_RELE(vp);
3507                 return (err);
3508         }
3509         /* falloc returns with f_tlock held on success */
3510         mutex_exit(&fp->f_tlock);
3511         *dhp = FTODH(fp);
3512         return (0);
3513 }

3515 int
3516 door_ki_info(door_handle_t dh, struct door_info *dip)
3517 {
3518         file_t *fp = DHTOF(dh);
3519         vnode_t *vp;

3521         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3522                 vp = fp->f_vnode;
3523         if (vp->v_type != VDOOR)
3524                 return (EINVAL);
3525         door_info_common(VTOD(vp), dip, fp);
3526         return (0);
3527 }

3529 door_handle_t
3530 door_ki_lookup(int did)
3531 {
3532         file_t *fp;
3533         door_handle_t dh;

3535         /* is the descriptor really a door? */
3536         if (door_lookup(did, &fp) == NULL)
3537                 return (NULL);
3538         /* got the door, put a hold on it and release the fd */
3539         dh = FTODH(fp);
3540         door_ki_hold(dh);
3541         releasef(did);
3542         return (dh);
```



```
3543 }

3545 int
3546 door_ki_setparam(door_handle_t dh, int type, size_t val)
3547 {
3548         file_t *fp = DHTOF(dh);
3549         vnode_t *vp;

3551         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3552                 vp = fp->f_vnode;
3553         if (vp->v_type != VDOOR)
3554                 return (EINVAL);
3555         return (door_setparam_common(VTOD(vp), 1, type, val));
3556 }

3558 int
3559 door_ki_getparam(door_handle_t dh, int type, size_t *out)
3560 {
3561         file_t *fp = DHTOF(dh);
3562         vnode_t *vp;

3564         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3565                 vp = fp->f_vnode;
3566         if (vp->v_type != VDOOR)
3567                 return (EINVAL);
3568         return (door_getparam_common(VTOD(vp), type, out));
3569 }
```

```
**********************************************************
   16819 Mon Aug 17 21:08:03 2015
new/usr/src/uts/common/fs/sockfs/sockcommon.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 439 /*
 440  * TODO Once the common vnode ops is available, then the vnops argument
 441  * should be removed.
 442  */
 443 /*ARGSUSED*/
 444 int
 445 sonode_constructor(void *buf, void *cdrarg, int kmflags)
 446 {
 447         struct sonode *so = buf;
 448         struct vnode *vp;

 450         vp = so->so_vnode = vn_alloc(kmflags);
 451         if (vp == NULL) {
 452                 return (-1);
 453         }
 454         vp->v_data = so;
 455         vn_setops(vp, socket_vnodeops);

 457         so->so_priv             = NULL;
 458         so->so_oobmsg           = NULL;

 460         so->so_proto_handle     = NULL;

 462         so->so_peercred         = NULL;

 464         so->so_rcv_queued       = 0;
 465         so->so_rcv_q_head       = NULL;
 466         so->so_rcv_q_last_head  = NULL;
 467         so->so_rcv_head         = NULL;
 468         so->so_rcv_last_head    = NULL;
 469         so->so_rcv_wanted       = 0;
 470         so->so_rcv_timer_interval = SOCKET_NO_RCVTIMER;
 471         so->so_rcv_timer_tid    = 0;
 472         so->so_rcv_thresh       = 0;

 474         list_create(&so->so_acceptq_list, sizeof (struct sonode),
 475             offsetof(struct sonode, so_acceptq_node));
 476         list_create(&so->so_acceptq_defer, sizeof (struct sonode),
 477             offsetof(struct sonode, so_acceptq_node));
 478         list_create(&so->so_pid_list, sizeof (pid_node_t),
 479                 offsetof(pid_node_t, pn_ref_link));
 480 #endif /* ! codereview */
 481         list_link_init(&so->so_acceptq_node);
 482         so->so_acceptq_len      = 0;
 483         so->so_backlog          = 0;
 484         so->so_listener         = NULL;

 486         so->so_snd_qfull        = B_FALSE;

 488         so->so_filter_active    = 0;
 489         so->so_filter_tx        = 0;
 490         so->so_filter_defertime = 0;
 491         so->so_filter_top       = NULL;
 492         so->so_filter_bottom    = NULL;

 494         mutex_init(&so->so_lock, NULL, MUTEX_DEFAULT, NULL);
 495         mutex_init(&so->so_acceptq_lock, NULL, MUTEX_DEFAULT, NULL);
 496         mutex_init(&so->so_pid_list_lock, NULL, MUTEX_DEFAULT, NULL);
 497 #endif /* ! codereview */
```



```
 498         rw_init(&so->so_fallback_rwlock, NULL, RW_DEFAULT, NULL);
 499         cv_init(&so->so_state_cv, NULL, CV_DEFAULT, NULL);
 500         cv_init(&so->so_single_cv, NULL, CV_DEFAULT, NULL);
 501         cv_init(&so->so_read_cv, NULL, CV_DEFAULT, NULL);

 503         cv_init(&so->so_acceptq_cv, NULL, CV_DEFAULT, NULL);
 504         cv_init(&so->so_snd_cv, NULL, CV_DEFAULT, NULL);
 505         cv_init(&so->so_rcv_cv, NULL, CV_DEFAULT, NULL);
 506         cv_init(&so->so_copy_cv, NULL, CV_DEFAULT, NULL);
 507         cv_init(&so->so_closing_cv, NULL, CV_DEFAULT, NULL);

 509         return (0);
 510 }

 512 /*ARGSUSED*/
 513 void
 514 sonode_destructor(void *buf, void *cdrarg)
 515 {
 516         struct sonode *so = buf;
 517         struct vnode *vp = SOTOV(so);

 519         ASSERT(so->so_priv == NULL);
 520         ASSERT(so->so_peercred == NULL);

 522         ASSERT(so->so_oobmsg == NULL);

 524         ASSERT(so->so_rcv_q_head == NULL);

 526         list_destroy(&so->so_acceptq_list);
 527         list_destroy(&so->so_acceptq_defer);
 528         list_destroy(&so->so_pid_list);
 529 #endif /* ! codereview */
 530         ASSERT(!list_link_active(&so->so_acceptq_node));
 531         ASSERT(so->so_listener == NULL);

 533         ASSERT(so->so_filter_active == 0);
 534         ASSERT(so->so_filter_tx == 0);
 535         ASSERT(so->so_filter_top == NULL);
 536         ASSERT(so->so_filter_bottom == NULL);

 538         ASSERT(vp->v_data == so);
 539         ASSERT(vn_matchops(vp, socket_vnodeops));

 541         vn_free(vp);

 543         mutex_destroy(&so->so_lock);
 544         mutex_destroy(&so->so_acceptq_lock);
 545         mutex_destroy(&so->so_pid_list_lock);
 546 #endif /* ! codereview */
 547         rw_destroy(&so->so_fallback_rwlock);

 549         cv_destroy(&so->so_state_cv);
 550         cv_destroy(&so->so_single_cv);
 551         cv_destroy(&so->so_read_cv);
 552         cv_destroy(&so->so_acceptq_cv);
 553         cv_destroy(&so->so_snd_cv);
 554         cv_destroy(&so->so_rcv_cv);
 555         cv_destroy(&so->so_closing_cv);
 556 }

 558 void
 559 sonode_init(struct sonode *so, struct sockparams *sp, int family,
 560     int type, int protocol, sonodeops_t *sops)
 561 {
 562         vnode_t *vp;
```

```
564         vp = SOTOV(so);

566         so->so_flag     = 0;

568         so->so_state    = 0;
569         so->so_mode     = 0;

571         so->so_count    = 0;

573         so->so_family   = family;
574         so->so_type     = type;
575         so->so_protocol = protocol;

577         SOCK_CONNID_INIT(so->so_proto_connid);

579         so->so_options  = 0;
580         so->so_linger.l_onoff   = 0;
581         so->so_linger.l_linger = 0;
582         so->so_sndbuf   = 0;
583         so->so_error    = 0;
584         so->so_rcvtimeo = 0;
585         so->so_sndtimeo = 0;
586         so->so_xpg_rcvbuf = 0;

588         ASSERT(so->so_oobmsg == NULL);
589         so->so_oobmark  = 0;
590         so->so_pgrp     = 0;

592         ASSERT(so->so_peercred == NULL);

594         so->so_zoneid = getzoneid();

596         so->so_sockparams = sp;

598         so->so_ops = sops;

600         so->so_not_str = (sops != &sotpi_sonodeops);

602         so->so_proto_handle = NULL;

604         so->so_downcalls = NULL;

606         so->so_copyflag = 0;

608         vn_reinit(vp);
609         vp->v_vfsp      = rootvfs;
610         vp->v_type      = VSOCK;
611         vp->v_rdev      = sockdev;

613         so->so_snd_qfull = B_FALSE;
614         so->so_minpsz = 0;

616         so->so_rcv_wakeup = B_FALSE;
617         so->so_snd_wakeup = B_FALSE;
618         so->so_flowctrld = B_FALSE;

620         so->so_pollev = 0;
621         bzero(&so->so_poll_list, sizeof (so->so_poll_list));
622         bzero(&so->so_proto_props, sizeof (struct sock_proto_props));

624         bzero(&(so->so_ksock_callbacks), sizeof (ksocket_callbacks_t));
625         so->so_ksock_cb_arg = NULL;

627         so->so_max_addr_len = sizeof (struct sockaddr_storage);

629         so->so_direct = NULL;
```



```
631         vn_exists(vp);
632 }

634 void
635 sonode_fini(struct sonode *so)
636 {
637         vnode_t *vp;
638         pid_node_t *pn;
639 #endif /* ! codereview */

641         ASSERT(so->so_count == 0);

643         if (so->so_rcv_timer_tid) {
644                 ASSERT(MUTEX_NOT_HELD(&so->so_lock));
645                 (void) untimeout(so->so_rcv_timer_tid);
646                 so->so_rcv_timer_tid = 0;
647         }

649         if (so->so_poll_list.ph_list != NULL) {
650                 pollwakeup(&so->so_poll_list, POLLERR);
651                 pollhead_clean(&so->so_poll_list);
652         }

654         if (so->so_direct != NULL)
655                 sod_sock_fini(so);

657         vp = SOTOV(so);
658         vn_invalid(vp);

660         if (so->so_peercred != NULL) {
661                 crfree(so->so_peercred);
662                 so->so_peercred = NULL;
663         }
664         /* Detach and destroy filters */
665         if (so->so_filter_top != NULL)
666                 sof_sonode_cleanup(so);

668         mutex_enter(&so->so_pid_list_lock);
669         while ((pn = list_head(&so->so_pid_list)) != NULL) {
670                 list_remove(&so->so_pid_list, pn);
671                 kmem_free(pn, sizeof (*pn));
672         }
673         mutex_exit(&so->so_pid_list_lock);

675 #endif /* ! codereview */
676         ASSERT(list_is_empty(&so->so_acceptq_list));
677         ASSERT(list_is_empty(&so->so_acceptq_defer));
678         ASSERT(!list_link_active(&so->so_acceptq_node));

680         ASSERT(so->so_rcv_queued == 0);
681         ASSERT(so->so_rcv_q_head == NULL);
682         ASSERT(so->so_rcv_q_last_head == NULL);
683         ASSERT(so->so_rcv_head == NULL);
684         ASSERT(so->so_rcv_last_head == NULL);
685 }

687 void
688 sonode_insert_pid(struct sonode *so, pid_t pid)
689 {
690         pid_node_t      *pn;

692         mutex_enter(&so->so_pid_list_lock);
693         for (pn = list_head(&so->so_pid_list);
694             pn != NULL && pn->pn_pid != pid;
695             pn = list_next(&so->so_pid_list, pn))
```

```
696                 ;

698         if (pn != NULL) {
699                 pn->pn_count++;
700         } else {
701                 pn = kmem_zalloc(sizeof (*pn), KM_SLEEP);
702                 list_link_init(&pn->pn_ref_link);
703                 pn->pn_pid = pid;
704                 pn->pn_count = 1;
705                 list_insert_tail(&so->so_pid_list, pn);
706         }
707         mutex_exit(&so->so_pid_list_lock);
708 }

710 void
711 sonode_remove_pid(struct sonode *so, pid_t pid)
712 {
713         pid_node_t *pn;

715         mutex_enter(&so->so_pid_list_lock);
716         for (pn = list_head(&so->so_pid_list);
717             pn != NULL && pn->pn_pid != pid;
718             pn = list_next(&so->so_pid_list, pn))
719                 ;

721         if (pn != NULL) {
722                 if (pn->pn_count > 1) {
723                                 pn->pn_count--;
724                 } else {
725                         list_remove(&so->so_pid_list, pn);
726                         kmem_free(pn, sizeof (*pn));
727                 }
728         }
729         mutex_exit(&so->so_pid_list_lock);
730 #endif /* ! codereview */
731 }
```

```
**********************************************************
    9811 Mon Aug 17 21:08:04 2015
new/usr/src/uts/common/fs/sockfs/sockcommon.h
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 106 /* Common sonode ops not support */
 107 extern int so_listen_notsupp(struct sonode *, int, struct cred *);
 108 extern int so_accept_notsupp(struct sonode *, int, struct cred *,
 109     struct sonode **);
 110 extern int so_getpeername_notsupp(struct sonode *, struct sockaddr *,
 111     socklen_t *, boolean_t, struct cred *);
 112 extern int so_shutdown_notsupp(struct sonode *, int, struct cred *);
 113 extern int so_sendmblk_notsupp(struct sonode *, struct nmsghdr *,
 114     int, struct cred *, mblk_t **);

 116 /* Common sonode ops */
 117 extern int so_init(struct sonode *, struct sonode *, struct cred *, int);
 118 extern int so_accept(struct sonode *, int, struct cred *, struct sonode **);
 119 extern int so_bind(struct sonode *, struct sockaddr *, socklen_t, int,
 120     struct cred *);
 121 extern int so_listen(struct sonode *, int, struct cred *);
 122 extern int so_connect(struct sonode *, struct sockaddr *,
 123     socklen_t, int, int, struct cred *);
 124 extern int so_getsockopt(struct sonode *, int, int, void *,
 125     socklen_t *, int, struct cred *);
 126 extern int so_setsockopt(struct sonode *, int, int, const void *,
 127     socklen_t, struct cred *);
 128 extern int so_getpeername(struct sonode *, struct sockaddr *,
 129     socklen_t *, boolean_t, struct cred *);
 130 extern int so_getsockname(struct sonode *, struct sockaddr *,
 131     socklen_t *, struct cred *);
 132 extern int so_ioctl(struct sonode *, int, intptr_t, int, struct cred *,
 133     int32_t *);
 134 extern int so_poll(struct sonode *, short, int, short *,
 135     struct pollhead **);
 136 extern int so_sendmsg(struct sonode *, struct nmsghdr *, struct uio *,
 137     struct cred *);
 138 extern int so_sendmblk_impl(struct sonode *, struct nmsghdr *, int,
 139     struct cred *, mblk_t **, struct sof_instance *, boolean_t);
 140 extern int so_sendmblk(struct sonode *, struct nmsghdr *, int,
 141     struct cred *, mblk_t **);
 142 extern int so_recvmsg(struct sonode *, struct nmsghdr *, struct uio *,
 143     struct cred *);
 144 extern int so_shutdown(struct sonode *, int, struct cred *);
 145 extern int so_close(struct sonode *, int, struct cred *);

 147 extern int so_tpi_fallback(struct sonode *, struct cred *);

 149 /* Common upcalls */
 150 extern sock_upper_handle_t so_newconn(sock_upper_handle_t,
 151     sock_lower_handle_t, sock_downcalls_t *, struct cred *, pid_t,
 152     sock_upcalls_t **);
 153 extern void     so_set_prop(sock_upper_handle_t,
 154         struct sock_proto_props *);
 155 extern ssize_t  so_queue_msg(sock_upper_handle_t, mblk_t *, size_t, int,
 156     int *, boolean_t *);
 157 extern ssize_t  so_queue_msg_impl(struct sonode *, mblk_t *, size_t, int,
 158     int *, boolean_t *, struct sof_instance *);
 159 extern void     so_signal_oob(sock_upper_handle_t, ssize_t);

 161 extern void     so_connected(sock_upper_handle_t, sock_connid_t, struct cred *,
 162     pid_t);
 163 extern int      so_disconnected(sock_upper_handle_t, sock_connid_t, int);
 164 extern void     so_txq_full(sock_upper_handle_t, boolean_t);
```



```
 165 extern void     so_opctl(sock_upper_handle_t, sock_opctl_action_t, uintptr_t);
 166 extern mblk_t   *so_get_sock_pid_mblk(sock_upper_handle_t);
 167 #endif /* ! codereview */
 168 /* Common misc. functions */

 170         /* accept queue */
 171 extern int      so_acceptq_enqueue(struct sonode *, struct sonode *);
 172 extern int      so_acceptq_enqueue_locked(struct sonode *, struct sonode *);
 173 extern int      so_acceptq_dequeue(struct sonode *, boolean_t,
 174     struct sonode **);
 175 extern void     so_acceptq_flush(struct sonode *, boolean_t);

 177         /* connect */
 178 extern int      so_wait_connected(struct sonode *, boolean_t, sock_connid_t);

 180         /* send */
 181 extern int      so_snd_wait_qnotfull(struct sonode *, boolean_t);
 182 extern void     so_snd_qfull(struct sonode *so);
 183 extern void     so_snd_qnotfull(struct sonode *so);

 185 extern int      socket_chgpgrp(struct sonode *, pid_t);
 186 extern void     socket_sendsig(struct sonode *, int);
 187 extern int      so_dequeue_msg(struct sonode *, mblk_t **, struct uio *,
 188     rval_t *, int);
 189 extern void     so_enqueue_msg(struct sonode *, mblk_t *, size_t);
 190 extern void     so_process_new_message(struct sonode *, mblk_t *, mblk_t *);
 191 extern boolean_t        so_check_flow_control(struct sonode *);

 193 extern mblk_t   *socopyinuio(uio_t *, ssize_t, size_t, ssize_t, size_t, int *);
 194 extern mblk_t   *socopyoutuio(mblk_t *, struct uio *, ssize_t, int *);

 196 extern boolean_t somsghasdata(mblk_t *);
 197 extern void     so_rcv_flush(struct sonode *);
 198 extern int      sorecvoob(struct sonode *, struct nmsghdr *, struct uio *,
 199                     int, boolean_t);

 201 extern void     so_timer_callback(void *);

 203 extern struct sonode *socket_sonode_create(struct sockparams *, int, int, int,
 204     int, int, int *, struct cred *);

 206 extern void socket_sonode_destroy(struct sonode *);
 207 extern int socket_init_common(struct sonode *, struct sonode *, int flags,
 208     struct cred *);
 209 extern int socket_getopt_common(struct sonode *, int, int, void *, socklen_t *,
 210     int);
 211 extern int socket_ioctl_common(struct sonode *, int, intptr_t, int,
 212     struct cred *, int32_t *);
 213 extern int socket_strioc_common(struct sonode *, int, intptr_t, int,
 214     struct cred *, int32_t *);

 216 extern int so_zcopy_wait(struct sonode *);
 217 extern int so_get_mod_version(struct sockparams *);

 219 /* Notification functions */
 220 extern void     so_notify_connected(struct sonode *);
 221 extern void     so_notify_disconnecting(struct sonode *);
 222 extern void     so_notify_disconnected(struct sonode *, boolean_t, int);
 223 extern void     so_notify_writable(struct sonode *);
 224 extern void     so_notify_data(struct sonode *, size_t);
 225 extern void     so_notify_oobsig(struct sonode *);
 226 extern void     so_notify_oobdata(struct sonode *, boolean_t);
 227 extern void     so_notify_eof(struct sonode *);
 228 extern void     so_notify_newconn(struct sonode *);
 229 extern void     so_notify_shutdown(struct sonode *);
 230 extern void     so_notify_error(struct sonode *);
```

```
232 /* Common sonode functions */
233 extern int      sonode_constructor(void *, void *, int);
234 extern void     sonode_destructor(void *, void *);
235 extern void     sonode_init(struct sonode *, struct sockparams *,
236     int, int, int, sonodeops_t *);
237 extern void     sonode_fini(struct sonode *);
238 extern void     sonode_insert_pid(struct sonode *, pid_t);
239 extern void     sonode_remove_pid(struct sonode *, pid_t);
240 #endif /* ! codereview */

242 /*
243  * Event flags to socket_sendsig().
244  */
245 #define SOCKETSIG_WRITE 0x1
246 #define SOCKETSIG_READ  0x2
247 #define SOCKETSIG_URG   0x4

249 extern sonodeops_t so_sonodeops;
250 extern sock_upcalls_t so_upcalls;

252 #ifdef  __cplusplus
253 }
254 #endif
255 #endif /* _SOCKCOMMON_H_ */
```

```
**********************************************************
   49005 Mon Aug 17 21:08:04 2015
new/usr/src/uts/common/fs/sockfs/sockcommon_sops.c
XXXX adding PID information to netstat output
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */

  26 /*
  27  * Copyright (c) 2014, Joyent, Inc.  All rights reserved.
  28  */

  30 #include <sys/types.h>
  31 #include <sys/param.h>
  32 #include <sys/systm.h>
  33 #include <sys/sysmacros.h>
  34 #include <sys/debug.h>
  35 #include <sys/cmn_err.h>

  37 #include <sys/stropts.h>
  38 #include <sys/socket.h>
  39 #include <sys/socketvar.h>
  40 #include <sys/fcntl.h>
  41 #endif /* ! codereview */

  43 #define _SUN_TPI_VERSION        2
  44 #include <sys/tihdr.h>
  45 #include <sys/sockio.h>
  46 #include <sys/kmem_impl.h>

  48 #include <sys/strsubr.h>
  49 #include <sys/strsun.h>
  50 #include <sys/ddi.h>
  51 #include <netinet/in.h>
  52 #include <inet/ip.h>

  54 #include <fs/sockfs/sockcommon.h>
  55 #include <fs/sockfs/sockfilter_impl.h>

  57 #include <sys/socket_proto.h>

  59 #include <fs/sockfs/socktpi_impl.h>
  60 #include <fs/sockfs/sodirect.h>
  61 #include <sys/tihdr.h>
```



```
  62 #include <fs/sockfs/nl7c.h>

  64 extern int xnet_skip_checks;
  65 extern int xnet_check_print;

  67 static void so_queue_oob(struct sonode *, mblk_t *, size_t);


  70 /*ARGSUSED*/
  71 int
  72 so_accept_notsupp(struct sonode *lso, int fflag,
  73     struct cred *cr, struct sonode **nsop)
  74 {
  75         return (EOPNOTSUPP);
  76 }

  78 /*ARGSUSED*/
  79 int
  80 so_listen_notsupp(struct sonode *so, int backlog, struct cred *cr)
  81 {
  82         return (EOPNOTSUPP);
  83 }

  85 /*ARGSUSED*/
  86 int
  87 so_getsockname_notsupp(struct sonode *so, struct sockaddr *sa,
  88     socklen_t *len, struct cred *cr)
  89 {
  90         return (EOPNOTSUPP);
  91 }

  93 /*ARGSUSED*/
  94 int
  95 so_getpeername_notsupp(struct sonode *so, struct sockaddr *addr,
  96     socklen_t *addrlen, boolean_t accept, struct cred *cr)
  97 {
  98         return (EOPNOTSUPP);
  99 }

 101 /*ARGSUSED*/
 102 int
 103 so_shutdown_notsupp(struct sonode *so, int how, struct cred *cr)
 104 {
 105         return (EOPNOTSUPP);
 106 }

 108 /*ARGSUSED*/
 109 int
 110 so_sendmblk_notsupp(struct sonode *so, struct msghdr *msg, int fflag,
 111     struct cred *cr, mblk_t **mpp)
 112 {
 113         return (EOPNOTSUPP);
 114 }

 116 /*
 117  * Generic Socket Ops
 118  */

 120 /* ARGSUSED */
 121 int
 122 so_init(struct sonode *so, struct sonode *pso, struct cred *cr, int flags)
 123 {
 124         return (socket_init_common(so, pso, flags, cr));
 125 }

 127 int
```

```
128 so_bind(struct sonode *so, struct sockaddr *name, socklen_t namelen,
129     int flags, struct cred *cr)
130 {
131         int error;

133         SO_BLOCK_FALLBACK(so, SOP_BIND(so, name, namelen, flags, cr));

135         ASSERT(flags == _SOBIND_XPG4_2 || flags == _SOBIND_SOCKBSD);

137         /* X/Open requires this check */
138         if ((so->so_state & SS_CANTSENDMORE) && !xnet_skip_checks) {
139                 if (xnet_check_print) {
140                         printf("sockfs: X/Open bind state check "
141                             "caused EINVAL\n");
142                 }
143                 error = EINVAL;
144                 goto done;
145         }

147         /*
148          * a bind to a NULL address is interpreted as unbind. So just
149          * do the downcall.
150          */
151         if (name == NULL)
152                 goto dobind;

154         switch (so->so_family) {
155         case AF_INET:
156                 if ((size_t)namelen != sizeof (sin_t)) {
157                         error = name->sa_family != so->so_family ?
158                             EAFNOSUPPORT : EINVAL;
159                         eprintsoline(so, error);
160                         goto done;
161                 }

163                 if ((flags & _SOBIND_XPG4_2) &&
164                     (name->sa_family != so->so_family)) {
165                         /*
166                          * This check has to be made for X/Open
167                          * sockets however application failures have
168                          * been observed when it is applied to
169                          * all sockets.
170                          */
171                         error = EAFNOSUPPORT;
172                         eprintsoline(so, error);
173                         goto done;
174                 }
175                 /*
176                  * Force a zero sa_family to match so_family.
177                  *
178                  * Some programs like inetd(1M) don't set the
179                  * family field. Other programs leave
180                  * sin_family set to garbage - SunOS 4.X does
181                  * not check the family field on a bind.
182                  * We use the family field that
183                  * was passed in to the socket() call.
184                  */
185                 name->sa_family = so->so_family;
186                 break;

188         case AF_INET6: {
189 #ifdef DEBUG
190                 sin6_t *sin6 = (sin6_t *)name;
191 #endif
192                 if ((size_t)namelen != sizeof (sin6_t)) {
193                         error = name->sa_family != so->so_family ?
```



```
194                             EAFNOSUPPORT : EINVAL;
195                         eprintsoline(so, error);
196                         goto done;
197                 }

199                 if (name->sa_family != so->so_family) {
200                         /*
201                          * With IPv6 we require the family to match
202                          * unlike in IPv4.
203                          */
204                         error = EAFNOSUPPORT;
205                         eprintsoline(so, error);
206                         goto done;
207                 }
208 #ifdef DEBUG
209                 /*
210                  * Verify that apps don't forget to clear
211                  * sin6_scope_id etc
212                  */
213                 if (sin6->sin6_scope_id != 0 &&
214                     !IN6_IS_ADDR_LINKSCOPE(&sin6->sin6_addr)) {
215                         zcmn_err(getzoneid(), CE_WARN,
216                             "bind with uninitialized sin6_scope_id "
217                             "(%d) on socket. Pid = %d\n",
218                             (int)sin6->sin6_scope_id,
219                             (int)curproc->p_pid);
220                 }
221                 if (sin6->__sin6_src_id != 0) {
222                         zcmn_err(getzoneid(), CE_WARN,
223                             "bind with uninitialized __sin6_src_id "
224                             "(%d) on socket. Pid = %d\n",
225                             (int)sin6->__sin6_src_id,
226                             (int)curproc->p_pid);
227                 }
228 #endif /* DEBUG */

230                 break;
231         }
232         default:
233                 /* Just pass the request to the protocol */
234                 goto dobind;
235         }

237         /*
238          * First we check if either NCA or KSSL has been enabled for
239          * the requested address, and if so, we fall back to TPI.
240          * If neither of those two services are enabled, then we just
241          * pass the request to the protocol.
242          *
243          * Note that KSSL can only be enabled on a socket if NCA is NOT
244          * enabled for that socket, hence the else-statement below.
245          */
246         if (nl7c_enabled && ((so->so_family == AF_INET ||
247             so->so_family == AF_INET6) &&
248             nl7c_lookup_addr(name, namelen) != NULL)) {
249                 /*
250                  * NL7C is not supported in non-global zones,
251                  * we enforce this restriction here.
252                  */
253                 if (so->so_zoneid == GLOBAL_ZONEID) {
254                         /* NCA should be used, so fall back to TPI */
255                         error = so_tpi_fallback(so, cr);
256                         SO_UNBLOCK_FALLBACK(so);
257                         if (error)
258                                 return (error);
259                         else
```

```
260                                 return (SOP_BIND(so, name, namelen, flags, cr));
261                 }
262         }

264 dobind:
265         if (so->so_filter_active == 0 ||
266             (error = sof_filter_bind(so, name, &namelen, cr)) < 0) {
267                 error = (*so->so_downcalls->sd_bind)
268                     (so->so_proto_handle, name, namelen, cr);
269         }
270 done:
271         SO_UNBLOCK_FALLBACK(so);

273         return (error);
274 }

276 int
277 so_listen(struct sonode *so, int backlog, struct cred *cr)
278 {
279         int     error = 0;

281         ASSERT(MUTEX_NOT_HELD(&so->so_lock));
282         SO_BLOCK_FALLBACK(so, SOP_LISTEN(so, backlog, cr));

284         if ((so)->so_filter_active == 0 ||
285             (error = sof_filter_listen(so, &backlog, cr)) < 0)
286                 error = (*so->so_downcalls->sd_listen)(so->so_proto_handle,
287                     backlog, cr);

289         SO_UNBLOCK_FALLBACK(so);

291         return (error);
292 }


295 int
296 so_connect(struct sonode *so, struct sockaddr *name,
297     socklen_t namelen, int fflag, int flags, struct cred *cr)
298 {
299         int error = 0;
300         sock_connid_t id;

302         ASSERT(MUTEX_NOT_HELD(&so->so_lock));
303         SO_BLOCK_FALLBACK(so, SOP_CONNECT(so, name, namelen, fflag, flags, cr));

305         /*
306          * If there is a pending error, return error
307          * This can happen if a non blocking operation caused an error.
308          */

310         if (so->so_error != 0) {
311                 mutex_enter(&so->so_lock);
312                 error = sogeterr(so, B_TRUE);
313                 mutex_exit(&so->so_lock);
314                 if (error != 0)
315                         goto done;
316         }

318         if (so->so_filter_active == 0 ||
319             (error = sof_filter_connect(so, (struct sockaddr *)name,
320             &namelen, cr)) < 0) {
321                 error = (*so->so_downcalls->sd_connect)(so->so_proto_handle,
322                     name, namelen, &id, cr);

324                 if (error == EINPROGRESS)
325                         error = so_wait_connected(so,
```



```
326                             fflag & (FNONBLOCK|FNDELAY), id);
327         }
328 done:
329         SO_UNBLOCK_FALLBACK(so);
330         return (error);
331 }

333 /*ARGSUSED*/
334 int
335 so_accept(struct sonode *so, int fflag, struct cred *cr, struct sonode **nsop)
336 {
337         int error = 0;
338         struct sonode *nso;

340         *nsop = NULL;

342         SO_BLOCK_FALLBACK(so, SOP_ACCEPT(so, fflag, cr, nsop));
343         if ((so->so_state & SS_ACCEPTCONN) == 0) {
344                 SO_UNBLOCK_FALLBACK(so);
345                 return ((so->so_type == SOCK_DGRAM || so->so_type == SOCK_RAW) ?
346                     EOPNOTSUPP : EINVAL);
347         }

349         if ((error = so_acceptq_dequeue(so, (fflag & (FNONBLOCK|FNDELAY)),
350             &nso)) == 0) {
351                 ASSERT(nso != NULL);

353                 /* finish the accept */
354                 if ((so->so_filter_active > 0 &&
355                     (error = sof_filter_accept(nso, cr)) > 0) ||
356                     (error = (*so->so_downcalls->sd_accept)(so->so_proto_handle,
357                     nso->so_proto_handle, (sock_upper_handle_t)nso, cr)) != 0) {
358                         (void) socket_close(nso, 0, cr);
359                         socket_destroy(nso);
360                 } else {
361                         *nsop = nso;
362                         if (!(curproc->p_flag & SSYS))
363                                 sonode_insert_pid(nso, curproc->p_pidp->pid_id);
364 #endif /* ! codereview */
365                 }
366         }

368         SO_UNBLOCK_FALLBACK(so);
369         return (error);
370 }

372 int
373 so_sendmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
374     struct cred *cr)
375 {
376         int error, flags;
377         boolean_t dontblock;
378         ssize_t orig_resid;
379         mblk_t  *mp;

381         SO_BLOCK_FALLBACK(so, SOP_SENDMSG(so, msg, uiop, cr));

383         flags = msg->msg_flags;
384         error = 0;
385         dontblock = (flags & MSG_DONTWAIT) ||
386             (uiop->uio_fmode & (FNONBLOCK|FNDELAY));

388         if (!(flags & MSG_XPG4_2) && msg->msg_controllen != 0) {
389                 /*
390                  * Old way of passing fd's is not supported
391                  */
```

```
392                 SO_UNBLOCK_FALLBACK(so);
393                 return (EOPNOTSUPP);
394         }

396         if ((so->so_mode & SM_ATOMIC) &&
397             uiop->uio_resid > so->so_proto_props.sopp_maxpsz &&
398             so->so_proto_props.sopp_maxpsz != -1) {
399                 SO_UNBLOCK_FALLBACK(so);
400                 return (EMSGSIZE);
401         }

403         /*
404          * For atomic sends we will only do one iteration.
405          */
406         do {
407                 if (so->so_state & SS_CANTSENDMORE) {
408                         error = EPIPE;
409                         break;
410                 }

412                 if (so->so_error != 0) {
413                         mutex_enter(&so->so_lock);
414                         error = sogeterr(so, B_TRUE);
415                         mutex_exit(&so->so_lock);
416                         if (error != 0)
417                                 break;
418                 }

420                 /*
421                  * Send down OOB messages even if the send path is being
422                  * flow controlled (assuming the protocol supports OOB data).
423                  */
424                 if (flags & MSG_OOB) {
425                         if ((so->so_mode & SM_EXDATA) == 0) {
426                                 error = EOPNOTSUPP;
427                                 break;
428                         }
429                 } else if (SO_SND_FLOWCTRLD(so)) {
430                         /*
431                          * Need to wait until the protocol is ready to receive
432                          * more data for transmission.
433                          */
434                         if ((error = so_snd_wait_qnotfull(so, dontblock)) != 0)
435                                 break;
436                 }

438                 /*
439                  * Time to send data to the protocol. We either copy the
440                  * data into mblks or pass the uio directly to the protocol.
441                  * We decide what to do based on the available down calls.
442                  */
443                 if (so->so_downcalls->sd_send_uio != NULL) {
444                         error = (*so->so_downcalls->sd_send_uio)
445                             (so->so_proto_handle, uiop, msg, cr);
446                         if (error != 0)
447                                 break;
448                 } else {
449                         /* save the resid in case of failure */
450                         orig_resid = uiop->uio_resid;

452                         if ((mp = socopyinuio(uiop,
453                             so->so_proto_props.sopp_maxpsz,
454                             so->so_proto_props.sopp_wroff,
455                             so->so_proto_props.sopp_maxblk,
456                             so->so_proto_props.sopp_tail, &error)) == NULL) {
457                                 break;
```



```
458                         }
459                         ASSERT(uiop->uio_resid >= 0);

461                         if (so->so_filter_active > 0 &&
462                             ((mp = SOF_FILTER_DATA_OUT(so, mp, msg, cr,
463                             &error)) == NULL)) {
464                                 if (error != 0)
465                                         break;
466                                 continue;
467                         }
468                         error = (*so->so_downcalls->sd_send)
469                             (so->so_proto_handle, mp, msg, cr);
470                         if (error != 0) {
471                                 /*
472                                  * The send failed. We do not have to free the
473                                  * mblks, because that is the protocol's
474                                  * responsibility. However, uio_resid must
475                                  * remain accurate, so adjust that here.
476                                  */
477                                 uiop->uio_resid = orig_resid;
478                                         break;
479                         }
480                 }
481         } while (uiop->uio_resid > 0);

483         SO_UNBLOCK_FALLBACK(so);

485         return (error);
486 }

488 int
489 so_sendmblk_impl(struct sonode *so, struct nmsghdr *msg, int fflag,
490     struct cred *cr, mblk_t **mpp, sof_instance_t *fil,
491     boolean_t fil_inject)
492 {
493         int error;
494         boolean_t dontblock;
495         size_t size;
496         mblk_t *mp = *mpp;

498         if (so->so_downcalls->sd_send == NULL)
499                 return (EOPNOTSUPP);

501         error = 0;
502         dontblock = (msg->msg_flags & MSG_DONTWAIT) ||
503             (fflag & (FNONBLOCK|FNDELAY));
504         size = msgdsize(mp);

506         if ((so->so_mode & SM_ATOMIC) &&
507             size > so->so_proto_props.sopp_maxpsz &&
508             so->so_proto_props.sopp_maxpsz != -1) {
509                 SO_UNBLOCK_FALLBACK(so);
510                 return (EMSGSIZE);
511         }

513         while (mp != NULL) {
514                 mblk_t *nmp, *last_mblk;
515                 size_t mlen;

517                 if (so->so_state & SS_CANTSENDMORE) {
518                         error = EPIPE;
519                         break;
520                 }
521                 if (so->so_error != 0) {
522                         mutex_enter(&so->so_lock);
523                         error = sogeterr(so, B_TRUE);
```

```
524                         mutex_exit(&so->so_lock);
525                         if (error != 0)
526                                 break;
527                 }
528                 /* Socket filters are not flow controlled */
529                 if (SO_SND_FLOWCTRLD(so) && !fil_inject) {
530                         /*
531                          * Need to wait until the protocol is ready to receive
532                          * more data for transmission.
533                          */
534                         if ((error = so_snd_wait_qnotfull(so, dontblock)) != 0)
535                                 break;
536                 }
537
538                 /*
539                  * We only allow so_maxpsz of data to be sent down to
540                  * the protocol at time.
541                  */
542                 mlen = MBLKL(mp);
543                 nmp = mp->b_cont;
544                 last_mblk = mp;
545                 while (nmp != NULL) {
546                         mlen += MBLKL(nmp);
547                         if (mlen > so->so_proto_props.sopp_maxpsz) {
548                                 last_mblk->b_cont = NULL;
549                                 break;
550                         }
551                         last_mblk = nmp;
552                         nmp = nmp->b_cont;
553                 }
554
555                 if (so->so_filter_active > 0 &&
556                     (mp = SOF_FILTER_DATA_OUT_FROM(so, fil, mp, msg,
557                     cr, &error)) == NULL) {
558                         *mpp = mp = nmp;
559                         if (error != 0)
560                                 break;
561                         continue;
562                 }
563                 error = (*so->so_downcalls->sd_send)
564                     (so->so_proto_handle, mp, msg, cr);
565                 if (error != 0) {
566                         /*
567                          * The send failed. The protocol will free the mblks
568                          * that were sent down. Let the caller deal with the
569                          * rest.
570                          */
571                         *mpp = nmp;
572                         break;
573                 }
574
575                 *mpp = mp = nmp;
576         }
577         /* Let the filter know whether the protocol is flow controlled */
578         if (fil_inject && error == 0 && SO_SND_FLOWCTRLD(so))
579                 error = ENOSPC;
580
581         return (error);
582 }
583
584 #pragma inline(so_sendmblk_impl)
585
586 int
587 so_sendmblk(struct sonode *so, struct nmsghdr *msg, int fflag,
588     struct cred *cr, mblk_t **mpp)
589 {
```



```
590         int error;
591
592         SO_BLOCK_FALLBACK(so, SOP_SENDMBLK(so, msg, fflag, cr, mpp));
593
594         if ((so->so_mode & SM_SENDFILESUPP) == 0) {
595                 SO_UNBLOCK_FALLBACK(so);
596                 return (EOPNOTSUPP);
597         }
598
599         error = so_sendmblk_impl(so, msg, fflag, cr, mpp, so->so_filter_top,
600             B_FALSE);
601
602         SO_UNBLOCK_FALLBACK(so);
603
604         return (error);
605 }
606
607 int
608 so_shutdown(struct sonode *so, int how, struct cred *cr)
609 {
610         int error;
611
612         SO_BLOCK_FALLBACK(so, SOP_SHUTDOWN(so, how, cr));
613
614         /*
615          * SunOS 4.X has no check for datagram sockets.
616          * 5.X checks that it is connected (ENOTCONN)
617          * X/Open requires that we check the connected state.
618          */
619         if (!(so->so_state & SS_ISCONNECTED)) {
620                 if (!xnet_skip_checks) {
621                         error = ENOTCONN;
622                         if (xnet_check_print) {
623                                 printf("sockfs: X/Open shutdown check "
624                                     "caused ENOTCONN\n");
625                         }
626                 }
627                 goto done;
628         }
629
630         if (so->so_filter_active == 0 ||
631             (error = sof_filter_shutdown(so, &how, cr)) < 0)
632                 error = ((*so->so_downcalls->sd_shutdown)(so->so_proto_handle,
633                     how, cr));
634
635         /*
636          * Protocol agreed to shutdown. We need to flush the
637          * receive buffer if the receive side is being shutdown.
638          */
639         if (error == 0 && how != SHUT_WR) {
640                 mutex_enter(&so->so_lock);
641                 /* wait for active reader to finish */
642                 (void) so_lock_read(so, 0);
643
644                 so_rcv_flush(so);
645
646                 so_unlock_read(so);
647                 mutex_exit(&so->so_lock);
648         }
649
650 done:
651         SO_UNBLOCK_FALLBACK(so);
652         return (error);
653 }
654
655 int
```

```
656 so_getsockname(struct sonode *so, struct sockaddr *addr,
657     socklen_t *addrlen, struct cred *cr)
658 {
659         int error;

661         SO_BLOCK_FALLBACK(so, SOP_GETSOCKNAME(so, addr, addrlen, cr));

663         if (so->so_filter_active == 0 ||
664             (error = sof_filter_getsockname(so, addr, addrlen, cr)) < 0)
665                 error = (*so->so_downcalls->sd_getsockname)
666                     (so->so_proto_handle, addr, addrlen, cr);

668         SO_UNBLOCK_FALLBACK(so);
669         return (error);
670 }

672 int
673 so_getpeername(struct sonode *so, struct sockaddr *addr,
674     socklen_t *addrlen, boolean_t accept, struct cred *cr)
675 {
676         int error;

678         SO_BLOCK_FALLBACK(so, SOP_GETPEERNAME(so, addr, addrlen, accept, cr));

680         if (accept) {
681                 error = (*so->so_downcalls->sd_getpeername)
682                     (so->so_proto_handle, addr, addrlen, cr);
683         } else if (!(so->so_state & SS_ISCONNECTED)) {
684                 error = ENOTCONN;
685         } else if ((so->so_state & SS_CANTSENDMORE) && !xnet_skip_checks) {
686                 /* Added this check for X/Open */
687                 error = EINVAL;
688                 if (xnet_check_print) {
689                         printf("sockfs: X/Open getpeername check => EINVAL\n");
690                 }
691         } else if (so->so_filter_active == 0 ||
692             (error = sof_filter_getpeername(so, addr, addrlen, cr)) < 0) {
693                 error = (*so->so_downcalls->sd_getpeername)
694                     (so->so_proto_handle, addr, addrlen, cr);
695         }

697         SO_UNBLOCK_FALLBACK(so);
698         return (error);
699 }

701 int
702 so_getsockopt(struct sonode *so, int level, int option_name,
703     void *optval, socklen_t *optlenp, int flags, struct cred *cr)
704 {
705         int error = 0;

707         if (level == SOL_FILTER)
708                 return (sof_getsockopt(so, option_name, optval, optlenp, cr));

710         SO_BLOCK_FALLBACK(so,
711             SOP_GETSOCKOPT(so, level, option_name, optval, optlenp, flags, cr));

713         if ((so->so_filter_active == 0 ||
714             (error = sof_filter_getsockopt(so, level, option_name, optval,
715             optlenp, cr)) < 0) &&
716             (error = socket_getopt_common(so, level, option_name, optval,
717             optlenp, flags)) < 0) {
718                 error = (*so->so_downcalls->sd_getsockopt)
719                     (so->so_proto_handle, level, option_name, optval, optlenp,
720                     cr);
721                 if (error ==  ENOPROTOOPT) {
```



```
722                         if (level == SOL_SOCKET) {
723                                 /*
724                                  * If a protocol does not support a particular
725                                  * socket option, set can fail (not allowed)
726                                  * but get can not fail. This is the previous
727                                  * sockfs bahvior.
728                                  */
729                                 switch (option_name) {
730                                 case SO_LINGER:
731                                         if (*optlenp < (t_uscalar_t)
732                                             sizeof (struct linger)) {
733                                                 error = EINVAL;
734                                                 break;
735                                         }
736                                         error = 0;
737                                         bzero(optval, sizeof (struct linger));
738                                         *optlenp = sizeof (struct linger);
739                                         break;
740                                 case SO_RCVTIMEO:
741                                 case SO_SNDTIMEO:
742                                         if (*optlenp < (t_uscalar_t)
743                                             sizeof (struct timeval)) {
744                                                 error = EINVAL;
745                                                 break;
746                                         }
747                                         error = 0;
748                                         bzero(optval, sizeof (struct timeval));
749                                         *optlenp = sizeof (struct timeval);
750                                         break;
751                                 case SO_SND_BUFINFO:
752                                         if (*optlenp < (t_uscalar_t)
753                                             sizeof (struct so_snd_bufinfo)) {
754                                                 error = EINVAL;
755                                                 break;
756                                         }
757                                         error = 0;
758                                         bzero(optval,
759                                             sizeof (struct so_snd_bufinfo));
760                                         *optlenp =
761                                             sizeof (struct so_snd_bufinfo);
762                                         break;
763                                 case SO_DEBUG:
764                                 case SO_REUSEADDR:
765                                 case SO_KEEPALIVE:
766                                 case SO_DONTROUTE:
767                                 case SO_BROADCAST:
768                                 case SO_USELOOPBACK:
769                                 case SO_OOBINLINE:
770                                 case SO_DGRAM_ERRIND:
771                                 case SO_SNDBUF:
772                                 case SO_RCVBUF:
773                                         error = 0;
774                                         *((int32_t *)optval) = 0;
775                                         *optlenp = sizeof (int32_t);
776                                         break;
777                                 default:
778                                         break;
779                                 }
780                         }
781                 }
782         }

784         SO_UNBLOCK_FALLBACK(so);
785         return (error);
786 }
```

```
788 int
789 so_setsockopt(struct sonode *so, int level, int option_name,
790     const void *optval, socklen_t optlen, struct cred *cr)
791 {
792         int error = 0;
793         struct timeval tl;
794         const void *opt = optval;

796         if (level == SOL_FILTER)
797                 return (sof_setsockopt(so, option_name, optval, optlen, cr));

799         SO_BLOCK_FALLBACK(so,
800             SOP_SETSOCKOPT(so, level, option_name, optval, optlen, cr));

802         /* X/Open requires this check */
803         if (so->so_state & SS_CANTSENDMORE && !xnet_skip_checks) {
804                 SO_UNBLOCK_FALLBACK(so);
805                 if (xnet_check_print)
806                         printf("sockfs: X/Open setsockopt check => EINVAL\n");
807                 return (EINVAL);
808         }

810         if (so->so_filter_active > 0 &&
811             (error = sof_filter_setsockopt(so, level, option_name,
812             (void *)optval, &optlen, cr)) >= 0)
813                 goto done;

815         if (level == SOL_SOCKET) {
816                 switch (option_name) {
817                 case SO_RCVTIMEO:
818                 case SO_SNDTIMEO: {
819                         /*
820                          * We pass down these two options to protocol in order
821                          * to support some third part protocols which need to
822                          * know them. For those protocols which don't care
823                          * these two options, simply return 0.
824                          */
825                         clock_t t_usec;

827                         if (get_udatamodel() == DATAMODEL_NONE ||
828                             get_udatamodel() == DATAMODEL_NATIVE) {
829                                 if (optlen != sizeof (struct timeval)) {
830                                         error = EINVAL;
831                                         goto done;
832                                 }
833                                 bcopy((struct timeval *)optval, &tl,
834                                     sizeof (struct timeval));
835                         } else {
836                                 if (optlen != sizeof (struct timeval32)) {
837                                         error = EINVAL;
838                                         goto done;
839                                 }
840                                 TIMEVAL32_TO_TIMEVAL(&tl,
841                                     (struct timeval32 *)optval);
842                         }
843                         opt = &tl;
844                         optlen = sizeof (tl);
845                         t_usec = tl.tv_sec * 1000 * 1000 + tl.tv_usec;
846                         mutex_enter(&so->so_lock);
847                         if (option_name == SO_RCVTIMEO)
848                                 so->so_rcvtimeo = drv_usectohz(t_usec);
849                         else
850                                 so->so_sndtimeo = drv_usectohz(t_usec);
851                         mutex_exit(&so->so_lock);
852                         break;
853                 }
```



```
854                 case SO_RCVBUF:
855                         /*
856                          * XXX XPG 4.2 applications retrieve SO_RCVBUF from
857                          * sockfs since the transport might adjust the value
858                          * and not return exactly what was set by the
859                          * application.
860                          */
861                         so->so_xpg_rcvbuf = *(int32_t *)optval;
862                         break;
863                 }
864         }
865         error = (*so->so_downcalls->sd_setsockopt)
866             (so->so_proto_handle, level, option_name, opt, optlen, cr);
867 done:
868         SO_UNBLOCK_FALLBACK(so);
869         return (error);
870 }

872 int
873 so_ioctl(struct sonode *so, int cmd, intptr_t arg, int mode,
874     struct cred *cr, int32_t *rvalp)
875 {
876         int error = 0;

878         SO_BLOCK_FALLBACK(so, SOP_IOCTL(so, cmd, arg, mode, cr, rvalp));

880         /*
881          * If there is a pending error, return error
882          * This can happen if a non blocking operation caused an error.
883          */
884         if (so->so_error != 0) {
885                 mutex_enter(&so->so_lock);
886                 error = sogeterr(so, B_TRUE);
887                 mutex_exit(&so->so_lock);
888                 if (error != 0)
889                         goto done;
890         }

892         /*
893          * calling strioc can result in the socket falling back to TPI,
894          * if that is supported.
895          */
896         if ((so->so_filter_active == 0 ||
897             (error = sof_filter_ioctl(so, cmd, arg, mode,
898             rvalp, cr)) < 0) &&
899             (error = socket_ioctl_common(so, cmd, arg, mode, cr, rvalp)) < 0 &&
900             (error = socket_strioc_common(so, cmd, arg, mode, cr, rvalp)) < 0) {
901                 error = (*so->so_downcalls->sd_ioctl)(so->so_proto_handle,
902                     cmd, arg, mode, rvalp, cr);
903         }

905 done:
906         SO_UNBLOCK_FALLBACK(so);

908         return (error);
909 }

911 int
912 so_poll(struct sonode *so, short events, int anyyet, short *reventsp,
913     struct pollhead **phpp)
914 {
915         int state = so->so_state, mask;
916         *reventsp = 0;

918         /*
919          * In sockets the errors are represented as input/output events
```

```
920          */
921         if (so->so_error != 0 &&
922             ((POLLIN|POLLRDNORM|POLLOUT) & events) != 0) {
923                 *reventsp = (POLLIN|POLLRDNORM|POLLOUT) & events;
924                 return (0);
925         }
926
927         /*
928          * If the socket is in a state where it can send data
929          * turn on POLLWRBAND and POLLOUT events.
930          */
931         if ((so->so_mode & SM_CONNREQUIRED) == 0 || (state & SS_ISCONNECTED)) {
932                 /*
933                  * out of band data is allowed even if the connection
934                  * is flow controlled
935                  */
936                 *reventsp |= POLLWRBAND & events;
937                 if (!SO_SND_FLOWCTRLD(so)) {
938                         /*
939                          * As long as there is buffer to send data
940                          * turn on POLLOUT events
941                          */
942                         *reventsp |= POLLOUT & events;
943                 }
944         }
945
946         /*
947          * Turn on POLLIN whenever there is data on the receive queue,
948          * or the socket is in a state where no more data will be received.
949          * Also, if the socket is accepting connections, flip the bit if
950          * there is something on the queue.
951          *
952          * We do an initial check for events without holding locks. However,
953          * if there are no event available, then we redo the check for POLLIN
954          * events under the lock.
955          */
956
957         /* Pending connections */
958         if (!list_is_empty(&so->so_acceptq_list))
959                 *reventsp |= (POLLIN|POLLRDNORM) & events;
960
961         /* Data */
962         /* so_downcalls is null for sctp */
963         if (so->so_downcalls != NULL && so->so_downcalls->sd_poll != NULL) {
964                 *reventsp |= (*so->so_downcalls->sd_poll)
965                     (so->so_proto_handle, events & SO_PROTO_POLLEV, anyyet,
966                     CRED()) & events;
967                 ASSERT((*reventsp & ~events) == 0);
968                 /* do not recheck events */
969                 events &= ~SO_PROTO_POLLEV;
970         } else {
971                 if (SO_HAVE_DATA(so))
972                         *reventsp |= (POLLIN|POLLRDNORM) & events;
973
974                 /* Urgent data */
975                 if ((state & SS_OOBPEND) != 0) {
976                         *reventsp |= (POLLRDBAND | POLLPRI) & events;
977                 }
978
979                 /*
980                  * If the socket has become disconnected, we set POLLHUP.
981                  * Note that if we are in this state, we will have set POLLIN
982                  * (SO_HAVE_DATA() is true on a disconnected socket), but not
983                  * POLLOUT (SS_ISCONNECTED is false).  This is in keeping with
984                  * the semantics of POLLHUP, which is defined to be mutually
985                  * exclusive with respect to POLLOUT but not POLLIN.  We are
```



```
986                  * therefore setting POLLHUP primarily for the benefit of
987                  * those not polling on POLLIN, as they have no other way of
988                  * knowing that the socket has been disconnected.
989                  */
990                 mask = SS_SENTLASTREADSIG | SS_SENTLASTWRITESIG;
991
992                 if ((state & (mask | SS_ISCONNECTED)) == mask)
993                         *reventsp |= POLLHUP;
994         }
995
996         if (!*reventsp && !anyyet) {
997                 /* Check for read events again, but this time under lock */
998                 if (events & (POLLIN|POLLRDNORM)) {
999                         mutex_enter(&so->so_lock);
1000                        if (SO_HAVE_DATA(so) ||
1001                            !list_is_empty(&so->so_acceptq_list)) {
1002                                mutex_exit(&so->so_lock);
1003                                *reventsp |= (POLLIN|POLLRDNORM) & events;
1004                                return (0);
1005                        } else {
1006                                so->so_pollev |= SO_POLLEV_IN;
1007                                mutex_exit(&so->so_lock);
1008                        }
1009                }
1010                *phpp = &so->so_poll_list;
1011        }
1012        return (0);
1013 }
1014
1015 /*
1016  * Generic Upcalls
1017  */
1018 void
1019 so_connected(sock_upper_handle_t sock_handle, sock_connid_t id,
1020     cred_t *peer_cred, pid_t peer_cpid)
1021 {
1022         struct sonode *so = (struct sonode *)sock_handle;
1023
1024         mutex_enter(&so->so_lock);
1025         ASSERT(so->so_proto_handle != NULL);
1026
1027         if (peer_cred != NULL) {
1028                 if (so->so_peercred != NULL)
1029                         crfree(so->so_peercred);
1030                 crhold(peer_cred);
1031                 so->so_peercred = peer_cred;
1032                 so->so_cpid = peer_cpid;
1033         }
1034
1035         so->so_proto_connid = id;
1036         soisconnected(so);
1037         /*
1038          * Wake ones who're waiting for conn to become established.
1039          */
1040         so_notify_connected(so);
1041 }
1042
1043 int
1044 so_disconnected(sock_upper_handle_t sock_handle, sock_connid_t id, int error)
1045 {
1046         struct sonode *so = (struct sonode *)sock_handle;
1047         boolean_t connect_failed;
1048
1049         mutex_enter(&so->so_lock);
1050
1051         /*
```

```
1052          * If we aren't currently connected, then this isn't a disconnect but
1053          * rather a failure to connect.
1054          */
1055         connect_failed = !(so->so_state & SS_ISCONNECTED);

1057         so->so_proto_connid = id;
1058         soisdisconnected(so, error);
1059         so_notify_disconnected(so, connect_failed, error);

1061         return (0);
1062 }

1064 void
1065 so_opctl(sock_upper_handle_t sock_handle, sock_opctl_action_t action,
1066     uintptr_t arg)
1067 {
1068         struct sonode *so = (struct sonode *)sock_handle;

1070         switch (action) {
1071         case SOCK_OPCTL_SHUT_SEND:
1072                 mutex_enter(&so->so_lock);
1073                 socantsendmore(so);
1074                 so_notify_disconnecting(so);
1075                 break;
1076         case SOCK_OPCTL_SHUT_RECV: {
1077                 mutex_enter(&so->so_lock);
1078                 socantrcvmore(so);
1079                 so_notify_eof(so);
1080                 break;
1081         }
1082         case SOCK_OPCTL_ENAB_ACCEPT:
1083                 mutex_enter(&so->so_lock);
1084                 so->so_state |= SS_ACCEPTCONN;
1085                 so->so_backlog = (unsigned int)arg;
1086                 /*
1087                  * The protocol can stop generating newconn upcalls when
1088                  * the backlog is full, so to make sure the listener does
1089                  * not end up with a queue full of deferred connections
1090                  * we reduce the backlog by one. Thus the listener will
1091                  * start closing deferred connections before the backlog
1092                  * is full.
1093                  */
1094                 if (so->so_filter_active > 0)
1095                         so->so_backlog = MAX(1, so->so_backlog - 1);
1096                 mutex_exit(&so->so_lock);
1097                 break;
1098         default:
1099                 ASSERT(0);
1100                 break;
1101         }
1102 }

1104 void
1105 so_txq_full(sock_upper_handle_t sock_handle, boolean_t qfull)
1106 {
1107         struct sonode *so = (struct sonode *)sock_handle;

1109         if (qfull) {
1110                 so_snd_qfull(so);
1111         } else {
1112                 so_snd_qnotfull(so);
1113                 mutex_enter(&so->so_lock);
1114                 /* so_notify_writable drops so_lock */
1115                 so_notify_writable(so);
1116         }
1117 }
```



```
1119 sock_upper_handle_t
1120 so_newconn(sock_upper_handle_t parenthandle,
1121     sock_lower_handle_t proto_handle, sock_downcalls_t *sock_downcalls,
1122     struct cred *peer_cred, pid_t peer_cpid, sock_upcalls_t **sock_upcallsp)
1123 {
1124         struct sonode   *so = (struct sonode *)parenthandle;
1125         struct sonode   *nso;
1126         int error;

1128         ASSERT(proto_handle != NULL);

1130         if ((so->so_state & SS_ACCEPTCONN) == 0 ||
1131             (so->so_acceptq_len >= so->so_backlog &&
1132             (so->so_filter_active == 0 || !sof_sonode_drop_deferred(so)))) {
1133                         return (NULL);
1134         }

1136         nso = socket_newconn(so, proto_handle, sock_downcalls, SOCKET_NOSLEEP,
1137             &error);
1138         if (nso == NULL)
1139                 return (NULL);

1141         if (peer_cred != NULL) {
1142                 crhold(peer_cred);
1143                 nso->so_peercred = peer_cred;
1144                 nso->so_cpid = peer_cpid;
1145         }
1146         nso->so_listener = so;

1148         /*
1149          * The new socket (nso), proto_handle and sock_upcallsp are all
1150          * valid at this point. But as soon as nso is placed in the accept
1151          * queue that can no longer be assumed (since an accept() thread may
1152          * pull it off the queue and close the socket).
1153          */
1154         *sock_upcallsp = &so_upcalls;

1156         mutex_enter(&so->so_acceptq_lock);
1157         if (so->so_state & (SS_CLOSING|SS_FALLBACK_PENDING|SS_FALLBACK_COMP)) {
1158                 mutex_exit(&so->so_acceptq_lock);
1159                 ASSERT(nso->so_count == 1);
1160                 nso->so_count--;
1161                 nso->so_listener = NULL;
1162                 /* drop proto ref */
1163                 VN_RELE(SOTOV(nso));
1164                 socket_destroy(nso);
1165                 return (NULL);
1166         } else {
1167                 so->so_acceptq_len++;
1168                 if (nso->so_state & SS_FIL_DEFER) {
1169                         list_insert_tail(&so->so_acceptq_defer, nso);
1170                         mutex_exit(&so->so_acceptq_lock);
1171                 } else {
1172                         list_insert_tail(&so->so_acceptq_list, nso);
1173                         cv_signal(&so->so_acceptq_cv);
1174                         mutex_exit(&so->so_acceptq_lock);
1175                         mutex_enter(&so->so_lock);
1176                         so_notify_newconn(so);
1177                 }

1179                 return ((sock_upper_handle_t)nso);
1180         }
1181 }

1183 void
```

```
1184 so_set_prop(sock_upper_handle_t sock_handle, struct sock_proto_props *soppp)
1185 {
1186         struct sonode *so;

1188         so = (struct sonode *)sock_handle;

1190         mutex_enter(&so->so_lock);

1192         if (soppp->sopp_flags & SOCKOPT_MAXBLK)
1193                 so->so_proto_props.sopp_maxblk = soppp->sopp_maxblk;
1194         if (soppp->sopp_flags & SOCKOPT_WROFF)
1195                 so->so_proto_props.sopp_wroff = soppp->sopp_wroff;
1196         if (soppp->sopp_flags & SOCKOPT_TAIL)
1197                 so->so_proto_props.sopp_tail = soppp->sopp_tail;
1198         if (soppp->sopp_flags & SOCKOPT_RCVHIWAT)
1199                 so->so_proto_props.sopp_rxhiwat = soppp->sopp_rxhiwat;
1200         if (soppp->sopp_flags & SOCKOPT_RCVLOWAT)
1201                 so->so_proto_props.sopp_rxlowat = soppp->sopp_rxlowat;
1202         if (soppp->sopp_flags & SOCKOPT_MAXPSZ)
1203                 so->so_proto_props.sopp_maxpsz = soppp->sopp_maxpsz;
1204         if (soppp->sopp_flags & SOCKOPT_MINPSZ)
1205                 so->so_proto_props.sopp_minpsz = soppp->sopp_minpsz;
1206         if (soppp->sopp_flags & SOCKOPT_ZCOPY) {
1207                 if (soppp->sopp_zcopyflag & ZCVMSAFE) {
1208                         so->so_proto_props.sopp_zcopyflag |= STZCVMSAFE;
1209                         so->so_proto_props.sopp_zcopyflag &= ~STZCVMUNSAFE;
1210                 } else if (soppp->sopp_zcopyflag & ZCVMUNSAFE) {
1211                         so->so_proto_props.sopp_zcopyflag |= STZCVMUNSAFE;
1212                         so->so_proto_props.sopp_zcopyflag &= ~STZCVMSAFE;
1213                 }

1215                 if (soppp->sopp_zcopyflag & COPYCACHED) {
1216                         so->so_proto_props.sopp_zcopyflag |= STRCOPYCACHED;
1217                 }
1218         }
1219         if (soppp->sopp_flags & SOCKOPT_OOBINLINE)
1220                 so->so_proto_props.sopp_oobinline = soppp->sopp_oobinline;
1221         if (soppp->sopp_flags & SOCKOPT_RCVTIMER)
1222                 so->so_proto_props.sopp_rcvtimer = soppp->sopp_rcvtimer;
1223         if (soppp->sopp_flags & SOCKOPT_RCVTHRESH)
1224                 so->so_proto_props.sopp_rcvthresh = soppp->sopp_rcvthresh;
1225         if (soppp->sopp_flags & SOCKOPT_MAXADDRLEN)
1226                 so->so_proto_props.sopp_maxaddrlen = soppp->sopp_maxaddrlen;
1227         if (soppp->sopp_flags & SOCKOPT_LOOPBACK)
1228                 so->so_proto_props.sopp_loopback = soppp->sopp_loopback;

1230         mutex_exit(&so->so_lock);

1232         if (so->so_filter_active > 0) {
1233                 sof_instance_t *inst;
1234                 ssize_t maxblk;
1235                 ushort_t wroff, tail;
1236                 maxblk = so->so_proto_props.sopp_maxblk;
1237                 wroff = so->so_proto_props.sopp_wroff;
1238                 tail = so->so_proto_props.sopp_tail;
1239                 for (inst = so->so_filter_bottom; inst != NULL;
1240                     inst = inst->sofi_prev) {
1241                         if (SOF_INTERESTED(inst, mblk_prop)) {
1242                                 (*inst->sofi_ops->sofop_mblk_prop)(
1243                                     (sof_handle_t)inst, inst->sofi_cookie,
1244                                     &maxblk, &wroff, &tail);
1245                         }
1246                 }
1247                 mutex_enter(&so->so_lock);
1248                 so->so_proto_props.sopp_maxblk = maxblk;
1249                 so->so_proto_props.sopp_wroff = wroff;
```



```
1250                 so->so_proto_props.sopp_tail = tail;
1251                 mutex_exit(&so->so_lock);
1252         }
1253 #ifdef DEBUG
1254         soppp->sopp_flags &= ~(SOCKOPT_MAXBLK | SOCKOPT_WROFF | SOCKOPT_TAIL |
1255             SOCKOPT_RCVHIWAT | SOCKOPT_RCVLOWAT | SOCKOPT_MAXPSZ |
1256             SOCKOPT_ZCOPY | SOCKOPT_OOBINLINE | SOCKOPT_RCVTIMER |
1257             SOCKOPT_RCVTHRESH | SOCKOPT_MAXADDRLEN | SOCKOPT_MINPSZ |
1258             SOCKOPT_LOOPBACK);
1259         ASSERT(soppp->sopp_flags == 0);
1260 #endif
1261 }

1263 /* ARGSUSED */
1264 ssize_t
1265 so_queue_msg_impl(struct sonode *so, mblk_t *mp,
1266     size_t msg_size, int flags, int *errorp,  boolean_t *force_pushp,
1267     sof_instance_t *filter)
1268 {
1269         boolean_t force_push = B_TRUE;
1270         int space_left;
1271         sodirect_t *sodp = so->so_direct;

1273         ASSERT(errorp != NULL);
1274         *errorp = 0;
1275         if (mp == NULL) {
1276                 if (so->so_downcalls->sd_recv_uio != NULL) {
1277                         mutex_enter(&so->so_lock);
1278                         /* the notify functions will drop the lock */
1279                         if (flags & MSG_OOB)
1280                                 so_notify_oobdata(so, IS_SO_OOB_INLINE(so));
1281                         else
1282                                 so_notify_data(so, msg_size);
1283                         return (0);
1284                 }
1285                 ASSERT(msg_size == 0);
1286                 mutex_enter(&so->so_lock);
1287                 goto space_check;
1288         }

1290         ASSERT(mp->b_next == NULL);
1291         ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_PROTO);
1292         ASSERT(msg_size == msgdsize(mp));

1294         if (DB_TYPE(mp) == M_PROTO && !__TPI_PRIM_ISALIGNED(mp->b_rptr)) {
1295                 /* The read pointer is not aligned correctly for TPI */
1296                 zcmn_err(getzoneid(), CE_WARN,
1297                     "sockfs: Unaligned TPI message received. rptr = %p\n",
1298                     (void *)mp->b_rptr);
1299                 freemsg(mp);
1300                 mutex_enter(&so->so_lock);
1301                 if (sodp != NULL)
1302                         SOD_UIOAFINI(sodp);
1303                 goto space_check;
1304         }

1306         if (so->so_filter_active > 0) {
1307                 for (; filter != NULL; filter = filter->sofi_prev) {
1308                         if (!SOF_INTERESTED(filter, data_in))
1309                                 continue;
1310                         mp = (*filter->sofi_ops->sofop_data_in)(
1311                             (sof_handle_t)filter, filter->sofi_cookie, mp,
1312                             flags, &msg_size);
1313                         ASSERT(msgdsize(mp) == msg_size);
1314                         DTRACE_PROBE2(filter__data, (sof_instance_t), filter,
1315                             (mblk_t *), mp);
```

```
1316                        /* Data was consumed/dropped, just do space check */
1317                        if (msg_size == 0) {
1318                                mutex_enter(&so->so_lock);
1319                                goto space_check;
1320                        }
1321                }
1322        }

1324        if (flags & MSG_OOB) {
1325                so_queue_oob(so, mp, msg_size);
1326                mutex_enter(&so->so_lock);
1327                goto space_check;
1328        }

1330        if (force_pushp != NULL)
1331                force_push = *force_pushp;

1333        mutex_enter(&so->so_lock);
1334        if (so->so_state & (SS_FALLBACK_DRAIN | SS_FALLBACK_COMP)) {
1335                if (sodp != NULL)
1336                        SOD_DISABLE(sodp);
1337                mutex_exit(&so->so_lock);
1338                *errorp = EOPNOTSUPP;
1339                return (-1);
1340        }
1341        if (so->so_state & (SS_CANTRCVMORE | SS_CLOSING)) {
1342                freemsg(mp);
1343                if (sodp != NULL)
1344                        SOD_DISABLE(sodp);
1345                mutex_exit(&so->so_lock);
1346                return (0);
1347        }

1349        /* process the mblk via I/OAT if capable */
1350        if (sodp != NULL && sodp->sod_enabled) {
1351                if (DB_TYPE(mp) == M_DATA) {
1352                        sod_uioa_mblk_init(sodp, mp, msg_size);
1353                } else {
1354                        SOD_UIOAFINI(sodp);
1355                }
1356        }

1358        if (mp->b_next == NULL) {
1359                so_enqueue_msg(so, mp, msg_size);
1360        } else {
1361                do {
1362                        mblk_t *nmp;

1364                        if ((nmp = mp->b_next) != NULL) {
1365                                mp->b_next = NULL;
1366                        }
1367                        so_enqueue_msg(so, mp, msgdsize(mp));
1368                        mp = nmp;
1369                } while (mp != NULL);
1370        }

1372        space_left = so->so_rcvbuf - so->so_rcv_queued;
1373        if (space_left <= 0) {
1374                so->so_flowctrld = B_TRUE;
1375                *errorp = ENOSPC;
1376                space_left = -1;
1377        }

1379        if (force_push || so->so_rcv_queued >= so->so_rcv_thresh ||
1380            so->so_rcv_queued >= so->so_rcv_wanted) {
1381                SOCKET_TIMER_CANCEL(so);
```



```
1382                /*
1383                 * so_notify_data will release the lock
1384                 */
1385                so_notify_data(so, so->so_rcv_queued);

1387                if (force_pushp != NULL)
1388                        *force_pushp = B_TRUE;
1389                goto done;
1390        } else if (so->so_rcv_timer_tid == 0) {
1391                /* Make sure the recv push timer is running */
1392                SOCKET_TIMER_START(so);
1393        }

1395 done_unlock:
1396        mutex_exit(&so->so_lock);
1397 done:
1398        return (space_left);

1400 space_check:
1401        space_left = so->so_rcvbuf - so->so_rcv_queued;
1402        if (space_left <= 0) {
1403                so->so_flowctrld = B_TRUE;
1404                *errorp = ENOSPC;
1405                space_left = -1;
1406        }
1407        goto done_unlock;
1408 }

1410 #pragma inline(so_queue_msg_impl)

1412 ssize_t
1413 so_queue_msg(sock_upper_handle_t sock_handle, mblk_t *mp,
1414     size_t msg_size, int flags, int *errorp,  boolean_t *force_pushp)
1415 {
1416        struct sonode *so = (struct sonode *)sock_handle;

1418        return (so_queue_msg_impl(so, mp, msg_size, flags, errorp, force_pushp,
1419            so->so_filter_bottom));
1420 }

1422 /*
1423  * Set the offset of where the oob data is relative to the bytes in
1424  * queued. Also generate SIGURG
1425  */
1426 void
1427 so_signal_oob(sock_upper_handle_t sock_handle, ssize_t offset)
1428 {
1429        struct sonode *so;

1431        ASSERT(offset >= 0);
1432        so = (struct sonode *)sock_handle;
1433        mutex_enter(&so->so_lock);
1434        if (so->so_direct != NULL)
1435                SOD_UIOAFINI(so->so_direct);

1437        /*
1438         * New urgent data on the way so forget about any old
1439         * urgent data.
1440         */
1441        so->so_state &= ~(SS_HAVEOOBDATA|SS_HADOOBDATA);

1443        /*
1444         * Record that urgent data is pending.
1445         */
1446        so->so_state |= SS_OOBPEND;
```

```
1448         if (so->so_oobmsg != NULL) {
1449                 dprintso(so, 1, ("sock: discarding old oob\n"));
1450                 freemsg(so->so_oobmsg);
1451                 so->so_oobmsg = NULL;
1452         }
1454         /*
1455          * set the offset where the urgent byte is
1456          */
1457         so->so_oobmark = so->so_rcv_queued + offset;
1458         if (so->so_oobmark == 0)
1459                 so->so_state |= SS_RCVATMARK;
1460         else
1461                 so->so_state &= ~SS_RCVATMARK;
1463         so_notify_oobsig(so);
1464 }
1466 /*
1467  * Queue the OOB byte
1468  */
1469 static void
1470 so_queue_oob(struct sonode *so, mblk_t *mp, size_t len)
1471 {
1472         mutex_enter(&so->so_lock);
1473         if (so->so_direct != NULL)
1474                 SOD_UIOAFINI(so->so_direct);
1476         ASSERT(mp != NULL);
1477         if (!IS_SO_OOB_INLINE(so)) {
1478                 so->so_oobmsg = mp;
1479                 so->so_state |= SS_HAVEOOBDATA;
1480         } else {
1481                 so_enqueue_msg(so, mp, len);
1482         }
1484         so_notify_oobdata(so, IS_SO_OOB_INLINE(so));
1485 }
1487 int
1488 so_close(struct sonode *so, int flag, struct cred *cr)
1489 {
1490         int error;
1492         /*
1493          * No new data will be enqueued once the CLOSING flag is set.
1494          */
1495         mutex_enter(&so->so_lock);
1496         so->so_state |= SS_CLOSING;
1497         ASSERT(so_verify_oobstate(so));
1498         so_rcv_flush(so);
1499         mutex_exit(&so->so_lock);
1501         if (so->so_filter_active > 0)
1502                 sof_sonode_closing(so);
1504         if (so->so_state & SS_ACCEPTCONN) {
1505                 /*
1506                  * We grab and release the accept lock to ensure that any
1507                  * thread about to insert a socket in so_newconn completes
1508                  * before we flush the queue. Any thread calling so_newconn
1509                  * after we drop the lock will observe the SS_CLOSING flag,
1510                  * which will stop it from inserting the socket in the queue.
1511                  */
1512                 mutex_enter(&so->so_acceptq_lock);
1513                 mutex_exit(&so->so_acceptq_lock);
```



```
1515                 so_acceptq_flush(so, B_TRUE);
1516         }
1518         error = (*so->so_downcalls->sd_close)(so->so_proto_handle, flag, cr);
1519         switch (error) {
1520         default:
1521                 /* Protocol made a synchronous close; remove proto ref */
1522                 VN_RELE(SOTOV(so));
1523                 break;
1524         case EINPROGRESS:
1525                 /*
1526                  * Protocol is in the process of closing, it will make a
1527                  * 'closed' upcall to remove the reference.
1528                  */
1529                 error = 0;
1530                 break;
1531         }
1533         return (error);
1534 }
1536 /*
1537  * Upcall made by the protocol when it's doing an asynchronous close. It
1538  * will drop the protocol's reference on the socket.
1539  */
1540 void
1541 so_closed(sock_upper_handle_t sock_handle)
1542 {
1543         struct sonode *so = (struct sonode *)sock_handle;
1545         VN_RELE(SOTOV(so));
1546 }
1548 mblk_t *
1549 so_get_sock_pid_mblk(sock_upper_handle_t sock_handle)
1550 {
1551         int sz, n = 0;
1552         mblk_t *mblk;
1553         pid_node_t *pn;
1554         pid_t *pids;
1555         conn_pid_info_t *cpi;
1556         struct sonode *so = (struct sonode *)sock_handle;
1558         mutex_enter(&so->so_pid_list_lock);
1560         n = list_numnodes(&so->so_pid_list);
1561         sz = sizeof (conn_pid_info_t);
1562         sz += (n > 1) ? ((n - 1) * sizeof (pid_t)) : 0;
1563         if ((mblk = allocb(sz, BPRI_HI)) == NULL) {
1564                 mutex_exit(&so->so_pid_list_lock);
1565                 return (NULL);
1566         }
1567         mblk->b_wptr += sz;
1568         cpi = (conn_pid_info_t *)mblk->b_datap->db_base;
1570         cpi->cpi_magic = CONN_PID_INFO_MGC;
1571         cpi->cpi_contents = CONN_PID_INFO_SOC;
1572         cpi->cpi_pids_cnt = n;
1573         cpi->cpi_tot_size = sz;
1574         cpi->cpi_pids[0] = 0;
1576         if (cpi->cpi_pids_cnt > 0) {
1577                 pids = cpi->cpi_pids;
1578                 for (pn = list_head(&so->so_pid_list); pn != NULL;
1579                     pids++, pn = list_next(&so->so_pid_list, pn))
```

```
1580                        *pids = pn->pn_pid;
1581        }
1582        mutex_exit(&so->so_pid_list_lock);
1583        return (mblk);
1584 }

1586 #endif /* ! codereview */
1587 void
1588 so_zcopy_notify(sock_upper_handle_t sock_handle)
1589 {
1590        struct sonode *so = (struct sonode *)sock_handle;

1592        mutex_enter(&so->so_lock);
1593        so->so_copyflag |= STZCNOTIFY;
1594        cv_broadcast(&so->so_copy_cv);
1595        mutex_exit(&so->so_lock);
1596 }

1598 void
1599 so_set_error(sock_upper_handle_t sock_handle, int error)
1600 {
1601        struct sonode *so = (struct sonode *)sock_handle;

1603        mutex_enter(&so->so_lock);

1605        soseterror(so, error);

1607        so_notify_error(so);
1608 }

1610 /*
1611  * so_recvmsg - read data from the socket
1612  *
1613  * There are two ways of obtaining data; either we ask the protocol to
1614  * copy directly into the supplied buffer, or we copy data from the
1615  * sonode's receive queue. The decision which one to use depends on
1616  * whether the protocol has a sd_recv_uio down call.
1617  */
1618 int
1619 so_recvmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
1620     struct cred *cr)
1621 {
1622        rval_t          rval;
1623        int             flags = 0;
1624        t_uscalar_t     controllen, namelen;
1625        int             error = 0;
1626        int ret;
1627        mblk_t          *mctlp = NULL;
1628        union T_primitives *tpr;
1629        void            *control;
1630        ssize_t         saved_resid;
1631        struct uio      *suiop;

1633        SO_BLOCK_FALLBACK(so, SOP_RECVMSG(so, msg, uiop, cr));

1635        if ((so->so_state & (SS_ISCONNECTED|SS_CANTRCVMORE)) == 0 &&
1636            (so->so_mode & SM_CONNREQUIRED)) {
1637                SO_UNBLOCK_FALLBACK(so);
1638                return (ENOTCONN);
1639        }

1641        if (msg->msg_flags & MSG_PEEK)
1642                msg->msg_flags &= ~MSG_WAITALL;

1644        if (so->so_mode & SM_ATOMIC)
1645                msg->msg_flags |= MSG_TRUNC;
```



```
1647        if (msg->msg_flags & MSG_OOB) {
1648                if ((so->so_mode & SM_EXDATA) == 0) {
1649                        error = EOPNOTSUPP;
1650                } else if (so->so_downcalls->sd_recv_uio != NULL) {
1651                        error = (*so->so_downcalls->sd_recv_uio)
1652                            (so->so_proto_handle, uiop, msg, cr);
1653                } else {
1654                        error = sorecvoob(so, msg, uiop, msg->msg_flags,
1655                            IS_SO_OOB_INLINE(so));
1656                }
1657                SO_UNBLOCK_FALLBACK(so);
1658                return (error);
1659        }

1661        /*
1662         * If the protocol has the recv down call, then pass the request
1663         * down.
1664         */
1665        if (so->so_downcalls->sd_recv_uio != NULL) {
1666                error = (*so->so_downcalls->sd_recv_uio)
1667                    (so->so_proto_handle, uiop, msg, cr);
1668                SO_UNBLOCK_FALLBACK(so);
1669                return (error);
1670        }

1672        /*
1673         * Reading data from the socket buffer
1674         */
1675        flags = msg->msg_flags;
1676        msg->msg_flags = 0;

1678        /*
1679         * Set msg_controllen and msg_namelen to zero here to make it
1680         * simpler in the cases that no control or name is returned.
1681         */
1682        controllen = msg->msg_controllen;
1683        namelen = msg->msg_namelen;
1684        msg->msg_controllen = 0;
1685        msg->msg_namelen = 0;

1687        mutex_enter(&so->so_lock);
1688        /* Set SOREADLOCKED */
1689        error = so_lock_read_intr(so,
1690            uiop->uio_fmode | ((flags & MSG_DONTWAIT) ? FNONBLOCK : 0));
1691        mutex_exit(&so->so_lock);
1692        if (error) {
1693                SO_UNBLOCK_FALLBACK(so);
1694                return (error);
1695        }

1697        suiop = sod_rcv_init(so, flags, &uiop);
1698 retry:
1699        saved_resid = uiop->uio_resid;
1700        error = so_dequeue_msg(so, &mctlp, uiop, &rval, flags);
1701        if (error != 0) {
1702                goto out;
1703        }
1704        /*
1705         * For datagrams the MOREDATA flag is used to set MSG_TRUNC.
1706         * For non-datagrams MOREDATA is used to set MSG_EOR.
1707         */
1708        ASSERT(!(rval.r_val1 & MORECTL));
1709        if ((rval.r_val1 & MOREDATA) && (so->so_mode & SM_ATOMIC))
1710                msg->msg_flags |= MSG_TRUNC;
1711        if (mctlp == NULL) {
```

```
1712                dprintso(so, 1, ("so_recvmsg: got M_DATA\n"));

1714                mutex_enter(&so->so_lock);
1715                /* Set MSG_EOR based on MOREDATA */
1716                if (!(rval.r_val1 & MOREDATA)) {
1717                        if (so->so_state & SS_SAVEDEOR) {
1718                                msg->msg_flags |= MSG_EOR;
1719                                so->so_state &= ~SS_SAVEDEOR;
1720                        }
1721                }
1722                /*
1723                 * If some data was received (i.e. not EOF) and the
1724                 * read/recv* has not been satisfied wait for some more.
1725                 */
1726                if ((flags & MSG_WAITALL) && !(msg->msg_flags & MSG_EOR) &&
1727                    uiop->uio_resid != saved_resid && uiop->uio_resid > 0) {
1728                        mutex_exit(&so->so_lock);
1729                        flags |= MSG_NOMARK;
1730                        goto retry;
1731                }

1733                goto out_locked;
1734        }
1735        /* so_queue_msg has already verified length and alignment */
1736        tpr = (union T_primitives *)mctlp->b_rptr;
1737        dprintso(so, 1, ("so_recvmsg: type %d\n", tpr->type));
1738        switch (tpr->type) {
1739        case T_DATA_IND: {
1740                /*
1741                 * Set msg_flags to MSG_EOR based on
1742                 * MORE_flag and MOREDATA.
1743                 */
1744                mutex_enter(&so->so_lock);
1745                so->so_state &= ~SS_SAVEDEOR;
1746                if (!(tpr->data_ind.MORE_flag & 1)) {
1747                        if (!(rval.r_val1 & MOREDATA))
1748                                msg->msg_flags |= MSG_EOR;
1749                        else
1750                                so->so_state |= SS_SAVEDEOR;
1751                }
1752                freemsg(mctlp);
1753                /*
1754                 * If some data was received (i.e. not EOF) and the
1755                 * read/recv* has not been satisfied wait for some more.
1756                 */
1757                if ((flags & MSG_WAITALL) && !(msg->msg_flags & MSG_EOR) &&
1758                    uiop->uio_resid != saved_resid && uiop->uio_resid > 0) {
1759                        mutex_exit(&so->so_lock);
1760                        flags |= MSG_NOMARK;
1761                        goto retry;
1762                }
1763                goto out_locked;
1764        }
1765        case T_UNITDATA_IND: {
1766                void *addr;
1767                t_uscalar_t addrlen;
1768                void *abuf;
1769                t_uscalar_t optlen;
1770                void *opt;

1772                if (namelen != 0) {
1773                        /* Caller wants source address */
1774                        addrlen = tpr->unitdata_ind.SRC_length;
1775                        addr = sogetoff(mctlp, tpr->unitdata_ind.SRC_offset,
1776                            addrlen, 1);
1777                        if (addr == NULL) {
```



```
1778                                freemsg(mctlp);
1779                                error = EPROTO;
1780                                eprintsoline(so, error);
1781                                goto out;
1782                        }
1783                        ASSERT(so->so_family != AF_UNIX);
1784                }
1785                optlen = tpr->unitdata_ind.OPT_length;
1786                if (optlen != 0) {
1787                        t_uscalar_t ncontrollen;

1789                        /*
1790                         * Extract any source address option.
1791                         * Determine how large cmsg buffer is needed.
1792                         */
1793                        opt = sogetoff(mctlp, tpr->unitdata_ind.OPT_offset,
1794                            optlen, __TPI_ALIGN_SIZE);

1796                        if (opt == NULL) {
1797                                freemsg(mctlp);
1798                                error = EPROTO;
1799                                eprintsoline(so, error);
1800                                goto out;
1801                        }
1802                        if (so->so_family == AF_UNIX)
1803                                so_getopt_srcaddr(opt, optlen, &addr, &addrlen);
1804                        ncontrollen = so_cmsglen(mctlp, opt, optlen,
1805                            !(flags & MSG_XPG4_2));
1806                        if (controllen != 0)
1807                                controllen = ncontrollen;
1808                        else if (ncontrollen != 0)
1809                                msg->msg_flags |= MSG_CTRUNC;
1810                } else {
1811                        controllen = 0;
1812                }

1814                if (namelen != 0) {
1815                        /*
1816                         * Return address to caller.
1817                         * Caller handles truncation if length
1818                         * exceeds msg_namelen.
1819                         * NOTE: AF_UNIX NUL termination is ensured by
1820                         * the sender's copyin_name().
1821                         */
1822                        abuf = kmem_alloc(addrlen, KM_SLEEP);

1824                        bcopy(addr, abuf, addrlen);
1825                        msg->msg_name = abuf;
1826                        msg->msg_namelen = addrlen;
1827                }

1829                if (controllen != 0) {
1830                        /*
1831                         * Return control msg to caller.
1832                         * Caller handles truncation if length
1833                         * exceeds msg_controllen.
1834                         */
1835                        control = kmem_zalloc(controllen, KM_SLEEP);

1837                        error = so_opt2cmsg(mctlp, opt, optlen,
1838                            !(flags & MSG_XPG4_2), control, controllen);
1839                        if (error) {
1840                                freemsg(mctlp);
1841                                if (msg->msg_namelen != 0)
1842                                        kmem_free(msg->msg_name,
1843                                            msg->msg_namelen);
```

```
1844                                kmem_free(control, controllen);
1845                                eprintsoline(so, error);
1846                                goto out;
1847                        }
1848                        msg->msg_control = control;
1849                        msg->msg_controllen = controllen;
1850                }

1852                freemsg(mctlp);
1853                goto out;
1854        }
1855        case T_OPTDATA_IND: {
1856                struct T_optdata_req *tdr;
1857                void *opt;
1858                t_uscalar_t optlen;

1860                tdr = (struct T_optdata_req *)mctlp->b_rptr;
1861                optlen = tdr->OPT_length;
1862                if (optlen != 0) {
1863                        t_uscalar_t ncontrollen;
1864                        /*
1865                         * Determine how large cmsg buffer is needed.
1866                         */
1867                        opt = sogetoff(mctlp,
1868                            tpr->optdata_ind.OPT_offset, optlen,
1869                            __TPI_ALIGN_SIZE);

1871                        if (opt == NULL) {
1872                                freemsg(mctlp);
1873                                error = EPROTO;
1874                                eprintsoline(so, error);
1875                                goto out;
1876                        }

1878                        ncontrollen = so_cmsglen(mctlp, opt, optlen,
1879                            !(flags & MSG_XPG4_2));
1880                        if (controllen != 0)
1881                                controllen = ncontrollen;
1882                        else if (ncontrollen != 0)
1883                                msg->msg_flags |= MSG_CTRUNC;
1884                } else {
1885                        controllen = 0;
1886                }

1888                if (controllen != 0) {
1889                        /*
1890                         * Return control msg to caller.
1891                         * Caller handles truncation if length
1892                         * exceeds msg_controllen.
1893                         */
1894                        control = kmem_zalloc(controllen, KM_SLEEP);

1896                        error = so_opt2cmsg(mctlp, opt, optlen,
1897                            !(flags & MSG_XPG4_2), control, controllen);
1898                        if (error) {
1899                                freemsg(mctlp);
1900                                kmem_free(control, controllen);
1901                                eprintsoline(so, error);
1902                                goto out;
1903                        }
1904                        msg->msg_control = control;
1905                        msg->msg_controllen = controllen;
1906                }

1908                /*
1909                 * Set msg_flags to MSG_EOR based on
```



```
1910                 * DATA_flag and MOREDATA.
1911                 */
1912                mutex_enter(&so->so_lock);
1913                so->so_state &= ~SS_SAVEDEOR;
1914                if (!(tpr->data_ind.MORE_flag & 1)) {
1915                        if (!(rval.r_val1 & MOREDATA))
1916                                msg->msg_flags |= MSG_EOR;
1917                        else
1918                                so->so_state |= SS_SAVEDEOR;
1919                }
1920                freemsg(mctlp);
1921                /*
1922                 * If some data was received (i.e. not EOF) and the
1923                 * read/recv* has not been satisfied wait for some more.
1924                 * Not possible to wait if control info was received.
1925                 */
1926                if ((flags & MSG_WAITALL) && !(msg->msg_flags & MSG_EOR) &&
1927                    controllen == 0 &&
1928                    uiop->uio_resid != saved_resid && uiop->uio_resid > 0) {
1929                        mutex_exit(&so->so_lock);
1930                        flags |= MSG_NOMARK;
1931                        goto retry;
1932                }
1933                goto out_locked;
1934        }
1935        default:
1936                cmn_err(CE_CONT, "so_recvmsg bad type %x \n",
1937                    tpr->type);
1938                freemsg(mctlp);
1939                error = EPROTO;
1940                ASSERT(0);
1941        }
1942 out:
1943        mutex_enter(&so->so_lock);
1944 out_locked:
1945        ret = sod_rcv_done(so, suiop, uiop);
1946        if (ret != 0 && error == 0)
1947                error = ret;

1949        so_unlock_read(so);     /* Clear SOREADLOCKED */
1950        mutex_exit(&so->so_lock);

1952        SO_UNBLOCK_FALLBACK(so);

1954        return (error);
1955 }

1957 sonodeops_t so_sonodeops = {
1958        so_init,                /* sop_init     */
1959        so_accept,              /* sop_accept   */
1960        so_bind,                /* sop_bind     */
1961        so_listen,              /* sop_listen   */
1962        so_connect,             /* sop_connect  */
1963        so_recvmsg,             /* sop_recvmsg  */
1964        so_sendmsg,             /* sop_sendmsg  */
1965        so_sendmblk,            /* sop_sendmblk */
1966        so_getpeername,         /* sop_getpeername */
1967        so_getsockname,         /* sop_getsockname */
1968        so_shutdown,            /* sop_shutdown */
1969        so_getsockopt,          /* sop_getsockopt */
1970        so_setsockopt,          /* sop_setsockopt */
1971        so_ioctl,               /* sop_ioctl    */
1972        so_poll,                /* sop_poll     */
1973        so_close,               /* sop_close */
1974 };
```

```
1976 sock_upcalls_t so_upcalls = {
1977         so_newconn,
1978         so_connected,
1979         so_disconnected,
1980         so_opctl,
1981         so_queue_msg,
1982         so_set_prop,
1983         so_txq_full,
1984         so_signal_oob,
1985         so_zcopy_notify,
1986         so_set_error,
1987         so_closed,
1988         so_get_sock_pid_mblk
  40         so_closed
1989 };
______unchanged_portion_omitted_
```

```
**********************************************************
   12990 Mon Aug 17 21:08:04 2015
new/usr/src/uts/common/fs/sockfs/sockcommon_vnops.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_


/*
 * generic vnode ops
 */

/*ARGSUSED*/
static int
socket_vop_open(struct vnode **vpp, int flag, struct cred *cr,
    caller_context_t *ct)
{
        struct vnode *vp = *vpp;
        struct sonode *so = VTOSO(vp);

        flag &= ~FCREAT;                /* paranoia */
        mutex_enter(&so->so_lock);
        so->so_count++;
        mutex_exit(&so->so_lock);

        if (!(curproc->p_flag & SSYS))
                sonode_insert_pid(so, curproc->p_pidp->pid_id);

#endif /* ! codereview */
        ASSERT(so->so_count != 0);      /* wraparound */
        ASSERT(vp->v_type == VSOCK);

        return (0);
}

/*ARGSUSED*/
static int
socket_vop_close(struct vnode *vp, int flag, int count, offset_t offset,
    struct cred *cr, caller_context_t *ct)
{
        struct sonode *so;
        int error = 0;

        so = VTOSO(vp);
        ASSERT(vp->v_type == VSOCK);

        cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
        cleanshares(vp, ttoproc(curthread)->p_pid);

        if (vp->v_stream)
                strclean(vp);

        if (count > 1) {
                dprint(2, ("socket_vop_close: count %d\n", count));
                return (0);
        }

        mutex_enter(&so->so_lock);
        if (--so->so_count == 0) {
                /*
                 * Initiate connection shutdown.
                 */
                mutex_exit(&so->so_lock);
                error = socket_close_internal(so, flag, cr);
        } else {
                mutex_exit(&so->so_lock);
```



```
        }

        return (error);
}

/*ARGSUSED2*/
static int
socket_vop_read(struct vnode *vp, struct uio *uiop, int ioflag, struct cred *cr,
    caller_context_t *ct)
{
        struct sonode *so = VTOSO(vp);
        struct nmsghdr lmsg;

        ASSERT(vp->v_type == VSOCK);
        bzero((void *)&lmsg, sizeof (lmsg));

        return (socket_recvmsg(so, &lmsg, uiop, cr));
}

/*ARGSUSED2*/
static int
socket_vop_write(struct vnode *vp, struct uio *uiop, int ioflag,
    struct cred *cr, caller_context_t *ct)
{
        struct sonode *so = VTOSO(vp);
        struct nmsghdr lmsg;

        ASSERT(vp->v_type == VSOCK);
        bzero((void *)&lmsg, sizeof (lmsg));

        if (!(so->so_mode & SM_BYTESTREAM)) {
                /*
                 * If the socket is not byte stream set MSG_EOR
                 */
                lmsg.msg_flags = MSG_EOR;
        }

        return (socket_sendmsg(so, &lmsg, uiop, cr));
}

/*ARGSUSED4*/
static int
socket_vop_ioctl(struct vnode *vp, int cmd, intptr_t arg, int mode,
    struct cred *cr, int32_t *rvalp, caller_context_t *ct)
{
        struct sonode *so = VTOSO(vp);

        ASSERT(vp->v_type == VSOCK);

        switch (cmd) {
        case F_ASSOCI_PID:
                if (cr != kcred)
                        return (EPERM);
                if (!(curproc->p_flag & SSYS))
                        sonode_insert_pid(so, (pid_t)arg);
                return (0);

        case F_DASSOC_PID:
                if (cr != kcred)
                        return (EPERM);
                if (!(curproc->p_flag & SSYS))
                        sonode_remove_pid(so, (pid_t)arg);
                return (0);
        }
#endif /* ! codereview */
```

```
233         return (socket_ioctl(so, cmd, arg, mode, cr, rvalp));
234 }

236 /*
237  * Allow any flags. Record FNDELAY and FNONBLOCK so that they can be inherited
238  * from listener to acceptor.
239  */
240 /* ARGSUSED */
241 static int
242 socket_vop_setfl(vnode_t *vp, int oflags, int nflags, cred_t *cr,
243     caller_context_t *ct)
244 {
245         struct sonode *so = VTOSO(vp);
246         int error = 0;

248         ASSERT(vp->v_type == VSOCK);

250         mutex_enter(&so->so_lock);
251         if (nflags & FNDELAY)
252                 so->so_state |= SS_NDELAY;
253         else
254                 so->so_state &= ~SS_NDELAY;
255         if (nflags & FNONBLOCK)
256                 so->so_state |= SS_NONBLOCK;
257         else
258                 so->so_state &= ~SS_NONBLOCK;
259         mutex_exit(&so->so_lock);

261         if (so->so_state & SS_ASYNC)
262                 oflags |= FASYNC;
263         /*
264          * Sets/clears the SS_ASYNC flag based on the presence/absence
265          * of the FASYNC flag passed to fcntl(F_SETFL).
266          * This exists solely for BSD fcntl() FASYNC compatibility.
267          */
268         if ((oflags ^ nflags) & FASYNC && so->so_version != SOV_STREAM) {
269                 int async = nflags & FASYNC;
270                 int32_t rv;

272                 /*
273                  * For non-TPI sockets all we have to do is set/remove the
274                  * SS_ASYNC bit, but for TPI it is more involved. For that
275                  * reason we delegate the job to the protocol's ioctl handler.
276                  */
277                 error = socket_ioctl(so, FIOASYNC, (intptr_t)&async, FKIOCTL,
278                     cr, &rv);
279         }
280         return (error);
281 }


284 /*
285  * Get the made up attributes for the vnode.
286  * 4.3BSD returns the current time for all the timestamps.
287  * 4.4BSD returns 0 for all the timestamps.
288  * Here we use the access and modified times recorded in the sonode.
289  *
290  * Just like in BSD there is not effect on the underlying file system node
291  * bound to an AF_UNIX pathname.
292  *
293  * When sockmod has been popped this will act just like a stream. Since
294  * a socket is always a clone there is no need to inspect the attributes
295  * of the "realvp".
296  */
297 /* ARGSUSED */
298 int
```



```
299 socket_vop_getattr(struct vnode *vp, struct vattr *vap, int flags,
300     struct cred *cr, caller_context_t *ct)
301 {
302         dev_t           fsid;
303         struct sonode   *so;
304         static int      sonode_shift = 0;

306         /*
307          * Calculate the amount of bitshift to a sonode pointer which will
308          * still keep it unique.  See below.
309          */
310         if (sonode_shift == 0)
311                 sonode_shift = highbit(sizeof (struct sonode));
312         ASSERT(sonode_shift > 0);

314         so = VTOSO(vp);
315         fsid = sockdev;

317         if (so->so_version == SOV_STREAM) {
318                 /*
319                  * The imaginary "sockmod" has been popped - act
320                  * as a stream
321                  */
322                 vap->va_type = VCHR;
323                 vap->va_mode = 0;
324         } else {
325                 vap->va_type = vp->v_type;
326                 vap->va_mode = S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|
327                     S_IROTH|S_IWOTH;
328         }
329         vap->va_uid = vap->va_gid = 0;
330         vap->va_fsid = fsid;
331         /*
332          * If the va_nodeid is > MAX_USHORT, then i386 stats might fail.
333          * So we shift down the sonode pointer to try and get the most
334          * uniqueness into 16-bits.
335          */
336         vap->va_nodeid = ((ino_t)so >> sonode_shift) & 0xFFFF;
337         vap->va_nlink = 0;
338         vap->va_size = 0;

340         /*
341          * We need to zero out the va_rdev to avoid some fstats getting
342          * EOVERFLOW.  This also mimics SunOS 4.x and BSD behavior.
343          */
344         vap->va_rdev = (dev_t)0;
345         vap->va_blksize = MAXBSIZE;
346         vap->va_nblocks = btod(vap->va_size);

348         if (!SOCK_IS_NONSTR(so)) {
349                 sotpi_info_t *sti = SOTOTPI(so);

351                 mutex_enter(&so->so_lock);
352                 vap->va_atime.tv_sec = sti->sti_atime;
353                 vap->va_mtime.tv_sec = sti->sti_mtime;
354                 vap->va_ctime.tv_sec = sti->sti_ctime;
355                 mutex_exit(&so->so_lock);
356         } else {
357                 vap->va_atime.tv_sec = 0;
358                 vap->va_mtime.tv_sec = 0;
359                 vap->va_ctime.tv_sec = 0;
360         }

362         vap->va_atime.tv_nsec = 0;
363         vap->va_mtime.tv_nsec = 0;
364         vap->va_ctime.tv_nsec = 0;
```

```
365         vap->va_seq = 0;

367         return (0);
368 }

370 /*
371  * Set attributes.
372  * Just like in BSD there is not effect on the underlying file system node
373  * bound to an AF_UNIX pathname.
374  *
375  * When sockmod has been popped this will act just like a stream. Since
376  * a socket is always a clone there is no need to modify the attributes
377  * of the "realvp".
378  */
379 /* ARGSUSED */
380 int
381 socket_vop_setattr(struct vnode *vp, struct vattr *vap, int flags,
382     struct cred *cr, caller_context_t *ct)
383 {
384         struct sonode *so = VTOSO(vp);

386         /*
387          * If times were changed, and we have a STREAMS socket, then update
388          * the sonode.
389          */
390         if (!SOCK_IS_NONSTR(so)) {
391                 sotpi_info_t *sti = SOTOTPI(so);

393                 mutex_enter(&so->so_lock);
394                 if (vap->va_mask & AT_ATIME)
395                         sti->sti_atime = vap->va_atime.tv_sec;
396                 if (vap->va_mask & AT_MTIME) {
397                         sti->sti_mtime = vap->va_mtime.tv_sec;
398                         sti->sti_ctime = gethrestime_sec();
399                 }
400                 mutex_exit(&so->so_lock);
401         }

403         return (0);
404 }

406 /*
407  * Check if user is allowed to access vp. For non-STREAMS based sockets,
408  * there might not be a device attached to the file system. So for those
409  * types of sockets there are no permissions to check.
410  *
411  * XXX Should there be some other mechanism to check access rights?
412  */
413 /*ARGSUSED*/
414 int
415 socket_vop_access(struct vnode *vp, int mode, int flags, struct cred *cr,
416     caller_context_t *ct)
417 {
418         struct sonode *so = VTOSO(vp);

420         if (!SOCK_IS_NONSTR(so)) {
421                 ASSERT(so->so_sockparams->sp_sdev_info.sd_vnode != NULL);
422                 return (VOP_ACCESS(so->so_sockparams->sp_sdev_info.sd_vnode,
423                     mode, flags, cr, NULL));
424         }
425         return (0);
426 }

428 /*
429  * 4.3BSD and 4.4BSD fail a fsync on a socket with EINVAL.
430  * This code does the same to be compatible and also to not give an
```



```
431  * application the impression that the data has actually been "synced"
432  * to the other end of the connection.
433  */
434 /* ARGSUSED */
435 int
436 socket_vop_fsync(struct vnode *vp, int syncflag, struct cred *cr,
437     caller_context_t *ct)
438 {
439         return (EINVAL);
440 }

442 /*ARGSUSED*/
443 static void
444 socket_vop_inactive(struct vnode *vp, struct cred *cr, caller_context_t *ct)
445 {
446         struct sonode *so = VTOSO(vp);

448         ASSERT(vp->v_type == VSOCK);

450         mutex_enter(&vp->v_lock);
451         /*
452          * If no one has reclaimed the vnode, remove from the
453          * cache now.
454          */
455         if (vp->v_count < 1)
456                 cmn_err(CE_PANIC, "socket_inactive: Bad v_count");

458         /*
459          * Drop the temporary hold by vn_rele now
460          */
461         if (--vp->v_count != 0) {
462                 mutex_exit(&vp->v_lock);
463                 return;
464         }
465         mutex_exit(&vp->v_lock);


468         ASSERT(!vn_has_cached_data(vp));

470         /* socket specfic clean-up */
471         socket_destroy_internal(so, cr);
472 }

474 /* ARGSUSED */
475 int
476 socket_vop_fid(struct vnode *vp, struct fid *fidp, caller_context_t *ct)
477 {
478         return (EINVAL);
479 }

481 /*
482  * Sockets are not seekable.
483  * (and there is a bug to fix STREAMS to make them fail this as well).
484  */
485 /*ARGSUSED*/
486 int
487 socket_vop_seek(struct vnode *vp, offset_t ooff, offset_t *noffp,
488     caller_context_t *ct)
489 {
490         return (ESPIPE);
491 }

493 /*ARGSUSED*/
494 static int
495 socket_vop_poll(struct vnode *vp, short events, int anyyet, short *reventsp,
496     struct pollhead **phpp, caller_context_t *ct)
```

```
497 {
498         struct sonode *so = VTOSO(vp);

500         ASSERT(vp->v_type == VSOCK);

502         return (socket_poll(so, events, anyyet, reventsp, phpp));
503 }
```

```
**********************************************************
   50094 Mon Aug 17 21:08:04 2015
new/usr/src/uts/common/fs/sockfs/socksubr.c
XXXX adding PID information to netstat output
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 1995, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */

  26 #include <sys/types.h>
  27 #include <sys/t_lock.h>
  28 #include <sys/param.h>
  29 #include <sys/systm.h>
  30 #include <sys/buf.h>
  31 #include <sys/conf.h>
  32 #include <sys/cred.h>
  33 #include <sys/kmem.h>
  34 #include <sys/sysmacros.h>
  35 #include <sys/vfs.h>
  36 #include <sys/vfs_opreg.h>
  37 #include <sys/vnode.h>
  38 #include <sys/debug.h>
  39 #include <sys/errno.h>
  40 #include <sys/time.h>
  41 #include <sys/file.h>
  42 #include <sys/open.h>
  43 #include <sys/user.h>
  44 #include <sys/termios.h>
  45 #include <sys/stream.h>
  46 #include <sys/strsubr.h>
  47 #include <sys/strsun.h>
  48 #include <sys/esunddi.h>
  49 #include <sys/flock.h>
  50 #include <sys/modctl.h>
  51 #include <sys/cmn_err.h>
  52 #include <sys/mkdev.h>
  53 #include <sys/pathname.h>
  54 #include <sys/ddi.h>
  55 #include <sys/stat.h>
  56 #include <sys/fs/snode.h>
  57 #include <sys/fs/dv_node.h>
  58 #include <sys/zone.h>

  60 #include <sys/socket.h>
  61 #include <sys/socketvar.h>
```



```
  62 #include <netinet/in.h>
  63 #include <sys/un.h>
  64 #include <sys/ucred.h>

  66 #include <sys/tiuser.h>
  67 #define _SUN_TPI_VERSION        2
  68 #include <sys/tihdr.h>

  70 #include <c2/audit.h>

  72 #include <fs/sockfs/nl7c.h>
  73 #include <fs/sockfs/sockcommon.h>
  74 #include <fs/sockfs/sockfilter_impl.h>
  75 #include <fs/sockfs/socktpi.h>
  76 #include <fs/sockfs/socktpi_impl.h>
  77 #include <fs/sockfs/sodirect.h>

  79 /*
  80  * Macros that operate on struct cmsghdr.
  81  * The CMSG_VALID macro does not assume that the last option buffer is padded.
  82  */
  83 #define CMSG_CONTENT(cmsg)      (&((cmsg)[1]))
  84 #define CMSG_CONTENTLEN(cmsg)   ((cmsg)->cmsg_len - sizeof (struct cmsghdr))
  85 #define CMSG_VALID(cmsg, start, end)                                    \
  86         (ISALIGNED_cmsghdr(cmsg) &&                                    \
  87         ((uintptr_t)(cmsg) >= (uintptr_t)(start)) &&                   \
  88         ((uintptr_t)(cmsg) < (uintptr_t)(end)) &&                      \
  89         ((ssize_t)(cmsg)->cmsg_len >= sizeof (struct cmsghdr)) &&      \
  90         ((uintptr_t)(cmsg) + (cmsg)->cmsg_len <= (uintptr_t)(end)))
  91 #define SO_LOCK_WAKEUP_TIME     3000    /* Wakeup time in milliseconds */

  93 dev_t sockdev;  /* For fsid in getattr */
  94 int sockfs_defer_nl7c_init = 0;

  96 struct socklist socklist;

  98 struct kmem_cache *socket_cache;

 100 /*
 101  * sockconf_lock protects the socket configuration (socket types and
 102  * socket filters) which is changed via the sockconfig system call.
 103  */
 104 krwlock_t sockconf_lock;

 106 static int sockfs_update(kstat_t *, int);
 107 static int sockfs_snapshot(kstat_t *, void *, int);
 108 extern smod_info_t *sotpi_smod_create(void);

 110 extern void sendfile_init();

 112 extern void nl7c_init(void);

 114 extern int modrootloaded;

 116 #define ADRSTRLEN (2 * sizeof (void *) + 1)
 117 /*
 118  * kernel structure for passing the sockinfo data back up to the user.
 119  * the strings array allows us to convert AF_UNIX addresses into strings
 120  * with a common method regardless of which n-bit kernel we're running.
 121  */
 122 struct k_sockinfo {
 123         struct sockinfo ks_si;
 124         char            ks_straddr[3][ADRSTRLEN];
 125 };

 116 /*
```

```
 117  * Translate from a device pathname (e.g. "/dev/tcp") to a vnode.
 118  * Returns with the vnode held.
 119  */
 120 int
 121 sogetvp(char *devpath, vnode_t **vpp, int uioflag)
 122 {
 123         struct snode *csp;
 124         vnode_t *vp, *dvp;
 125         major_t maj;
 126         int error;

 128         ASSERT(uioflag == UIO_SYSSPACE || uioflag == UIO_USERSPACE);

 130         /*
 131          * Lookup the underlying filesystem vnode.
 132          */
 133         error = lookupname(devpath, uioflag, FOLLOW, NULLVPP, &vp);
 134         if (error)
 135                 return (error);

 137         /* Check that it is the correct vnode */
 138         if (vp->v_type != VCHR) {
 139                 VN_RELE(vp);
 140                 return (ENOTSOCK);
 141         }

 143         /*
 144          * If devpath went through devfs, the device should already
 145          * be configured. If devpath is a mknod file, however, we
 146          * need to make sure the device is properly configured.
 147          * To do this, we do something similar to spec_open()
 148          * except that we resolve to the minor/leaf level since
 149          * we need to return a vnode.
 150          */
 151         csp = VTOS(VTOS(vp)->s_commonvp);
 152         if (!(csp->s_flag & SDIPSET)) {
 153                 char *pathname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
 154                 error = ddi_dev_pathname(vp->v_rdev, S_IFCHR, pathname);
 155                 if (error == 0)
 156                         error = devfs_lookupname(pathname, NULLVPP, &dvp);
 157                 VN_RELE(vp);
 158                 kmem_free(pathname, MAXPATHLEN);
 159                 if (error != 0)
 160                         return (ENXIO);
 161                 vp = dvp;       /* use the devfs vp */
 162         }

 164         /* device is configured at this point */
 165         maj = getmajor(vp->v_rdev);
 166         if (!STREAMSTAB(maj)) {
 167                 VN_RELE(vp);
 168                 return (ENOSTR);
 169         }

 171         *vpp = vp;
 172         return (0);
 173 }
_____unchanged_portion_omitted_

 717 /*
 718  * Extract file descriptors from a fdbuf.
 719  * Return list in rights/rightslen.
 720  */
 721 /*ARGSUSED*/
 722 static int
 723 fdbuf_extract(struct fdbuf *fdbuf, void *rights, int rightslen)
```



```
 724 {
 725         int     i, fd;
 726         int     *rp;
 727         struct file *fp;
 728         int     numfd;

 730         dprint(1, ("fdbuf_extract: %d fds, len %d\n",
 731             fdbuf->fd_numfd, rightslen));

 733         numfd = fdbuf->fd_numfd;
 734         ASSERT(rightslen == numfd * (int)sizeof (int));

 736         /*
 737          * Allocate a file descriptor and increment the f_count.
 738          * The latter is needed since we always call fdbuf_free
 739          * which performs a closef.
 740          */
 741         rp = (int *)rights;
 742         for (i = 0; i < numfd; i++) {
 743                 if ((fd = ufalloc(0)) == -1)
 744                         goto cleanup;
 745                 /*
 746                  * We need pointer size alignment for fd_fds. On a LP64
 747                  * kernel, the required alignment is 8 bytes while
 748                  * the option headers and values are only 4 bytes
 749                  * aligned. So its safer to do a bcopy compared to
 750                  * assigning fdbuf->fd_fds[i] to fp.
 751                  */
 752                 bcopy((char *)&fdbuf->fd_fds[i], (char *)&fp, sizeof (fp));
 753                 mutex_enter(&fp->f_tlock);
 754                 fp->f_count++;
 755                 mutex_exit(&fp->f_tlock);
 756                 setf(fd, fp);
 757                 *rp++ = fd;

 759                 /*
 760                  * Add the current pid to the list associated with this
 761                  * descriptor.
 762                  */
 763                 if (fp->f_vnode != NULL)
 764                         (void) VOP_IOCTL(fp->f_vnode, F_ASSOCI_PID,
 765                             (intptr_t)curproc->p_pidp->pid_id, FKIOCTL, kcred,
 766                             NULL, NULL);

 768 #endif /* ! codereview */
 769                 if (AU_AUDITING())
 770                         audit_fdrecv(fd, fp);
 771                 dprint(1, ("fdbuf_extract: [%d] = %d, %p refcnt %d\n",
 772                     i, fd, (void *)fp, fp->f_count));
 773         }
 774         return (0);

 776 cleanup:
 777         /*
 778          * Undo whatever partial work the loop above has done.
 779          */
 780         {
 781                 int j;

 783                 rp = (int *)rights;
 784                 for (j = 0; j < i; j++) {
 785                         dprint(0,
 786                             ("fdbuf_extract: cleanup[%d] = %d\n", j, *rp));
 787                         (void) closeandsetf(*rp++, NULL);
 788                 }
 789         }
```

```
791         return (EMFILE);
792 }

794 /*
795  * Insert file descriptors into an fdbuf.
796  * Returns a kmem_alloc'ed fdbuf. The fdbuf should be freed
797  * by calling fdbuf_free().
798  */
799 int
800 fdbuf_create(void *rights, int rightslen, struct fdbuf **fdbufp)
801 {
802         int             numfd, i;
803         int             *fds;
804         struct file     *fp;
805         struct fdbuf    *fdbuf;
806         int             fdbufsize;

808         dprint(1, ("fdbuf_create: len %d\n", rightslen));

810         numfd = rightslen / (int)sizeof (int);

812         fdbufsize = (int)FDBUF_HDRSIZE + (numfd * (int)sizeof (struct file *));
813         fdbuf = kmem_alloc(fdbufsize, KM_SLEEP);
814         fdbuf->fd_size = fdbufsize;
815         fdbuf->fd_numfd = 0;
816         fdbuf->fd_ebuf = NULL;
817         fdbuf->fd_ebuflen = 0;
818         fds = (int *)rights;
819         for (i = 0; i < numfd; i++) {
820                 if ((fp = getf(fds[i])) == NULL) {
821                         fdbuf_free(fdbuf);
822                         return (EBADF);
823                 }
824                 dprint(1, ("fdbuf_create: [%d] = %d, %p refcnt %d\n",
825                     i, fds[i], (void *)fp, fp->f_count));
826                 mutex_enter(&fp->f_tlock);
827                 fp->f_count++;
828                 mutex_exit(&fp->f_tlock);
829                 /*
830                  * The maximum alignment for fdbuf (or any option header
831                  * and its value) it 4 bytes. On a LP64 kernel, the alignment
832                  * is not sufficient for pointers (fd_fds in this case). Since
833                  * we just did a kmem_alloc (we get a double word alignment),
834                  * we don't need to do anything on the send side (we loose
835                  * the double word alignment because fdbuf goes after an
836                  * option header (eg T_unitdata_req) which is only 4 byte
837                  * aligned). We take care of this when we extract the file
838                  * descriptor in fdbuf_extract or fdbuf_free.
839                  */
840                 fdbuf->fd_fds[i] = fp;
841                 fdbuf->fd_numfd++;
842                 releasef(fds[i]);
843                 if (AU_AUDITING())
844                         audit_fdsend(fds[i], fp, 0);
845         }
846         *fdbufp = fdbuf;
847         return (0);
848 }

850 static int
851 fdbuf_optlen(int rightslen)
852 {
853         int numfd;

855         numfd = rightslen / (int)sizeof (int);
```



```
857         return ((int)FDBUF_HDRSIZE + (numfd * (int)sizeof (struct file *)));
858 }

860 static t_uscalar_t
861 fdbuf_cmsglen(int fdbuflen)
862 {
863         return (t_uscalar_t)((fdbuflen - FDBUF_HDRSIZE) /
864             (int)sizeof (struct file *) * (int)sizeof (int));
865 }


868 /*
869  * Return non-zero if the mblk and fdbuf are consistent.
870  */
871 static int
872 fdbuf_verify(mblk_t *mp, struct fdbuf *fdbuf, int fdbuflen)
873 {
874         if (fdbuflen >= FDBUF_HDRSIZE &&
875             fdbuflen == fdbuf->fd_size) {
876                 frtn_t *frp = mp->b_datap->db_frtnp;
877                 /*
878                  * Check that the SO_FILEP portion of the
879                  * message has not been modified by
880                  * the loopback transport. The sending sockfs generates
881                  * a message that is esballoc'ed with the free function
882                  * being fdbuf_free() and where free_arg contains the
883                  * identical information as the SO_FILEP content.
884                  *
885                  * If any of these constraints are not satisfied we
886                  * silently ignore the option.
887                  */
888                 ASSERT(mp);
889                 if (frp != NULL &&
890                     frp->free_func == fdbuf_free &&
891                     frp->free_arg != NULL &&
892                     bcmp(frp->free_arg, fdbuf, fdbuflen) == 0) {
893                         dprint(1, ("fdbuf_verify: fdbuf %p len %d\n",
894                             (void *)fdbuf, fdbuflen));
895                         return (1);
896                 } else {
897                         zcmn_err(getzoneid(), CE_WARN,
898                             "sockfs: mismatched fdbuf content (%p)",
899                             (void *)mp);
900                         return (0);
901                 }
902         } else {
903                 zcmn_err(getzoneid(), CE_WARN,
904                     "sockfs: mismatched fdbuf len %d, %d\n",
905                     fdbuflen, fdbuf->fd_size);
906                 return (0);
907         }
908 }

910 /*
911  * When the file descriptors returned by sorecvmsg can not be passed
912  * to the application this routine will cleanup the references on
913  * the files. Start at startoff bytes into the buffer.
914  */
915 static void
916 close_fds(void *fdbuf, int fdbuflen, int startoff)
917 {
918         int *fds = (int *)fdbuf;
919         int numfd = fdbuflen / (int)sizeof (int);
920         int i;
```

```
922         dprint(1, ("close_fds(%p, %d, %d)\n", fdbuf, fdbuflen, startoff));
923
924         for (i = 0; i < numfd; i++) {
925                 if (startoff < 0)
926                         startoff = 0;
927                 if (startoff < (int)sizeof (int)) {
928                         /*
929                          * This file descriptor is partially or fully after
930                          * the offset
931                          */
932                         dprint(0,
933                             ("close_fds: cleanup[%d] = %d\n", i, fds[i]));
934                         (void) closeandsetf(fds[i], NULL);
935                 }
936                 startoff -= (int)sizeof (int);
937         }
938 }
939
940 /*
941  * Close all file descriptors contained in the control part starting at
942  * the startoffset.
943  */
944 void
945 so_closefds(void *control, t_uscalar_t controllen, int oldflg,
946     int startoff)
947 {
948         struct cmsghdr *cmsg;
949
950         if (control == NULL)
951                 return;
952
953         if (oldflg) {
954                 close_fds(control, controllen, startoff);
955                 return;
956         }
957         /* Scan control part for file descriptors. */
958         for (cmsg = (struct cmsghdr *)control;
959             CMSG_VALID(cmsg, control, (uintptr_t)control + controllen);
960             cmsg = CMSG_NEXT(cmsg)) {
961                 if (cmsg->cmsg_level == SOL_SOCKET &&
962                     cmsg->cmsg_type == SCM_RIGHTS) {
963                         close_fds(CMSG_CONTENT(cmsg),
964                             (int)CMSG_CONTENTLEN(cmsg),
965                             startoff - (int)sizeof (struct cmsghdr));
966                 }
967                 startoff -= cmsg->cmsg_len;
968         }
969 }
970
971 /*
972  * Returns a pointer/length for the file descriptors contained
973  * in the control buffer. Returns with *fdlenp == -1 if there are no
974  * file descriptor options present. This is different than there being
975  * a zero-length file descriptor option.
976  * Fail if there are multiple SCM_RIGHT cmsgs.
977  */
978 int
979 so_getfdopt(void *control, t_uscalar_t controllen, int oldflg,
980     void **fdsp, int *fdlenp)
981 {
982         struct cmsghdr *cmsg;
983         void *fds;
984         int fdlen;
985
986         if (control == NULL) {
987                 *fdsp = NULL;
```



```
988                 *fdlenp = -1;
989                 return (0);
990         }
991
992         if (oldflg) {
993                 *fdsp = control;
994                 if (controllen == 0)
995                         *fdlenp = -1;
996                 else
997                         *fdlenp = controllen;
998                 dprint(1, ("so_getfdopt: old %d\n", *fdlenp));
999                 return (0);
1000        }
1001
1002        fds = NULL;
1003        fdlen = 0;
1004
1005        for (cmsg = (struct cmsghdr *)control;
1006            CMSG_VALID(cmsg, control, (uintptr_t)control + controllen);
1007            cmsg = CMSG_NEXT(cmsg)) {
1008                if (cmsg->cmsg_level == SOL_SOCKET &&
1009                    cmsg->cmsg_type == SCM_RIGHTS) {
1010                        if (fds != NULL)
1011                                return (EINVAL);
1012                        fds = CMSG_CONTENT(cmsg);
1013                        fdlen = (int)CMSG_CONTENTLEN(cmsg);
1014                        dprint(1, ("so_getfdopt: new %lu\n",
1015                            (size_t)CMSG_CONTENTLEN(cmsg)));
1016                }
1017        }
1018        if (fds == NULL) {
1019                dprint(1, ("so_getfdopt: NONE\n"));
1020                *fdlenp = -1;
1021        } else
1022                *fdlenp = fdlen;
1023        *fdsp = fds;
1024        return (0);
1025 }
1026
1027 /*
1028  * Return the length of the options including any file descriptor options.
1029  */
1030 t_uscalar_t
1031 so_optlen(void *control, t_uscalar_t controllen, int oldflg)
1032 {
1033         struct cmsghdr *cmsg;
1034         t_uscalar_t optlen = 0;
1035         t_uscalar_t len;
1036
1037         if (control == NULL)
1038                 return (0);
1039
1040         if (oldflg)
1041                 return ((t_uscalar_t)(sizeof (struct T_opthdr) +
1042                     fdbuf_optlen(controllen)));
1043
1044         for (cmsg = (struct cmsghdr *)control;
1045             CMSG_VALID(cmsg, control, (uintptr_t)control + controllen);
1046             cmsg = CMSG_NEXT(cmsg)) {
1047                 if (cmsg->cmsg_level == SOL_SOCKET &&
1048                     cmsg->cmsg_type == SCM_RIGHTS) {
1049                         len = fdbuf_optlen((int)CMSG_CONTENTLEN(cmsg));
1050                 } else {
1051                         len = (t_uscalar_t)CMSG_CONTENTLEN(cmsg);
1052                 }
1053                 optlen += (t_uscalar_t)(_TPI_ALIGN_TOPT(len) +
```

```
                    sizeof (struct T_opthdr));
        }
        dprint(1, ("so_optlen: controllen %d, flg %d -> optlen %d\n",
            controllen, oldflg, optlen));
        return (optlen);
}

/*
 * Copy options from control to the mblk. Skip any file descriptor options.
 */
void
so_cmsg2opt(void *control, t_uscalar_t controllen, int oldflg, mblk_t *mp)
{
        struct T_opthdr toh;
        struct cmsghdr *cmsg;

        if (control == NULL)
                return;

        if (oldflg) {
                /* No real options - caller has handled file descriptors */
                return;
        }
        for (cmsg = (struct cmsghdr *)control;
            CMSG_VALID(cmsg, control, (uintptr_t)control + controllen);
            cmsg = CMSG_NEXT(cmsg)) {
                /*
                 * Note: The caller handles file descriptors prior
                 * to calling this function.
                 */
                t_uscalar_t len;

                if (cmsg->cmsg_level == SOL_SOCKET &&
                    cmsg->cmsg_type == SCM_RIGHTS)
                        continue;

                len = (t_uscalar_t)CMSG_CONTENTLEN(cmsg);
                toh.level = cmsg->cmsg_level;
                toh.name = cmsg->cmsg_type;
                toh.len = len + (t_uscalar_t)sizeof (struct T_opthdr);
                toh.status = 0;

                soappendmsg(mp, &toh, sizeof (toh));
                soappendmsg(mp, CMSG_CONTENT(cmsg), len);
                mp->b_wptr += _TPI_ALIGN_TOPT(len) - len;
                ASSERT(mp->b_wptr <= mp->b_datap->db_lim);
        }
}

/*
 * Return the length of the control message derived from the options.
 * Exclude SO_SRCADDR and SO_UNIX_CLOSE options. Include SO_FILEP.
 * When oldflg is set only include SO_FILEP.
 * so_opt2cmsg and so_cmsglen are inter-related since so_cmsglen
 * allocates the space that so_opt2cmsg fills. If one changes, the other should
 * also be checked for any possible impacts.
 */
t_uscalar_t
so_cmsglen(mblk_t *mp, void *opt, t_uscalar_t optlen, int oldflg)
{
        t_uscalar_t cmsglen = 0;
        struct T_opthdr *tohp;
        t_uscalar_t len;
        t_uscalar_t last_roundup = 0;

        ASSERT(__TPI_TOPT_ISALIGNED(opt));
```



```
        for (tohp = (struct T_opthdr *)opt;
            tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
            tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
                dprint(1, ("so_cmsglen: level 0x%x, name %d, len %d\n",
                    tohp->level, tohp->name, tohp->len));
                if (tohp->level == SOL_SOCKET &&
                    (tohp->name == SO_SRCADDR ||
                    tohp->name == SO_UNIX_CLOSE)) {
                        continue;
                }
                if (tohp->level == SOL_SOCKET && tohp->name == SO_FILEP) {
                        struct fdbuf *fdbuf;
                        int fdbuflen;

                        fdbuf = (struct fdbuf *)_TPI_TOPT_DATA(tohp);
                        fdbuflen = (int)_TPI_TOPT_DATALEN(tohp);

                        if (!fdbuf_verify(mp, fdbuf, fdbuflen))
                                continue;
                        if (oldflg) {
                                cmsglen += fdbuf_cmsglen(fdbuflen);
                                continue;
                        }
                        len = fdbuf_cmsglen(fdbuflen);
                } else if (tohp->level == SOL_SOCKET &&
                    tohp->name == SCM_TIMESTAMP) {
                        if (oldflg)
                                continue;

                        if (get_udatamodel() == DATAMODEL_NATIVE) {
                                len = sizeof (struct timeval);
                        } else {
                                len = sizeof (struct timeval32);
                        }
                } else {
                        if (oldflg)
                                continue;
                        len = (t_uscalar_t)_TPI_TOPT_DATALEN(tohp);
                }
                /*
                 * Exclude roundup for last option to not set
                 * MSG_CTRUNC when the cmsg fits but the padding doesn't fit.
                 */
                last_roundup = (t_uscalar_t)
                    (ROUNDUP_cmsglen(len + (int)sizeof (struct cmsghdr)) -
                    (len + (int)sizeof (struct cmsghdr)));
                cmsglen += (t_uscalar_t)(len + (int)sizeof (struct cmsghdr)) +
                    last_roundup;
        }
        cmsglen -= last_roundup;
        dprint(1, ("so_cmsglen: optlen %d, flg %d -> cmsglen %d\n",
            optlen, oldflg, cmsglen));
        return (cmsglen);
}

/*
 * Copy options from options to the control. Convert SO_FILEP to
 * file descriptors.
 * Returns errno or zero.
 * so_opt2cmsg and so_cmsglen are inter-related since so_cmsglen
 * allocates the space that so_opt2cmsg fills. If one changes, the other should
 * also be checked for any possible impacts.
 */
int
so_opt2cmsg(mblk_t *mp, void *opt, t_uscalar_t optlen, int oldflg,
```

```
1186     void *control, t_uscalar_t controllen)
1187 {
1188         struct T_opthdr *tohp;
1189         struct cmsghdr *cmsg;
1190         struct fdbuf *fdbuf;
1191         int fdbuflen;
1192         int error;
1193 #if defined(DEBUG) || defined(__lint)
1194         struct cmsghdr *cend = (struct cmsghdr *)
1195             (((uint8_t *)control) + ROUNDUP_cmsglen(controllen));
1196 #endif
1197         cmsg = (struct cmsghdr *)control;

1199         ASSERT(__TPI_TOPT_ISALIGNED(opt));

1201         for (tohp = (struct T_opthdr *)opt;
1202             tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1203             tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1204                 dprint(1, ("so_opt2cmsg: level 0x%x, name %d, len %d\n",
1205                     tohp->level, tohp->name, tohp->len));

1207                 if (tohp->level == SOL_SOCKET &&
1208                     (tohp->name == SO_SRCADDR ||
1209                     tohp->name == SO_UNIX_CLOSE)) {
1210                         continue;
1211                 }
1212                 ASSERT((uintptr_t)cmsg <= (uintptr_t)control + controllen);
1213                 if (tohp->level == SOL_SOCKET && tohp->name == SO_FILEP) {
1214                         fdbuf = (struct fdbuf *)_TPI_TOPT_DATA(tohp);
1215                         fdbuflen = (int)_TPI_TOPT_DATALEN(tohp);

1217                         if (!fdbuf_verify(mp, fdbuf, fdbuflen))
1218                                 return (EPROTO);
1219                         if (oldflg) {
1220                                 error = fdbuf_extract(fdbuf, control,
1221                                     (int)controllen);
1222                                 if (error != 0)
1223                                         return (error);
1224                                 continue;
1225                         } else {
1226                                 int fdlen;

1228                                 fdlen = (int)fdbuf_cmsglen(
1229                                     (int)_TPI_TOPT_DATALEN(tohp));

1231                                 cmsg->cmsg_level = tohp->level;
1232                                 cmsg->cmsg_type = SCM_RIGHTS;
1233                                 cmsg->cmsg_len = (socklen_t)(fdlen +
1234                                     sizeof (struct cmsghdr));

1236                                 error = fdbuf_extract(fdbuf,
1237                                     CMSG_CONTENT(cmsg), fdlen);
1238                                 if (error != 0)
1239                                         return (error);
1240                         }
1241                 } else if (tohp->level == SOL_SOCKET &&
1242                     tohp->name == SCM_TIMESTAMP) {
1243                         timestruc_t *timestamp;

1245                         if (oldflg)
1246                                 continue;

1248                         cmsg->cmsg_level = tohp->level;
1249                         cmsg->cmsg_type = tohp->name;

1251                         timestamp =
```



```
1252                             (timestruc_t *)P2ROUNDUP((intptr_t)&tohp[1],
1253                             sizeof (intptr_t));

1255                         if (get_udatamodel() == DATAMODEL_NATIVE) {
1256                                 struct timeval tv;

1258                                 cmsg->cmsg_len = sizeof (struct timeval) +
1259                                     sizeof (struct cmsghdr);
1260                                 tv.tv_sec = timestamp->tv_sec;
1261                                 tv.tv_usec = timestamp->tv_nsec /
1262                                     (NANOSEC / MICROSEC);
1263                                 /*
1264                                  * on LP64 systems, the struct timeval in
1265                                  * the destination will not be 8-byte aligned,
1266                                  * so use bcopy to avoid alignment trouble
1267                                  */
1268                                 bcopy(&tv, CMSG_CONTENT(cmsg), sizeof (tv));
1269                         } else {
1270                                 struct timeval32 *time32;

1272                                 cmsg->cmsg_len = sizeof (struct timeval32) +
1273                                     sizeof (struct cmsghdr);
1274                                 time32 = (struct timeval32 *)CMSG_CONTENT(cmsg);
1275                                 time32->tv_sec = (time32_t)timestamp->tv_sec;
1276                                 time32->tv_usec =
1277                                     (int32_t)(timestamp->tv_nsec /
1278                                     (NANOSEC / MICROSEC));
1279                         }

1281                 } else {
1282                         if (oldflg)
1283                                 continue;

1285                         cmsg->cmsg_level = tohp->level;
1286                         cmsg->cmsg_type = tohp->name;
1287                         cmsg->cmsg_len = (socklen_t)(_TPI_TOPT_DATALEN(tohp) +
1288                             sizeof (struct cmsghdr));

1290                         /* copy content to control data part */
1291                         bcopy(&tohp[1], CMSG_CONTENT(cmsg),
1292                             CMSG_CONTENTLEN(cmsg));
1293                 }
1294                 /* move to next CMSG structure! */
1295                 cmsg = CMSG_NEXT(cmsg);
1296         }
1297         dprint(1, ("so_opt2cmsg: buf %p len %d; cend %p; final cmsg %p\n",
1298             control, controllen, (void *)cend, (void *)cmsg));
1299         ASSERT(cmsg <= cend);
1300         return (0);
1301 }

1303 /*
1304  * Extract the SO_SRCADDR option value if present.
1305  */
1306 void
1307 so_getopt_srcaddr(void *opt, t_uscalar_t optlen, void **srcp,
1308     t_uscalar_t *srclenp)
1309 {
1310         struct T_opthdr         *tohp;

1312         ASSERT(__TPI_TOPT_ISALIGNED(opt));

1314         ASSERT(srcp != NULL && srclenp != NULL);
1315         *srcp = NULL;
1316         *srclenp = 0;
```

```
1318         for (tohp = (struct T_opthdr *)opt;
1319             tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1320             tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1321                 dprint(1, ("so_getopt_srcaddr: level 0x%x, name %d, len %d\n",
1322                     tohp->level, tohp->name, tohp->len));
1323                 if (tohp->level == SOL_SOCKET &&
1324                     tohp->name == SO_SRCADDR) {
1325                         *srcp = _TPI_TOPT_DATA(tohp);
1326                         *srclenp = (t_uscalar_t)_TPI_TOPT_DATALEN(tohp);
1327                 }
1328         }
1329 }

1331 /*
1332  * Verify if the SO_UNIX_CLOSE option is present.
1333  */
1334 int
1335 so_getopt_unix_close(void *opt, t_uscalar_t optlen)
1336 {
1337         struct T_opthdr         *tohp;

1339         ASSERT(__TPI_TOPT_ISALIGNED(opt));

1341         for (tohp = (struct T_opthdr *)opt;
1342             tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1343             tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1344                 dprint(1,
1345                     ("so_getopt_unix_close: level 0x%x, name %d, len %d\n",
1346                     tohp->level, tohp->name, tohp->len));
1347                 if (tohp->level == SOL_SOCKET &&
1348                     tohp->name == SO_UNIX_CLOSE)
1349                         return (1);
1350         }
1351         return (0);
1352 }

1354 /*
1355  * Allocate an M_PROTO message.
1356  *
1357  * If allocation fails the behavior depends on sleepflg:
1358  *      _ALLOC_NOSLEEP  fail immediately
1359  *      _ALLOC_INTR     sleep for memory until a signal is caught
1360  *      _ALLOC_SLEEP    sleep forever. Don't return NULL.
1361  */
1362 mblk_t *
1363 soallocproto(size_t size, int sleepflg, cred_t *cr)
1364 {
1365         mblk_t  *mp;

1367         /* Round up size for reuse */
1368         size = MAX(size, 64);
1369         if (cr != NULL)
1370                 mp = allocb_cred(size, cr, curproc->p_pid);
1371         else
1372                 mp = allocb(size, BPRI_MED);

1374         if (mp == NULL) {
1375                 int error;      /* Dummy - error not returned to caller */

1377                 switch (sleepflg) {
1378                 case _ALLOC_SLEEP:
1379                         if (cr != NULL) {
1380                                 mp = allocb_cred_wait(size, STR_NOSIG, &error,
1381                                     cr, curproc->p_pid);
1382                         } else {
1383                                 mp = allocb_wait(size, BPRI_MED, STR_NOSIG,
```



```
1384                                     &error);
1385                         }
1386                         ASSERT(mp);
1387                         break;
1388                 case _ALLOC_INTR:
1389                         if (cr != NULL) {
1390                                 mp = allocb_cred_wait(size, 0, &error, cr,
1391                                     curproc->p_pid);
1392                         } else {
1393                                 mp = allocb_wait(size, BPRI_MED, 0, &error);
1394                         }
1395                         if (mp == NULL) {
1396                                 /* Caught signal while sleeping for memory */
1397                                 eprintline(ENOBUFS);
1398                                 return (NULL);
1399                         }
1400                         break;
1401                 case _ALLOC_NOSLEEP:
1402                 default:
1403                         eprintline(ENOBUFS);
1404                         return (NULL);
1405                 }
1406         }
1407         DB_TYPE(mp) = M_PROTO;
1408         return (mp);
1409 }

1411 /*
1412  * Allocate an M_PROTO message with a single component.
1413  * len is the length of buf. size is the amount to allocate.
1414  *
1415  * buf can be NULL with a non-zero len.
1416  * This results in a bzero'ed chunk being placed the message.
1417  */
1418 mblk_t *
1419 soallocproto1(const void *buf, ssize_t len, ssize_t size, int sleepflg,
1420     cred_t *cr)
1421 {
1422         mblk_t  *mp;

1424         if (size == 0)
1425                 size = len;

1427         ASSERT(size >= len);
1428         /* Round up size for reuse */
1429         size = MAX(size, 64);
1430         mp = soallocproto(size, sleepflg, cr);
1431         if (mp == NULL)
1432                 return (NULL);
1433         mp->b_datap->db_type = M_PROTO;
1434         if (len != 0) {
1435                 if (buf != NULL)
1436                         bcopy(buf, mp->b_wptr, len);
1437                 else
1438                         bzero(mp->b_wptr, len);
1439                 mp->b_wptr += len;
1440         }
1441         return (mp);
1442 }

1444 /*
1445  * Append buf/len to mp.
1446  * The caller has to ensure that there is enough room in the mblk.
1447  *
1448  * buf can be NULL with a non-zero len.
1449  * This results in a bzero'ed chunk being placed the message.
```

```
 */
void
soappendmsg(mblk_t *mp, const void *buf, ssize_t len)
{
        ASSERT(mp);

        if (len != 0) {
                /* Assert for room left */
                ASSERT(mp->b_datap->db_lim - mp->b_wptr >= len);
                if (buf != NULL)
                        bcopy(buf, mp->b_wptr, len);
                else
                        bzero(mp->b_wptr, len);
        }
        mp->b_wptr += len;
}

/*
 * Create a message using two kernel buffers.
 * If size is set that will determine the allocation size (e.g. for future
 * soappendmsg calls). If size is zero it is derived from the buffer
 * lengths.
 */
mblk_t *
soallocproto2(const void *buf1, ssize_t len1, const void *buf2, ssize_t len2,
    ssize_t size, int sleepflg, cred_t *cr)
{
        mblk_t *mp;

        if (size == 0)
                size = len1 + len2;
        ASSERT(size >= len1 + len2);

        mp = soallocproto1(buf1, len1, size, sleepflg, cr);
        if (mp)
                soappendmsg(mp, buf2, len2);
        return (mp);
}

/*
 * Create a message using three kernel buffers.
 * If size is set that will determine the allocation size (for future
 * soappendmsg calls). If size is zero it is derived from the buffer
 * lengths.
 */
mblk_t *
soallocproto3(const void *buf1, ssize_t len1, const void *buf2, ssize_t len2,
    const void *buf3, ssize_t len3, ssize_t size, int sleepflg, cred_t *cr)
{
        mblk_t *mp;

        if (size == 0)
                size = len1 + len2 +len3;
        ASSERT(size >= len1 + len2 + len3);

        mp = soallocproto1(buf1, len1, size, sleepflg, cr);
        if (mp != NULL) {
                soappendmsg(mp, buf2, len2);
                soappendmsg(mp, buf3, len3);
        }
        return (mp);
}

#ifdef DEBUG
char *
pr_state(uint_t state, uint_t mode)
```



```
{
        static char buf[1024];

        buf[0] = 0;
        if (state & SS_ISCONNECTED)
                (void) strcat(buf, "ISCONNECTED ");
        if (state & SS_ISCONNECTING)
                (void) strcat(buf, "ISCONNECTING ");
        if (state & SS_ISDISCONNECTING)
                (void) strcat(buf, "ISDISCONNECTING ");
        if (state & SS_CANTSENDMORE)
                (void) strcat(buf, "CANTSENDMORE ");

        if (state & SS_CANTRCVMORE)
                (void) strcat(buf, "CANTRCVMORE ");
        if (state & SS_ISBOUND)
                (void) strcat(buf, "ISBOUND ");
        if (state & SS_NDELAY)
                (void) strcat(buf, "NDELAY ");
        if (state & SS_NONBLOCK)
                (void) strcat(buf, "NONBLOCK ");

        if (state & SS_ASYNC)
                (void) strcat(buf, "ASYNC ");
        if (state & SS_ACCEPTCONN)
                (void) strcat(buf, "ACCEPTCONN ");
        if (state & SS_SAVEDEOR)
                (void) strcat(buf, "SAVEDEOR ");

        if (state & SS_RCVATMARK)
                (void) strcat(buf, "RCVATMARK ");
        if (state & SS_OOBPEND)
                (void) strcat(buf, "OOBPEND ");
        if (state & SS_HAVEOOBDATA)
                (void) strcat(buf, "HAVEOOBDATA ");
        if (state & SS_HADOOBDATA)
                (void) strcat(buf, "HADOOBDATA ");

        if (mode & SM_PRIV)
                (void) strcat(buf, "PRIV ");
        if (mode & SM_ATOMIC)
                (void) strcat(buf, "ATOMIC ");
        if (mode & SM_ADDR)
                (void) strcat(buf, "ADDR ");
        if (mode & SM_CONNREQUIRED)
                (void) strcat(buf, "CONNREQUIRED ");

        if (mode & SM_FDPASSING)
                (void) strcat(buf, "FDPASSING ");
        if (mode & SM_EXDATA)
                (void) strcat(buf, "EXDATA ");
        if (mode & SM_OPTDATA)
                (void) strcat(buf, "OPTDATA ");
        if (mode & SM_BYTESTREAM)
                (void) strcat(buf, "BYTESTREAM ");
        return (buf);
}

char *
pr_addr(int family, struct sockaddr *addr, t_uscalar_t addrlen)
{
        static char buf[1024];

        if (addr == NULL || addrlen == 0) {
                (void) sprintf(buf, "(len %d) %p", addrlen, (void *)addr);
                return (buf);
```

```
1582         }
1583         switch (family) {
1584         case AF_INET: {
1585                 struct sockaddr_in sin;

1587                 bcopy(addr, &sin, sizeof (sin));

1589                 (void) sprintf(buf, "(len %d) %x/%d",
1590                     addrlen, ntohl(sin.sin_addr.s_addr), ntohs(sin.sin_port));
1591                 break;
1592         }
1593         case AF_INET6: {
1594                 struct sockaddr_in6 sin6;
1595                 uint16_t *piece = (uint16_t *)&sin6.sin6_addr;

1597                 bcopy((char *)addr, (char *)&sin6, sizeof (sin6));
1598                 (void) sprintf(buf, "(len %d) %x:%x:%x:%x:%x:%x:%x:%x/%d",
1599                     addrlen,
1600                     ntohs(piece[0]), ntohs(piece[1]),
1601                     ntohs(piece[2]), ntohs(piece[3]),
1602                     ntohs(piece[4]), ntohs(piece[5]),
1603                     ntohs(piece[6]), ntohs(piece[7]),
1604                     ntohs(sin6.sin6_port));
1605                 break;
1606         }
1607         case AF_UNIX: {
1608                 struct sockaddr_un *soun = (struct sockaddr_un *)addr;

1610                 (void) sprintf(buf, "(len %d) %s", addrlen,
1611                     (soun == NULL) ? "(none)" : soun->sun_path);
1612                 break;
1613         }
1614         default:
1615                 (void) sprintf(buf, "(unknown af %d)", family);
1616                 break;
1617         }
1618         return (buf);
1619 }

1621 /* The logical equivalence operator (a if-and-only-if b) */
1622 #define EQUIVALENT(a, b)        (((a) && (b)) || (!(a) && (!(b))))

1624 /*
1625  * Verify limitations and invariants on oob state.
1626  * Return 1 if OK, otherwise 0 so that it can be used as
1627  *      ASSERT(verify_oobstate(so));
1628  */
1629 int
1630 so_verify_oobstate(struct sonode *so)
1631 {
1632         boolean_t havemark;

1634         ASSERT(MUTEX_HELD(&so->so_lock));

1636         /*
1637          * The possible state combinations are:
1638          *      0
1639          *      SS_OOBPEND
1640          *      SS_OOBPEND|SS_HAVEOOBDATA
1641          *      SS_OOBPEND|SS_HADOOBDATA
1642          *      SS_HADOOBDATA
1643          */
1644         switch (so->so_state & (SS_OOBPEND|SS_HAVEOOBDATA|SS_HADOOBDATA)) {
1645         case 0:
1646         case SS_OOBPEND:
1647         case SS_OOBPEND|SS_HAVEOOBDATA:
```



```
1648         case SS_OOBPEND|SS_HADOOBDATA:
1649         case SS_HADOOBDATA:
1650                 break;
1651         default:
1652                 printf("Bad oob state 1 (%p): state %s\n",
1653                     (void *)so, pr_state(so->so_state, so->so_mode));
1654                 return (0);
1655         }

1657         /* SS_RCVATMARK should only be set when SS_OOBPEND is set */
1658         if ((so->so_state & (SS_RCVATMARK|SS_OOBPEND)) == SS_RCVATMARK) {
1659                 printf("Bad oob state 2 (%p): state %s\n",
1660                     (void *)so, pr_state(so->so_state, so->so_mode));
1661                 return (0);
1662         }

1664         /*
1665          * (havemark != 0 or SS_RCVATMARK) iff SS_OOBPEND
1666          * For TPI, the presence of a "mark" is indicated by sti_oobsigcnt.
1667          */
1668         havemark = (SOCK_IS_NONSTR(so)) ? so->so_oobmark > 0 :
1669             SOTOTPI(so)->sti_oobsigcnt > 0;

1671         if (!EQUIVALENT(havemark || (so->so_state & SS_RCVATMARK),
1672             so->so_state & SS_OOBPEND)) {
1673                 printf("Bad oob state 3 (%p): state %s\n",
1674                     (void *)so, pr_state(so->so_state, so->so_mode));
1675                 return (0);
1676         }

1678         /*
1679          * Unless SO_OOBINLINE we have so_oobmsg != NULL iff SS_HAVEOOBDATA
1680          */
1681         if (!(so->so_options & SO_OOBINLINE) &&
1682             !EQUIVALENT(so->so_oobmsg != NULL, so->so_state & SS_HAVEOOBDATA)) {
1683                 printf("Bad oob state 4 (%p): state %s\n",
1684                     (void *)so, pr_state(so->so_state, so->so_mode));
1685                 return (0);
1686         }

1688         if (!SOCK_IS_NONSTR(so) &&
1689             SOTOTPI(so)->sti_oobsigcnt < SOTOTPI(so)->sti_oobcnt) {
1690                 printf("Bad oob state 5 (%p): counts %d/%d state %s\n",
1691                     (void *)so, SOTOTPI(so)->sti_oobsigcnt,
1692                     SOTOTPI(so)->sti_oobcnt,
1693                     pr_state(so->so_state, so->so_mode));
1694                 return (0);
1695         }

1697         return (1);
1698 }
1699 #undef  EQUIVALENT
1700 #endif /* DEBUG */

1702 /* initialize sockfs zone specific kstat related items                  */
1703 void *
1704 sock_kstat_init(zoneid_t zoneid)
1705 {
1706         kstat_t *ksp;

1708         ksp = kstat_create_zone("sockfs", 0, "sock_unix_list", "misc",
1709             KSTAT_TYPE_RAW, 0, KSTAT_FLAG_VAR_SIZE|KSTAT_FLAG_VIRTUAL, zoneid);

1711         if (ksp != NULL) {
1712                 ksp->ks_update = sockfs_update;
1713                 ksp->ks_snapshot = sockfs_snapshot;
```

```
1714                ksp->ks_lock = &socklist.sl_lock;
1715                ksp->ks_private = (void *)(uintptr_t)zoneid;
1716                kstat_install(ksp);
1717        }

1719        return (ksp);
1720 }

1722 /* tear down sockfs zone specific kstat related items                   */
1723 /*ARGSUSED*/
1724 void
1725 sock_kstat_fini(zoneid_t zoneid, void *arg)
1726 {
1727        kstat_t *ksp = (kstat_t *)arg;

1729        if (ksp != NULL) {
1730                ASSERT(zoneid == (zoneid_t)(uintptr_t)ksp->ks_private);
1731                kstat_delete(ksp);
1732        }
1733 }

1735 /*
1736  * Zones:
1737  * Note that nactive is going to be different for each zone.
1738  * This means we require kstat to call sockfs_update and then sockfs_snapshot
1739  * for the same zone, or sockfs_snapshot will be taken into the wrong size
1740  * buffer. This is safe, but if the buffer is too small, user will not be
1741  * given details of all sockets. However, as this kstat has a ks_lock, kstat
1742  * driver will keep it locked between the update and the snapshot, so no
1743  * other process (zone) can currently get inbetween resulting in a wrong size
1744  * buffer allocation.
1745  */
1746 static int
1747 sockfs_update(kstat_t *ksp, int rw)
1748 {
1749        uint_t  n, nactive = 0;         /* # of active AF_UNIX sockets  */
1750        uint_t  tsze;
 769        uint_t  nactive = 0;            /* # of active AF_UNIX sockets  */
1751        struct sonode   *so;            /* current sonode on socklist   */
1752        zoneid_t        myzoneid = (zoneid_t)(uintptr_t)ksp->ks_private;

1754        tsze = 0;

1756 #endif /* ! codereview */
1757        ASSERT((zoneid_t)(uintptr_t)ksp->ks_private == getzoneid());

1759        if (rw == KSTAT_WRITE) {        /* bounce all writes            */
1760                return (EACCES);
1761        }

1763        for (so = socklist.sl_list; so != NULL; so = SOTOTPI(so)->sti_next_so) {
1764                if (so->so_count != 0 && so->so_zoneid == myzoneid) {

1766 #endif /* ! codereview */
1767                        nactive++;

1769                        mutex_enter(&so->so_pid_list_lock);
1770                        n = list_numnodes(&so->so_pid_list);
1771                        mutex_exit(&so->so_pid_list_lock);

1773                        tsze += sizeof (struct sockinfo);
1774                        tsze += (n > 1) ? ((n - 1) * sizeof (pid_t)) : 0;
1775 #endif /* ! codereview */
1776                }
1777        }
1778        ksp->ks_ndata = nactive;
```



```
1779        ksp->ks_data_size = tsze;
 773        ksp->ks_data_size = nactive * sizeof (struct k_sockinfo);

1781        return (0);
1782 }

1784 static int
1785 sockfs_snapshot(kstat_t *ksp, void *buf, int rw)
1786 {
1787        int                     ns;     /* # of sonodes we've copied    */
1788        struct sonode           *so;    /* current sonode on socklist   */
1789        struct sockinfo         *psi;   /* where we put sockinfo data   */
 783        struct k_sockinfo       *pksi;  /* where we put sockinfo data   */
1790        t_uscalar_t             sn_len; /* soa_len                      */
1791        zoneid_t                myzoneid = (zoneid_t)(uintptr_t)ksp->ks_private;
1792        sotpi_info_t            *sti;

1794        uint_t                          sze;
1795        mblk_t                          *mblk;
1796        conn_pid_info_t                 *cpi;

1798 #endif /* ! codereview */
1799        ASSERT((zoneid_t)(uintptr_t)ksp->ks_private == getzoneid());

1801        ksp->ks_snaptime = gethrtime();

1803        if (rw == KSTAT_WRITE) {        /* bounce all writes            */
1804                return (EACCES);
1805        }

1807        /*
1808         * for each sonode on the socklist, we massage the important
1809         * info into buf, in k_sockinfo format.
1810         */
1811        psi = (struct sockinfo *)buf;
 788        pksi = (struct k_sockinfo *)buf;
1812        ns = 0;
1813        for (so = socklist.sl_list; so != NULL; so = SOTOTPI(so)->sti_next_so) {
1814                /* only stuff active sonodes and the same zone:         */
1815                if (so->so_count == 0 || so->so_zoneid != myzoneid) {
1816                        continue;
1817                }

1819                mblk = so_get_sock_pid_mblk((sock_upper_handle_t)so);
1820                if (mblk == NULL) {
1821                        continue;
1822                }
1823                cpi = (conn_pid_info_t *)mblk->b_datap->db_base;
1824                sze = sizeof (struct sockinfo);
1825                sze += (cpi->cpi_pids_cnt > 1) ?
1826                        ((cpi->cpi_pids_cnt - 1) * sizeof (pid_t)) : 0;

1828 #endif /* ! codereview */
1829                /*
1830                 * If the sonode was activated between the update and the
1831                 * snapshot, we're done - as this is only a snapshot. We need
1832                 * to make sure that we have space for this sockinfo. In the
1833                 * time window between the update and the snapshot, the size of
1834                 * sockinfo my change, as new pids are added/removed to/from
1835                 * the list. We have to take that into consideration and only
1836                 * include the sockinfo if we have enough space. That means the
1837                 * number of entries we return by snapshot might not equal the
1838                 * the number of entries calculated by update.
 796                 * snapshot, we're done - as this is only a snapshot.
1839                 */
1840                if (((caddr_t)(psi) + sze) >
```

```
1841                        ((caddr_t)buf + ksp->ks_data_size)) {
 798                if ((caddr_t)(pksi) >= (caddr_t)buf + ksp->ks_data_size) {
1842                        break;
1843                }

1845                sti = SOTOTPI(so);
1846                /* copy important info into buf:                        */
1847                psi->si_size = sze;
1848                psi->si_family = so->so_family;
1849                psi->si_type = so->so_type;
1850                psi->si_flag = so->so_flag;
1851                psi->si_state = so->so_state;
1852                psi->si_serv_type = sti->sti_serv_type;
1853                psi->si_ux_laddr_sou_magic =
 804                pksi->ks_si.si_size = sizeof (struct k_sockinfo);
 805                pksi->ks_si.si_family = so->so_family;
 806                pksi->ks_si.si_type = so->so_type;
 807                pksi->ks_si.si_flag = so->so_flag;
 808                pksi->ks_si.si_state = so->so_state;
 809                pksi->ks_si.si_serv_type = sti->sti_serv_type;
 810                pksi->ks_si.si_ux_laddr_sou_magic =
1854                    sti->sti_ux_laddr.soua_magic;
1855                psi->si_ux_faddr_sou_magic =
 812                pksi->ks_si.si_ux_faddr_sou_magic =
1856                    sti->sti_ux_faddr.soua_magic;
1857                psi->si_laddr_soa_len = sti->sti_laddr.soa_len;
1858                psi->si_faddr_soa_len = sti->sti_faddr.soa_len;
1859                psi->si_szoneid = so->so_zoneid;
1860                psi->si_faddr_noxlate = sti->sti_faddr_noxlate;

 814                pksi->ks_si.si_laddr_soa_len = sti->sti_laddr.soa_len;
 815                pksi->ks_si.si_faddr_soa_len = sti->sti_faddr.soa_len;
 816                pksi->ks_si.si_szoneid = so->so_zoneid;
 817                pksi->ks_si.si_faddr_noxlate = sti->sti_faddr_noxlate;

1863                mutex_enter(&so->so_lock);

1865                if (sti->sti_laddr_sa != NULL) {
1866                        ASSERT(sti->sti_laddr_sa->sa_data != NULL);
1867                        sn_len = sti->sti_laddr_len;
1868                        ASSERT(sn_len <= sizeof (short) +
1869                            sizeof (psi->si_laddr_sun_path));
 825                            sizeof (pksi->ks_si.si_laddr_sun_path));

1871                        psi->si_laddr_family =
 827                        pksi->ks_si.si_laddr_family =
1872                            sti->sti_laddr_sa->sa_family;
1873                        if (sn_len != 0) {
1874                                /* AF_UNIX socket names are NULL terminated */
1875                                (void) strncpy(psi->si_laddr_sun_path,
 831                                (void) strncpy(pksi->ks_si.si_laddr_sun_path,
1876                                    sti->sti_laddr_sa->sa_data,
1877                                    sizeof (psi->si_laddr_sun_path));
1878                                sn_len = strlen(psi->si_laddr_sun_path);
 833                                    sizeof (pksi->ks_si.si_laddr_sun_path));
 834                                sn_len = strlen(pksi->ks_si.si_laddr_sun_path);
1879                        }
1880                        psi->si_laddr_sun_path[sn_len] = 0;
 836                        pksi->ks_si.si_laddr_sun_path[sn_len] = 0;
1881                }

1883                if (sti->sti_faddr_sa != NULL) {
1884                        ASSERT(sti->sti_faddr_sa->sa_data != NULL);
1885                        sn_len = sti->sti_faddr_len;
1886                        ASSERT(sn_len <= sizeof (short) +
1887                            sizeof (psi->si_faddr_sun_path));
```



```
 843                            sizeof (pksi->ks_si.si_faddr_sun_path));

1889                        psi->si_faddr_family =
 845                        pksi->ks_si.si_faddr_family =
1890                            sti->sti_faddr_sa->sa_family;
1891                        if (sn_len != 0) {
1892                                (void) strncpy(psi->si_faddr_sun_path,
 848                                (void) strncpy(pksi->ks_si.si_faddr_sun_path,
1893                                    sti->sti_faddr_sa->sa_data,
1894                                    sizeof (psi->si_faddr_sun_path));
1895                                sn_len = strlen(psi->si_faddr_sun_path);
 850                                    sizeof (pksi->ks_si.si_faddr_sun_path));
 851                                sn_len = strlen(pksi->ks_si.si_faddr_sun_path);
1896                        }
1897                        psi->si_faddr_sun_path[sn_len] = 0;
 853                        pksi->ks_si.si_faddr_sun_path[sn_len] = 0;
1898                }

1900                mutex_exit(&so->so_lock);

1902                (void) sprintf(psi->si_son_straddr, "%p", (void *)so);
1903                (void) sprintf(psi->si_lvn_straddr, "%p",
 858                (void) sprintf(pksi->ks_straddr[0], "%p", (void *)so);
 859                (void) sprintf(pksi->ks_straddr[1], "%p",
1904                    (void *)sti->sti_ux_laddr.soua_vp);
1905                (void) sprintf(psi->si_fvn_straddr, "%p",
 861                (void) sprintf(pksi->ks_straddr[2], "%p",
1906                    (void *)sti->sti_ux_faddr.soua_vp);

1908                psi->si_pids[0] = 0;
1909                if ((psi->si_pn_cnt = cpi->cpi_pids_cnt) > 0) {
1910                        (void) memcpy(psi->si_pids, cpi->cpi_pids,
1911                            psi->si_pn_cnt * sizeof (pid_t));
1912                }

1914                freemsg(mblk);

1916                psi = (struct sockinfo *)((caddr_t)psi + psi->si_size);
1917 #endif /* ! codereview */
1918                ns++;
 864                pksi++;
1919        }

1921        ksp->ks_ndata = ns;
1922        return (0);
1923 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   80817 Mon Aug 17 21:08:04 2015
new/usr/src/uts/common/inet/ip/ipclassifier.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_
2724 #endif

2726 mblk_t *
2727 conn_get_pid_mblk(conn_t *connp)
2728 {
2729         mblk_t *mblk;
2730         conn_pid_info_t *cpi;

2732         if (connp->conn_upper_handle != NULL) {
2733                 return (*connp->conn_upcalls->su_get_sock_pid_mblk)
2734                     (connp->conn_upper_handle);
2735         } else if (!IPCL_IS_NONSTR(connp) && connp->conn_rq != NULL &&
2736             connp->conn_rq->q_stream != NULL) {
2737                 return (sh_get_pid_mblk(connp->conn_rq->q_stream));
2738         }

2740         /* return an empty mblk */
2741         if ((mblk = allocb(sizeof (conn_pid_info_t), BPRI_HI)) == NULL)
2742                 return (NULL);
2743         mblk->b_wptr += sizeof (conn_pid_info_t);
2744         cpi = (conn_pid_info_t *) mblk->b_datap->db_base;
2745         cpi->cpi_magic = CONN_PID_INFO_MGC;
2746         cpi->cpi_contents = CONN_PID_INFO_NON;
2747         cpi->cpi_pids_cnt = 0;
2748         cpi->cpi_tot_size = sizeof (conn_pid_info_t);
2749         cpi->cpi_pids[0] = 0;
2750         return (mblk);
2751 }
2752 #endif /* ! codereview */
```

```
**********************************************************
   26483 Mon Aug 17 21:08:05 2015
new/usr/src/uts/common/inet/ipclassifier.h
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 508 /*
 509  * For use with subsystems within ip which use ALL_ZONES as a wildcard
 510  */
 511 #define IPCL_ZONEID(connp)                                              \
 512         ((connp)->conn_allzones ? ALL_ZONES : (connp)->conn_zoneid)

 514 /*
 515  * For matching between a conn_t and a zoneid.
 516  */
 517 #define IPCL_ZONE_MATCH(connp, zoneid)                                  \
 518         (((connp)->conn_allzones) ||                                    \
 519             ((zoneid) == ALL_ZONES) ||                                  \
 520             (connp)->conn_zoneid == (zoneid))

 522 /*
 523  * On a labeled system, we must treat bindings to ports
 524  * on shared IP addresses by sockets with MAC exemption
 525  * privilege as being in all zones, as there's
 526  * otherwise no way to identify the right receiver.
 527  */

 529 #define IPCL_CONNS_MAC(conn1, conn2)                                    \
 530         (((conn1)->conn_mac_mode != CONN_MAC_DEFAULT) ||                \
 531         ((conn2)->conn_mac_mode != CONN_MAC_DEFAULT))

 533 #define IPCL_BIND_ZONE_MATCH(conn1, conn2)                              \
 534         (IPCL_CONNS_MAC(conn1, conn2) ||                               \
 535         IPCL_ZONE_MATCH(conn1, conn2->conn_zoneid) ||                  \
 536         IPCL_ZONE_MATCH(conn2, conn1->conn_zoneid))


 539 #define _IPCL_V4_MATCH(v6addr, v4addr)  \
 540         (V4_PART_OF_V6((v6addr)) == (v4addr) && IN6_IS_ADDR_V4MAPPED(&(v6addr)))

 542 #define _IPCL_V4_MATCH_ANY(addr)        \
 543         (IN6_IS_ADDR_V4MAPPED_ANY(&(addr)) || IN6_IS_ADDR_UNSPECIFIED(&(addr)))


 546 /*
 547  * IPCL_PROTO_MATCH() and IPCL_PROTO_MATCH_V6() only matches conns with
 548  * the specified ira_zoneid or conn_allzones by calling conn_wantpacket.
 549  */
 550 #define IPCL_PROTO_MATCH(connp, ira, ipha)                              \
 551         ((((connp)->conn_laddr_v4 == INADDR_ANY) ||                    \
 552         (((connp)->conn_laddr_v4 == ((ipha)->ipha_dst)) &&             \
 553             (((connp)->conn_faddr_v4 == INADDR_ANY) ||                 \
 554         ((connp)->conn_faddr_v4 == ((ipha)->ipha_src))))) &&           \
 555         conn_wantpacket((connp), (ira), (ipha)))

 557 #define IPCL_PROTO_MATCH_V6(connp, ira, ip6h)                           \
 558         ((IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) ||          \
 559         (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &((ip6h)->ip6_dst)) &&   \
 560         (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) ||           \
 561         IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &((ip6h)->ip6_src))))) && \
 562         (conn_wantpacket_v6((connp), (ira), (ip6h))))

 564 #define IPCL_CONN_HASH(src, ports, ipst)                                \
 565         ((unsigned)(ntohl((src)) ^ ((ports) >> 24) ^ ((ports) >> 16) ^  \
 566         ((ports) >> 8) ^ (ports)) % (ipst)->ips_ipcl_conn_fanout_size)
```



```
 568 #define IPCL_CONN_HASH_V6(src, ports, ipst)                             \
 569         IPCL_CONN_HASH(V4_PART_OF_V6((src)), (ports), (ipst))

 571 #define IPCL_CONN_MATCH(connp, proto, src, dst, ports)                  \
 572         ((connp)->conn_proto == (proto) &&                             \
 573                 (connp)->conn_ports == (ports) &&                      \
 574                 _IPCL_V4_MATCH((connp)->conn_faddr_v6, (src)) &&       \
 575                 _IPCL_V4_MATCH((connp)->conn_laddr_v6, (dst)) &&       \
 576                 !(connp)->conn_ipv6_v6only)

 578 #define IPCL_CONN_MATCH_V6(connp, proto, src, dst, ports)               \
 579         ((connp)->conn_proto == (proto) &&                             \
 580                 (connp)->conn_ports == (ports) &&                      \
 581                 IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(src)) &&  \
 582                 IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(dst)))

 584 #define IPCL_PORT_HASH(port, size) \
 585         ((((port) >> 8) ^ (port)) & ((size) - 1))

 587 #define IPCL_BIND_HASH(lport, ipst)                                     \
 588         ((unsigned)(((lport) >> 8) ^ (lport)) % \
 589             (ipst)->ips_ipcl_bind_fanout_size)

 591 #define IPCL_BIND_MATCH(connp, proto, laddr, lport)                     \
 592         ((connp)->conn_proto == (proto) &&                             \
 593                 (connp)->conn_lport == (lport) &&                      \
 594                 (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) ||         \
 595                 _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))) &&    \
 596                 !(connp)->conn_ipv6_v6only)

 598 #define IPCL_BIND_MATCH_V6(connp, proto, laddr, lport)                  \
 599         ((connp)->conn_proto == (proto) &&                             \
 600                 (connp)->conn_lport == (lport) &&                      \
 601                 (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) || \
 602                 IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6)))

 604 /*
 605  * We compare conn_laddr since it captures both connected and a bind to
 606  * a multicast or broadcast address.
 607  * The caller needs to match the zoneid and also call conn_wantpacket
 608  * for multicast, broadcast, or when conn_incoming_ifindex is set.
 609  */
 610 #define IPCL_UDP_MATCH(connp, lport, laddr, fport, faddr)               \
 611         (((connp)->conn_lport == (lport)) &&                           \
 612         ((_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) ||                \
 613         (_IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) &&            \
 614         (_IPCL_V4_MATCH_ANY((connp)->conn_faddr_v6) ||                 \
 615         (_IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)) &&            \
 616         (connp)->conn_fport == (fport)))))) &&                         \
 617         !(connp)->conn_ipv6_v6only)

 619 /*
 620  * We compare conn_laddr since it captures both connected and a bind to
 621  * a multicast or broadcast address.
 622  * The caller needs to match the zoneid and also call conn_wantpacket_v6
 623  * for multicast or when conn_incoming_ifindex is set.
 624  */
 625 #define IPCL_UDP_MATCH_V6(connp, lport, laddr, fport, faddr)    \
 626         (((connp)->conn_lport == (lport)) &&                    \
 627         (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) ||    \
 628         (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) &&       \
 629         (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) ||    \
 630         (IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(faddr)) &&       \
 631         (connp)->conn_fport == (fport))))))
```

```
633 #define IPCL_IPTUN_HASH(laddr, faddr)                                   \
634         ((ntohl(laddr) ^ ((ntohl(faddr) << 24) | (ntohl(faddr) >> 8))) % \
635         ipcl_iptun_fanout_size)

637 #define IPCL_IPTUN_HASH_V6(laddr, faddr)                                \
638         IPCL_IPTUN_HASH((laddr)->s6_addr32[0] ^ (laddr)->s6_addr32[1] ^ \
639             (faddr)->s6_addr32[2] ^ (faddr)->s6_addr32[3],              \
640             (faddr)->s6_addr32[0] ^ (faddr)->s6_addr32[1] ^             \
641             (laddr)->s6_addr32[2] ^ (laddr)->s6_addr32[3])

643 #define IPCL_IPTUN_MATCH(connp, laddr, faddr)                   \
644         (_IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) &&     \
645         _IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)))

647 #define IPCL_IPTUN_MATCH_V6(connp, laddr, faddr)                \
648         (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, (laddr)) &&        \
649         IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, (faddr)))

651 #define IPCL_UDP_HASH(lport, ipst)      \
652         IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_udp_fanout_size)

654 #define CONN_G_HASH_SIZE        1024

656 /* Raw socket hash function. */
657 #define IPCL_RAW_HASH(lport, ipst)      \
658         IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_raw_fanout_size)

660 /*
661  * This is similar to IPCL_BIND_MATCH except that the local port check
662  * is changed to a wildcard port check.
663  * We compare conn_laddr since it captures both connected and a bind to
664  * a multicast or broadcast address.
665  */
666 #define IPCL_RAW_MATCH(connp, proto, laddr)                     \
667         ((connp)->conn_proto == (proto) &&                      \
668         (connp)->conn_lport == 0 &&                             \
669         (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) ||          \
670         _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))))

672 #define IPCL_RAW_MATCH_V6(connp, proto, laddr)                  \
673         ((connp)->conn_proto == (proto) &&                      \
674         (connp)->conn_lport == 0 &&                             \
675         (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) ||    \
676         IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr))))

678 /* Function prototypes */
679 extern void ipcl_g_init(void);
680 extern void ipcl_init(ip_stack_t *);
681 extern void ipcl_g_destroy(void);
682 extern void ipcl_destroy(ip_stack_t *);
683 extern conn_t *ipcl_conn_create(uint32_t, int, netstack_t *);
684 extern void ipcl_conn_destroy(conn_t *);

686 void ipcl_hash_insert_wildcard(connf_t *, conn_t *);
687 void ipcl_hash_remove(conn_t *);
688 void ipcl_hash_remove_locked(conn_t *connp, connf_t *connfp);

690 extern int      ipcl_bind_insert(conn_t *);
691 extern int      ipcl_bind_insert_v4(conn_t *);
692 extern int      ipcl_bind_insert_v6(conn_t *);
693 extern int      ipcl_conn_insert(conn_t *);
694 extern int      ipcl_conn_insert_v4(conn_t *);
695 extern int      ipcl_conn_insert_v6(conn_t *);
696 extern conn_t   *ipcl_get_next_conn(connf_t *, conn_t *, uint32_t);

698 conn_t *ipcl_classify_v4(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,
```



```
699             ip_stack_t *);
700 conn_t *ipcl_classify_v6(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,
701             ip_stack_t *);
702 conn_t *ipcl_classify(mblk_t *, ip_recv_attr_t *, ip_stack_t *);
703 conn_t *ipcl_classify_raw(mblk_t *, uint8_t, uint32_t, ipha_t *,
704     ip6_t *, ip_recv_attr_t *, ip_stack_t *);
705 conn_t *ipcl_iptun_classify_v4(ipaddr_t *, ipaddr_t *, ip_stack_t *);
706 conn_t *ipcl_iptun_classify_v6(in6_addr_t *, in6_addr_t *, ip_stack_t *);
707 void    ipcl_globalhash_insert(conn_t *);
708 void    ipcl_globalhash_remove(conn_t *);
709 void    ipcl_walk(pfv_t, void *, ip_stack_t *);
710 conn_t  *ipcl_tcp_lookup_reversed_ipv4(ipha_t *, tcpha_t *, int, ip_stack_t *);
711 conn_t  *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpha_t *, int, uint_t,
712             ip_stack_t *);
713 conn_t  *ipcl_lookup_listener_v4(uint16_t, ipaddr_t, zoneid_t, ip_stack_t *);
714 conn_t  *ipcl_lookup_listener_v6(uint16_t, in6_addr_t *, uint_t, zoneid_t,
715             ip_stack_t *);
716 int     conn_trace_ref(conn_t *);
717 int     conn_untrace_ref(conn_t *);
718 void    ipcl_conn_cleanup(conn_t *);
719 extern uint_t   conn_recvancillary_size(conn_t *, crb_t, ip_recv_attr_t *,
720     mblk_t *, ip_pkt_t *);
721 extern void     conn_recvancillary_add(conn_t *, crb_t, ip_recv_attr_t *,
722     ip_pkt_t *, uchar_t *, uint_t);
723 conn_t *ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *, ipha_t *, tcpha_t *,
724             ip_stack_t *);
725 conn_t *ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *, ip6_t *, tcpha_t *,
726             ip_stack_t *);

728 extern int ip_create_helper_stream(conn_t *, ldi_ident_t);
729 extern void ip_free_helper_stream(conn_t *);
730 extern int      ip_helper_stream_setup(queue_t *, dev_t *, int, int,
731     cred_t *, boolean_t);
732 extern mblk_t *conn_get_pid_mblk(conn_t *);
733 #endif /* ! codereview */

735 #ifdef  __cplusplus
736 }
737 #endif

739 #endif  /* _INET_IPCLASSIFIER_H */
```

```
**********************************************************
   60158 Mon Aug 17 21:08:05 2015
new/usr/src/uts/common/inet/mib2.h
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

typedef uint32_t        Counter;
typedef uint32_t        Counter32;
typedef uint64_t        Counter64;
typedef uint32_t        Gauge;
typedef uint32_t        IpAddress;
typedef struct in6_addr Ip6Address;
typedef Octet_t         DeviceName;
typedef Octet_t         PhysAddress;
typedef uint32_t        DeviceIndex;    /* Interface index */

#define MIB2_UNKNOWN_INTERFACE  0
#define MIB2_UNKNOWN_PROCESS    0

/*
 *  IP group
 */
#define MIB2_IP_ADDR            20      /* ipAddrEntry */
#define MIB2_IP_ROUTE           21      /* ipRouteEntry */
#define MIB2_IP_MEDIA           22      /* ipNetToMediaEntry */
#define MIB2_IP6_ROUTE          23      /* ipv6RouteEntry */
#define MIB2_IP6_MEDIA          24      /* ipv6NetToMediaEntry */
#define MIB2_IP6_ADDR           25      /* ipv6AddrEntry */
#define MIB2_IP_TRAFFIC_STATS   31      /* ipIfStatsEntry (IPv4) */
#define EXPER_IP_GROUP_MEMBERSHIP       100
#define EXPER_IP6_GROUP_MEMBERSHIP      101
#define EXPER_IP_GROUP_SOURCES          102
#define EXPER_IP6_GROUP_SOURCES         103
#define EXPER_IP_RTATTR                 104
#define EXPER_IP_DCE                    105

/*
 * There can be one of each of these tables per transport (MIB2_* above).
 */
#define EXPER_XPORT_MLP         105     /* transportMLPEntry */
#define EXPER_XPORT_PROC_INFO   106     /* conn_pid_node entry */
#endif /* ! codereview */

/* Old names retained for compatibility */
#define MIB2_IP_20      MIB2_IP_ADDR
#define MIB2_IP_21      MIB2_IP_ROUTE
#define MIB2_IP_22      MIB2_IP_MEDIA

typedef struct mib2_ip {
                /* forwarder?  1 gateway, 2 NOT gateway {ip 1} RW */
        int     ipForwarding;
                /* default Time-to-Live for iph         {ip 2} RW */
        int     ipDefaultTTL;
                /* # of input datagrams                 {ip 3} */
        Counter ipInReceives;
                /* # of dg discards for iph error       {ip 4} */
        Counter ipInHdrErrors;
                /* # of dg discards for bad addr        {ip 5} */
        Counter ipInAddrErrors;
                /* # of dg being forwarded              {ip 6} */
        Counter ipForwDatagrams;
                /* # of dg discards for unk protocol    {ip 7} */
        Counter ipInUnknownProtos;
                /* # of dg discards of good dg's        {ip 8} */
        Counter ipInDiscards;
```



```
                /* # of dg sent upstream                {ip 9} */
        Counter ipInDelivers;
                /* # of outdgs recv'd from upstream     {ip 10} */
        Counter ipOutRequests;
                /* # of good outdgs discarded           {ip 11} */
        Counter ipOutDiscards;
                /* # of outdg discards: no route found  {ip 12} */
        Counter ipOutNoRoutes;
                /* sec's recv'd frags held for reass.   {ip 13} */
        int     ipReasmTimeout;
                /* # of ip frags needing reassembly     {ip 14} */
        Counter ipReasmReqds;
                /* # of dg's reassembled                {ip 15} */
        Counter ipReasmOKs;
                /* # of reassembly failures (not dg cnt){ip 16} */
        Counter ipReasmFails;
                /* # of dg's fragged                    {ip 17} */
        Counter ipFragOKs;
                /* # of dg discards for no frag set     {ip 18} */
        Counter ipFragFails;
                /* # of dg frags from fragmentation     {ip 19} */
        Counter ipFragCreates;
                /* {ip 20} */
        int     ipAddrEntrySize;
                /* {ip 21} */
        int     ipRouteEntrySize;
                /* {ip 22} */
        int     ipNetToMediaEntrySize;
                /* # of valid route entries discarded   {ip 23} */
        Counter ipRoutingDiscards;
/*
 * following defined in MIB-II as part of TCP & UDP groups:
 */
                /* total # of segments recv'd with error        { tcp 14 } */
        Counter tcpInErrs;
                /* # of recv'd dg's not deliverable (no appl.)  { udp 2 } */
        Counter udpNoPorts;
/*
 * In addition to MIB-II
 */
                /* # of bad IP header checksums */
        Counter ipInCksumErrs;
                /* # of complete duplicates in reassembly */
        Counter ipReasmDuplicates;
                /* # of partial duplicates in reassembly */
        Counter ipReasmPartDups;
                /* # of packets not forwarded due to adminstrative reasons */
        Counter ipForwProhibits;
                /* # of UDP packets with bad UDP checksums */
        Counter udpInCksumErrs;
                /* # of UDP packets droped due to queue overflow */
        Counter udpInOverflows;
                /*
                 * # of RAW IP packets (all IP protocols except UDP, TCP
                 * and ICMP) droped due to queue overflow
                 */
        Counter rawipInOverflows;

        /*
         * Folowing are private IPSEC MIB.
         */
        /* # of incoming packets that succeeded policy checks */
        Counter ipsecInSucceeded;
        /* # of incoming packets that failed policy checks */
        Counter ipsecInFailed;
/* Compatible extensions added here */
```

```
279         int     ipMemberEntrySize;      /* Size of ip_member_t */
280         int     ipGroupSourceEntrySize; /* Size of ip_grpsrc_t */

282         Counter ipInIPv6; /* # of IPv6 packets received by IPv4 and dropped */
283         Counter ipOutIPv6;              /* No longer used */
284         Counter ipOutSwitchIPv6;        /* No longer used */

286         int     ipRouteAttributeSize;   /* Size of mib2_ipAttributeEntry_t */
287         int     transportMLPSize;       /* Size of mib2_transportMLPEntry_t */
288         int     ipDestEntrySize;        /* Size of dest_cache_entry_t */
289 } mib2_ip_t;

291 /*
292  *      ipv6IfStatsEntry OBJECT-TYPE
293  *              SYNTAX     Ipv6IfStatsEntry
294  *              MAX-ACCESS not-accessible
295  *              STATUS     current
296  *              DESCRIPTION
297  *                      "An interface statistics entry containing objects
298  *                      at a particular IPv6 interface."
299  *              AUGMENTS { ipv6IfEntry }
300  *              ::= { ipv6IfStatsTable 1 }
301  *
302  * Per-interface IPv6 statistics table
303  */

305 typedef struct mib2_ipv6IfStatsEntry {
306         /* Local ifindex to identify the interface */
307         DeviceIndex     ipv6IfIndex;

309                 /* forwarder?  1 gateway, 2 NOT gateway {ipv6MIBObjects 1} RW */
310         int     ipv6Forwarding;
311                 /* default Hoplimit for IPv6            {ipv6MIBObjects 2} RW */
312         int     ipv6DefaultHopLimit;

314         int     ipv6IfStatsEntrySize;
315         int     ipv6AddrEntrySize;
316         int     ipv6RouteEntrySize;
317         int     ipv6NetToMediaEntrySize;
318         int     ipv6MemberEntrySize;            /* Size of ipv6_member_t */
319         int     ipv6GroupSourceEntrySize;       /* Size of ipv6_grpsrc_t */

321         /* # input datagrams (incl errors)      { ipv6IfStatsEntry 1 } */
322         Counter ipv6InReceives;
323         /* # errors in IPv6 headers and options { ipv6IfStatsEntry 2 } */
324         Counter ipv6InHdrErrors;
325         /* # exceeds outgoing link MTU          { ipv6IfStatsEntry 3 } */
326         Counter ipv6InTooBigErrors;
327         /* # discarded due to no route to dest  { ipv6IfStatsEntry 4 } */
328         Counter ipv6InNoRoutes;
329         /* # invalid or unsupported addresses   { ipv6IfStatsEntry 5 } */
330         Counter ipv6InAddrErrors;
331         /* # unknown next header                { ipv6IfStatsEntry 6 } */
332         Counter ipv6InUnknownProtos;
333         /* # too short packets                  { ipv6IfStatsEntry 7 } */
334         Counter ipv6InTruncatedPkts;
335         /* # discarded e.g. due to no buffers   { ipv6IfStatsEntry 8 } */
336         Counter ipv6InDiscards;
337         /* # delivered to upper layer protocols { ipv6IfStatsEntry 9 } */
338         Counter ipv6InDelivers;
339         /* # forwarded out interface            { ipv6IfStatsEntry 10 } */
340         Counter ipv6OutForwDatagrams;
341         /* # originated out interface           { ipv6IfStatsEntry 11 } */
342         Counter ipv6OutRequests;
343         /* # discarded e.g. due to no buffers   { ipv6IfStatsEntry 12 } */
344         Counter ipv6OutDiscards;
```



```
345         /* # sucessfully fragmented packets     { ipv6IfStatsEntry 13 } */
346         Counter ipv6OutFragOKs;
347         /* # fragmentation failed               { ipv6IfStatsEntry 14 } */
348         Counter ipv6OutFragFails;
349         /* # fragments created                  { ipv6IfStatsEntry 15 } */
350         Counter ipv6OutFragCreates;
351         /* # fragments to reassemble            { ipv6IfStatsEntry 16 } */
352         Counter ipv6ReasmReqds;
353         /* # packets after reassembly           { ipv6IfStatsEntry 17 } */
354         Counter ipv6ReasmOKs;
355         /* # reassembly failed                  { ipv6IfStatsEntry 18 } */
356         Counter ipv6ReasmFails;
357         /* # received multicast packets         { ipv6IfStatsEntry 19 } */
358         Counter ipv6InMcastPkts;
359         /* # transmitted multicast packets      { ipv6IfStatsEntry 20 } */
360         Counter ipv6OutMcastPkts;
361 /*
362  * In addition to defined MIBs
363  */
364                 /* # discarded due to no route to dest */
365         Counter ipv6OutNoRoutes;
366                 /* # of complete duplicates in reassembly */
367         Counter ipv6ReasmDuplicates;
368                 /* # of partial duplicates in reassembly */
369         Counter ipv6ReasmPartDups;
370                 /* # of packets not forwarded due to adminstrative reasons */
371         Counter ipv6ForwProhibits;
372                 /* # of UDP packets with bad UDP checksums */
373         Counter udpInCksumErrs;
374                 /* # of UDP packets droped due to queue overflow */
375         Counter udpInOverflows;
376                 /*
377                  * # of RAW IPv6 packets (all IPv6 protocols except UDP, TCP
378                  * and ICMPv6) droped due to queue overflow
379                  */
380         Counter rawipInOverflows;

382                 /* # of IPv4 packets received by IPv6 and dropped */
383         Counter ipv6InIPv4;
384                 /* # of IPv4 packets transmitted by ip_wput_wput */
385         Counter ipv6OutIPv4;
386                 /* # of times ip_wput_v6 has switched to become ip_wput */
387         Counter ipv6OutSwitchIPv4;
388 } mib2_ipv6IfStatsEntry_t;

390 /*
391  * Per interface IP statistics, both v4 and v6.
392  *
393  * Some applications expect to get mib2_ipv6IfStatsEntry_t structs back when
394  * making a request. To ensure backwards compatability, the first
395  * sizeof(mib2_ipv6IfStatsEntry_t) bytes of the structure is identical to
396  * mib2_ipv6IfStatsEntry_t. This should work as long the application is
397  * written correctly (i.e., using ipv6IfStatsEntrySize to get the size of
398  * the struct)
399  *
400  * RFC4293 introduces several new counters, as well as defining 64-bit
401  * versions of existing counters. For a new counters, if they have both 32-
402  * and 64-bit versions, then we only added the latter. However, for already
403  * existing counters, we have added the 64-bit versions without removing the
404  * old (32-bit) ones. The 64- and 32-bit counters will only be synchronized
405  * when the structure contains IPv6 statistics, which is done to ensure
406  * backwards compatibility.
407  */

409 /* The following are defined in RFC 4001 and are used for ipIfStatsIPVersion */
410 #define MIB2_INETADDRESSTYPE_unknown    0
```

```
#define MIB2_INETADDRESSTYPE_ipv4       1
#define MIB2_INETADDRESSTYPE_ipv6       2

/*
 * On amd64, the alignment requirements for long long's is different for
 * 32 and 64 bits. If we have a struct containing long long's that is being
 * passed between a 64-bit kernel to a 32-bit application, then it is very
 * likely that the size of the struct will differ due to padding. Therefore, we
 * pack the data to ensure that the struct size is the same for 32- and
 * 64-bits.
 */
#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack(4)
#endif

typedef struct mib2_ipIfStatsEntry {

        /* Local ifindex to identify the interface */
        DeviceIndex     ipIfStatsIfIndex;

        /* forwarder?  1 gateway, 2 NOT gateway { ipv6MIBObjects 1} RW */
        int     ipIfStatsForwarding;
        /* default Hoplimit for IPv6            { ipv6MIBObjects 2} RW */
        int     ipIfStatsDefaultHopLimit;
#define ipIfStatsDefaultTTL     ipIfStatsDefaultHopLimit

        int     ipIfStatsEntrySize;
        int     ipIfStatsAddrEntrySize;
        int     ipIfStatsRouteEntrySize;
        int     ipIfStatsNetToMediaEntrySize;
        int     ipIfStatsMemberEntrySize;
        int     ipIfStatsGroupSourceEntrySize;

        /* # input datagrams (incl errors)      { ipIfStatsEntry 3 } */
        Counter ipIfStatsInReceives;
        /* # errors in IP headers and options   { ipIfStatsEntry 7 } */
        Counter ipIfStatsInHdrErrors;
        /* # exceeds outgoing link MTU(v6 only) { ipv6IfStatsEntry 3 } */
        Counter ipIfStatsInTooBigErrors;
        /* # discarded due to no route to dest  { ipIfStatsEntry 8 } */
        Counter ipIfStatsInNoRoutes;
        /* # invalid or unsupported addresses   { ipIfStatsEntry 9 } */
        Counter ipIfStatsInAddrErrors;
        /* # unknown next header                { ipIfStatsEntry 10 } */
        Counter ipIfStatsInUnknownProtos;
        /* # too short packets                  { ipIfStatsEntry 11 } */
        Counter ipIfStatsInTruncatedPkts;
        /* # discarded e.g. due to no buffers   { ipIfStatsEntry 17 } */
        Counter ipIfStatsInDiscards;
        /* # delivered to upper layer protocols { ipIfStatsEntry 18 } */
        Counter ipIfStatsInDelivers;
        /* # forwarded out interface            { ipIfStatsEntry 23 } */
        Counter ipIfStatsOutForwDatagrams;
        /* # originated out interface           { ipIfStatsEntry 20 } */
        Counter ipIfStatsOutRequests;
        /* # discarded e.g. due to no buffers   { ipIfStatsEntry 25 } */
        Counter ipIfStatsOutDiscards;
        /* # sucessfully fragmented packets     { ipIfStatsEntry 27 } */
        Counter ipIfStatsOutFragOKs;
        /* # fragmentation failed               { ipIfStatsEntry 28 } */
        Counter ipIfStatsOutFragFails;
        /* # fragments created                  { ipIfStatsEntry 29 } */
        Counter ipIfStatsOutFragCreates;
        /* # fragments to reassemble            { ipIfStatsEntry 14 } */
        Counter ipIfStatsReasmReqds;
        /* # packets after reassembly           { ipIfStatsEntry 15 } */
```



```
        Counter ipIfStatsReasmOKs;
        /* # reassembly failed                  { ipIfStatsEntry 16 } */
        Counter ipIfStatsReasmFails;
        /* # received multicast packets         { ipIfStatsEntry 34 } */
        Counter ipIfStatsInMcastPkts;
        /* # transmitted multicast packets      { ipIfStatsEntry 38 } */
        Counter ipIfStatsOutMcastPkts;

        /*
         * In addition to defined MIBs
         */

        /* # discarded due to no route to dest  { ipSystemStatsEntry 22 } */
        Counter ipIfStatsOutNoRoutes;
        /* # of complete duplicates in reassembly */
        Counter ipIfStatsReasmDuplicates;
        /* # of partial duplicates in reassembly */
        Counter ipIfStatsReasmPartDups;
        /* # of packets not forwarded due to adminstrative reasons */
        Counter ipIfStatsForwProhibits;
        /* # of UDP packets with bad UDP checksums */
        Counter udpInCksumErrs;
#define udpIfStatsInCksumErrs   udpInCksumErrs
        /* # of UDP packets droped due to queue overflow */
        Counter udpInOverflows;
#define udpIfStatsInOverflows   udpInOverflows
        /*
         * # of RAW IP packets (all IP protocols except UDP, TCP
         * and ICMP) droped due to queue overflow
         */
        Counter rawipInOverflows;
#define rawipIfStatsInOverflows rawipInOverflows

        /*
         * # of IP packets received with the wrong version (i.e., not equal
         * to ipIfStatsIPVersion) and that were dropped.
         */
        Counter ipIfStatsInWrongIPVersion;
        /*
         * This counter is no longer used
         */
        Counter ipIfStatsOutWrongIPVersion;
        /*
         * This counter is no longer used
         */
        Counter ipIfStatsOutSwitchIPVersion;

        /*
         * Fields defined in RFC 4293
         */

        /* ip version                           { ipIfStatsEntry 1 } */
        int             ipIfStatsIPVersion;
        /* # input datagrams (incl errors)      { ipIfStatsEntry 4 } */
        Counter64       ipIfStatsHCInReceives;
        /* # input octets (incl errors)         { ipIfStatsEntry 6 } */
        Counter64       ipIfStatsHCInOctets;
        /*
         *                                      { ipIfStatsEntry 13 }
         * # input datagrams for which a forwarding attempt was made
         */
        Counter64       ipIfStatsHCInForwDatagrams;
        /* # delivered to upper layer protocols { ipIfStatsEntry 19 } */
        Counter64       ipIfStatsHCInDelivers;
        /* # originated out interface           { ipIfStatsEntry 21 } */
        Counter64       ipIfStatsHCOutRequests;
```

```
543         /* # forwarded out interface            { ipIfStatsEntry 23 } */
544         Counter64       ipIfStatsHCOutForwDatagrams;
545         /* # dg's requiring fragmentation       { ipIfStatsEntry 26 } */
546         Counter         ipIfStatsOutFragReqds;
547         /* # output datagrams                   { ipIfStatsEntry 31 } */
548         Counter64       ipIfStatsHCOutTransmits;
549         /* # output octets                      { ipIfStatsEntry 33 } */
550         Counter64       ipIfStatsHCOutOctets;
551         /* # received multicast datagrams       { ipIfStatsEntry 35 } */
552         Counter64       ipIfStatsHCInMcastPkts;
553         /* # received multicast octets          { ipIfStatsEntry 37 } */
554         Counter64       ipIfStatsHCInMcastOctets;
555         /* # transmitted multicast datagrams    { ipIfStatsEntry 39 } */
556         Counter64       ipIfStatsHCOutMcastPkts;
557         /* # transmitted multicast octets       { ipIfStatsEntry 41 } */
558         Counter64       ipIfStatsHCOutMcastOctets;
559         /* # received broadcast datagrams       { ipIfStatsEntry 43 } */
560         Counter64       ipIfStatsHCInBcastPkts;
561         /* # transmitted broadcast datagrams    { ipIfStatsEntry 45 } */
562         Counter64       ipIfStatsHCOutBcastPkts;

564         /*
565          * Fields defined in mib2_ip_t
566          */

568         /* # of incoming packets that succeeded policy checks */
569         Counter         ipsecInSucceeded;
570 #define ipsecIfStatsInSucceeded ipsecInSucceeded
571         /* # of incoming packets that failed policy checks */
572         Counter         ipsecInFailed;
573 #define ipsecIfStatsInFailed    ipsecInFailed
574         /* # of bad IP header checksums */
575         Counter         ipInCksumErrs;
576 #define ipIfStatsInCksumErrs    ipInCksumErrs
577         /* total # of segments recv'd with error        { tcp 14 } */
578         Counter         tcpInErrs;
579 #define tcpIfStatsInErrs        tcpInErrs
580         /* # of recv'd dg's not deliverable (no appl.)  { udp 2 } */
581         Counter         udpNoPorts;
582 #define udpIfStatsNoPorts       udpNoPorts
583 } mib2_ipIfStatsEntry_t;
584 #define MIB_FIRST_NEW_ELM_mib2_ipIfStatsEntry_t ipIfStatsIPVersion

586 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
587 #pragma pack()
588 #endif

590 /*
591  * The IP address table contains this entity's IP addressing information.
592  *
593  *      ipAddrTable OBJECT-TYPE
594  *              SYNTAX  SEQUENCE OF IpAddrEntry
595  *              ACCESS  not-accessible
596  *              STATUS  mandatory
597  *              DESCRIPTION
598  *                      "The table of addressing information relevant to
599  *                      this entity's IP addresses."
600  *              ::= { ip 20 }
601  */

603 typedef struct mib2_ipAddrEntry {
604                         /* IP address of this entry     {ipAddrEntry 1} */
605         IpAddress       ipAdEntAddr;
606                         /* Unique interface index       {ipAddrEntry 2} */
607         DeviceName      ipAdEntIfIndex;
608                         /* Subnet mask for this IP addr {ipAddrEntry 3} */
```



```
609         IpAddress       ipAdEntNetMask;
610                         /* 2^lsb of IP broadcast addr   {ipAddrEntry 4} */
611         int             ipAdEntBcastAddr;
612                         /* max size for dg reassembly   {ipAddrEntry 5} */
613         int             ipAdEntReasmMaxSize;
614                         /* additional ipif_t fields */
615         struct ipAdEntInfo_s {
616                 Gauge           ae_mtu;
617                                 /* BSD if metric */
618                 int             ae_metric;
619                                 /* ipif broadcast addr.  relation to above?? */
620                 IpAddress       ae_broadcast_addr;
621                                 /* point-point dest addr */
622                 IpAddress       ae_pp_dst_addr;
623                 int             ae_flags;       /* IFF_* flags in if.h */
624                 Counter         ae_ibcnt;       /* Inbound packets */
625                 Counter         ae_obcnt;       /* Outbound packets */
626                 Counter         ae_focnt;       /* Forwarded packets */
627                 IpAddress       ae_subnet;      /* Subnet prefix */
628                 int             ae_subnet_len;  /* Subnet prefix length */
629                 IpAddress       ae_src_addr;    /* Source address */
630         }               ipAdEntInfo;
631         uint32_t        ipAdEntRetransmitTime;  /* ipInterfaceRetransmitTime */
632 } mib2_ipAddrEntry_t;
633 #define MIB_FIRST_NEW_ELM_mib2_ipAddrEntry_t    ipAdEntRetransmitTime

635 /*
636  *      ipv6AddrTable OBJECT-TYPE
637  *              SYNTAX      SEQUENCE OF Ipv6AddrEntry
638  *              MAX-ACCESS  not-accessible
639  *              STATUS      current
640  *              DESCRIPTION
641  *                      "The table of addressing information relevant to
642  *                      this node's interface addresses."
643  *              ::= { ipv6MIBObjects 8 }
644  */

646 typedef struct mib2_ipv6AddrEntry {
647         /* Unique interface index                       { Part of INDEX } */
648         DeviceName      ipv6AddrIfIndex;

650         /* IPv6 address of this entry                   { ipv6AddrEntry 1 } */
651         Ip6Address      ipv6AddrAddress;
652         /* Prefix length                                { ipv6AddrEntry 2 } */
653         uint_t          ipv6AddrPfxLength;
654         /* Type: stateless(1), stateful(2), unknown(3)  { ipv6AddrEntry 3 } */
655         uint_t          ipv6AddrType;
656         /* Anycast: true(1), false(2)                   { ipv6AddrEntry 4 } */
657         uint_t          ipv6AddrAnycastFlag;
658         /*
659          * Address status: preferred(1), deprecated(2), invalid(3),
660          * inaccessible(4), unknown(5)                  { ipv6AddrEntry 5 }
661          */
662         uint_t          ipv6AddrStatus;
663         struct ipv6AddrInfo_s {
664                 Gauge           ae_mtu;
665                                 /* BSD if metric */
666                 int             ae_metric;
667                                 /* point-point dest addr */
668                 Ip6Address      ae_pp_dst_addr;
669                 int             ae_flags;       /* IFF_* flags in if.h */
670                 Counter         ae_ibcnt;       /* Inbound packets */
671                 Counter         ae_obcnt;       /* Outbound packets */
672                 Counter         ae_focnt;       /* Forwarded packets */
673                 Ip6Address      ae_subnet;      /* Subnet prefix */
674                 int             ae_subnet_len;  /* Subnet prefix length */
```

```
675                 Ip6Address      ae_src_addr;    /* Source address */
676         }               ipv6AddrInfo;
677         uint32_t        ipv6AddrReasmMaxSize;   /* InterfaceReasmMaxSize */
678         Ip6Address      ipv6AddrIdentifier;     /* InterfaceIdentifier */
679         uint32_t        ipv6AddrIdentifierLen;
680         uint32_t        ipv6AddrReachableTime;  /* InterfaceReachableTime */
681         uint32_t        ipv6AddrRetransmitTime; /* InterfaceRetransmitTime */
682 } mib2_ipv6AddrEntry_t;
683 #define MIB_FIRST_NEW_ELM_mib2_ipv6AddrEntry_t  ipv6AddrReasmMaxSize

685 /*
686  * The IP routing table contains an entry for each route presently known to
687  * this entity. (for IPv4 routes)
688  *
689  *      ipRouteTable OBJECT-TYPE
690  *              SYNTAX  SEQUENCE OF IpRouteEntry
691  *              ACCESS  not-accessible
692  *              STATUS  mandatory
693  *              DESCRIPTION
694  *                      "This entity's IP Routing table."
695  *              ::= { ip 21 }
696  */

698 typedef struct mib2_ipRouteEntry {
699                 /* dest ip addr for this route          {ipRouteEntry 1 } RW */
700         IpAddress       ipRouteDest;
701                 /* unique interface index for this hop  {ipRouteEntry 2 } RW */
702         DeviceName      ipRouteIfIndex;
703                 /* primary route metric                 {ipRouteEntry 3 } RW */
704         int             ipRouteMetric1;
705                 /* alternate route metric               {ipRouteEntry 4 } RW */
706         int             ipRouteMetric2;
707                 /* alternate route metric               {ipRouteEntry 5 } RW */
708         int             ipRouteMetric3;
709                 /* alternate route metric               {ipRouteEntry 6 } RW */
710         int             ipRouteMetric4;
711                 /* ip addr of next hop on this route    {ipRouteEntry 7 } RW */
712         IpAddress       ipRouteNextHop;
713                 /* other(1), inval(2), dir(3), indir(4) {ipRouteEntry 8 } RW */
714         int             ipRouteType;
715                 /* mechanism by which route was learned {ipRouteEntry 9 } */
716         int             ipRouteProto;
717                 /* sec's since last update of route     {ipRouteEntry 10} RW */
718         int             ipRouteAge;
719                 /*                                      {ipRouteEntry 11} RW */
720         IpAddress       ipRouteMask;
721                 /* alternate route metric               {ipRouteEntry 12} RW */
722         int             ipRouteMetric5;
723                 /* additional info from ire's           {ipRouteEntry 13 } */
724         struct ipRouteInfo_s {
725                 Gauge           re_max_frag;
726                 Gauge           re_rtt;
727                 Counter         re_ref;
728                 int             re_frag_flag;
729                 IpAddress       re_src_addr;
730                 int             re_ire_type;
731                 Counter         re_obpkt;
732                 Counter         re_ibpkt;
733                 int             re_flags;
734                 /*
735                  * The following two elements (re_in_ill and re_in_src_addr)
736                  * are no longer used but are left here for the benefit of
737                  * old Apps that won't be able to handle the change in the
738                  * size of this struct. These elements will always be
739                  * set to zeroes.
740                  */
```



```
741                 DeviceName      re_in_ill;      /* Input interface */
742                 IpAddress       re_in_src_addr; /* Input source address */
743         }               ipRouteInfo;
744 } mib2_ipRouteEntry_t;

746 /*
747  * The IPv6 routing table contains an entry for each route presently known to
748  * this entity.
749  *
750  *      ipv6RouteTable OBJECT-TYPE
751  *              SYNTAX  SEQUENCE OF IpRouteEntry
752  *              ACCESS  not-accessible
753  *              STATUS  current
754  *              DESCRIPTION
755  *                      "IPv6 Routing table. This table contains
756  *                      an entry for each valid IPv6 unicast route
757  *                      that can be used for packet forwarding
758  *                      determination."
759  *              ::= { ipv6MIBObjects 11 }
760  */

762 typedef struct mib2_ipv6RouteEntry {
763                 /* dest ip addr for this route          { ipv6RouteEntry 1 } */
764         Ip6Address      ipv6RouteDest;
765                 /* prefix length                        { ipv6RouteEntry 2 } */
766         int             ipv6RoutePfxLength;
767                 /* unique route index                   { ipv6RouteEntry 3 } */
768         unsigned        ipv6RouteIndex;
769                 /* unique interface index for this hop  { ipv6RouteEntry 4 } */
770         DeviceName      ipv6RouteIfIndex;
771                 /* IPv6 addr of next hop on this route  { ipv6RouteEntry 5 } */
772         Ip6Address      ipv6RouteNextHop;
773                 /* other(1), discard(2), local(3), remote(4) */
774                 /*                                      { ipv6RouteEntry 6 } */
775         int             ipv6RouteType;
776                 /* mechanism by which route was learned { ipv6RouteEntry 7 } */
777                 /*
778                  * other(1), local(2), netmgmt(3), ndisc(4), rip(5), ospf(6),
779                  * bgp(7), idrp(8), igrp(9)
780                  */
781         int             ipv6RouteProtocol;
782                 /* policy hook or traffic class         { ipv6RouteEntry 8 } */
783         unsigned        ipv6RoutePolicy;
784                 /* sec's since last update of route     { ipv6RouteEntry 9} */
785         int             ipv6RouteAge;
786                 /* Routing domain ID of the next hop    { ipv6RouteEntry 10 } */
787         unsigned        ipv6RouteNextHopRDI;
788                 /* route metric                         { ipv6RouteEntry 11 } */
789         unsigned        ipv6RouteMetric;
790                 /* preference (impl specific)           { ipv6RouteEntry 12 } */
791         unsigned        ipv6RouteWeight;
792                 /* additional info from ire's           { } */
793         struct ipv6RouteInfo_s {
794                 Gauge           re_max_frag;
795                 Gauge           re_rtt;
796                 Counter         re_ref;
797                 int             re_frag_flag;
798                 Ip6Address      re_src_addr;
799                 int             re_ire_type;
800                 Counter         re_obpkt;
801                 Counter         re_ibpkt;
802                 int             re_flags;
803         }               ipv6RouteInfo;
804 } mib2_ipv6RouteEntry_t;

806 /*
```

```
 * The IPv4 and IPv6 routing table entries on a trusted system also have
 * security attributes in the form of label ranges.  This experimental
 * interface provides information about these labels.
 *
 * Each entry in this table contains a label range and an index that refers
 * back to the entry in the routing table to which it applies.  There may be 0,
 * 1, or many label ranges for each routing table entry.
 *
 * (opthdr.level is set to MIB2_IP for IPv4 entries and MIB2_IP6 for IPv6.
 * opthdr.name is set to EXPER_IP_GWATTR.)
 *
 *      ipRouteAttributeTable OBJECT-TYPE
 *              SYNTAX  SEQUENCE OF IpAttributeEntry
 *              ACCESS  not-accessible
 *              STATUS  current
 *              DESCRIPTION
 *                      "IPv4 routing attributes table.  This table contains
 *                      an entry for each valid trusted label attached to a
 *                      route in the system."
 *              ::= { ip 102 }
 *
 *      ipv6RouteAttributeTable OBJECT-TYPE
 *              SYNTAX  SEQUENCE OF IpAttributeEntry
 *              ACCESS  not-accessible
 *              STATUS  current
 *              DESCRIPTION
 *                      "IPv6 routing attributes table.  This table contains
 *                      an entry for each valid trusted label attached to a
 *                      route in the system."
 *              ::= { ip6 102 }
 */

typedef struct mib2_ipAttributeEntry {
        uint_t          iae_routeidx;
        int             iae_doi;
        brange_t        iae_slrange;
} mib2_ipAttributeEntry_t;

/*
 * The IP address translation table contain the IpAddress to
 * 'physical' address equivalences.  Some interfaces do not
 * use translation tables for determining address
 * equivalences (e.g., DDN-X.25 has an algorithmic method);
 * if all interfaces are of this type, then the Address
 * Translation table is empty, i.e., has zero entries.
 *
 *      ipNetToMediaTable OBJECT-TYPE
 *              SYNTAX  SEQUENCE OF IpNetToMediaEntry
 *              ACCESS  not-accessible
 *              STATUS  mandatory
 *              DESCRIPTION
 *                      "The IP Address Translation table used for mapping
 *                      from IP addresses to physical addresses."
 *              ::= { ip 22 }
 */

typedef struct mib2_ipNetToMediaEntry {
        /* Unique interface index               { ipNetToMediaEntry 1 } RW */
        DeviceName      ipNetToMediaIfIndex;
        /* Media dependent physical addr        { ipNetToMediaEntry 2 } RW */
        PhysAddress     ipNetToMediaPhysAddress;
        /* ip addr for this physical addr       { ipNetToMediaEntry 3 } RW */
        IpAddress       ipNetToMediaNetAddress;
        /* other(1), inval(2), dyn(3), stat(4)  { ipNetToMediaEntry 4 } RW */
        int             ipNetToMediaType;
        struct ipNetToMediaInfo_s {
```



```
                PhysAddress     ntm_mask;       /* subnet mask for entry */
                int             ntm_flags;      /* ACE_F_* flags in arp.h */
        }               ipNetToMediaInfo;
} mib2_ipNetToMediaEntry_t;

/*
 *      ipv6NetToMediaTable OBJECT-TYPE
 *               SYNTAX      SEQUENCE OF Ipv6NetToMediaEntry
 *               MAX-ACCESS  not-accessible
 *               STATUS      current
 *               DESCRIPTION
 *                      "The IPv6 Address Translation table used for
 *                      mapping from IPv6 addresses to physical addresses.
 *
 *                      The IPv6 address translation table contain the
 *                      Ipv6Address to 'physical' address equivalencies.
 *                      Some interfaces do not use translation tables
 *                      for determining address equivalencies; if all
 *                      interfaces are of this type, then the Address
 *                      Translation table is empty, i.e., has zero
 *                      entries."
 *              ::= { ipv6MIBObjects 12 }
 */

typedef struct mib2_ipv6NetToMediaEntry {
        /* Unique interface index               { Part of INDEX } */
        DeviceIndex     ipv6NetToMediaIfIndex;

        /* ip addr for this physical addr       { ipv6NetToMediaEntry 1 } */
        Ip6Address      ipv6NetToMediaNetAddress;
        /* Media dependent physical addr        { ipv6NetToMediaEntry 2 } */
        PhysAddress     ipv6NetToMediaPhysAddress;
        /*
         * Type of mapping
         * other(1), dynamic(2), static(3), local(4)
         *                                      { ipv6NetToMediaEntry 3 }
         */
        int             ipv6NetToMediaType;
        /*
         * NUD state
         * reachable(1), stale(2), delay(3), probe(4), invalid(5), unknown(6)
         * Note: The kernel returns ND_* states.
         *                                      { ipv6NetToMediaEntry 4 }
         */
        int             ipv6NetToMediaState;
        /* sysUpTime last time entry was updated { ipv6NetToMediaEntry 5 } */
        int             ipv6NetToMediaLastUpdated;
} mib2_ipv6NetToMediaEntry_t;


/*
 * List of group members per interface
 */
typedef struct ip_member {
        /* Interface index */
        DeviceName      ipGroupMemberIfIndex;
        /* IP Multicast address */
        IpAddress       ipGroupMemberAddress;
        /* Number of member sockets */
        Counter         ipGroupMemberRefCnt;
        /* Filter mode: 1 => include, 2 => exclude */
        int             ipGroupMemberFilterMode;
} ip_member_t;


/*
```

```
 * List of IPv6 group members per interface
 */
typedef struct ipv6_member {
        /* Interface index */
        DeviceIndex     ipv6GroupMemberIfIndex;
        /* IP Multicast address */
        Ip6Address      ipv6GroupMemberAddress;
        /* Number of member sockets */
        Counter         ipv6GroupMemberRefCnt;
        /* Filter mode: 1 => include, 2 => exclude */
        int             ipv6GroupMemberFilterMode;
} ipv6_member_t;

/*
 * This is used to mark transport layer entities (e.g., TCP connections) that
 * are capable of receiving packets from a range of labels.  'level' is set to
 * the protocol of interest (e.g., MIB2_TCP), and 'name' is set to
 * EXPER_XPORT_MLP.  The tme_connidx refers back to the entry in MIB2_TCP_CONN,
 * MIB2_TCP6_CONN, or MIB2_SCTP_CONN.
 *
 * It is also used to report connections that receive packets at a single label
 * that's other than the zone's label.  This is the case when a TCP connection
 * is accepted from a particular peer using an MLP listener.
 */
typedef struct mib2_transportMLPEntry {
        uint_t          tme_connidx;
        uint_t          tme_flags;
        int             tme_doi;
        bslabel_t       tme_label;
} mib2_transportMLPEntry_t;

#define MIB2_TMEF_PRIVATE       0x00000001      /* MLP on private addresses */
#define MIB2_TMEF_SHARED        0x00000002      /* MLP on shared addresses */
#define MIB2_TMEF_ANONMLP       0x00000004      /* Anonymous MLP port */
#define MIB2_TMEF_MACEXEMPT     0x00000008      /* MAC-Exempt port */
#define MIB2_TMEF_IS_LABELED    0x00000010      /* tme_doi & tme_label exists */
#define MIB2_TMEF_MACIMPLICIT   0x00000020      /* MAC-Implicit */
/*
 * List of IPv4 source addresses being filtered per interface
 */
typedef struct ip_grpsrc {
        /* Interface index */
        DeviceName      ipGroupSourceIfIndex;
        /* IP Multicast address */
        IpAddress       ipGroupSourceGroup;
        /* IP Source address */
        IpAddress       ipGroupSourceAddress;
} ip_grpsrc_t;


/*
 * List of IPv6 source addresses being filtered per interface
 */
typedef struct ipv6_grpsrc {
        /* Interface index */
        DeviceIndex     ipv6GroupSourceIfIndex;
        /* IP Multicast address */
        Ip6Address      ipv6GroupSourceGroup;
        /* IP Source address */
        Ip6Address      ipv6GroupSourceAddress;
} ipv6_grpsrc_t;


/*
 * List of destination cache entries
 */
```



```
typedef struct dest_cache_entry {
        /* IP Multicast address */
        IpAddress       DestIpv4Address;
        Ip6Address      DestIpv6Address;
        uint_t          DestFlags;      /* DCEF_* */
        uint32_t        DestPmtu;       /* Path MTU if DCEF_PMTU */
        uint32_t        DestIdent;      /* Per destination IP ident. */
        DeviceIndex     DestIfindex;    /* For IPv6 link-locals */
        uint32_t        DestAge;        /* Age of MTU info in seconds */
} dest_cache_entry_t;


/*
 * ICMP Group
 */
typedef struct mib2_icmp {
        /* total # of recv'd ICMP msgs                  { icmp 1 } */
        Counter icmpInMsgs;
        /* recv'd ICMP msgs with errors                 { icmp 2 } */
        Counter icmpInErrors;
        /* recv'd "dest unreachable" msg's              { icmp 3 } */
        Counter icmpInDestUnreachs;
        /* recv'd "time exceeded" msg's                 { icmp 4 } */
        Counter icmpInTimeExcds;
        /* recv'd "parameter problem" msg's             { icmp 5 } */
        Counter icmpInParmProbs;
        /* recv'd "source quench" msg's                 { icmp 6 } */
        Counter icmpInSrcQuenchs;
        /* recv'd "ICMP redirect" msg's                 { icmp 7 } */
        Counter icmpInRedirects;
        /* recv'd "echo request" msg's                  { icmp 8 } */
        Counter icmpInEchos;
        /* recv'd "echo reply" msg's                    { icmp 9 } */
        Counter icmpInEchoReps;
        /* recv'd "timestamp" msg's                     { icmp 10 } */
        Counter icmpInTimestamps;
        /* recv'd "timestamp reply" msg's               { icmp 11 } */
        Counter icmpInTimestampReps;
        /* recv'd "address mask request" msg's          { icmp 12 } */
        Counter icmpInAddrMasks;
        /* recv'd "address mask reply" msg's            { icmp 13 } */
        Counter icmpInAddrMaskReps;
        /* total # of sent ICMP msg's                   { icmp 14 } */
        Counter icmpOutMsgs;
        /* # of msg's not sent for internal icmp errors { icmp 15 } */
        Counter icmpOutErrors;
        /* # of "dest unreachable" msg's sent           { icmp 16 } */
        Counter icmpOutDestUnreachs;
        /* # of "time exceeded" msg's sent              { icmp 17 } */
        Counter icmpOutTimeExcds;
        /* # of "parameter problme" msg's sent          { icmp 18 } */
        Counter icmpOutParmProbs;
        /* # of "source quench" msg's sent              { icmp 19 } */
        Counter icmpOutSrcQuenchs;
        /* # of "ICMP redirect" msg's sent              { icmp 20 } */
        Counter icmpOutRedirects;
        /* # of "Echo request" msg's sent               { icmp 21 } */
        Counter icmpOutEchos;
        /* # of "Echo reply" msg's sent                 { icmp 22 } */
        Counter icmpOutEchoReps;
        /* # of "timestamp request" msg's sent          { icmp 23 } */
        Counter icmpOutTimestamps;
        /* # of "timestamp reply" msg's sent            { icmp 24 } */
        Counter icmpOutTimestampReps;
        /* # of "address mask request" msg's sent       { icmp 25 } */
        Counter icmpOutAddrMasks;
```

```
1071         /* # of "address mask reply" msg's sent         { icmp 26 } */
1072         Counter icmpOutAddrMaskReps;
1073 /*
1074  * In addition to MIB-II
1075  */
1076         /* # of received packets with checksum errors */
1077         Counter icmpInCksumErrs;
1078         /* # of received packets with unknow codes */
1079         Counter icmpInUnknowns;
1080         /* # of received unreachables with "fragmentation needed" */
1081         Counter icmpInFragNeeded;
1082         /* # of sent unreachables with "fragmentation needed" */
1083         Counter icmpOutFragNeeded;
1084         /*
1085          * # of msg's not sent since original packet was broadcast/multicast
1086          * or an ICMP error packet
1087          */
1088         Counter icmpOutDrops;
1089         /* # of ICMP packets droped due to queue overflow */
1090         Counter icmpInOverflows;
1091         /* recv'd "ICMP redirect" msg's that are bad thus ignored */
1092         Counter icmpInBadRedirects;
1093 } mib2_icmp_t;


1096 /*
1097  *      ipv6IfIcmpEntry OBJECT-TYPE
1098  *              SYNTAX      Ipv6IfIcmpEntry
1099  *              MAX-ACCESS  not-accessible
1100  *              STATUS      current
1101  *              DESCRIPTION
1102  *                      "An ICMPv6 statistics entry containing
1103  *                      objects at a particular IPv6 interface.
1104  *
1105  *                      Note that a receiving interface is
1106  *                      the interface to which a given ICMPv6 message
1107  *                      is addressed which may not be necessarily
1108  *                      the input interface for the message.
1109  *
1110  *                      Similarly, the sending interface is
1111  *                      the interface that sources a given
1112  *                      ICMP message which is usually but not
1113  *                      necessarily the output interface for the message."
1114  *              AUGMENTS { ipv6IfEntry }
1115  *              ::= { ipv6IfIcmpTable 1 }
1116  *
1117  * Per-interface ICMPv6 statistics table
1118  */

1120 typedef struct mib2_ipv6IfIcmpEntry {
1121         /* Local ifindex to identify the interface */
1122         DeviceIndex     ipv6IfIcmpIfIndex;

1124         int             ipv6IfIcmpEntrySize;    /* Size of ipv6IfIcmpEntry */

1126         /* The total # ICMP msgs rcvd includes ipv6IfIcmpInErrors */
1127         Counter32       ipv6IfIcmpInMsgs;
1128         /* # ICMP with ICMP-specific errors (bad checkum, length, etc) */
1129         Counter32       ipv6IfIcmpInErrors;
1130         /* # ICMP Destination Unreachable */
1131         Counter32       ipv6IfIcmpInDestUnreachs;
1132         /* # ICMP destination unreachable/communication admin prohibited */
1133         Counter32       ipv6IfIcmpInAdminProhibs;
1134         Counter32       ipv6IfIcmpInTimeExcds;
1135         Counter32       ipv6IfIcmpInParmProblems;
1136         Counter32       ipv6IfIcmpInPktTooBigs;
```



```
1137         Counter32       ipv6IfIcmpInEchos;
1138         Counter32       ipv6IfIcmpInEchoReplies;
1139         Counter32       ipv6IfIcmpInRouterSolicits;
1140         Counter32       ipv6IfIcmpInRouterAdvertisements;
1141         Counter32       ipv6IfIcmpInNeighborSolicits;
1142         Counter32       ipv6IfIcmpInNeighborAdvertisements;
1143         Counter32       ipv6IfIcmpInRedirects;
1144         Counter32       ipv6IfIcmpInGroupMembQueries;
1145         Counter32       ipv6IfIcmpInGroupMembResponses;
1146         Counter32       ipv6IfIcmpInGroupMembReductions;
1147         /* Total # ICMP messages attempted to send (includes OutErrors) */
1148         Counter32       ipv6IfIcmpOutMsgs;
1149         /* # ICMP messages not sent due to ICMP problems (e.g. no buffers) */
1150         Counter32       ipv6IfIcmpOutErrors;
1151         Counter32       ipv6IfIcmpOutDestUnreachs;
1152         Counter32       ipv6IfIcmpOutAdminProhibs;
1153         Counter32       ipv6IfIcmpOutTimeExcds;
1154         Counter32       ipv6IfIcmpOutParmProblems;
1155         Counter32       ipv6IfIcmpOutPktTooBigs;
1156         Counter32       ipv6IfIcmpOutEchos;
1157         Counter32       ipv6IfIcmpOutEchoReplies;
1158         Counter32       ipv6IfIcmpOutRouterSolicits;
1159         Counter32       ipv6IfIcmpOutRouterAdvertisements;
1160         Counter32       ipv6IfIcmpOutNeighborSolicits;
1161         Counter32       ipv6IfIcmpOutNeighborAdvertisements;
1162         Counter32       ipv6IfIcmpOutRedirects;
1163         Counter32       ipv6IfIcmpOutGroupMembQueries;
1164         Counter32       ipv6IfIcmpOutGroupMembResponses;
1165         Counter32       ipv6IfIcmpOutGroupMembReductions;
1166 /* Additions beyond the MIB */
1167         Counter32       ipv6IfIcmpInOverflows;
1168         /* recv'd "ICMPv6 redirect" msg's that are bad thus ignored */
1169         Counter32       ipv6IfIcmpBadHoplimit;
1170         Counter32       ipv6IfIcmpInBadNeighborAdvertisements;
1171         Counter32       ipv6IfIcmpInBadNeighborSolicitations;
1172         Counter32       ipv6IfIcmpInBadRedirects;
1173         Counter32       ipv6IfIcmpInGroupMembTotal;
1174         Counter32       ipv6IfIcmpInGroupMembBadQueries;
1175         Counter32       ipv6IfIcmpInGroupMembBadReports;
1176         Counter32       ipv6IfIcmpInGroupMembOurReports;
1177 } mib2_ipv6IfIcmpEntry_t;

1179 /*
1180  * the TCP group
1181  *
1182  * Note that instances of object types that represent
1183  * information about a particular TCP connection are
1184  * transient; they persist only as long as the connection
1185  * in question.
1186  */
1187 #define MIB2_TCP_CONN   13      /* tcpConnEntry */
1188 #define MIB2_TCP6_CONN  14      /* tcp6ConnEntry */

1190 /* Old name retained for compatibility */
1191 #define MIB2_TCP_13     MIB2_TCP_CONN

1193 /* Pack data in mib2_tcp to make struct size the same for 32- and 64-bits */
1194 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1195 #pragma pack(4)
1196 #endif
1197 typedef struct mib2_tcp {
1198                 /* algorithm used for transmit timeout value    { tcp 1 } */
1199         int     tcpRtoAlgorithm;
1200                 /* minimum retransmit timeout (ms)              { tcp 2 } */
1201         int     tcpRtoMin;
1202                 /* maximum retransmit timeout (ms)              { tcp 3 } */
```

```
1203         int     tcpRtoMax;
1204                 /* maximum # of connections supported           { tcp 4 } */
1205         int     tcpMaxConn;
1206                 /* # of direct transitions CLOSED -> SYN-SENT   { tcp 5 } */
1207         Counter tcpActiveOpens;
1208                 /* # of direct transitions LISTEN -> SYN-RCVD   { tcp 6 } */
1209         Counter tcpPassiveOpens;
1210                 /* # of direct SIN-SENT/RCVD -> CLOSED/LISTEN   { tcp 7 } */
1211         Counter tcpAttemptFails;
1212                 /* # of direct ESTABLISHED/CLOSE-WAIT -> CLOSED { tcp 8 } */
1213         Counter tcpEstabResets;
1214                 /* # of connections ESTABLISHED or CLOSE-WAIT   { tcp 9 } */
1215         Gauge   tcpCurrEstab;
1216                 /* total # of segments recv'd                   { tcp 10 } */
1217         Counter tcpInSegs;
1218                 /* total # of segments sent                     { tcp 11 } */
1219         Counter tcpOutSegs;
1220                 /* total # of segments retransmitted            { tcp 12 } */
1221         Counter tcpRetransSegs;
1222                 /* {tcp 13} */
1223         int     tcpConnTableSize;       /* Size of tcpConnEntry_t */
1224         /* in ip                                {tcp 14} */
1225                 /* # of segments sent with RST flag             { tcp 15 } */
1226         Counter tcpOutRsts;
1227 /* In addition to MIB-II */
1228 /* Sender */
1229         /* total # of data segments sent */
1230         Counter tcpOutDataSegs;
1231         /* total # of bytes in data segments sent */
1232         Counter tcpOutDataBytes;
1233         /* total # of bytes in segments retransmitted */
1234         Counter tcpRetransBytes;
1235         /* total # of acks sent */
1236         Counter tcpOutAck;
1237         /* total # of delayed acks sent */
1238         Counter tcpOutAckDelayed;
1239         /* total # of segments sent with the urg flag on */
1240         Counter tcpOutUrg;
1241         /* total # of window updates sent */
1242         Counter tcpOutWinUpdate;
1243         /* total # of zero window probes sent */
1244         Counter tcpOutWinProbe;
1245         /* total # of control segments sent (syn, fin, rst) */
1246         Counter tcpOutControl;
1247         /* total # of segments sent due to "fast retransmit" */
1248         Counter tcpOutFastRetrans;
1249 /* Receiver */
1250         /* total # of ack segments received */
1251         Counter tcpInAckSegs;
1252         /* total # of bytes acked */
1253         Counter tcpInAckBytes;
1254         /* total # of duplicate acks */
1255         Counter tcpInDupAck;
1256         /* total # of acks acking unsent data */
1257         Counter tcpInAckUnsent;
1258         /* total # of data segments received in order */
1259         Counter tcpInDataInorderSegs;
1260         /* total # of data bytes received in order */
1261         Counter tcpInDataInorderBytes;
1262         /* total # of data segments received out of order */
1263         Counter tcpInDataUnorderSegs;
1264         /* total # of data bytes received out of order */
1265         Counter tcpInDataUnorderBytes;
1266         /* total # of complete duplicate data segments received */
1267         Counter tcpInDataDupSegs;
1268         /* total # of bytes in the complete duplicate data segments received */
```



```
1269         Counter tcpInDataDupBytes;
1270         /* total # of partial duplicate data segments received */
1271         Counter tcpInDataPartDupSegs;
1272         /* total # of bytes in the partial duplicate data segments received */
1273         Counter tcpInDataPartDupBytes;
1274         /* total # of data segments received past the window */
1275         Counter tcpInDataPastWinSegs;
1276         /* total # of data bytes received part the window */
1277         Counter tcpInDataPastWinBytes;
1278         /* total # of zero window probes received */
1279         Counter tcpInWinProbe;
1280         /* total # of window updates received */
1281         Counter tcpInWinUpdate;
1282         /* total # of data segments received after the connection has closed */
1283         Counter tcpInClosed;
1284 /* Others */
1285         /* total # of failed attempts to update the rtt estimate */
1286         Counter tcpRttNoUpdate;
1287         /* total # of successful attempts to update the rtt estimate */
1288         Counter tcpRttUpdate;
1289         /* total # of retransmit timeouts */
1290         Counter tcpTimRetrans;
1291         /* total # of retransmit timeouts dropping the connection */
1292         Counter tcpTimRetransDrop;
1293         /* total # of keepalive timeouts */
1294         Counter tcpTimKeepalive;
1295         /* total # of keepalive timeouts sending a probe */
1296         Counter tcpTimKeepaliveProbe;
1297         /* total # of keepalive timeouts dropping the connection */
1298         Counter tcpTimKeepaliveDrop;
1299         /* total # of connections refused due to backlog full on listen */
1300         Counter tcpListenDrop;
1301         /* total # of connections refused due to half-open queue (q0) full */
1302         Counter tcpListenDropQ0;
1303         /* total # of connections dropped from a full half-open queue (q0) */
1304         Counter tcpHalfOpenDrop;
1305         /* total # of retransmitted segments by SACK retransmission */
1306         Counter tcpOutSackRetransSegs;

1308         int     tcp6ConnTableSize;      /* Size of tcp6ConnEntry_t */

1310         /*
1311          * fields from RFC 4022
1312          */

1314         /* total # of segments recv'd                           { tcp 17 } */
1315         Counter64       tcpHCInSegs;
1316         /* total # of segments sent                             { tcp 18 } */
1317         Counter64       tcpHCOutSegs;
1318 } mib2_tcp_t;
1319 #define MIB_FIRST_NEW_ELM_mib2_tcp_t    tcpHCInSegs

1321 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1322 #pragma pack()
1323 #endif

1325 /*
1326  * The TCP/IPv4 connection table {tcp 13} contains information about this
1327  * entity's existing TCP connections over IPv4.
1328  */
1329 /* For tcpConnState and tcp6ConnState */
1330 #define MIB2_TCP_closed         1
1331 #define MIB2_TCP_listen         2
1332 #define MIB2_TCP_synSent        3
1333 #define MIB2_TCP_synReceived    4
1334 #define MIB2_TCP_established    5
```

```
#define MIB2_TCP_finWait1       6
#define MIB2_TCP_finWait2       7
#define MIB2_TCP_closeWait      8
#define MIB2_TCP_lastAck        9
#define MIB2_TCP_closing        10
#define MIB2_TCP_timeWait       11
#define MIB2_TCP_deleteTCB      12              /* only writeable value */

/* Pack data to make struct size the same for 32- and 64-bits */
#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack(4)
#endif
typedef struct mib2_tcpConnEntry {
                /* state of tcp connection              { tcpConnEntry 1} RW */
        int             tcpConnState;
                /* local ip addr for this connection    { tcpConnEntry 2 } */
        IpAddress       tcpConnLocalAddress;
                /* local port for this connection       { tcpConnEntry 3 } */
        int             tcpConnLocalPort;       /* In host byte order */
                /* remote ip addr for this connection   { tcpConnEntry 4 } */
        IpAddress       tcpConnRemAddress;
                /* remote port for this connection      { tcpConnEntry 5 } */
        int             tcpConnRemPort;         /* In host byte order */
        struct tcpConnEntryInfo_s {
                        /* seq # of next segment to send */
                Gauge           ce_snxt;
                                /* seq # of of last segment unacknowledged */
                Gauge           ce_suna;
                                /* currect send window size */
                Gauge           ce_swnd;
                                /* seq # of next expected segment */
                Gauge           ce_rnxt;
                                /* seq # of last ack'd segment */
                Gauge           ce_rack;
                                /* currenct receive window size */
                Gauge           ce_rwnd;
                                        /* current rto (retransmit timeout) */
                Gauge           ce_rto;
                                        /* current max segment size */
                Gauge           ce_mss;
                                /* actual internal state */
                int             ce_state;
        }               tcpConnEntryInfo;

        /* pid of the processes that created this connection */
        uint32_t        tcpConnCreationProcess;
        /* system uptime when the connection was created */
        uint64_t        tcpConnCreationTime;
} mib2_tcpConnEntry_t;
#define MIB_FIRST_NEW_ELM_mib2_tcpConnEntry_t   tcpConnCreationProcess

#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack()
#endif


/*
 * The TCP/IPv6 connection table {tcp 14} contains information about this
 * entity's existing TCP connections over IPv6.
 */

/* Pack data to make struct size the same for 32- and 64-bits */
#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack(4)
#endif
typedef struct mib2_tcp6ConnEntry {
```



```
        /* local ip addr for this connection    { ipv6TcpConnEntry 1 } */
        Ip6Address      tcp6ConnLocalAddress;
        /* local port for this connection       { ipv6TcpConnEntry 2 } */
        int             tcp6ConnLocalPort;
        /* remote ip addr for this connection   { ipv6TcpConnEntry 3 } */
        Ip6Address      tcp6ConnRemAddress;
        /* remote port for this connection      { ipv6TcpConnEntry 4 } */
        int             tcp6ConnRemPort;
        /* interface index or zero              { ipv6TcpConnEntry 5 } */
        DeviceIndex     tcp6ConnIfIndex;
        /* state of tcp6 connection             { ipv6TcpConnEntry 6 } RW */
        int             tcp6ConnState;
        struct tcp6ConnEntryInfo_s {
                        /* seq # of next segment to send */
                Gauge           ce_snxt;
                                /* seq # of of last segment unacknowledged */
                Gauge           ce_suna;
                                /* currect send window size */
                Gauge           ce_swnd;
                                /* seq # of next expected segment */
                Gauge           ce_rnxt;
                                /* seq # of last ack'd segment */
                Gauge           ce_rack;
                                /* currenct receive window size */
                Gauge           ce_rwnd;
                                        /* current rto (retransmit timeout) */
                Gauge           ce_rto;
                                        /* current max segment size */
                Gauge           ce_mss;
                                /* actual internal state */
                int             ce_state;
        }               tcp6ConnEntryInfo;

        /* pid of the processes that created this connection */
        uint32_t        tcp6ConnCreationProcess;
        /* system uptime when the connection was created */
        uint64_t        tcp6ConnCreationTime;
} mib2_tcp6ConnEntry_t;
#define MIB_FIRST_NEW_ELM_mib2_tcp6ConnEntry_t  tcp6ConnCreationProcess

#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack()
#endif

/*
 * the UDP group
 */
#define MIB2_UDP_ENTRY  5       /* udpEntry */
#define MIB2_UDP6_ENTRY 6       /* udp6Entry */

/* Old name retained for compatibility */
#define MIB2_UDP_5      MIB2_UDP_ENTRY

/* Pack data to make struct size the same for 32- and 64-bits */
#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack(4)
#endif
typedef struct mib2_udp {
                /* total # of UDP datagrams sent upstream       { udp 1 } */
        Counter udpInDatagrams;
        /* in ip                                        { udp 2 } */
                /* # of recv'd dg's not deliverable (other)     { udp 3 }  */
        Counter udpInErrors;
                /* total # of dg's sent                         { udp 4 } */
        Counter udpOutDatagrams;
                /* { udp 5 } */
```

```
        int     udpEntrySize;                   /* Size of udpEntry_t */
        int     udp6EntrySize;                  /* Size of udp6Entry_t */
        Counter udpOutErrors;

        /*
         * fields from RFC 4113
         */

        /* total # of UDP datagrams sent upstream               { udp 8 } */
        Counter64       udpHCInDatagrams;
        /* total # of dg's sent                                 { udp 9 } */
        Counter64       udpHCOutDatagrams;
} mib2_udp_t;
#define MIB_FIRST_NEW_ELM_mib2_udp_t    udpHCInDatagrams

#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack()
#endif

/*
 * The UDP listener table contains information about this entity's UDP
 * end-points on which a local application is currently accepting datagrams.
 */

/* For both IPv4 and IPv6 ue_state: */
#define MIB2_UDP_unbound        1
#define MIB2_UDP_idle           2
#define MIB2_UDP_connected      3
#define MIB2_UDP_unknown        4

/* Pack data to make struct size the same for 32- and 64-bits */
#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack(4)
#endif
typedef struct mib2_udpEntry {
                /* local ip addr of listener            { udpEntry 1 } */
        IpAddress       udpLocalAddress;
                /* local port of listener               { udpEntry 2 } */
        int             udpLocalPort;           /* In host byte order */
        struct udpEntryInfo_s {
                int             ue_state;
                IpAddress       ue_RemoteAddress;
                int             ue_RemotePort;  /* In host byte order */
        }               udpEntryInfo;

        /*
         * RFC 4113
         */

        /* Unique id for this 4-tuple           { udpEndpointEntry 7 } */
        uint32_t        udpInstance;
        /* pid of the processes that created this endpoint */
        uint32_t        udpCreationProcess;
        /* system uptime when the endpoint was created */
        uint64_t        udpCreationTime;
} mib2_udpEntry_t;
#define MIB_FIRST_NEW_ELM_mib2_udpEntry_t       udpInstance

#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack()
#endif

/*
 * The UDP (for IPv6) listener table contains information about this
 * entity's UDP end-points on which a local application is
 * currently accepting datagrams.
```



```
 */

/* Pack data to make struct size the same for 32- and 64-bits */
#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack(4)
#endif
typedef struct mib2_udp6Entry {
                /* local ip addr of listener            { ipv6UdpEntry 1 } */
        Ip6Address      udp6LocalAddress;
                /* local port of listener               { ipv6UdpEntry 2 } */
        int             udp6LocalPort;          /* In host byte order */
                /* interface index or zero              { ipv6UdpEntry 3 } */
        DeviceIndex     udp6IfIndex;
        struct udp6EntryInfo_s {
                int     ue_state;
                Ip6Address      ue_RemoteAddress;
                int             ue_RemotePort;  /* In host byte order */
        }               udp6EntryInfo;

        /*
         * RFC 4113
         */

        /* Unique id for this 4-tuple           { udpEndpointEntry 7 } */
        uint32_t        udp6Instance;
        /* pid of the processes that created this endpoint */
        uint32_t        udp6CreationProcess;
        /* system uptime when the endpoint was created */
        uint64_t        udp6CreationTime;
} mib2_udp6Entry_t;
#define MIB_FIRST_NEW_ELM_mib2_udp6Entry_t      udp6Instance

#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack()
#endif

/*
 * the RAWIP group
 */
typedef struct mib2_rawip {
                /* total # of RAWIP datagrams sent upstream */
        Counter rawipInDatagrams;
                /* # of RAWIP packets with bad IPV6_CHECKSUM checksums */
        Counter rawipInCksumErrs;
                /* # of recv'd dg's not deliverable (other) */
        Counter rawipInErrors;
                /* total # of dg's sent */
        Counter rawipOutDatagrams;
                /* total # of dg's not sent (e.g. no memory) */
        Counter rawipOutErrors;
} mib2_rawip_t;

/* DVMRP group */
#define EXPER_DVMRP_VIF         1
#define EXPER_DVMRP_MRT         2


/*
 * The SCTP group
 */
#define MIB2_SCTP_CONN                  15
#define MIB2_SCTP_CONN_LOCAL            16
#define MIB2_SCTP_CONN_REMOTE           17

#define MIB2_SCTP_closed                1
#define MIB2_SCTP_cookieWait            2
```

```
#define MIB2_SCTP_cookieEchoed          3
#define MIB2_SCTP_established           4
#define MIB2_SCTP_shutdownPending       5
#define MIB2_SCTP_shutdownSent          6
#define MIB2_SCTP_shutdownReceived      7
#define MIB2_SCTP_shutdownAckSent       8
#define MIB2_SCTP_deleteTCB             9
#define MIB2_SCTP_listen                10      /* Not in the MIB */

#define MIB2_SCTP_ACTIVE                1
#define MIB2_SCTP_INACTIVE              2

#define MIB2_SCTP_ADDR_V4               1
#define MIB2_SCTP_ADDR_V6               2

#define MIB2_SCTP_RTOALGO_OTHER         1
#define MIB2_SCTP_RTOALGO_VANJ          2

typedef struct mib2_sctpConnEntry {
                /* connection identifier        { sctpAssocEntry 1 } */
        uint32_t        sctpAssocId;
                /* remote hostname (not used)   { sctpAssocEntry 2 } */
        Octet_t         sctpAssocRemHostName;
                /* local port number            { sctpAssocEntry 3 } */
        uint32_t        sctpAssocLocalPort;
                /* remote port number           { sctpAssocEntry 4 } */
        uint32_t        sctpAssocRemPort;
                /* type of primary remote addr  { sctpAssocEntry 5 } */
        int             sctpAssocRemPrimAddrType;
                /* primary remote address       { sctpAssocEntry 6 } */
        Ip6Address      sctpAssocRemPrimAddr;
                /* local address */
        Ip6Address      sctpAssocLocPrimAddr;
                /* current heartbeat interval   { sctpAssocEntry 7 } */
        uint32_t        sctpAssocHeartBeatInterval;
                /* state of this association    { sctpAssocEntry 8 } */
        int             sctpAssocState;
                /* # of inbound streams         { sctpAssocEntry 9 } */
        uint32_t        sctpAssocInStreams;
                /* # of outbound streams        { sctpAssocEntry 10 } */
        uint32_t        sctpAssocOutStreams;
                /* max # of data retans         { sctpAssocEntry 11 } */
        uint32_t        sctpAssocMaxRetr;
                /* sysId for assoc owner        { sctpAssocEntry 12 } */
        uint32_t        sctpAssocPrimProcess;
                /* # of rxmit timeouts during hanshake */
        Counter32       sctpAssocT1expired;     /* { sctpAssocEntry 13 } */
                /* # of rxmit timeouts during shutdown */
        Counter32       sctpAssocT2expired;     /* { sctpAssocEntry 14 } */
                /* # of rxmit timeouts during data transfer */
        Counter32       sctpAssocRtxChunks;     /* { sctpAssocEntry 15 } */
                /* assoc start-up time          { sctpAssocEntry 16 } */
        uint32_t        sctpAssocStartTime;
        struct sctpConnEntryInfo_s {
                                /* amount of data in send Q */
                Gauge           ce_sendq;
                                /* amount of data in recv Q */
                Gauge           ce_recvq;
                                /* currect send window size */
                Gauge           ce_swnd;
                                /* currenct receive window size */
                Gauge           ce_rwnd;
                                /* current max segment size */
                Gauge           ce_mss;
        } sctpConnEntryInfo;
} mib2_sctpConnEntry_t;
```



```
typedef struct mib2_sctpConnLocalAddrEntry {
                /* connection identifier */
        uint32_t        sctpAssocId;
                /* type of local addr           { sctpAssocLocalEntry 1 } */
        int             sctpAssocLocalAddrType;
                /* local address                { sctpAssocLocalEntry 2 } */
        Ip6Address      sctpAssocLocalAddr;
} mib2_sctpConnLocalEntry_t;

typedef struct mib2_sctpConnRemoteAddrEntry {
                /* connection identier */
        uint32_t        sctpAssocId;
                /* remote addr type             { sctpAssocRemEntry 1 } */
        int             sctpAssocRemAddrType;
                /* remote address               { sctpAssocRemEntry 2 } */
        Ip6Address      sctpAssocRemAddr;
                /* is the address active        { sctpAssocRemEntry 3 } */
        int             sctpAssocRemAddrActive;
                /* whether hearbeat is active   { sctpAssocRemEntry 4 } */
        int             sctpAssocRemAddrHBActive;
                /* current RTO                  { sctpAssocRemEntry 5 } */
        uint32_t        sctpAssocRemAddrRTO;
                /* max # of rexmits before becoming inactive */
        uint32_t        sctpAssocRemAddrMaxPathRtx; /* {sctpAssocRemEntry 6} */
                /* # of rexmits to this dest    { sctpAssocRemEntry 7 } */
        uint32_t        sctpAssocRemAddrRtx;
} mib2_sctpConnRemoteEntry_t;



/* Pack data in mib2_sctp to make struct size the same for 32- and 64-bits */
#if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
#pragma pack(4)
#endif

typedef struct mib2_sctp {
                /* algorithm used to determine rto      { sctpParams 1 } */
        int             sctpRtoAlgorithm;
                /* min RTO in msecs                     { sctpParams 2 } */
        uint32_t        sctpRtoMin;
                /* max RTO in msecs                     { sctpParams 3 } */
        uint32_t        sctpRtoMax;
                /* initial RTO in msecs                 { sctpParams 4 } */
        uint32_t        sctpRtoInitial;
                /* max # of assocs                      { sctpParams 5 } */
        int32_t         sctpMaxAssocs;
                /* cookie lifetime in msecs             { sctpParams 6 } */
        uint32_t        sctpValCookieLife;
                /* max # of retrans in startup          { sctpParams 7 } */
        uint32_t        sctpMaxInitRetr;
        /* # of conns ESTABLISHED, SHUTDOWN-RECEIVED or SHUTDOWN-PENDING */
        Counter32       sctpCurrEstab;          /* { sctpStats 1 } */
                /* # of active opens                    { sctpStats 2 } */
        Counter32       sctpActiveEstab;
                /* # of passive opens                   { sctpStats 3 } */
        Counter32       sctpPassiveEstab;
                /* # of aborted conns                   { sctpStats 4 } */
        Counter32       sctpAborted;
                /* # of graceful shutdowns              { sctpStats 5 } */
        Counter32       sctpShutdowns;
                /* # of OOB packets                     { sctpStats 6 } */
        Counter32       sctpOutOfBlue;
                /* # of packets discarded due to cksum  { sctpStats 7 } */
        Counter32       sctpChecksumError;
                /* # of control chunks sent             { sctpStats 8 } */
```

```
1731         Counter64       sctpOutCtrlChunks;
1732                 /* # of ordered data chunks sent        { sctpStats 9 } */
1733         Counter64       sctpOutOrderChunks;
1734                 /* # of unordered data chunks sent      { sctpStats 10 } */
1735         Counter64       sctpOutUnorderChunks;
1736                 /* # of retransmitted data chunks */
1737         Counter64       sctpRetransChunks;
1738                 /* # of SACK chunks sent */
1739         Counter         sctpOutAck;
1740                 /* # of delayed ACK timeouts */
1741         Counter         sctpOutAckDelayed;
1742                 /* # of SACK chunks sent to update window */
1743         Counter         sctpOutWinUpdate;
1744                 /* # of fast retransmits */
1745         Counter         sctpOutFastRetrans;
1746                 /* # of window probes sent */
1747         Counter         sctpOutWinProbe;
1748                 /* # of control chunks received         { sctpStats 11 } */
1749         Counter64       sctpInCtrlChunks;
1750                 /* # of ordered data chunks rcvd        { sctpStats 12 } */
1751         Counter64       sctpInOrderChunks;
1752                 /* # of unord data chunks rcvd          { sctpStats 13 } */
1753         Counter64       sctpInUnorderChunks;
1754                 /* # of received SACK chunks */
1755         Counter         sctpInAck;
1756                 /* # of received SACK chunks with duplicate TSN */
1757         Counter         sctpInDupAck;
1758                 /* # of SACK chunks acking unsent data */
1759         Counter         sctpInAckUnsent;
1760                 /* # of Fragmented User Messages        { sctpStats 14 } */
1761         Counter64       sctpFragUsrMsgs;
1762                 /* # of Reassembled User Messages       { sctpStats 15 } */
1763         Counter64       sctpReasmUsrMsgs;
1764                 /* # of Sent SCTP Packets               { sctpStats 16 } */
1765         Counter64       sctpOutSCTPPkts;
1766                 /* # of Received SCTP Packets           { sctpStats 17 } */
1767         Counter64       sctpInSCTPPkts;
1768                 /* # of invalid cookies received */
1769         Counter         sctpInInvalidCookie;
1770                 /* total # of retransmit timeouts */
1771         Counter         sctpTimRetrans;
1772                 /* total # of retransmit timeouts dropping the connection */
1773         Counter         sctpTimRetransDrop;
1774                 /* total # of heartbeat probes */
1775         Counter         sctpTimHeartBeatProbe;
1776                 /* total # of heartbeat timeouts dropping the connection */
1777         Counter         sctpTimHeartBeatDrop;
1778                 /* total # of conns refused due to backlog full on listen */
1779         Counter         sctpListenDrop;
1780                 /* total # of pkts received after the association has closed */
1781         Counter         sctpInClosed;
1782         int             sctpEntrySize;
1783         int             sctpLocalEntrySize;
1784         int             sctpRemoteEntrySize;
1785 } mib2_sctp_t;

1787 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1788 #pragma pack()
1789 #endif


1792 #ifdef  __cplusplus
1793 }
1794 #endif

1796 #endif  /* _INET_MIB2_H */
```

```
**********************************************************
   32229 Mon Aug 17 21:08:05 2015
new/usr/src/uts/common/inet/sctp/sctp_snmp.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 518 /*
 519  * Return SNMP global stats in buffer in mpdata.
 520  * Return associatiation table in mp_conn_data,
 521  * local address table in mp_local_data, and
 522  * remote address table in mp_rem_data.
 523  */
 524 mblk_t *
 525 sctp_snmp_get_mib2(queue_t *q, mblk_t *mpctl, sctp_stack_t *sctps)
 526 {
 527         mblk_t                  *mpdata, *mp_ret;
 528         mblk_t                  *mp_conn_ctl = NULL;
 529         mblk_t                  *mp_conn_data;
 530         mblk_t                  *mp_conn_tail = NULL;
 531         mblk_t                  *mp_pidnode_ctl = NULL;
 532         mblk_t                  *mp_pidnode_data;
 533         mblk_t                  *mp_pidnode_tail = NULL;
 534 #endif /* ! codereview */
 535         mblk_t                  *mp_local_ctl = NULL;
 536         mblk_t                  *mp_local_data;
 537         mblk_t                  *mp_local_tail = NULL;
 538         mblk_t                  *mp_rem_ctl = NULL;
 539         mblk_t                  *mp_rem_data;
 540         mblk_t                  *mp_rem_tail = NULL;
 541         mblk_t                  *mp_attr_ctl = NULL;
 542         mblk_t                  *mp_attr_data;
 543         mblk_t                  *mp_attr_tail = NULL;
 544         struct opthdr           *optp;
 545         sctp_t                  *sctp, *sctp_prev = NULL;
 546         sctp_faddr_t            *fp;
 547         mib2_sctpConnEntry_t    sce;
 548         mib2_sctpConnLocalEntry_t       scle;
 549         mib2_sctpConnRemoteEntry_t      scre;
 550         mib2_transportMLPEntry_t        mlp;
 551         int                     i;
 552         int                     l;
 553         int                     scanned = 0;
 554         zoneid_t                zoneid = Q_TO_CONN(q)->conn_zoneid;
 555         conn_t                  *connp;
 556         boolean_t               needattr;
 557         int                     idx;
 558         mib2_sctp_t             sctp_mib;

 560         /*
 561          * Make copies of the original message.
 562          * mpctl will hold SCTP counters,
 563          * mp_conn_ctl will hold list of connections.
 564          */
 565         mp_ret = copymsg(mpctl);
 566         mp_conn_ctl = copymsg(mpctl);
 567         mp_pidnode_ctl = copymsg(mpctl);
 568 #endif /* ! codereview */
 569         mp_local_ctl = copymsg(mpctl);
 570         mp_rem_ctl = copymsg(mpctl);
 571         mp_attr_ctl = copymsg(mpctl);

 573         mpdata = mpctl->b_cont;

 575         if (mp_conn_ctl == NULL || mp_pidnode_ctl == NULL ||
 576             mp_local_ctl == NULL || mp_rem_ctl == NULL || mp_attr_ctl == NULL ||
```



```
 577             mpdata == NULL) {
 531         if (mp_conn_ctl == NULL || mp_local_ctl == NULL ||
 532             mp_rem_ctl == NULL || mp_attr_ctl == NULL || mpdata == NULL) {
 578                 freemsg(mp_attr_ctl);
 579                 freemsg(mp_rem_ctl);
 580                 freemsg(mp_local_ctl);
 581                 freemsg(mp_pidnode_ctl);
 582 #endif /* ! codereview */
 583                 freemsg(mp_conn_ctl);
 584                 freemsg(mp_ret);
 585                 freemsg(mpctl);
 586                 return (NULL);
 587         }
 588         mp_conn_data = mp_conn_ctl->b_cont;
 589         mp_pidnode_data = mp_pidnode_ctl->b_cont;
 590 #endif /* ! codereview */
 591         mp_local_data = mp_local_ctl->b_cont;
 592         mp_rem_data = mp_rem_ctl->b_cont;
 593         mp_attr_data = mp_attr_ctl->b_cont;

 595         bzero(&sctp_mib, sizeof (sctp_mib));

 597         /* hostname address parameters are not supported in Solaris */
 598         sce.sctpAssocRemHostName.o_length = 0;
 599         sce.sctpAssocRemHostName.o_bytes[0] = 0;

 601         /* build table of connections -- need count in fixed part */

 603         idx = 0;
 604         mutex_enter(&sctps->sctps_g_lock);
 605         sctp = list_head(&sctps->sctps_g_list);
 606         while (sctp != NULL) {
 607                 mutex_enter(&sctp->sctp_reflock);
 608                 if (sctp->sctp_condemned) {
 609                         mutex_exit(&sctp->sctp_reflock);
 610                         sctp = list_next(&sctps->sctps_g_list, sctp);
 611                         continue;
 612                 }
 613                 sctp->sctp_refcnt++;
 614                 mutex_exit(&sctp->sctp_reflock);
 615                 mutex_exit(&sctps->sctps_g_lock);
 616                 if (sctp_prev != NULL)
 617                         SCTP_REFRELE(sctp_prev);
 618                 if (sctp->sctp_connp->conn_zoneid != zoneid)
 619                         goto next_sctp;
 620                 if (sctp->sctp_state == SCTPS_ESTABLISHED ||
 621                     sctp->sctp_state == SCTPS_SHUTDOWN_PENDING ||
 622                     sctp->sctp_state == SCTPS_SHUTDOWN_RECEIVED) {
 623                         /*
 624                          * Just bump the local sctp_mib.  The number of
 625                          * existing associations is not kept in kernel.
 626                          */
 627                         BUMP_MIB(&sctp_mib, sctpCurrEstab);
 628                 }
 629                 SCTPS_UPDATE_MIB(sctps, sctpOutSCTPPkts, sctp->sctp_opkts);
 630                 sctp->sctp_opkts = 0;
 631                 SCTPS_UPDATE_MIB(sctps, sctpOutCtrlChunks, sctp->sctp_obchunks);
 632                 UPDATE_LOCAL(sctp->sctp_cum_obchunks,
 633                     sctp->sctp_obchunks);
 634                 sctp->sctp_obchunks = 0;
 635                 SCTPS_UPDATE_MIB(sctps, sctpOutOrderChunks,
 636                     sctp->sctp_odchunks);
 637                 UPDATE_LOCAL(sctp->sctp_cum_odchunks,
 638                     sctp->sctp_odchunks);
 639                 sctp->sctp_odchunks = 0;
 640                 SCTPS_UPDATE_MIB(sctps, sctpOutUnorderChunks,
```

```
641                     sctp->sctp_oudchunks);
642                 UPDATE_LOCAL(sctp->sctp_cum_oudchunks,
643                     sctp->sctp_oudchunks);
644                 sctp->sctp_oudchunks = 0;
645                 SCTPS_UPDATE_MIB(sctps, sctpRetransChunks,
646                     sctp->sctp_rxtchunks);
647                 UPDATE_LOCAL(sctp->sctp_cum_rxtchunks,
648                     sctp->sctp_rxtchunks);
649                 sctp->sctp_rxtchunks = 0;
650                 SCTPS_UPDATE_MIB(sctps, sctpInSCTPPkts, sctp->sctp_ipkts);
651                 sctp->sctp_ipkts = 0;
652                 SCTPS_UPDATE_MIB(sctps, sctpInCtrlChunks, sctp->sctp_ibchunks);
653                 UPDATE_LOCAL(sctp->sctp_cum_ibchunks,
654                     sctp->sctp_ibchunks);
655                 sctp->sctp_ibchunks = 0;
656                 SCTPS_UPDATE_MIB(sctps, sctpInOrderChunks, sctp->sctp_idchunks);
657                 UPDATE_LOCAL(sctp->sctp_cum_idchunks,
658                     sctp->sctp_idchunks);
659                 sctp->sctp_idchunks = 0;
660                 SCTPS_UPDATE_MIB(sctps, sctpInUnorderChunks,
661                     sctp->sctp_iudchunks);
662                 UPDATE_LOCAL(sctp->sctp_cum_iudchunks,
663                     sctp->sctp_iudchunks);
664                 sctp->sctp_iudchunks = 0;
665                 SCTPS_UPDATE_MIB(sctps, sctpFragUsrMsgs, sctp->sctp_fragdmsgs);
666                 sctp->sctp_fragdmsgs = 0;
667                 SCTPS_UPDATE_MIB(sctps, sctpReasmUsrMsgs, sctp->sctp_reassmsgs);
668                 sctp->sctp_reassmsgs = 0;

670                 sce.sctpAssocId = ntohl(sctp->sctp_lvtag);
671                 sce.sctpAssocLocalPort = ntohs(sctp->sctp_connp->conn_lport);
672                 sce.sctpAssocRemPort = ntohs(sctp->sctp_connp->conn_fport);

674                 RUN_SCTP(sctp);
675                 if (sctp->sctp_primary != NULL) {
676                         fp = sctp->sctp_primary;

678                         if (IN6_IS_ADDR_V4MAPPED(&fp->sf_faddr)) {
679                                 sce.sctpAssocRemPrimAddrType =
680                                     MIB2_SCTP_ADDR_V4;
681                         } else {
682                                 sce.sctpAssocRemPrimAddrType =
683                                     MIB2_SCTP_ADDR_V6;
684                         }
685                         sce.sctpAssocRemPrimAddr = fp->sf_faddr;
686                         sce.sctpAssocLocPrimAddr = fp->sf_saddr;
687                         sce.sctpAssocHeartBeatInterval = TICK_TO_MSEC(
688                             fp->sf_hb_interval);
689                 } else {
690                         sce.sctpAssocRemPrimAddrType = MIB2_SCTP_ADDR_V4;
691                         bzero(&sce.sctpAssocRemPrimAddr,
692                             sizeof (sce.sctpAssocRemPrimAddr));
693                         bzero(&sce.sctpAssocLocPrimAddr,
694                             sizeof (sce.sctpAssocLocPrimAddr));
695                         sce.sctpAssocHeartBeatInterval =
696                             sctps->sctps_heartbeat_interval;
697                 }

699                 /*
700                  * Table for local addresses
701                  */
702                 scanned = 0;
703                 for (i = 0; i < SCTP_IPIF_HASH; i++) {
704                         sctp_saddr_ipif_t       *obj;

706                         if (sctp->sctp_saddrs[i].ipif_count == 0)
```



```
707                                 continue;
708                         obj = list_head(&sctp->sctp_saddrs[i].sctp_ipif_list);
709                         for (l = 0; l < sctp->sctp_saddrs[i].ipif_count; l++) {
710                                 sctp_ipif_t     *sctp_ipif;
711                                 in6_addr_t      addr;

713                                 sctp_ipif = obj->saddr_ipifp;
714                                 addr = sctp_ipif->sctp_ipif_saddr;
715                                 scanned++;
716                                 scle.sctpAssocId = ntohl(sctp->sctp_lvtag);
717                                 if (IN6_IS_ADDR_V4MAPPED(&addr)) {
718                                         scle.sctpAssocLocalAddrType =
719                                             MIB2_SCTP_ADDR_V4;
720                                 } else {
721                                         scle.sctpAssocLocalAddrType =
722                                             MIB2_SCTP_ADDR_V6;
723                                 }
724                                 scle.sctpAssocLocalAddr = addr;
725                                 (void) snmp_append_data2(mp_local_data,
726                                     &mp_local_tail, (char *)&scle,
727                                     sizeof (scle));
728                                 if (scanned >= sctp->sctp_nsaddrs)
729                                         goto done;
730                                 obj = list_next(&sctp->
731                                     sctp_saddrs[i].sctp_ipif_list, obj);
732                         }
733                 }
734 done:
735                 /*
736                  * Table for remote addresses
737                  */
738                 for (fp = sctp->sctp_faddrs; fp; fp = fp->sf_next) {
739                         scre.sctpAssocId = ntohl(sctp->sctp_lvtag);
740                         if (IN6_IS_ADDR_V4MAPPED(&fp->sf_faddr)) {
741                                 scre.sctpAssocRemAddrType = MIB2_SCTP_ADDR_V4;
742                         } else {
743                                 scre.sctpAssocRemAddrType = MIB2_SCTP_ADDR_V6;
744                         }
745                         scre.sctpAssocRemAddr = fp->sf_faddr;
746                         if (fp->sf_state == SCTP_FADDRS_ALIVE) {
747                                 scre.sctpAssocRemAddrActive =
748                                     scre.sctpAssocRemAddrHBActive =
749                                     MIB2_SCTP_ACTIVE;
750                         } else {
751                                 scre.sctpAssocRemAddrActive =
752                                     scre.sctpAssocRemAddrHBActive =
753                                     MIB2_SCTP_INACTIVE;
754                         }
755                         scre.sctpAssocRemAddrRTO = TICK_TO_MSEC(fp->sf_rto);
756                         scre.sctpAssocRemAddrMaxPathRtx = fp->sf_max_retr;
757                         scre.sctpAssocRemAddrRtx = fp->sf_T3expire;
758                         (void) snmp_append_data2(mp_rem_data, &mp_rem_tail,
759                             (char *)&scre, sizeof (scre));
760                 }
761                 connp = sctp->sctp_connp;
762                 needattr = B_FALSE;
763                 bzero(&mlp, sizeof (mlp));
764                 if (connp->conn_mlp_type != mlptSingle) {
765                         if (connp->conn_mlp_type == mlptShared ||
766                             connp->conn_mlp_type == mlptBoth)
767                                 mlp.tme_flags |= MIB2_TMEF_SHARED;
768                         if (connp->conn_mlp_type == mlptPrivate ||
769                             connp->conn_mlp_type == mlptBoth)
770                                 mlp.tme_flags |= MIB2_TMEF_PRIVATE;
771                         needattr = B_TRUE;
772                 }
```

```
773                 if (connp->conn_anon_mlp) {
774                         mlp.tme_flags |= MIB2_TMEF_ANONMLP;
775                         needattr = B_TRUE;
776                 }
777                 switch (connp->conn_mac_mode) {
778                 case CONN_MAC_DEFAULT:
779                         break;
780                 case CONN_MAC_AWARE:
781                         mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
782                         needattr = B_TRUE;
783                         break;
784                 case CONN_MAC_IMPLICIT:
785                         mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
786                         needattr = B_TRUE;
787                         break;
788                 }
789                 if (sctp->sctp_connp->conn_ixa->ixa_tsl != NULL) {
790                         ts_label_t *tsl;

792                         tsl = sctp->sctp_connp->conn_ixa->ixa_tsl;
793                         mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
794                         mlp.tme_doi = label2doi(tsl);
795                         mlp.tme_label = *label2bslabel(tsl);
796                         needattr = B_TRUE;
797                 }
798                 WAKE_SCTP(sctp);
799                 sce.sctpAssocState = sctp_snmp_state(sctp);
800                 sce.sctpAssocInStreams = sctp->sctp_num_istr;
801                 sce.sctpAssocOutStreams = sctp->sctp_num_ostr;
802                 sce.sctpAssocMaxRetr = sctp->sctp_pa_max_rxt;
803                 /* A 0 here indicates that no primary process is known */
804                 sce.sctpAssocPrimProcess = 0;
805                 sce.sctpAssocT1expired = sctp->sctp_T1expire;
806                 sce.sctpAssocT2expired = sctp->sctp_T2expire;
807                 sce.sctpAssocRtxChunks = sctp->sctp_T3expire;
808                 sce.sctpAssocStartTime = sctp->sctp_assoc_start_time;
809                 sce.sctpConnEntryInfo.ce_sendq = sctp->sctp_unacked +
810                     sctp->sctp_unsent;
811                 sce.sctpConnEntryInfo.ce_recvq = sctp->sctp_rxqueued;
812                 sce.sctpConnEntryInfo.ce_swnd = sctp->sctp_frwnd;
813                 sce.sctpConnEntryInfo.ce_rwnd = sctp->sctp_rwnd;
814                 sce.sctpConnEntryInfo.ce_mss = sctp->sctp_mss;
815                 (void) snmp_append_data2(mp_conn_data, &mp_conn_tail,
816                     (char *)&sce, sizeof (sce));

818                 (void) snmp_append_data2(mp_pidnode_data, &mp_pidnode_tail,
819                     (char *)&sce, sizeof (sce));

821                 (void) snmp_append_mblk2(mp_pidnode_data, &mp_pidnode_tail,
822                     conn_get_pid_mblk(connp));

824 #endif /* ! codereview */
825                 mlp.tme_connidx = idx++;
826                 if (needattr)
827                         (void) snmp_append_data2(mp_attr_ctl->b_cont,
828                             &mp_attr_tail, (char *)&mlp, sizeof (mlp));
829 next_sctp:
830                 sctp_prev = sctp;
831                 mutex_enter(&sctps->sctps_g_lock);
832                 sctp = list_next(&sctps->sctps_g_list, sctp);
833         }
834         mutex_exit(&sctps->sctps_g_lock);
835         if (sctp_prev != NULL)
836                 SCTP_REFRELE(sctp_prev);

838         sctp_sum_mib(sctps, &sctp_mib);
```



```
840         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
841         optp->level = MIB2_SCTP;
842         optp->name = 0;
843         (void) snmp_append_data(mpdata, (char *)&sctp_mib, sizeof (sctp_mib));
844         optp->len = msgdsize(mpdata);
845         qreply(q, mpctl);

847         /* table of connections... */
848         optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
849             sizeof (struct T_optmgmt_ack)];
850         optp->level = MIB2_SCTP;
851         optp->name = MIB2_SCTP_CONN;
852         optp->len = msgdsize(mp_conn_data);
853         qreply(q, mp_conn_ctl);

855         /* table of EXPER_XPORT_PROC_INFO */
856         optp = (struct opthdr *)&mp_pidnode_ctl->b_rptr[
857             sizeof (struct T_optmgmt_ack)];
858         optp->level = MIB2_SCTP;
859         optp->name = EXPER_XPORT_PROC_INFO;
860         optp->len = msgdsize(mp_pidnode_data);
861         qreply(q, mp_pidnode_ctl);
862 #endif /* ! codereview */

864         /* assoc local address table */
865         optp = (struct opthdr *)&mp_local_ctl->b_rptr[
866             sizeof (struct T_optmgmt_ack)];
867         optp->level = MIB2_SCTP;
868         optp->name = MIB2_SCTP_CONN_LOCAL;
869         optp->len = msgdsize(mp_local_data);
870         qreply(q, mp_local_ctl);

872         /* assoc remote address table */
873         optp = (struct opthdr *)&mp_rem_ctl->b_rptr[
874             sizeof (struct T_optmgmt_ack)];
875         optp->level = MIB2_SCTP;
876         optp->name = MIB2_SCTP_CONN_REMOTE;
877         optp->len = msgdsize(mp_rem_data);
878         qreply(q, mp_rem_ctl);

880         /* table of MLP attributes */
881         optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
882             sizeof (struct T_optmgmt_ack)];
883         optp->level = MIB2_SCTP;
884         optp->name = EXPER_XPORT_MLP;
885         optp->len = msgdsize(mp_attr_data);
886         if (optp->len == 0)
887                 freemsg(mp_attr_ctl);
888         else
889                 qreply(q, mp_attr_ctl);

891         return (mp_ret);
892 }

894 /* Translate SCTP state to MIB2 SCTP state. */
895 static int
896 sctp_snmp_state(sctp_t *sctp)
897 {
898         if (sctp == NULL)
899                 return (0);

901         switch (sctp->sctp_state) {
902         case SCTPS_IDLE:
903         case SCTPS_BOUND:
904                 return (MIB2_SCTP_closed);
```

```
905         case SCTPS_LISTEN:
906                 return (MIB2_SCTP_listen);
907         case SCTPS_COOKIE_WAIT:
908                 return (MIB2_SCTP_cookieWait);
909         case SCTPS_COOKIE_ECHOED:
910                 return (MIB2_SCTP_cookieEchoed);
911         case SCTPS_ESTABLISHED:
912                 return (MIB2_SCTP_established);
913         case SCTPS_SHUTDOWN_PENDING:
914                 return (MIB2_SCTP_shutdownPending);
915         case SCTPS_SHUTDOWN_SENT:
916                 return (MIB2_SCTP_shutdownSent);
917         case SCTPS_SHUTDOWN_RECEIVED:
918                 return (MIB2_SCTP_shutdownReceived);
919         case SCTPS_SHUTDOWN_ACK_SENT:
920                 return (MIB2_SCTP_shutdownAckSent);
921         default:
922                 return (0);
923         }
924 }

926 /*
927  * To sum up all MIB2 stats for a sctp_stack_t from all per CPU stats.  The
928  * caller should initialize the target mib2_sctp_t properly as this function
929  * just adds up all the per CPU stats.
930  */
931 static void
932 sctp_sum_mib(sctp_stack_t *sctps, mib2_sctp_t *sctp_mib)
933 {
934         int i;
935         int cnt;

937         /* Static componets of mib2_sctp_t. */
938         SET_MIB(sctp_mib->sctpRtoAlgorithm, MIB2_SCTP_RTOALGO_VANJ);
939         SET_MIB(sctp_mib->sctpRtoMin, sctps->sctps_rto_ming);
940         SET_MIB(sctp_mib->sctpRtoMax, sctps->sctps_rto_maxg);
941         SET_MIB(sctp_mib->sctpRtoInitial, sctps->sctps_rto_initialg);
942         SET_MIB(sctp_mib->sctpMaxAssocs, -1);
943         SET_MIB(sctp_mib->sctpValCookieLife, sctps->sctps_cookie_life);
944         SET_MIB(sctp_mib->sctpMaxInitRetr, sctps->sctps_max_init_retr);

946         /* fixed length structure for IPv4 and IPv6 counters */
947         SET_MIB(sctp_mib->sctpEntrySize, sizeof (mib2_sctpConnEntry_t));
948         SET_MIB(sctp_mib->sctpLocalEntrySize,
949             sizeof (mib2_sctpConnLocalEntry_t));
950         SET_MIB(sctp_mib->sctpRemoteEntrySize,
951             sizeof (mib2_sctpConnRemoteEntry_t));

953         /*
954          * sctps_sc_cnt may change in the middle of the loop.  It is better
955          * to get its value first.
956          */
957         cnt = sctps->sctps_sc_cnt;
958         for (i = 0; i < cnt; i++)
959                 sctp_add_mib(&sctps->sctps_sc[i]->sctp_sc_mib, sctp_mib);
960 }

962 static void
963 sctp_add_mib(mib2_sctp_t *from, mib2_sctp_t *to)
964 {
965         to->sctpActiveEstab += from->sctpActiveEstab;
966         to->sctpPassiveEstab += from->sctpPassiveEstab;
967         to->sctpAborted += from->sctpAborted;
968         to->sctpShutdowns += from->sctpShutdowns;
969         to->sctpOutOfBlue += from->sctpOutOfBlue;
970         to->sctpChecksumError += from->sctpChecksumError;
```



```
971         to->sctpOutCtrlChunks += from->sctpOutCtrlChunks;
972         to->sctpOutOrderChunks += from->sctpOutOrderChunks;
973         to->sctpOutUnorderChunks += from->sctpOutUnorderChunks;
974         to->sctpRetransChunks += from->sctpRetransChunks;
975         to->sctpOutAck += from->sctpOutAck;
976         to->sctpOutAckDelayed += from->sctpOutAckDelayed;
977         to->sctpOutWinUpdate += from->sctpOutWinUpdate;
978         to->sctpOutFastRetrans += from->sctpOutFastRetrans;
979         to->sctpOutWinProbe += from->sctpOutWinProbe;
980         to->sctpInCtrlChunks += from->sctpInCtrlChunks;
981         to->sctpInOrderChunks += from->sctpInOrderChunks;
982         to->sctpInUnorderChunks += from->sctpInUnorderChunks;
983         to->sctpInAck += from->sctpInAck;
984         to->sctpInDupAck += from->sctpInDupAck;
985         to->sctpInAckUnsent += from->sctpInAckUnsent;
986         to->sctpFragUsrMsgs += from->sctpFragUsrMsgs;
987         to->sctpReasmUsrMsgs += from->sctpReasmUsrMsgs;
988         to->sctpOutSCTPPkts += from->sctpOutSCTPPkts;
989         to->sctpInSCTPPkts += from->sctpInSCTPPkts;
990         to->sctpInInvalidCookie += from->sctpInInvalidCookie;
991         to->sctpTimRetrans += from->sctpTimRetrans;
992         to->sctpTimRetransDrop += from->sctpTimRetransDrop;
993         to->sctpTimHeartBeatProbe += from->sctpTimHeartBeatProbe;
994         to->sctpTimHeartBeatDrop += from->sctpTimHeartBeatDrop;
995         to->sctpListenDrop += from->sctpListenDrop;
996         to->sctpInClosed += from->sctpInClosed;
997 }
```

```
**********************************************************
    9932 Mon Aug 17 21:08:05 2015
new/usr/src/uts/common/inet/snmpcom.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_
```

```
110 int
111 snmp_append_mblk(mblk_t *mpdata, mblk_t *mblk)
112 {
113         if (!mpdata || !mblk)
114                 return (0);
115         while (mpdata->b_cont)
116                 mpdata = mpdata->b_cont;
117         mpdata->b_cont = mblk;
118         return (1);
119 }

121 #endif /* ! codereview */
122 /*
123  * Need a form which avoids O(n^2) behavior locating the end of the
124  * chain every time.  This is it.
125  */
126 int
127 snmp_append_data2(mblk_t *mpdata, mblk_t **last_mpp, char *blob, int len)
128 {

130         if (!mpdata)
131                 return (0);
132         if (*last_mpp == NULL) {
133                 while (mpdata->b_cont)
134                         mpdata = mpdata->b_cont;
135                 *last_mpp = mpdata;
136         }
137         if ((*last_mpp)->b_wptr + len >= (*last_mpp)->b_datap->db_lim) {
138                 (*last_mpp)->b_cont = allocb(DATA_MBLK_SIZE, BPRI_HI);
139                 *last_mpp = (*last_mpp)->b_cont;
140                 if (!*last_mpp)
141                         return (0);
142         }
143         bcopy(blob, (char *)(*last_mpp)->b_wptr, len);
144         (*last_mpp)->b_wptr += len;
145         return (1);
146 }

148 int
149 snmp_append_mblk2(mblk_t *mpdata, mblk_t **last_mpp, mblk_t *mblk)
150 {
151         if (!mpdata || !mblk)
152                 return (0);
153         if (*last_mpp == NULL) {
154                 while (mpdata->b_cont)
155                         mpdata = mpdata->b_cont;
156                 *last_mpp = mpdata;
157         }
158         (*last_mpp)->b_cont = mblk;
159         *last_mpp = (*last_mpp)->b_cont;
160 #endif /* ! codereview */
161         return (1);
162 }

164 /*
165  * SNMP requests are issued using putmsg() on a stream containing all
166  * relevant modules.  The ctl part contains a O_T_OPTMGMT_REQ message,
167  * and the data part is NULL
168  * to process this msg. If snmpcom_req() returns FALSE, then the module
```



```
169  * will try optcom_req to see if its some sort of SOCKET or IP option.
170  * snmpcom_req returns TRUE whenever the first option is recognized as
171  * an SNMP request, even if a bad one.
172  *
173  * "get" is done by a single O_T_OPTMGMT_REQ with MGMT_flags set to T_CURRENT.
174  * All modules respond with one or msg's about what they know.  Responses
175  * are in T_OPTMGMT_ACK format.  The opthdr level/name fields identify what
176  * is begin returned, the len field how big it is (in bytes).  The info
177  * itself is in the data portion of the msg.  Fixed length info returned
178  * in one msg; each table in a separate msg.
179  *
180  * setfn() returns 1 if things ok, 0 if set request invalid or otherwise
181  * messed up.
182  *
183  * If the passed q is at the bottom of the module chain (q_next == NULL,
184  * a ctl msg with req->name, level, len all zero is sent upstream.  This
185  * is and EOD flag to the caller.
186  *
187  * IMPORTANT:
188  * - The msg type is M_PROTO, not M_PCPROTO!!!  This is by design,
189  *   since multiple messages will be sent to stream head and we want
190  *   them queued for reading, not discarded.
191  * - All requests which match a table entry are sent to all get/set functions
192  *   of each module.  The functions must simply ignore requests not meant
193  *   for them: getfn() returns 0, setfn() returns 1.
194  */
195 boolean_t
196 snmpcom_req(queue_t *q, mblk_t *mp, pfi_t setfn, pfi_t getfn, cred_t *credp)
197 {
198         mblk_t                  *mpctl;
199         struct opthdr           *req;
200         struct opthdr           *next_req;
201         struct opthdr           *req_end;
202         struct opthdr           *req_start;
203         sor_t                   *sreq;
204         struct T_optmgmt_req    *tor = (struct T_optmgmt_req *)mp->b_rptr;
205         struct T_optmgmt_ack    *toa;
206         boolean_t               legacy_req;

208         if (mp->b_cont) {       /* don't deal with multiple mblk's */
209                 freemsg(mp->b_cont);
210                 mp->b_cont = (mblk_t *)0;
211                 optcom_err_ack(q, mp, TSYSERR, EBADMSG);
212                 return (B_TRUE);
213         }
214         if ((mp->b_wptr - mp->b_rptr) < sizeof (struct T_optmgmt_req) ||
215             !(req_start = (struct opthdr *)mi_offset_param(mp,
216             tor->OPT_offset, tor->OPT_length)))
217                 goto bad_req1;
218         if (! __TPI_OPT_ISALIGNED(req_start))
219                 goto bad_req1;

221         /*
222          * if first option not in the MIB2 or EXPER range, return false so
223          * optcom_req can scope things out.  Otherwise it's passed to each
224          * calling module to process or ignore as it sees fit.
225          */
226         if ((!(req_start->level >= MIB2_RANGE_START &&
227             req_start->level <= MIB2_RANGE_END)) &&
228             (!(req_start->level >= EXPER_RANGE_START &&
229             req_start->level <= EXPER_RANGE_END)))
230                 return (B_FALSE);

232         switch (tor->MGMT_flags) {

234         case T_NEGOTIATE:
```

```
235                 if (secpolicy_ip_config(credp, B_FALSE) != 0) {
236                         optcom_err_ack(q, mp, TACCES, 0);
237                         return (B_TRUE);
238                 }
239                 req_end = (struct opthdr *)((uchar_t *)req_start +
240                     tor->OPT_length);
241                 for (req = req_start; req < req_end; req = next_req) {
242                         next_req =
243                             (struct opthdr *)((uchar_t *)&req[1] +
244                             _TPI_ALIGN_OPT(req->len));
245                         if (next_req > req_end)
246                                 goto bad_req2;
247                         for (sreq = req_arr; sreq < A_END(req_arr); sreq++) {
248                                 if (req->level == sreq->sor_group &&
249                                     req->name == sreq->sor_code)
250                                         break;
251                         }
252                         if (sreq >= A_END(req_arr))
253                                 goto bad_req3;
254                         if (!(*setfn)(q, req->level, req->name,
255                             (uchar_t *)&req[1], req->len))
256                                 goto bad_req4;
257                 }
258                 if (q->q_next != NULL)
259                         putnext(q, mp);
260                 else
261                         freemsg(mp);
262                 return (B_TRUE);

264         case OLD_T_CURRENT:
265         case T_CURRENT:
266                 mpctl = allocb(TOAHDR_SIZE, BPRI_MED);
267                 if (!mpctl) {
268                         optcom_err_ack(q, mp, TSYSERR, ENOMEM);
269                         return (B_TRUE);
270                 }
271                 mpctl->b_cont = allocb(DATA_MBLK_SIZE, BPRI_MED);
272                 if (!mpctl->b_cont) {
273                         freemsg(mpctl);
274                         optcom_err_ack(q, mp, TSYSERR, ENOMEM);
275                         return (B_TRUE);
276                 }
277                 mpctl->b_datap->db_type = M_PROTO;
278                 mpctl->b_wptr += TOAHDR_SIZE;
279                 toa = (struct T_optmgmt_ack *)mpctl->b_rptr;
280                 toa->PRIM_type = T_OPTMGMT_ACK;
281                 toa->OPT_offset = sizeof (struct T_optmgmt_ack);
282                 toa->OPT_length = sizeof (struct opthdr);
283                 toa->MGMT_flags = T_SUCCESS;
284                 /*
285                  * If the current process is running inside a solaris10-
286                  * branded zone and len is 0 then it's a request for
287                  * legacy data.
288                  */
289                 if (PROC_IS_BRANDED(curproc) &&
290                     (strcmp(curproc->p_brand->b_name, "solaris10") == 0) &&
291                     (req_start->len == 0))
292                         legacy_req = B_TRUE;
293                 else
294                         legacy_req = B_FALSE;
295                 if (!(*getfn)(q, mpctl, req_start->level, legacy_req))
296                         freemsg(mpctl);
297                 /*
298                  * all data for this module has now been sent upstream.  If
299                  * this is bottom module of stream, send up an EOD ctl msg,
300                  * otherwise pass onto the next guy for processing.
```



```
301                  */
302                 if (q->q_next != NULL) {
303                         putnext(q, mp);
304                         return (B_TRUE);
305                 }
306                 if (mp->b_cont) {
307                         freemsg(mp->b_cont);
308                         mp->b_cont = NULL;
309                 }
310                 mpctl = reallocb(mp, TOAHDR_SIZE, 1);
311                 if (!mpctl) {
312                         optcom_err_ack(q, mp, TSYSERR, ENOMEM);
313                         return (B_TRUE);
314                 }
315                 mpctl->b_datap->db_type = M_PROTO;
316                 mpctl->b_wptr = mpctl->b_rptr + TOAHDR_SIZE;
317                 toa = (struct T_optmgmt_ack *)mpctl->b_rptr;
318                 toa->PRIM_type = T_OPTMGMT_ACK;
319                 toa->OPT_offset = sizeof (struct T_optmgmt_ack);
320                 toa->OPT_length = sizeof (struct opthdr);
321                 toa->MGMT_flags = T_SUCCESS;
322                 req = (struct opthdr *)&toa[1];
323                 req->level = 0;
324                 req->name = 0;
325                 req->len = 0;
326                 qreply(q, mpctl);
327                 return (B_TRUE);

329         default:
330                 optcom_err_ack(q, mp, TBADFLAG, 0);
331                 return (B_TRUE);
332         }

334 bad_req1:;
335         printf("snmpcom bad_req1\n");
336         goto bad_req;
337 bad_req2:;
338         printf("snmpcom bad_req2\n");
339         goto bad_req;
340 bad_req3:;
341         printf("snmpcom bad_req3\n");
342         goto bad_req;
343 bad_req4:;
344         printf("snmpcom bad_req4\n");
345         /* FALLTHRU */
346 bad_req:;
347         optcom_err_ack(q, mp, TBADOPT, 0);
348         return (B_TRUE);

350 }
```

```
**********************************************************
    1768 Mon Aug 17 21:08:06 2015
new/usr/src/uts/common/inet/snmpcom.h
XXXX adding PID information to netstat output
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/* Copyright (c) 1990 Mentat Inc. */

#ifndef _INET_SNMPCOM_H
#define _INET_SNMPCOM_H

#ifdef  __cplusplus
extern "C" {
#endif

#if defined(_KERNEL) && defined(__STDC__)

/* snmpcom_req function prototypes */
typedef int (*snmp_setf_t)(queue_t *, int, int, uchar_t *, int);
typedef int (*snmp_getf_t)(queue_t *, mblk_t *, int, boolean_t);

extern int      snmp_append_data(mblk_t *mpdata, char *blob, int len);
extern int      snmp_append_mblk(mblk_t *mpdata, mblk_t *mblk);
#endif /* ! codereview */
extern int      snmp_append_data2(mblk_t *mpdata, mblk_t **last_mpp,
                    char *blob, int len);
extern int      snmp_append_mblk2(mblk_t *mpdata, mblk_t **last_mpp,
                    mblk_t *mblk);
#endif /* ! codereview */

extern boolean_t        snmpcom_req(queue_t *q, mblk_t *mp,
    snmp_setf_t setfn, snmp_getf_t getfn, cred_t *cr);

#endif  /* defined(_KERNEL) && defined(__STDC__) */

#ifdef  __cplusplus
}
#endif

#endif  /* _INET_SNMPCOM_H */
```

```
**********************************************************
   54234 Mon Aug 17 21:08:06 2015
new/usr/src/uts/common/inet/sockmods/socksctp.c
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/buf.h>
#include <sys/vfs.h>
#include <sys/vnode.h>
#include <sys/fcntl.h>
#endif /* ! codereview */
#include <sys/debug.h>
#include <sys/errno.h>
#include <sys/stropts.h>
#include <sys/cmn_err.h>
#include <sys/sysmacros.h>
#include <sys/filio.h>
#include <sys/policy.h>

#include <sys/project.h>
#include <sys/tihdr.h>
#include <sys/strsubr.h>
#include <sys/esunddi.h>
#include <sys/ddi.h>

#include <sys/sockio.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/strsun.h>

#include <netinet/sctp.h>
#include <inet/sctp_itf.h>
#include <fs/sockfs/sockcommon.h>
#include "socksctp.h"

/*
 * SCTP sockfs sonode operations, 1-1 socket
 */
```



```
static int sosctp_init(struct sonode *, struct sonode *, struct cred *, int);
static int sosctp_accept(struct sonode *, int, struct cred *, struct sonode **);
static int sosctp_bind(struct sonode *, struct sockaddr *, socklen_t, int,
    struct cred *);
static int sosctp_listen(struct sonode *, int, struct cred *);
static int sosctp_connect(struct sonode *, struct sockaddr *, socklen_t,
    int, int, struct cred *);
static int sosctp_recvmsg(struct sonode *, struct nmsghdr *, struct uio *,
    struct cred *);
static int sosctp_sendmsg(struct sonode *, struct nmsghdr *, struct uio *,
    struct cred *);
static int sosctp_getpeername(struct sonode *, struct sockaddr *, socklen_t *,
    boolean_t, struct cred *);
static int sosctp_getsockname(struct sonode *, struct sockaddr *, socklen_t *,
    struct cred *);
static int sosctp_shutdown(struct sonode *, int, struct cred *);
static int sosctp_getsockopt(struct sonode *, int, int, void *, socklen_t *,
    int, struct cred *);
static int sosctp_setsockopt(struct sonode *, int, int, const void *,
    socklen_t, struct cred *);
static int sosctp_ioctl(struct sonode *, int, intptr_t, int, struct cred *,
    int32_t *);
static int sosctp_close(struct sonode *, int, struct cred *);
void sosctp_fini(struct sonode *, struct cred *);

/*
 * SCTP sockfs sonode operations, 1-N socket
 */
static int sosctp_seq_connect(struct sonode *, struct sockaddr *,
    socklen_t, int, int, struct cred *);
static int sosctp_seq_sendmsg(struct sonode *, struct nmsghdr *, struct uio *,
    struct cred *);

/*
 * Socket association upcalls, 1-N socket connection
 */
sock_upper_handle_t sctp_assoc_newconn(sock_upper_handle_t,
    sock_lower_handle_t, sock_downcalls_t *, struct cred *, pid_t,
    sock_upcalls_t **);
static void sctp_assoc_connected(sock_upper_handle_t, sock_connid_t,
    struct cred *, pid_t);
static int sctp_assoc_disconnected(sock_upper_handle_t, sock_connid_t, int);
static void sctp_assoc_disconnecting(sock_upper_handle_t, sock_opctl_action_t,
    uintptr_t arg);
static ssize_t sctp_assoc_recv(sock_upper_handle_t, mblk_t *, size_t, int,
    int *, boolean_t *);
static void sctp_assoc_xmitted(sock_upper_handle_t, boolean_t);
static void sctp_assoc_properties(sock_upper_handle_t,
    struct sock_proto_props *);
static mblk_t *sctp_get_sock_pid_mblk(sock_upper_handle_t);
#endif /* ! codereview */

sonodeops_t sosctp_sonodeops = {
        sosctp_init,                    /* sop_init     */
        sosctp_accept,                  /* sop_accept   */
        sosctp_bind,                    /* sop_bind     */
        sosctp_listen,                  /* sop_listen   */
        sosctp_connect,                 /* sop_connect  */
        sosctp_recvmsg,                 /* sop_recvmsg  */
        sosctp_sendmsg,                 /* sop_sendmsg  */
        so_sendmblk_notsupp,            /* sop_sendmblk */
        sosctp_getpeername,             /* sop_getpeername */
        sosctp_getsockname,             /* sop_getsockname */
        sosctp_shutdown,                /* sop_shutdown */
        sosctp_getsockopt,              /* sop_getsockopt */
        sosctp_setsockopt,              /* sop_setsockopt */
```

```
128         sosctp_ioctl,                   /* sop_ioctl    */
129         so_poll,                        /* sop_poll     */
130         sosctp_close,                   /* sop_close    */
131 };

133 sonodeops_t sosctp_seq_sonodeops = {
134         sosctp_init,                    /* sop_init     */
135         so_accept_notsupp,              /* sop_accept   */
136         sosctp_bind,                    /* sop_bind     */
137         sosctp_listen,                  /* sop_listen   */
138         sosctp_seq_connect,             /* sop_connect  */
139         sosctp_recvmsg,                 /* sop_recvmsg  */
140         sosctp_seq_sendmsg,             /* sop_sendmsg  */
141         so_sendmblk_notsupp,            /* sop_sendmblk */
142         so_getpeername_notsupp,         /* sop_getpeername */
143         sosctp_getsockname,             /* sop_getsockname */
144         so_shutdown_notsupp,            /* sop_shutdown */
145         sosctp_getsockopt,              /* sop_getsockopt */
146         sosctp_setsockopt,              /* sop_setsockopt */
147         sosctp_ioctl,                   /* sop_ioctl    */
148         so_poll,                        /* sop_poll     */
149         sosctp_close,                   /* sop_close    */
150 };

152 /* All the upcalls expect the upper handle to be sonode. */
153 sock_upcalls_t sosctp_sock_upcalls = {
154         so_newconn,
155         so_connected,
156         so_disconnected,
157         so_opctl,
158         so_queue_msg,
159         so_set_prop,
160         so_txq_full,
161         NULL,                   /* su_signal_oob */
162 };

164 /* All the upcalls expect the upper handle to be sctp_sonode/sctp_soassoc. */
165 sock_upcalls_t sosctp_assoc_upcalls = {
166         sctp_assoc_newconn,
167         sctp_assoc_connected,
168         sctp_assoc_disconnected,
169         sctp_assoc_disconnecting,
170         sctp_assoc_recv,
171         sctp_assoc_properties,
172         sctp_assoc_xmitted,
173         NULL,                   /* su_recv_space */
174         NULL,                   /* su_signal_oob */
175         NULL,                   /* su_set_error */
176         NULL,                   /* su_closed */
177         sctp_get_sock_pid_mblk
178 #endif /* ! codereview */
179 };

181 /* ARGSUSED */
182 static int
183 sosctp_init(struct sonode *so, struct sonode *pso, struct cred *cr, int flags)
184 {
185         struct sctp_sonode *ss;
186         struct sctp_sonode *pss;
187         sctp_sockbuf_limits_t sbl;
188         int err;

190         ss = SOTOSSO(so);

192         if (pso != NULL) {
193                 /*
```



```
194                  * Passive open, just inherit settings from parent. We should
195                  * not end up here for SOCK_SEQPACKET type sockets, since no
196                  * new sonode is created in that case.
197                  */
198                 ASSERT(so->so_type == SOCK_STREAM);
199                 pss = SOTOSSO(pso);

201                 mutex_enter(&pso->so_lock);
202                 so->so_state |= (SS_ISBOUND | SS_ISCONNECTED |
203                     (pso->so_state & SS_ASYNC));
204                 sosctp_so_inherit(pss, ss);
205                 so->so_proto_props = pso->so_proto_props;
206                 so->so_mode = pso->so_mode;
207                 mutex_exit(&pso->so_lock);

209                 return (0);
210         }

212         if ((err = secpolicy_basic_net_access(cr)) != 0)
213                 return (err);

215         if (so->so_type == SOCK_STREAM) {
216                 so->so_proto_handle = (sock_lower_handle_t)sctp_create(so,
217                     NULL, so->so_family, so->so_type, SCTP_CAN_BLOCK,
218                     &sosctp_sock_upcalls, &sbl, cr);
219                 so->so_mode = SM_CONNREQUIRED;
220         } else {
221                 ASSERT(so->so_type == SOCK_SEQPACKET);
222                 so->so_proto_handle = (sock_lower_handle_t)sctp_create(ss,
223                     NULL, so->so_family, so->so_type, SCTP_CAN_BLOCK,
224                     &sosctp_assoc_upcalls, &sbl, cr);
225         }

227         if (so->so_proto_handle == NULL)
228                 return (ENOMEM);

230         so->so_rcvbuf = sbl.sbl_rxbuf;
231         so->so_rcvlowat = sbl.sbl_rxlowat;
232         so->so_sndbuf = sbl.sbl_txbuf;
233         so->so_sndlowat = sbl.sbl_txlowat;

235         return (0);
236 }

238 /*
239  * Accept incoming connection.
240  */
241 /*ARGSUSED*/
242 static int
243 sosctp_accept(struct sonode *so, int fflag, struct cred *cr,
244     struct sonode **nsop)
245 {
246         int error = 0;

248         if ((so->so_state & SS_ACCEPTCONN) == 0)
249                 return (EINVAL);

251         error = so_acceptq_dequeue(so, (fflag & (FNONBLOCK|FNDELAY)), nsop);

253         return (error);
254 }

256 /*
257  * Bind local endpoint.
258  */
259 /*ARGSUSED*/
```

```
260 static int
261 sosctp_bind(struct sonode *so, struct sockaddr *name, socklen_t namelen,
262     int flags, struct cred *cr)
263 {
264         int error;

266         if (!(flags & _SOBIND_LOCK_HELD)) {
267                 mutex_enter(&so->so_lock);
268                 so_lock_single(so);     /* Set SOLOCKED */
269         } else {
270                 ASSERT(MUTEX_HELD(&so->so_lock));
271         }

273         /*
274          * X/Open requires this check
275          */
276         if (so->so_state & SS_CANTSENDMORE) {
277                 error = EINVAL;
278                 goto done;
279         }


282         /*
283          * Protocol module does address family checks.
284          */
285         mutex_exit(&so->so_lock);

287         error = sctp_bind((struct sctp_s *)so->so_proto_handle, name, namelen);

289         mutex_enter(&so->so_lock);
290         if (error == 0) {
291                 so->so_state |= SS_ISBOUND;
292         } else {
293                 eprintsoline(so, error);
294         }
295 done:
296         if (!(flags & _SOBIND_LOCK_HELD)) {
297                 so_unlock_single(so, SOLOCKED);
298                 mutex_exit(&so->so_lock);
299         } else {
300                 /* If the caller held the lock don't release it here */
301                 ASSERT(MUTEX_HELD(&so->so_lock));
302                 ASSERT(so->so_flag & SOLOCKED);
303         }

305         return (error);
306 }

308 /*
309  * Turn socket into a listen socket.
310  */
311 /* ARGSUSED */
312 static int
313 sosctp_listen(struct sonode *so, int backlog, struct cred *cr)
314 {
315         int error = 0;

317         mutex_enter(&so->so_lock);
318         so_lock_single(so);

320         /*
321          * If this socket is trying to do connect, or if it has
322          * been connected, disallow.
323          */
324         if (so->so_state & (SS_ISCONNECTING | SS_ISCONNECTED |
325             SS_ISDISCONNECTING | SS_CANTRCVMORE | SS_CANTSENDMORE)) {
```



```
326                 error = EINVAL;
327                 eprintsoline(so, error);
328                 goto done;
329         }

331         if (backlog < 0) {
332                 backlog = 0;
333         }

335         /*
336          * If listen() is only called to change backlog, we don't
337          * need to notify protocol module.
338          */
339         if (so->so_state & SS_ACCEPTCONN) {
340                 so->so_backlog = backlog;
341                 goto done;
342         }

344         mutex_exit(&so->so_lock);
345         error = sctp_listen((struct sctp_s *)so->so_proto_handle);
346         mutex_enter(&so->so_lock);
347         if (error == 0) {
348                 so->so_state |= (SS_ACCEPTCONN|SS_ISBOUND);
349                 so->so_backlog = backlog;
350         } else {
351                 eprintsoline(so, error);
352         }
353 done:
354         so_unlock_single(so, SOLOCKED);
355         mutex_exit(&so->so_lock);

357         return (error);
358 }

360 /*
361  * Active open.
362  */
363 /*ARGSUSED*/
364 static int
365 sosctp_connect(struct sonode *so, struct sockaddr *name,
366     socklen_t namelen, int fflag, int flags, struct cred *cr)
367 {
368         int error = 0;
369         pid_t pid = curproc->p_pid;

371         ASSERT(so->so_type == SOCK_STREAM);

373         mutex_enter(&so->so_lock);
374         so_lock_single(so);

376         /*
377          * Can't connect() after listen(), or if the socket is already
378          * connected.
379          */
380         if (so->so_state & (SS_ACCEPTCONN|SS_ISCONNECTED|SS_ISCONNECTING)) {
381                 if (so->so_state & SS_ISCONNECTED) {
382                         error = EISCONN;
383                 } else if (so->so_state & SS_ISCONNECTING) {
384                         error = EALREADY;
385                 } else {
386                         error = EOPNOTSUPP;
387                 }
388                 eprintsoline(so, error);
389                 goto done;
390         }
```

```
392         /*
393          * Check for failure of an earlier call
394          */
395         if (so->so_error != 0) {
396                 error = sogeterr(so, B_TRUE);
397                 eprintsoline(so, error);
398                 goto done;
399         }

401         /*
402          * Connection is closing, or closed, don't allow reconnect.
403          * TCP allows this to proceed, but the socket remains unwriteable.
404          * BSD returns EINVAL.
405          */
406         if (so->so_state & (SS_ISDISCONNECTING|SS_CANTRCVMORE|
407             SS_CANTSENDMORE)) {
408                 error = EINVAL;
409                 eprintsoline(so, error);
410                 goto done;
411         }

413         if (name == NULL || namelen == 0) {
414                 mutex_exit(&so->so_lock);
415                 error = EINVAL;
416                 eprintsoline(so, error);
417                 goto done;
418         }

420         soisconnecting(so);
421         mutex_exit(&so->so_lock);

423         error = sctp_connect((struct sctp_s *)so->so_proto_handle,
424             name, namelen, cr, pid);

426         mutex_enter(&so->so_lock);
427         if (error == 0) {
428                 /*
429                  * Allow other threads to access the socket
430                  */
431                 error = sowaitconnected(so, fflag, 0);
432         }
433 done:
434         so_unlock_single(so, SOLOCKED);
435         mutex_exit(&so->so_lock);
436         return (error);
437 }

439 /*
440  * Active open for 1-N sockets, create a new association and
441  * call connect on that.
442  * If there parent hasn't been bound yet (this is the first association),
443  * make it so.
444  */
445 static int
446 sosctp_seq_connect(struct sonode *so, struct sockaddr *name,
447     socklen_t namelen, int fflag, int flags, struct cred *cr)
448 {
449         struct sctp_soassoc *ssa;
450         struct sctp_sonode *ss;
451         int error;

453         ASSERT(so->so_type == SOCK_SEQPACKET);

455         mutex_enter(&so->so_lock);
456         so_lock_single(so);
```



```
458         if (name == NULL || namelen == 0) {
459                 error = EINVAL;
460                 eprintsoline(so, error);
461                 goto done;
462         }

464         ss = SOTOSSO(so);

466         error = sosctp_assoc_createconn(ss, name, namelen, NULL, 0, fflag,
467             cr, &ssa);
468         if (error != 0) {
469                 if ((error == EHOSTUNREACH) && (flags & _SOCONNECT_XPG4_2)) {
470                         error = ENETUNREACH;
471                 }
472         }
473         if (ssa != NULL) {
474                 SSA_REFRELE(ss, ssa);
475         }

477 done:
478         so_unlock_single(so, SOLOCKED);
479         mutex_exit(&so->so_lock);
480         return (error);
481 }

483 /*
484  * Receive data.
485  */
486 /* ARGSUSED */
487 static int
488 sosctp_recvmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
489     struct cred *cr)
490 {
491         struct sctp_sonode *ss = SOTOSSO(so);
492         struct sctp_soassoc *ssa = NULL;
493         int flags, error = 0;
494         struct T_unitdata_ind *tind;
495         ssize_t orig_resid = uiop->uio_resid;
496         int len, count, readcnt = 0;
497         socklen_t controllen, namelen;
498         void *opt;
499         mblk_t *mp;
500         rval_t  rval;

502         controllen = msg->msg_controllen;
503         namelen = msg->msg_namelen;
504         flags = msg->msg_flags;
505         msg->msg_flags = 0;
506         msg->msg_controllen = 0;
507         msg->msg_namelen = 0;

509         if (so->so_type == SOCK_STREAM) {
510                 if (!(so->so_state & (SS_ISCONNECTED|SS_ISCONNECTING|
511                     SS_CANTRCVMORE))) {
512                         return (ENOTCONN);
513                 }
514         } else {
515                 /* NOTE: Will come here from vop_read() as well */
516                 /* For 1-N socket, recv() cannot be used. */
517                 if (namelen == 0)
518                         return (EOPNOTSUPP);
519                 /*
520                  * If there are no associations, and no new connections are
521                  * coming in, there's not going to be new messages coming
522                  * in either.
523                  */
```

```
524                 if (so->so_rcv_q_head == NULL && so->so_rcv_head == NULL &&
525                     ss->ss_assoccnt == 0 && !(so->so_state & SS_ACCEPTCONN)) {
526                         return (ENOTCONN);
527                 }
528         }

530         /*
531          * out-of-band data not supported.
532          */
533         if (flags & MSG_OOB) {
534                 return (EOPNOTSUPP);
535         }

537         /*
538          * flag possibilities:
539          *
540          * MSG_PEEK     Don't consume data
541          * MSG_WAITALL  Wait for full quantity of data (ignored if MSG_PEEK)
542          * MSG_DONTWAIT Non-blocking (same as FNDELAY | FNONBLOCK)
543          *
544          * MSG_WAITALL can return less than the full buffer if either
545          *
546          * 1. we would block and we are non-blocking
547          * 2. a full message cannot be delivered
548          *
549          * Given that we always get a full message from proto below,
550          * MSG_WAITALL is not meaningful.
551          */

553         mutex_enter(&so->so_lock);

555         /*
556          * Allow just one reader at a time.
557          */
558         error = so_lock_read_intr(so,
559             uiop->uio_fmode | ((flags & MSG_DONTWAIT) ? FNONBLOCK : 0));
560         if (error) {
561                 mutex_exit(&so->so_lock);
562                 return (error);
563         }
564         mutex_exit(&so->so_lock);
565 again:
566         error = so_dequeue_msg(so, &mp, uiop, &rval, flags | MSG_DUPCTRL);
567         if (mp != NULL) {
568                 if (so->so_type == SOCK_SEQPACKET) {
569                         ssa = *(struct sctp_soassoc **)DB_BASE(mp);
570                 }

572                 tind = (struct T_unitdata_ind *)mp->b_rptr;

574                 len = tind->SRC_length;

576                 if (namelen > 0 && len > 0) {

578                         opt = sogetoff(mp, tind->SRC_offset, len, 1);

580                         ASSERT(opt != NULL);

582                         msg->msg_name = kmem_alloc(len, KM_SLEEP);
583                         msg->msg_namelen = len;

585                         bcopy(opt, msg->msg_name, len);
586                 }

588                 len = tind->OPT_length;
589                 if (controllen == 0) {
```



```
590                         if (len > 0) {
591                                 msg->msg_flags |= MSG_CTRUNC;
592                         }
593                 } else if (len > 0) {
594                         opt = sogetoff(mp, tind->OPT_offset, len,
595                             __TPI_ALIGN_SIZE);

597                         ASSERT(opt != NULL);
598                         sosctp_pack_cmsg(opt, msg, len);
599                 }

601                 if (mp->b_flag & SCTP_NOTIFICATION) {
602                         msg->msg_flags |= MSG_NOTIFICATION;
603                 }

605                 if (!(mp->b_flag & SCTP_PARTIAL_DATA) &&
606                     !(rval.r_val1 & MOREDATA)) {
607                         msg->msg_flags |= MSG_EOR;
608                 }
609                 freemsg(mp);
610         }
611 done:
612         if (!(flags & MSG_PEEK))
613                 readcnt = orig_resid - uiop->uio_resid;
614         /*
615          * Determine if we need to update SCTP about the buffer
616          * space.  For performance reason, we cannot update SCTP
617          * every time a message is read.  The socket buffer low
618          * watermark is used as the threshold.
619          */
620         if (ssa == NULL) {
621                 mutex_enter(&so->so_lock);
622                 count = so->so_rcvbuf - so->so_rcv_queued;

624                 ASSERT(so->so_rcv_q_head != NULL ||
625                     so->so_rcv_head != NULL ||
626                     so->so_rcv_queued == 0);

628                 so_unlock_read(so);

630                 /*
631                  * so_dequeue_msg() sets r_val2 to true if flow control was
632                  * cleared and we need to update SCTP.  so_flowctrld was
633                  * cleared in so_dequeue_msg() via so_check_flow_control().
634                  */
635                 if (rval.r_val2) {
636                         mutex_exit(&so->so_lock);
637                         sctp_recvd((struct sctp_s *)so->so_proto_handle, count);
638                 } else {
639                         mutex_exit(&so->so_lock);
640                 }
641         } else {
642                 /*
643                  * Each association keeps track of how much data it has
644                  * queued; we need to update the value here. Note that this
645                  * is slightly different from SOCK_STREAM type sockets, which
646                  * does not need to update the byte count, as it is already
647                  * done in so_dequeue_msg().
648                  */
649                 mutex_enter(&so->so_lock);
650                 ssa->ssa_rcv_queued -= readcnt;
651                 count = so->so_rcvbuf - ssa->ssa_rcv_queued;

653                 so_unlock_read(so);

655                 if (readcnt > 0 && ssa->ssa_flowctrld &&
```

```
656                     ssa->ssa_rcv_queued < so->so_rcvlowat) {
657                         /*
658                          * Need to clear ssa_flowctrld, different from 1-1
659                          * style.
660                          */
661                         ssa->ssa_flowctrld = B_FALSE;
662                         mutex_exit(&so->so_lock);
663                         sctp_recvd(ssa->ssa_conn, count);
664                         mutex_enter(&so->so_lock);
665                 }
666
667                 /*
668                  * MOREDATA flag is set if all data could not be copied
669                  */
670                 if (!(flags & MSG_PEEK) && !(rval.r_val1 & MOREDATA)) {
671                         SSA_REFRELE(ss, ssa);
672                 }
673                 mutex_exit(&so->so_lock);
674         }
675
676         return (error);
677 }
678
679 int
680 sosctp_uiomove(mblk_t *hdr_mp, ssize_t count, ssize_t blk_size, int wroff,
681     struct uio *uiop, int flags)
682 {
683         ssize_t size;
684         int error;
685         mblk_t *mp;
686         dblk_t *dp;
687
688         if (blk_size == INFPSZ)
689                 blk_size = count;
690
691         /*
692          * Loop until we have all data copied into mblk's.
693          */
694         while (count > 0) {
695                 size = MIN(count, blk_size);
696
697                 /*
698                  * As a message can be splitted up and sent in different
699                  * packets, each mblk will have the extra space before
700                  * data to accommodate what SCTP wants to put in there.
701                  */
702                 while ((mp = allocb(size + wroff, BPRI_MED)) == NULL) {
703                         if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
704                             (flags & MSG_DONTWAIT)) {
705                                 return (EAGAIN);
706                         }
707                         if ((error = strwaitbuf(size + wroff, BPRI_MED))) {
708                                 return (error);
709                         }
710                 }
711
712                 dp = mp->b_datap;
713                 dp->db_cpid = curproc->p_pid;
714                 ASSERT(wroff <= dp->db_lim - mp->b_wptr);
715                 mp->b_rptr += wroff;
716                 error = uiomove(mp->b_rptr, size, UIO_WRITE, uiop);
717                 if (error != 0) {
718                         freeb(mp);
719                         return (error);
720                 }
721                 mp->b_wptr = mp->b_rptr + size;
```



```
722                 count -= size;
723                 hdr_mp->b_cont = mp;
724                 hdr_mp = mp;
725         }
726         return (0);
727 }
728
729 /*
730  * Send message.
731  */
732 static int
733 sosctp_sendmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
734     struct cred *cr)
735 {
736         mblk_t *mctl;
737         struct cmsghdr *cmsg;
738         struct sctp_sndrcvinfo *sinfo;
739         int optlen, flags, fflag;
740         ssize_t count, msglen;
741         int error;
742
743         ASSERT(so->so_type == SOCK_STREAM);
744
745         flags = msg->msg_flags;
746         if (flags & MSG_OOB) {
747                 /*
748                  * No out-of-band data support.
749                  */
750                 return (EOPNOTSUPP);
751         }
752
753         if (msg->msg_controllen != 0) {
754                 optlen = msg->msg_controllen;
755                 cmsg = sosctp_find_cmsg(msg->msg_control, optlen, SCTP_SNDRCV);
756                 if (cmsg != NULL) {
757                         if (cmsg->cmsg_len <
758                             (sizeof (*sinfo) + sizeof (*cmsg))) {
759                                 eprintsoline(so, EINVAL);
760                                 return (EINVAL);
761                         }
762                         sinfo = (struct sctp_sndrcvinfo *)(cmsg + 1);
763
764                         /* Both flags should not be set together. */
765                         if ((sinfo->sinfo_flags & MSG_EOF) &&
766                             (sinfo->sinfo_flags & MSG_ABORT)) {
767                                 eprintsoline(so, EINVAL);
768                                 return (EINVAL);
769                         }
770
771                         /* Initiate a graceful shutdown. */
772                         if (sinfo->sinfo_flags & MSG_EOF) {
773                                 /* Can't include data in MSG_EOF message. */
774                                 if (uiop->uio_resid != 0) {
775                                         eprintsoline(so, EINVAL);
776                                         return (EINVAL);
777                                 }
778
779                                 /*
780                                  * This is the same sequence as done in
781                                  * shutdown(SHUT_WR).
782                                  */
783                                 mutex_enter(&so->so_lock);
784                                 so_lock_single(so);
785                                 socantsendmore(so);
786                                 cv_broadcast(&so->so_snd_cv);
787                                 so->so_state |= SS_ISDISCONNECTING;
```

```
788                                 mutex_exit(&so->so_lock);

790                                 pollwakeup(&so->so_poll_list, POLLOUT);
791                                 sctp_recvd((struct sctp_s *)so->so_proto_handle,
792                                     so->so_rcvbuf);
793                                 error = sctp_disconnect(
794                                     (struct sctp_s *)so->so_proto_handle);

796                                 mutex_enter(&so->so_lock);
797                                 so_unlock_single(so, SOLOCKED);
798                                 mutex_exit(&so->so_lock);
799                                 return (error);
800                         }
801                 }
802         } else {
803                 optlen = 0;
804         }

806         mutex_enter(&so->so_lock);
807         for (;;) {
808                 if (so->so_state & SS_CANTSENDMORE) {
809                         mutex_exit(&so->so_lock);
810                         return (EPIPE);
811                 }

813                 if (so->so_error != 0) {
814                         error = sogeterr(so, B_TRUE);
815                         mutex_exit(&so->so_lock);
816                         return (error);
817                 }

819                 if (!so->so_snd_qfull)
820                         break;

822                 if (so->so_state & SS_CLOSING) {
823                         mutex_exit(&so->so_lock);
824                         return (EINTR);
825                 }
826                 /*
827                  * Xmit window full in a blocking socket.
828                  */
829                 if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
830                     (flags & MSG_DONTWAIT)) {
831                         mutex_exit(&so->so_lock);
832                         return (EAGAIN);
833                 } else {
834                         /*
835                          * Wait for space to become available and try again.
836                          */
837                         error = cv_wait_sig(&so->so_snd_cv, &so->so_lock);
838                         if (!error) { /* signal */
839                                 mutex_exit(&so->so_lock);
840                                 return (EINTR);
841                         }
842                 }
843         }
844         msglen = count = uiop->uio_resid;

846         /* Don't allow sending a message larger than the send buffer size. */
847         /* XXX Transport module need to enforce this */
848         if (msglen > so->so_sndbuf) {
849                 mutex_exit(&so->so_lock);
850                 return (EMSGSIZE);
851         }

853         /*
```



```
854          * Allow piggybacking data on handshake messages (SS_ISCONNECTING).
855          */
856         if (!(so->so_state & (SS_ISCONNECTING | SS_ISCONNECTED))) {
857                 /*
858                  * We need to check here for listener so that the
859                  * same error will be returned as with a TCP socket.
860                  * In this case, sosctp_connect() returns EOPNOTSUPP
861                  * while a TCP socket returns ENOTCONN instead.  Catch it
862                  * here to have the same behavior as a TCP socket.
863                  *
864                  * We also need to make sure that the peer address is
865                  * provided before we attempt to do the connect.
866                  */
867                 if ((so->so_state & SS_ACCEPTCONN) ||
868                     msg->msg_name == NULL) {
869                         mutex_exit(&so->so_lock);
870                         error = ENOTCONN;
871                         goto error_nofree;
872                 }
873                 mutex_exit(&so->so_lock);
874                 fflag = uiop->uio_fmode;
875                 if (flags & MSG_DONTWAIT) {
876                         fflag |= FNDELAY;
877                 }
878                 error = sosctp_connect(so, msg->msg_name, msg->msg_namelen,
879                     fflag, (so->so_version == SOV_XPG4_2) * _SOCONNECT_XPG4_2,
880                     cr);
881                 if (error) {
882                         /*
883                          * Check for non-fatal errors, socket connected
884                          * while the lock had been lifted.
885                          */
886                         if (error != EISCONN && error != EALREADY) {
887                                 goto error_nofree;
888                         }
889                         error = 0;
890                 }
891         } else {
892                 mutex_exit(&so->so_lock);
893         }

895         mctl = sctp_alloc_hdr(msg->msg_name, msg->msg_namelen,
896             msg->msg_control, optlen, SCTP_CAN_BLOCK);
897         if (mctl == NULL) {
898                 error = EINTR;
899                 goto error_nofree;
900         }

902         /* Copy in the message. */
903         if ((error = sosctp_uiomove(mctl, count, so->so_proto_props.sopp_maxblk,
904             so->so_proto_props.sopp_wroff, uiop, flags)) != 0) {
905                 goto error_ret;
906         }
907         error = sctp_sendmsg((struct sctp_s *)so->so_proto_handle, mctl, 0);
908         if (error == 0)
909                 return (0);

911 error_ret:
912         freemsg(mctl);
913 error_nofree:
914         mutex_enter(&so->so_lock);
915         if ((error == EPIPE) && (so->so_state & SS_CANTSENDMORE)) {
916                 /*
917                  * We received shutdown between the time lock was
918                  * lifted and call to sctp_sendmsg().
919                  */
```

```
920                 mutex_exit(&so->so_lock);
921                 return (EPIPE);
922         }
923         mutex_exit(&so->so_lock);
924         return (error);
925 }

927 /*
928  * Send message on 1-N socket. Connects automatically if there is
929  * no association.
930  */
931 static int
932 sosctp_seq_sendmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
933     struct cred *cr)
934 {
935         struct sctp_sonode *ss;
936         struct sctp_soassoc *ssa;
937         struct cmsghdr *cmsg;
938         struct sctp_sndrcvinfo *sinfo;
939         int aid = 0;
940         mblk_t *mctl;
941         int namelen, optlen, flags;
942         ssize_t count, msglen;
943         int error;
944         uint16_t s_flags = 0;

946         ASSERT(so->so_type == SOCK_SEQPACKET);

948         /*
949          * There shouldn't be problems with alignment, as the memory for
950          * msg_control was alloced with kmem_alloc.
951          */
952         cmsg = sosctp_find_cmsg(msg->msg_control, msg->msg_controllen,
953             SCTP_SNDRCV);
954         if (cmsg != NULL) {
955                 if (cmsg->cmsg_len < (sizeof (*sinfo) + sizeof (*cmsg))) {
956                         eprintsoline(so, EINVAL);
957                         return (EINVAL);
958                 }
959                 sinfo = (struct sctp_sndrcvinfo *)(cmsg + 1);
960                 s_flags = sinfo->sinfo_flags;
961                 aid = sinfo->sinfo_assoc_id;
962         }

964         ss = SOTOSSO(so);
965         namelen = msg->msg_namelen;

967         if (msg->msg_controllen > 0) {
968                 optlen = msg->msg_controllen;
969         } else {
970                 optlen = 0;
971         }

973         mutex_enter(&so->so_lock);

975         /*
976          * If there is no association id, connect to address specified
977          * in msg_name.  Otherwise look up the association using the id.
978          */
979         if (aid == 0) {
980                 /*
981                  * Connect and shutdown cannot be done together, so check for
982                  * MSG_EOF.
983                  */
984                 if (msg->msg_name == NULL || namelen == 0 ||
985                     (s_flags & MSG_EOF)) {
```



```
986                         error = EINVAL;
987                         eprintsoline(so, error);
988                         goto done;
989                 }
990                 flags = uiop->uio_fmode;
991                 if (msg->msg_flags & MSG_DONTWAIT) {
992                         flags |= FNDELAY;
993                 }
994                 so_lock_single(so);
995                 error = sosctp_assoc_createconn(ss, msg->msg_name, namelen,
996                     msg->msg_control, optlen, flags, cr, &ssa);
997                 if (error) {
998                         if ((so->so_version == SOV_XPG4_2) &&
999                             (error == EHOSTUNREACH)) {
1000                                error = ENETUNREACH;
1001                        }
1002                        if (ssa == NULL) {
1003                                /*
1004                                 * Fatal error during connect(). Bail out.
1005                                 * If ssa exists, it means that the handshake
1006                                 * is in progress.
1007                                 */
1008                                eprintsoline(so, error);
1009                                so_unlock_single(so, SOLOCKED);
1010                                goto done;
1011                        }
1012                        /*
1013                         * All the errors are non-fatal ones, don't return
1014                         * e.g. EINPROGRESS from sendmsg().
1015                         */
1016                        error = 0;
1017                }
1018                so_unlock_single(so, SOLOCKED);
1019        } else {
1020                if ((error = sosctp_assoc(ss, aid, &ssa)) != 0) {
1021                        eprintsoline(so, error);
1022                        goto done;
1023                }
1024        }

1026        /*
1027         * Now we have an association.
1028         */
1029        flags = msg->msg_flags;

1031        /*
1032         * MSG_EOF initiates graceful shutdown.
1033         */
1034        if (s_flags & MSG_EOF) {
1035                if (uiop->uio_resid) {
1036                        /*
1037                         * Can't include data in MSG_EOF message.
1038                         */
1039                        error = EINVAL;
1040                } else {
1041                        mutex_exit(&so->so_lock);
1042                        ssa->ssa_state |= SS_ISDISCONNECTING;
1043                        sctp_recvd(ssa->ssa_conn, so->so_rcvbuf);
1044                        error = sctp_disconnect(ssa->ssa_conn);
1045                        mutex_enter(&so->so_lock);
1046                }
1047                goto refrele;
1048        }

1050        for (;;) {
1051                if (ssa->ssa_state & SS_CANTSENDMORE) {
```

```
1052                        SSA_REFRELE(ss, ssa);
1053                        mutex_exit(&so->so_lock);
1054                        return (EPIPE);
1055                }
1056                if (ssa->ssa_error != 0) {
1057                        error = ssa->ssa_error;
1058                        ssa->ssa_error = 0;
1059                        goto refrele;
1060                }

1062                if (!ssa->ssa_snd_qfull)
1063                        break;

1065                if (so->so_state & SS_CLOSING) {
1066                        error = EINTR;
1067                        goto refrele;
1068                }
1069                if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
1070                    (flags & MSG_DONTWAIT)) {
1071                        error = EAGAIN;
1072                        goto refrele;
1073                } else {
1074                        /*
1075                         * Wait for space to become available and try again.
1076                         */
1077                        error = cv_wait_sig(&so->so_snd_cv, &so->so_lock);
1078                        if (!error) { /* signal */
1079                                error = EINTR;
1080                                goto refrele;
1081                        }
1082                }
1083        }

1085        msglen = count = uiop->uio_resid;

1087        /* Don't allow sending a message larger than the send buffer size. */
1088        if (msglen > so->so_sndbuf) {
1089                error = EMSGSIZE;
1090                goto refrele;
1091        }

1093        /*
1094         * Update TX buffer usage here so that we can lift the socket lock.
1095         */
1096        mutex_exit(&so->so_lock);

1098        mctl = sctp_alloc_hdr(msg->msg_name, namelen, msg->msg_control,
1099            optlen, SCTP_CAN_BLOCK);
1100        if (mctl == NULL) {
1101                error = EINTR;
1102                goto lock_rele;
1103        }

1105        /* Copy in the message. */
1106        if ((error = sosctp_uiomove(mctl, count, ssa->ssa_wrsize,
1107            ssa->ssa_wroff, uiop, flags)) != 0) {
1108                goto lock_rele;
1109        }
1110        error = sctp_sendmsg((struct sctp_s *)ssa->ssa_conn, mctl, 0);
1111 lock_rele:
1112        mutex_enter(&so->so_lock);
1113        if (error != 0) {
1114                freemsg(mctl);
1115                if ((error == EPIPE) && (ssa->ssa_state & SS_CANTSENDMORE)) {
1116                        /*
1117                         * We received shutdown between the time lock was
```



```
1118                         * lifted and call to sctp_sendmsg().
1119                         */
1120                        SSA_REFRELE(ss, ssa);
1121                        mutex_exit(&so->so_lock);
1122                        return (EPIPE);
1123                }
1124        }

1126 refrele:
1127        SSA_REFRELE(ss, ssa);
1128 done:
1129        mutex_exit(&so->so_lock);
1130        return (error);
1131 }

1133 /*
1134  * Get address of remote node.
1135  */
1136 /* ARGSUSED */
1137 static int
1138 sosctp_getpeername(struct sonode *so, struct sockaddr *addr, socklen_t *addrlen,
1139     boolean_t accept, struct cred *cr)
1140 {
1141        return (sctp_getpeername((struct sctp_s *)so->so_proto_handle, addr,
1142            addrlen));
1143 }

1145 /*
1146  * Get local address.
1147  */
1148 /* ARGSUSED */
1149 static int
1150 sosctp_getsockname(struct sonode *so, struct sockaddr *addr, socklen_t *addrlen,
1151     struct cred *cr)
1152 {
1153        return (sctp_getsockname((struct sctp_s *)so->so_proto_handle, addr,
1154            addrlen));
1155 }

1157 /*
1158  * Called from shutdown().
1159  */
1160 /* ARGSUSED */
1161 static int
1162 sosctp_shutdown(struct sonode *so, int how, struct cred *cr)
1163 {
1164        uint_t state_change;
1165        int wakesig = 0;
1166        int error = 0;

1168        mutex_enter(&so->so_lock);
1169        /*
1170         * Record the current state and then perform any state changes.
1171         * Then use the difference between the old and new states to
1172         * determine which needs to be done.
1173         */
1174        state_change = so->so_state;

1176        switch (how) {
1177        case SHUT_RD:
1178                socantrcvmore(so);
1179                break;
1180        case SHUT_WR:
1181                socantsendmore(so);
1182                break;
1183        case SHUT_RDWR:
```

```
1184                    socantsendmore(so);
1185                    socantrcvmore(so);
1186                    break;
1187            default:
1188                    mutex_exit(&so->so_lock);
1189                    return (EINVAL);
1190            }

1192            state_change = so->so_state & ~state_change;

1194            if (state_change & SS_CANTRCVMORE) {
1195                    if (so->so_rcv_q_head == NULL) {
1196                            cv_signal(&so->so_rcv_cv);
1197                    }
1198                    wakesig = POLLIN|POLLRDNORM;

1200                    socket_sendsig(so, SOCKETSIG_READ);
1201            }
1202            if (state_change & SS_CANTSENDMORE) {
1203                    cv_broadcast(&so->so_snd_cv);
1204                    wakesig |= POLLOUT;

1206                    so->so_state |= SS_ISDISCONNECTING;
1207            }
1208            mutex_exit(&so->so_lock);

1210            pollwakeup(&so->so_poll_list, wakesig);

1212            if (state_change & SS_CANTSENDMORE) {
1213                    sctp_recvd((struct sctp_s *)so->so_proto_handle, so->so_rcvbuf);
1214                    error = sctp_disconnect((struct sctp_s *)so->so_proto_handle);
1215            }

1217            /*
1218             * HACK: sctp_disconnect() may return EWOULDBLOCK.  But this error is
1219             * not documented in standard socket API.  Catch it here.
1220             */
1221            if (error == EWOULDBLOCK)
1222                    error = 0;
1223            return (error);
1224    }

1226    /*
1227     * Get socket options.
1228     */
1229    /*ARGSUSED5*/
1230    static int
1231    sosctp_getsockopt(struct sonode *so, int level, int option_name,
1232        void *optval, socklen_t *optlenp, int flags, struct cred *cr)
1233    {
1234            socklen_t maxlen = *optlenp;
1235            socklen_t len;
1236            socklen_t optlen;
1237            uint8_t buffer[4];
1238            void    *optbuf = &buffer;
1239            int     error = 0;

1241            if (level == SOL_SOCKET) {
1242                    switch (option_name) {
1243                    /* Not supported options */
1244                    case SO_SNDTIMEO:
1245                    case SO_RCVTIMEO:
1246                    case SO_EXCLBIND:
1247                            eprintsoline(so, ENOPROTOOPT);
1248                            return (ENOPROTOOPT);
1249                    default:
```



```
1250                            error = socket_getopt_common(so, level, option_name,
1251                                optval, optlenp, flags);
1252                            if (error >= 0)
1253                                    return (error);
1254                            /* Pass the request to the protocol */
1255                            break;
1256                    }
1257            }

1259            if (level == IPPROTO_SCTP) {
1260                    /*
1261                     * Should go through ioctl().
1262                     */
1263                    return (EINVAL);
1264            }

1266            if (maxlen > sizeof (buffer)) {
1267                    optbuf = kmem_alloc(maxlen, KM_SLEEP);
1268            }
1269            optlen = maxlen;

1271            /*
1272             * If the resulting optlen is greater than the provided maxlen, then
1273             * we sliently trucate.
1274             */
1275            error = sctp_get_opt((struct sctp_s *)so->so_proto_handle, level,
1276                option_name, optbuf, &optlen);

1278            if (error != 0) {
1279                    eprintsoline(so, error);
1280                    goto free;
1281            }
1282            len = optlen;

1284    copyout:

1286            len = MIN(len, maxlen);
1287            bcopy(optbuf, optval, len);
1288            *optlenp = optlen;
1289    free:
1290            if (optbuf != &buffer) {
1291                    kmem_free(optbuf, maxlen);
1292            }

1294            return (error);
1295    }

1297    /*
1298     * Set socket options
1299     */
1300    /* ARGSUSED */
1301    static int
1302    sosctp_setsockopt(struct sonode *so, int level, int option_name,
1303        const void *optval, t_uscalar_t optlen, struct cred *cr)
1304    {
1305            struct sctp_sonode *ss = SOTOSSO(so);
1306            struct sctp_soassoc *ssa = NULL;
1307            sctp_assoc_t id;
1308            int error, rc;
1309            void *conn = NULL;

1311            mutex_enter(&so->so_lock);

1313            /*
1314             * For some SCTP level options, one can select the association this
1315             * applies to.
```

```
1316          */
1317         if (so->so_type == SOCK_STREAM) {
1318                 conn = so->so_proto_handle;
1319         } else {
1320                 /*
1321                  * SOCK_SEQPACKET only
1322                  */
1323                 id = 0;
1324                 if (level == IPPROTO_SCTP) {
1325                         switch (option_name) {
1326                         case SCTP_RTOINFO:
1327                         case SCTP_ASSOCINFO:
1328                         case SCTP_SET_PEER_PRIMARY_ADDR:
1329                         case SCTP_PRIMARY_ADDR:
1330                         case SCTP_PEER_ADDR_PARAMS:
1331                                 /*
1332                                  * Association ID is the first element
1333                                  * params struct
1334                                  */
1335                                 if (optlen < sizeof (sctp_assoc_t)) {
1336                                         error = EINVAL;
1337                                         eprintsoline(so, error);
1338                                         goto done;
1339                                 }
1340                                 id = *(sctp_assoc_t *)optval;
1341                                 break;
1342                         case SCTP_DEFAULT_SEND_PARAM:
1343                                 if (optlen != sizeof (struct sctp_sndrcvinfo)) {
1344                                         error = EINVAL;
1345                                         eprintsoline(so, error);
1346                                         goto done;
1347                                 }
1348                                 id = ((struct sctp_sndrcvinfo *)
1349                                     optval)->sinfo_assoc_id;
1350                                 break;
1351                         case SCTP_INITMSG:
1352                                 /*
1353                                  * Only applies to future associations
1354                                  */
1355                                 conn = so->so_proto_handle;
1356                                 break;
1357                         default:
1358                                 break;
1359                         }
1360                 } else if (level == SOL_SOCKET) {
1361                         if (option_name == SO_LINGER) {
1362                                 error = EOPNOTSUPP;
1363                                 eprintsoline(so, error);
1364                                 goto done;
1365                         }
1366                         /*
1367                          * These 2 options are applied to all associations.
1368                          * The other socket level options are only applied
1369                          * to the socket (not associations).
1370                          */
1371                         if ((option_name != SO_RCVBUF) &&
1372                             (option_name != SO_SNDBUF)) {
1373                                 conn = so->so_proto_handle;
1374                         }
1375                 } else {
1376                         conn = NULL;
1377                 }

1379                 /*
1380                  * If association ID was specified, do op on that assoc.
1381                  * Otherwise set the default setting of a socket.
```



```
1382                  */
1383                 if (id != 0) {
1384                         if ((error = sosctp_assoc(ss, id, &ssa)) != 0) {
1385                                 eprintsoline(so, error);
1386                                 goto done;
1387                         }
1388                         conn = ssa->ssa_conn;
1389                 }
1390         }
1391         dprint(2, ("sosctp_setsockopt %p (%d) - conn %p %d %d id:%d\n",
1392             (void *)ss, so->so_type, (void *)conn, level, option_name, id));

1394         ASSERT(ssa == NULL || (ssa != NULL && conn != NULL));
1395         if (conn != NULL) {
1396                 mutex_exit(&so->so_lock);
1397                 error = sctp_set_opt((struct sctp_s *)conn, level, option_name,
1398                     optval, optlen);
1399                 mutex_enter(&so->so_lock);
1400                 if (ssa != NULL)
1401                         SSA_REFRELE(ss, ssa);
1402         } else {
1403                 /*
1404                  * 1-N socket, and we have to apply the operation to ALL
1405                  * associations. Like with anything of this sort, the
1406                  * problem is what to do if the operation fails.
1407                  * Just try to apply the setting to everyone, but store
1408                  * error number if someone returns such.  And since we are
1409                  * looping through all possible aids, some of them can be
1410                  * invalid.  We just ignore this kind (sosctp_assoc()) of
1411                  * errors.
1412                  */
1413                 sctp_assoc_t aid;

1415                 mutex_exit(&so->so_lock);
1416                 error = sctp_set_opt((struct sctp_s *)so->so_proto_handle,
1417                     level, option_name, optval, optlen);
1418                 mutex_enter(&so->so_lock);
1419                 for (aid = 1; aid < ss->ss_maxassoc; aid++) {
1420                         if (sosctp_assoc(ss, aid, &ssa) != 0)
1421                                 continue;
1422                         mutex_exit(&so->so_lock);
1423                         rc = sctp_set_opt((struct sctp_s *)ssa->ssa_conn, level,
1424                             option_name, optval, optlen);
1425                         mutex_enter(&so->so_lock);
1426                         SSA_REFRELE(ss, ssa);
1427                         if (error == 0) {
1428                                 error = rc;
1429                         }
1430                 }
1431         }
1432 done:
1433         mutex_exit(&so->so_lock);
1434         return (error);
1435 }

1437 /*ARGSUSED*/
1438 static int
1439 sosctp_ioctl(struct sonode *so, int cmd, intptr_t arg, int mode,
1440     struct cred *cr, int32_t *rvalp)
1441 {
1442         struct sctp_sonode      *ss;
1443         int32_t                 value;
1444         int                     error;
1445         int                     intval;
1446         pid_t                   pid;
1447         struct sctp_soassoc     *ssa;
```

```
1448        void                    *conn;
1449        void                    *buf;
1450        STRUCT_DECL(sctpopt, opt);
1451        uint32_t                optlen;
1452        int                     buflen;

1454        ss = SOTOSSO(so);

1456        /* handle socket specific ioctls */
1457        switch (cmd) {
1458        case FIONBIO:
1459                if (so_copyin((void *)arg, &value, sizeof (int32_t),
1460                    (mode & (int)FKIOCTL))) {
1461                        return (EFAULT);
1462                }
1463                mutex_enter(&so->so_lock);
1464                if (value) {
1465                        so->so_state |= SS_NDELAY;
1466                } else {
1467                        so->so_state &= ~SS_NDELAY;
1468                }
1469                mutex_exit(&so->so_lock);
1470                return (0);

1472        case FIOASYNC:
1473                if (so_copyin((void *)arg, &value, sizeof (int32_t),
1474                    (mode & (int)FKIOCTL))) {
1475                        return (EFAULT);
1476                }
1477                mutex_enter(&so->so_lock);

1479                if (value) {
1480                        /* Turn on SIGIO */
1481                        so->so_state |= SS_ASYNC;
1482                } else {
1483                        /* Turn off SIGIO */
1484                        so->so_state &= ~SS_ASYNC;
1485                }
1486                mutex_exit(&so->so_lock);
1487                return (0);

1489        case SIOCSPGRP:
1490        case FIOSETOWN:
1491                if (so_copyin((void *)arg, &pid, sizeof (pid_t),
1492                    (mode & (int)FKIOCTL))) {
1493                        return (EFAULT);
1494                }
1495                mutex_enter(&so->so_lock);

1497                error = (pid != so->so_pgrp) ? socket_chgpgrp(so, pid) : 0;
1498                mutex_exit(&so->so_lock);
1499                return (error);

1501        case SIOCGPGRP:
1502        case FIOGETOWN:
1503                if (so_copyout(&so->so_pgrp, (void *)arg,
1504                    sizeof (pid_t), (mode & (int)FKIOCTL)))
1505                        return (EFAULT);
1506                return (0);

1508        case FIONREAD:
1509                /* XXX: Cannot be used unless standard buffer is used */
1510                /*
1511                 * Return number of bytes of data in all data messages
1512                 * in queue in "arg".
1513                 * For stream socket, amount of available data.
```



```
1514                 * For sock_dgram, # of available bytes + addresses.
1515                 */
1516                intval = (so->so_state & SS_ACCEPTCONN) ? 0 :
1517                    MIN(so->so_rcv_queued, INT_MAX);
1518                if (so_copyout(&intval, (void *)arg, sizeof (intval),
1519                    (mode & (int)FKIOCTL)))
1520                        return (EFAULT);
1521                return (0);
1522        case SIOCATMARK:
1523                /*
1524                 * No support for urgent data.
1525                 */
1526                intval = 0;

1528                if (so_copyout(&intval, (void *)arg, sizeof (int),
1529                    (mode & (int)FKIOCTL)))
1530                        return (EFAULT);
1531                return (0);
1532        case _I_GETPEERCRED: {
1533                int error = 0;

1535                if ((mode & FKIOCTL) == 0)
1536                        return (EINVAL);

1538                mutex_enter(&so->so_lock);
1539                if ((so->so_mode & SM_CONNREQUIRED) == 0) {
1540                        error = ENOTSUP;
1541                } else if ((so->so_state & SS_ISCONNECTED) == 0) {
1542                        error = ENOTCONN;
1543                } else if (so->so_peercred != NULL) {
1544                        k_peercred_t *kp = (k_peercred_t *)arg;
1545                        kp->pc_cr = so->so_peercred;
1546                        kp->pc_cpid = so->so_cpid;
1547                        crhold(so->so_peercred);
1548                } else {
1549                        error = EINVAL;
1550                }
1551                mutex_exit(&so->so_lock);
1552                return (error);
1553        }
1554        case SIOCSCTPGOPT:
1555                STRUCT_INIT(opt, mode);

1557                if (so_copyin((void *)arg, STRUCT_BUF(opt), STRUCT_SIZE(opt),
1558                    (mode & (int)FKIOCTL))) {
1559                        return (EFAULT);
1560                }
1561                if ((optlen = STRUCT_FGET(opt, sopt_len)) > SO_MAXARGSIZE)
1562                        return (EINVAL);

1564                /*
1565                 * Find the correct sctp_t based on whether it is 1-N socket
1566                 * or not.
1567                 */
1568                intval = STRUCT_FGET(opt, sopt_aid);
1569                mutex_enter(&so->so_lock);
1570                if ((so->so_type == SOCK_SEQPACKET) && intval) {
1571                        if ((error = sosctp_assoc(ss, intval, &ssa)) != 0) {
1572                                mutex_exit(&so->so_lock);
1573                                return (error);
1574                        }
1575                        conn = ssa->ssa_conn;
1576                        ASSERT(conn != NULL);
1577                } else {
1578                        conn = so->so_proto_handle;
1579                        ssa = NULL;
```

```
1580                }
1581                mutex_exit(&so->so_lock);

1583                /* Copyin the option buffer and then call sctp_get_opt(). */
1584                buflen = optlen;
1585                /* Let's allocate a buffer enough to hold an int */
1586                if (buflen < sizeof (uint32_t))
1587                        buflen = sizeof (uint32_t);
1588                buf = kmem_alloc(buflen, KM_SLEEP);
1589                if (so_copyin(STRUCT_FGETP(opt, sopt_val), buf, optlen,
1590                    (mode & (int)FKIOCTL))) {
1591                        if (ssa != NULL) {
1592                                mutex_enter(&so->so_lock);
1593                                SSA_REFRELE(ss, ssa);
1594                                mutex_exit(&so->so_lock);
1595                        }
1596                        kmem_free(buf, buflen);
1597                        return (EFAULT);
1598                }
1599                /* The option level has to be IPPROTO_SCTP */
1600                error = sctp_get_opt((struct sctp_s *)conn, IPPROTO_SCTP,
1601                    STRUCT_FGET(opt, sopt_name), buf, &optlen);
1602                if (ssa != NULL) {
1603                        mutex_enter(&so->so_lock);
1604                        SSA_REFRELE(ss, ssa);
1605                        mutex_exit(&so->so_lock);
1606                }
1607                optlen = MIN(buflen, optlen);
1608                /* No error, copyout the result with the correct buf len. */
1609                if (error == 0) {
1610                        STRUCT_FSET(opt, sopt_len, optlen);
1611                        if (so_copyout(STRUCT_BUF(opt), (void *)arg,
1612                            STRUCT_SIZE(opt), (mode & (int)FKIOCTL))) {
1613                                error = EFAULT;
1614                        } else if (so_copyout(buf, STRUCT_FGETP(opt, sopt_val),
1615                            optlen, (mode & (int)FKIOCTL))) {
1616                                error = EFAULT;
1617                        }
1618                }
1619                kmem_free(buf, buflen);
1620                return (error);

1622        case SIOCSCTPSOPT:
1623                STRUCT_INIT(opt, mode);

1625                if (so_copyin((void *)arg, STRUCT_BUF(opt), STRUCT_SIZE(opt),
1626                    (mode & (int)FKIOCTL))) {
1627                        return (EFAULT);
1628                }
1629                if ((optlen = STRUCT_FGET(opt, sopt_len)) > SO_MAXARGSIZE)
1630                        return (EINVAL);

1632                /*
1633                 * Find the correct sctp_t based on whether it is 1-N socket
1634                 * or not.
1635                 */
1636                intval = STRUCT_FGET(opt, sopt_aid);
1637                mutex_enter(&so->so_lock);
1638                if (intval != 0) {
1639                        if ((error = sosctp_assoc(ss, intval, &ssa)) != 0) {
1640                                mutex_exit(&so->so_lock);
1641                                return (error);
1642                        }
1643                        conn = ssa->ssa_conn;
1644                        ASSERT(conn != NULL);
1645                } else {
```



```
1646                        conn = so->so_proto_handle;
1647                        ssa = NULL;
1648                }
1649                mutex_exit(&so->so_lock);

1651                /* Copyin the option buffer and then call sctp_set_opt(). */
1652                buf = kmem_alloc(optlen, KM_SLEEP);
1653                if (so_copyin(STRUCT_FGETP(opt, sopt_val), buf, optlen,
1654                    (mode & (int)FKIOCTL))) {
1655                        if (ssa != NULL) {
1656                                mutex_enter(&so->so_lock);
1657                                SSA_REFRELE(ss, ssa);
1658                                mutex_exit(&so->so_lock);
1659                        }
1660                        kmem_free(buf, intval);
1661                        return (EFAULT);
1662                }
1663                /* The option level has to be IPPROTO_SCTP */
1664                error = sctp_set_opt((struct sctp_s *)conn, IPPROTO_SCTP,
1665                    STRUCT_FGET(opt, sopt_name), buf, optlen);
1666                if (ssa) {
1667                        mutex_enter(&so->so_lock);
1668                        SSA_REFRELE(ss, ssa);
1669                        mutex_exit(&so->so_lock);
1670                }
1671                kmem_free(buf, optlen);
1672                return (error);

1674        case SIOCSCTPPEELOFF: {
1675                struct sonode *nso;
1676                struct sctp_uc_swap us;
1677                int nfd;
1678                struct file *nfp;
1679                struct vnode *nvp = NULL;
1680                struct sockparams *sp;

1682                dprint(2, ("sctppeeloff %p\n", (void *)ss));

1684                if (so->so_type != SOCK_SEQPACKET) {
1685                        return (EOPNOTSUPP);
1686                }
1687                if (so_copyin((void *)arg, &intval, sizeof (intval),
1688                    (mode & (int)FKIOCTL))) {
1689                        return (EFAULT);
1690                }
1691                if (intval == 0) {
1692                        return (EINVAL);
1693                }

1695                /*
1696                 * Find sockparams. This is different from parent's entry,
1697                 * as the socket type is different.
1698                 */
1699                error = solookup(so->so_family, SOCK_STREAM, so->so_protocol,
1700                    &sp);
1701                if (error != 0)
1702                        return (error);

1704                /*
1705                 * Allocate the user fd.
1706                 */
1707                if ((nfd = ufalloc(0)) == -1) {
1708                        eprintsoline(so, EMFILE);
1709                        SOCKPARAMS_DEC_REF(sp);
1710                        return (EMFILE);
1711                }
```

```
1713                /*
1714                 * Copy the fd out.
1715                 */
1716                if (so_copyout(&nfd, (void *)arg, sizeof (nfd),
1717                    (mode & (int)FKIOCTL))) {
1718                        error = EFAULT;
1719                        goto err;
1720                }
1721                mutex_enter(&so->so_lock);

1723                /*
1724                 * Don't use sosctp_assoc() in order to peel off disconnected
1725                 * associations.
1726                 */
1727                ssa = ((uint32_t)intval >= ss->ss_maxassoc) ? NULL :
1728                    ss->ss_assocs[intval].ssi_assoc;
1729                if (ssa == NULL) {
1730                        mutex_exit(&so->so_lock);
1731                        error = EINVAL;
1732                        goto err;
1733                }
1734                SSA_REFHOLD(ssa);

1736                nso = socksctp_create(sp, so->so_family, SOCK_STREAM,
1737                    so->so_protocol, so->so_version, SOCKET_NOSLEEP,
1738                    &error, cr);
1739                if (nso == NULL) {
1740                        SSA_REFRELE(ss, ssa);
1741                        mutex_exit(&so->so_lock);
1742                        goto err;
1743                }
1744                nvp = SOTOV(nso);
1745                so_lock_single(so);
1746                mutex_exit(&so->so_lock);

1748                /* cannot fail, only inheriting properties */
1749                (void) sosctp_init(nso, so, CRED(), 0);

1751                /*
1752                 * We have a single ref on the new socket. This is normally
1753                 * handled by socket_{create,newconn}, but since they are not
1754                 * used we have to do it here.
1755                 */
1756                nso->so_count = 1;

1758                us.sus_handle = nso;
1759                us.sus_upcalls = &sosctp_sock_upcalls;

1761                /*
1762                 * Upcalls to new socket are blocked for the duration of
1763                 * downcall.
1764                 */
1765                mutex_enter(&nso->so_lock);

1767                error = sctp_set_opt((struct sctp_s *)ssa->ssa_conn,
1768                    IPPROTO_SCTP, SCTP_UC_SWAP, &us, sizeof (us));
1769                if (error) {
1770                        goto peelerr;
1771                }
1772                error = falloc(nvp, FWRITE|FREAD, &nfp, NULL);
1773                if (error) {
1774                        goto peelerr;
1775                }

1777                /*
```



```
1778                 * fill in the entries that falloc reserved
1779                 */
1780                nfp->f_vnode = nvp;
1781                mutex_exit(&nfp->f_tlock);
1782                setf(nfd, nfp);

1784                /* Add pid to the list associated with that socket. */
1785                if (nfp->f_vnode != NULL) {
1786                        (void) VOP_IOCTL(nfp->f_vnode, F_ASSOCI_PID,
1787                            (intptr_t)curproc->p_pidp->pid_id, FKIOCTL, kcred,
1788                            NULL, NULL);
1789                }

1791 #endif /* ! codereview */
1792                mutex_enter(&so->so_lock);

1794                sosctp_assoc_move(ss, SOTOSSO(nso), ssa);

1796                mutex_exit(&nso->so_lock);

1798                ssa->ssa_conn = NULL;
1799                sosctp_assoc_free(ss, ssa);

1801                so_unlock_single(so, SOLOCKED);
1802                mutex_exit(&so->so_lock);

1804                return (0);

1806 err:
1807                SOCKPARAMS_DEC_REF(sp);
1808                setf(nfd, NULL);
1809                eprintsoline(so, error);
1810                return (error);

1812 peelerr:
1813                mutex_exit(&nso->so_lock);
1814                mutex_enter(&so->so_lock);
1815                ASSERT(nso->so_count == 1);
1816                nso->so_count = 0;
1817                so_unlock_single(so, SOLOCKED);
1818                SSA_REFRELE(ss, ssa);
1819                mutex_exit(&so->so_lock);

1821                setf(nfd, NULL);
1822                ASSERT(nvp->v_count == 1);
1823                socket_destroy(nso);
1824                eprintsoline(so, error);
1825                return (error);
1826        }
1827        default:
1828                return (EINVAL);
1829        }
1830 }

1832 /*ARGSUSED*/
1833 static int
1834 sosctp_close(struct sonode *so, int flag, struct cred *cr)
1835 {
1836        struct sctp_sonode *ss;
1837        struct sctp_sa_id *ssi;
1838        struct sctp_soassoc *ssa;
1839        int32_t i;

1841        ss = SOTOSSO(so);

1843        /*
```

```
1844          * Initiate connection shutdown.  Tell SCTP if there is any data
1845          * left unread.
1846          */
1847         sctp_recvd((struct sctp_s *)so->so_proto_handle,
1848             so->so_rcvbuf - so->so_rcv_queued);
1849         (void) sctp_disconnect((struct sctp_s *)so->so_proto_handle);

1851         /*
1852          * New associations can't come in, but old ones might get
1853          * closed in upcall. Protect against that by taking a reference
1854          * on the association.
1855          */
1856         mutex_enter(&so->so_lock);
1857         ssi = ss->ss_assocs;
1858         for (i = 0; i < ss->ss_maxassoc; i++, ssi++) {
1859                 if ((ssa = ssi->ssi_assoc) != NULL) {
1860                         SSA_REFHOLD(ssa);
1861                         sosctp_assoc_isdisconnected(ssa, 0);
1862                         mutex_exit(&so->so_lock);

1864                         sctp_recvd(ssa->ssa_conn, so->so_rcvbuf -
1865                             ssa->ssa_rcv_queued);
1866                         (void) sctp_disconnect(ssa->ssa_conn);

1868                         mutex_enter(&so->so_lock);
1869                         SSA_REFRELE(ss, ssa);
1870                 }
1871         }
1872         mutex_exit(&so->so_lock);

1874         return (0);
1875 }

1877 /*
1878  * Closes incoming connections which were never accepted, frees
1879  * resources.
1880  */
1881 /* ARGSUSED */
1882 void
1883 sosctp_fini(struct sonode *so, struct cred *cr)
1884 {
1885         struct sctp_sonode *ss;
1886         struct sctp_sa_id *ssi;
1887         struct sctp_soassoc *ssa;
1888         int32_t i;

1890         ss = SOTOSSO(so);

1892         ASSERT(so->so_ops == &sosctp_sonodeops ||
1893             so->so_ops == &sosctp_seq_sonodeops);

1895         /* We are the sole owner of so now */
1896         mutex_enter(&so->so_lock);

1898         /* Free all pending connections */
1899         so_acceptq_flush(so, B_TRUE);

1901         ssi = ss->ss_assocs;
1902         for (i = 0; i < ss->ss_maxassoc; i++, ssi++) {
1903                 if ((ssa = ssi->ssi_assoc) != NULL) {
1904                         SSA_REFHOLD(ssa);
1905                         mutex_exit(&so->so_lock);

1907                         sctp_close((struct sctp_s *)ssa->ssa_conn);

1909                         mutex_enter(&so->so_lock);
```



```
1910                         ssa->ssa_conn = NULL;
1911                         sosctp_assoc_free(ss, ssa);
1912                 }
1913         }
1914         if (ss->ss_assocs != NULL) {
1915                 ASSERT(ss->ss_assoccnt == 0);
1916                 kmem_free(ss->ss_assocs,
1917                     ss->ss_maxassoc * sizeof (struct sctp_sa_id));
1918         }
1919         mutex_exit(&so->so_lock);

1921         if (so->so_proto_handle)
1922                 sctp_close((struct sctp_s *)so->so_proto_handle);
1923         so->so_proto_handle = NULL;

1925         /*
1926          * Note until sctp_close() is called, SCTP can still send up
1927          * messages, such as event notifications.  So we should flush
1928          * the recevie buffer after calling sctp_close().
1929          */
1930         mutex_enter(&so->so_lock);
1931         so_rcv_flush(so);
1932         mutex_exit(&so->so_lock);

1934         sonode_fini(so);
1935 }

1937 /*
1938  * Upcalls from SCTP
1939  */

1941 /*
1942  * This is the upcall function for 1-N (SOCK_SEQPACKET) socket when a new
1943  * association is created.  Note that the first argument (handle) is of type
1944  * sctp_sonode *, which is the one changed to a listener for new
1945  * associations.  All the other upcalls for 1-N socket take sctp_soassoc *
1946  * as handle.  The only exception is the su_properties upcall, which
1947  * can take both types as handle.
1948  */
1949 /* ARGSUSED */
1950 sock_upper_handle_t
1951 sctp_assoc_newconn(sock_upper_handle_t parenthandle,
1952     sock_lower_handle_t connind, sock_downcalls_t *dc,
1953     struct cred *peer_cred, pid_t peer_cpid, sock_upcalls_t **ucp)
1954 {
1955         struct sctp_sonode *lss = (struct sctp_sonode *)parenthandle;
1956         struct sonode *lso = &lss->ss_so;
1957         struct sctp_soassoc *ssa;
1958         sctp_assoc_t id;

1960         ASSERT(lss->ss_type == SOSCTP_SOCKET);
1961         ASSERT(lso->so_state & SS_ACCEPTCONN);
1962         ASSERT(lso->so_proto_handle != NULL); /* closed conn */
1963         ASSERT(lso->so_type == SOCK_SEQPACKET);

1965         mutex_enter(&lso->so_lock);

1967         if ((id = sosctp_aid_get(lss)) == -1) {
1968                 /*
1969                  * Array not large enough; increase size.
1970                  */
1971                 if (sosctp_aid_grow(lss, lss->ss_maxassoc, KM_NOSLEEP) < 0) {
1972                         mutex_exit(&lso->so_lock);
1973                         return (NULL);
1974                 }
1975                 id = sosctp_aid_get(lss);
```

```
1976                    ASSERT(id != -1);
1977            }

1979            /*
1980             * Create soassoc for this connection
1981             */
1982            ssa = sosctp_assoc_create(lss, KM_NOSLEEP);
1983            if (ssa == NULL) {
1984                    mutex_exit(&lso->so_lock);
1985                    return (NULL);
1986            }
1987            sosctp_aid_reserve(lss, id, 1);
1988            lss->ss_assocs[id].ssi_assoc = ssa;
1989            ++lss->ss_assoccnt;
1990            ssa->ssa_id = id;
1991            ssa->ssa_conn = (struct sctp_s *)connind;
1992            ssa->ssa_state = (SS_ISBOUND | SS_ISCONNECTED);
1993            ssa->ssa_wroff = lss->ss_wroff;
1994            ssa->ssa_wrsize = lss->ss_wrsize;

1996            mutex_exit(&lso->so_lock);

1998            *ucp = &sosctp_assoc_upcalls;

2000            return ((sock_upper_handle_t)ssa);
2001    }

2003    /* ARGSUSED */
2004    static void
2005    sctp_assoc_connected(sock_upper_handle_t handle, sock_connid_t id,
2006        struct cred *peer_cred, pid_t peer_cpid)
2007    {
2008            struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2009            struct sonode *so = &ssa->ssa_sonode->ss_so;

2011            ASSERT(so->so_type == SOCK_SEQPACKET);
2012            ASSERT(ssa->ssa_conn);

2014            mutex_enter(&so->so_lock);
2015            sosctp_assoc_isconnected(ssa);
2016            mutex_exit(&so->so_lock);
2017    }

2019    /* ARGSUSED */
2020    static int
2021    sctp_assoc_disconnected(sock_upper_handle_t handle, sock_connid_t id, int error)
2022    {
2023            struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2024            struct sonode *so = &ssa->ssa_sonode->ss_so;
2025            int ret;

2027            ASSERT(so->so_type == SOCK_SEQPACKET);
2028            ASSERT(ssa->ssa_conn != NULL);

2030            mutex_enter(&so->so_lock);
2031            sosctp_assoc_isdisconnected(ssa, error);
2032            if (ssa->ssa_refcnt == 1) {
2033                    ret = 1;
2034                    ssa->ssa_conn = NULL;
2035            } else {
2036                    ret = 0;
2037            }
2038            SSA_REFRELE(SOTOSSO(so), ssa);

2040            cv_broadcast(&so->so_snd_cv);
```



```
2042            mutex_exit(&so->so_lock);

2044            return (ret);
2045    }

2047    /* ARGSUSED */
2048    static void
2049    sctp_assoc_disconnecting(sock_upper_handle_t handle, sock_opctl_action_t action,
2050        uintptr_t arg)
2051    {
2052            struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2053            struct sonode *so = &ssa->ssa_sonode->ss_so;

2055            ASSERT(so->so_type == SOCK_SEQPACKET);
2056            ASSERT(ssa->ssa_conn != NULL);
2057            ASSERT(action == SOCK_OPCTL_SHUT_SEND);

2059            mutex_enter(&so->so_lock);
2060            sosctp_assoc_isdisconnecting(ssa);
2061            mutex_exit(&so->so_lock);
2062    }

2064    /* ARGSUSED */
2065    static ssize_t
2066    sctp_assoc_recv(sock_upper_handle_t handle, mblk_t *mp, size_t len, int flags,
2067        int *errorp, boolean_t *forcepush)
2068    {
2069            struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2070            struct sctp_sonode *ss = ssa->ssa_sonode;
2071            struct sonode *so = &ss->ss_so;
2072            struct T_unitdata_ind *tind;
2073            mblk_t *mp2;
2074            union sctp_notification *sn;
2075            struct sctp_sndrcvinfo *sinfo;
2076            ssize_t space_available;

2078            ASSERT(ssa->ssa_type == SOSCTP_ASSOC);
2079            ASSERT(so->so_type == SOCK_SEQPACKET);
2080            ASSERT(ssa->ssa_conn != NULL); /* closed conn */
2081            ASSERT(mp != NULL);

2083            ASSERT(errorp != NULL);
2084            *errorp = 0;

2086            /*
2087             * Should be getting T_unitdata_req's only.
2088             * Must have address as part of packet.
2089             */
2090            tind = (struct T_unitdata_ind *)mp->b_rptr;
2091            ASSERT((DB_TYPE(mp) == M_PROTO) &&
2092                (tind->PRIM_type == T_UNITDATA_IND));
2093            ASSERT(tind->SRC_length);

2095            mutex_enter(&so->so_lock);

2097            /*
2098             * For notify messages, need to fill in association id.
2099             * For data messages, sndrcvinfo could be in ancillary data.
2100             */
2101            if (mp->b_flag & SCTP_NOTIFICATION) {
2102                    mp2 = mp->b_cont;
2103                    sn = (union sctp_notification *)mp2->b_rptr;
2104                    switch (sn->sn_header.sn_type) {
2105                    case SCTP_ASSOC_CHANGE:
2106                            sn->sn_assoc_change.sac_assoc_id = ssa->ssa_id;
2107                            break;
```

```
2108                case SCTP_PEER_ADDR_CHANGE:
2109                        sn->sn_paddr_change.spc_assoc_id = ssa->ssa_id;
2110                        break;
2111                case SCTP_REMOTE_ERROR:
2112                        sn->sn_remote_error.sre_assoc_id = ssa->ssa_id;
2113                        break;
2114                case SCTP_SEND_FAILED:
2115                        sn->sn_send_failed.ssf_assoc_id = ssa->ssa_id;
2116                        break;
2117                case SCTP_SHUTDOWN_EVENT:
2118                        sn->sn_shutdown_event.sse_assoc_id = ssa->ssa_id;
2119                        break;
2120                case SCTP_ADAPTATION_INDICATION:
2121                        sn->sn_adaptation_event.sai_assoc_id = ssa->ssa_id;
2122                        break;
2123                case SCTP_PARTIAL_DELIVERY_EVENT:
2124                        sn->sn_pdapi_event.pdapi_assoc_id = ssa->ssa_id;
2125                        break;
2126                default:
2127                        ASSERT(0);
2128                        break;
2129                }
2130        } else {
2131                if (tind->OPT_length > 0) {
2132                        struct cmsghdr  *cmsg;
2133                        char            *cend;
2134
2135                        cmsg = (struct cmsghdr *)
2136                            ((uchar_t *)mp->b_rptr + tind->OPT_offset);
2137                        cend = (char *)cmsg + tind->OPT_length;
2138                        for (;;) {
2139                                if ((char *)(cmsg + 1) > cend ||
2140                                    ((char *)cmsg + cmsg->cmsg_len) > cend) {
2141                                        break;
2142                                }
2143                                if ((cmsg->cmsg_level == IPPROTO_SCTP) &&
2144                                    (cmsg->cmsg_type == SCTP_SNDRCV)) {
2145                                        sinfo = (struct sctp_sndrcvinfo *)
2146                                            (cmsg + 1);
2147                                        sinfo->sinfo_assoc_id = ssa->ssa_id;
2148                                        break;
2149                                }
2150                                if (cmsg->cmsg_len > 0) {
2151                                        cmsg = (struct cmsghdr *)
2152                                            ((uchar_t *)cmsg + cmsg->cmsg_len);
2153                                } else {
2154                                        break;
2155                                }
2156                        }
2157                }
2158        }
2159
2160        /*
2161         * SCTP has reserved space in the header for storing a pointer.
2162         * Put the pointer to assocation there, and queue the data.
2163         */
2164        SSA_REFHOLD(ssa);
2165        ASSERT((mp->b_rptr - DB_BASE(mp)) >= sizeof (ssa));
2166        *(struct sctp_soassoc **)DB_BASE(mp) = ssa;
2167
2168        ssa->ssa_rcv_queued += len;
2169        space_available = so->so_rcvbuf - ssa->ssa_rcv_queued;
2170        if (space_available <= 0)
2171                ssa->ssa_flowctrld = B_TRUE;
2172
2173        so_enqueue_msg(so, mp, len);
```



```
2175        /* so_notify_data drops so_lock */
2176        so_notify_data(so, len);
2177
2178        return (space_available);
2179 }
2180
2181 static void
2182 sctp_assoc_xmitted(sock_upper_handle_t handle, boolean_t qfull)
2183 {
2184        struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2185        struct sctp_sonode *ss = ssa->ssa_sonode;
2186
2187        ASSERT(ssa->ssa_type == SOSCTP_ASSOC);
2188        ASSERT(ss->ss_so.so_type == SOCK_SEQPACKET);
2189        ASSERT(ssa->ssa_conn != NULL);
2190
2191        mutex_enter(&ss->ss_so.so_lock);
2192
2193        ssa->ssa_snd_qfull = qfull;
2194
2195        /*
2196         * Wake blocked writers.
2197         */
2198        cv_broadcast(&ss->ss_so.so_snd_cv);
2199
2200        mutex_exit(&ss->ss_so.so_lock);
2201 }
2202
2203 static void
2204 sctp_assoc_properties(sock_upper_handle_t handle,
2205     struct sock_proto_props *soppp)
2206 {
2207        struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2208        struct sonode *so;
2209
2210        if (ssa->ssa_type == SOSCTP_ASSOC) {
2211                so = &ssa->ssa_sonode->ss_so;
2212
2213                mutex_enter(&so->so_lock);
2214
2215                /* Per assoc_id properties. */
2216                if (soppp->sopp_flags & SOCKOPT_WROFF)
2217                        ssa->ssa_wroff = soppp->sopp_wroff;
2218                if (soppp->sopp_flags & SOCKOPT_MAXBLK)
2219                        ssa->ssa_wrsize = soppp->sopp_maxblk;
2220        } else {
2221                so = &((struct sctp_sonode *)handle)->ss_so;
2222                mutex_enter(&so->so_lock);
2223
2224                if (soppp->sopp_flags & SOCKOPT_WROFF)
2225                        so->so_proto_props.sopp_wroff = soppp->sopp_wroff;
2226                if (soppp->sopp_flags & SOCKOPT_MAXBLK)
2227                        so->so_proto_props.sopp_maxblk = soppp->sopp_maxblk;
2228                if (soppp->sopp_flags & SOCKOPT_RCVHIWAT) {
2229                        ssize_t lowat;
2230
2231                        so->so_rcvbuf = soppp->sopp_rxhiwat;
2232                        /*
2233                         * The low water mark should be adjusted properly
2234                         * if the high water mark is changed.  It should
2235                         * not be bigger than 1/4 of high water mark.
2236                         */
2237                        lowat = soppp->sopp_rxhiwat >> 2;
2238                        if (so->so_rcvlowat > lowat) {
2239                                /* Sanity check... */
```

```
2240                                if (lowat == 0)
2241                                        so->so_rcvlowat = soppp->sopp_rxhiwat;
2242                                else
2243                                        so->so_rcvlowat = lowat;
2244                        }
2245                }
2246        }
2247        mutex_exit(&so->so_lock);
2248 }

2250 static mblk_t *
2251 sctp_get_sock_pid_mblk(sock_upper_handle_t handle)
2252 {
2253        struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2254        struct sonode *so;

2256        if (ssa->ssa_type == SOSCTP_ASSOC)
2257                so = &ssa->ssa_sonode->ss_so;
2258        else
2259                so = &((struct sctp_sonode *)handle)->ss_so;

2261        return (so_get_sock_pid_mblk((sock_upper_handle_t)so));
2262 #endif /* ! codereview */
2263 }
```

```
**********************************************************
   35172 Mon Aug 17 21:08:06 2015
new/usr/src/uts/common/inet/tcp/tcp_stats.c
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011, Joyent Inc. All rights reserved.
 */

#include <sys/types.h>
#include <sys/tihdr.h>
#include <sys/policy.h>
#include <sys/tsol/tnet.h>
#include <sys/kstat.h>

#include <sys/strsun.h>
#include <sys/stropts.h>
#include <sys/strsubr.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/uio.h>

#endif /* ! codereview */
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>
#include <inet/tcp_stats.h>
#include <inet/kstatcom.h>
#include <inet/snmpcom.h>

static int      tcp_kstat_update(kstat_t *, int);
static int      tcp_kstat2_update(kstat_t *, int);
static void     tcp_sum_mib(tcp_stack_t *, mib2_tcp_t *);

static void     tcp_add_mib(mib2_tcp_t *, mib2_tcp_t *);
static void     tcp_add_stats(tcp_stat_counter_t *, tcp_stat_t *);
static void     tcp_clr_stats(tcp_stat_t *);

tcp_g_stat_t    tcp_g_statistics;
kstat_t         *tcp_g_kstat;

/* Translate TCP state to MIB2 TCP state. */
static int
```



```
tcp_snmp_state(tcp_t *tcp)
{
        if (tcp == NULL)
                return (0);

        switch (tcp->tcp_state) {
        case TCPS_CLOSED:
        case TCPS_IDLE: /* RFC1213 doesn't have analogue for IDLE & BOUND */
        case TCPS_BOUND:
                return (MIB2_TCP_closed);
        case TCPS_LISTEN:
                return (MIB2_TCP_listen);
        case TCPS_SYN_SENT:
                return (MIB2_TCP_synSent);
        case TCPS_SYN_RCVD:
                return (MIB2_TCP_synReceived);
        case TCPS_ESTABLISHED:
                return (MIB2_TCP_established);
        case TCPS_CLOSE_WAIT:
                return (MIB2_TCP_closeWait);
        case TCPS_FIN_WAIT_1:
                return (MIB2_TCP_finWait1);
        case TCPS_CLOSING:
                return (MIB2_TCP_closing);
        case TCPS_LAST_ACK:
                return (MIB2_TCP_lastAck);
        case TCPS_FIN_WAIT_2:
                return (MIB2_TCP_finWait2);
        case TCPS_TIME_WAIT:
                return (MIB2_TCP_timeWait);
        default:
                return (0);
        }
}

/*
 * Return SNMP stuff in buffer in mpdata.
 */
mblk_t *
tcp_snmp_get(queue_t *q, mblk_t *mpctl, boolean_t legacy_req)
{
        mblk_t                  *mpdata;
        mblk_t                  *mp_conn_ctl = NULL;
        mblk_t                  *mp_conn_tail;
        mblk_t                  *mp_attr_ctl = NULL;
        mblk_t                  *mp_attr_tail;
        mblk_t                  *mp_pidnode_ctl = NULL;
        mblk_t                  *mp_pidnode_tail;
#endif /* ! codereview */
        mblk_t                  *mp6_conn_ctl = NULL;
        mblk_t                  *mp6_conn_tail;
        mblk_t                  *mp6_attr_ctl = NULL;
        mblk_t                  *mp6_attr_tail;
        mblk_t                  *mp6_pidnode_ctl = NULL;
        mblk_t                  *mp6_pidnode_tail;
#endif /* ! codereview */
        struct opthdr           *optp;
        mib2_tcpConnEntry_t     tce;
        mib2_tcp6ConnEntry_t    tce6;
        mib2_transportMLPEntry_t mlp;
        connf_t                 *connfp;
        int                     i;
        boolean_t               ispriv;
        zoneid_t                zoneid;
        int                     v4_conn_idx;
        int                     v6_conn_idx;
```

```
128         conn_t                  *connp = Q_TO_CONN(q);
129         tcp_stack_t             *tcps;
130         ip_stack_t              *ipst;
131         mblk_t                  *mp2ctl;
132         mib2_tcp_t              tcp_mib;
133         size_t                  tcp_mib_size, tce_size, tce6_size;

135         /*
136          * make a copy of the original message
137          */
138         mp2ctl = copymsg(mpctl);

140         if (mpctl == NULL ||
141             (mpdata = mpctl->b_cont) == NULL ||
142             (mp_conn_ctl = copymsg(mpctl)) == NULL ||
143             (mp_attr_ctl = copymsg(mpctl)) == NULL ||
144             (mp_pidnode_ctl = copymsg(mpctl)) == NULL ||
145 #endif /* ! codereview */
146             (mp6_conn_ctl = copymsg(mpctl)) == NULL ||
147             (mp6_attr_ctl = copymsg(mpctl)) == NULL ||
148             (mp6_pidnode_ctl = copymsg(mpctl)) == NULL) {
 33             (mp6_attr_ctl = copymsg(mpctl)) == NULL) {
149                 freemsg(mp_conn_ctl);
150                 freemsg(mp_attr_ctl);
151                 freemsg(mp_pidnode_ctl);
152 #endif /* ! codereview */
153                 freemsg(mp6_conn_ctl);
154                 freemsg(mp6_attr_ctl);
155                 freemsg(mp6_pidnode_ctl);
156 #endif /* ! codereview */
157                 freemsg(mpctl);
158                 freemsg(mp2ctl);
159                 return (NULL);
160         }

162         ipst = connp->conn_netstack->netstack_ip;
163         tcps = connp->conn_netstack->netstack_tcp;

165         if (legacy_req) {
166                 tcp_mib_size = LEGACY_MIB_SIZE(&tcp_mib, mib2_tcp_t);
167                 tce_size = LEGACY_MIB_SIZE(&tce, mib2_tcpConnEntry_t);
168                 tce6_size = LEGACY_MIB_SIZE(&tce6, mib2_tcp6ConnEntry_t);
169         } else {
170                 tcp_mib_size = sizeof (mib2_tcp_t);
171                 tce_size = sizeof (mib2_tcpConnEntry_t);
172                 tce6_size = sizeof (mib2_tcp6ConnEntry_t);
173         }

175         bzero(&tcp_mib, sizeof (tcp_mib));

177         /* build table of connections -- need count in fixed part */
178         SET_MIB(tcp_mib.tcpRtoAlgorithm, 4);   /* vanj */
179         SET_MIB(tcp_mib.tcpRtoMin, tcps->tcps_rexmit_interval_min);
180         SET_MIB(tcp_mib.tcpRtoMax, tcps->tcps_rexmit_interval_max);
181         SET_MIB(tcp_mib.tcpMaxConn, -1);
182         SET_MIB(tcp_mib.tcpCurrEstab, 0);

184         ispriv =
185             secpolicy_ip_config((Q_TO_CONN(q))->conn_cred, B_TRUE) == 0;
186         zoneid = Q_TO_CONN(q)->conn_zoneid;

188         v4_conn_idx = v6_conn_idx = 0;
189         mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;
190         mp_pidnode_tail = mp6_pidnode_tail = NULL;
191 #endif /* ! codereview */
```



```
193         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
194                 ipst = tcps->tcps_netstack->netstack_ip;

196                 connfp = &ipst->ips_ipcl_globalhash_fanout[i];

198                 connp = NULL;

200                 while ((connp =
201                     ipcl_get_next_conn(connfp, connp, IPCL_TCPCONN)) != NULL) {
202                         tcp_t *tcp;
203                         boolean_t needattr;

205                         if (connp->conn_zoneid != zoneid)
206                                 continue;       /* not in this zone */

208                         tcp = connp->conn_tcp;
209                         TCPS_UPDATE_MIB(tcps, tcpHCInSegs, tcp->tcp_ibsegs);
210                         tcp->tcp_ibsegs = 0;
211                         TCPS_UPDATE_MIB(tcps, tcpHCOutSegs, tcp->tcp_obsegs);
212                         tcp->tcp_obsegs = 0;

214                         tce6.tcp6ConnState = tce.tcpConnState =
215                             tcp_snmp_state(tcp);
216                         if (tce.tcpConnState == MIB2_TCP_established ||
217                             tce.tcpConnState == MIB2_TCP_closeWait)
218                                 BUMP_MIB(&tcp_mib, tcpCurrEstab);

220                         needattr = B_FALSE;
221                         bzero(&mlp, sizeof (mlp));
222                         if (connp->conn_mlp_type != mlptSingle) {
223                                 if (connp->conn_mlp_type == mlptShared ||
224                                     connp->conn_mlp_type == mlptBoth)
225                                         mlp.tme_flags |= MIB2_TMEF_SHARED;
226                                 if (connp->conn_mlp_type == mlptPrivate ||
227                                     connp->conn_mlp_type == mlptBoth)
228                                         mlp.tme_flags |= MIB2_TMEF_PRIVATE;
229                                 needattr = B_TRUE;
230                         }
231                         if (connp->conn_anon_mlp) {
232                                 mlp.tme_flags |= MIB2_TMEF_ANONMLP;
233                                 needattr = B_TRUE;
234                         }
235                         switch (connp->conn_mac_mode) {
236                         case CONN_MAC_DEFAULT:
237                                 break;
238                         case CONN_MAC_AWARE:
239                                 mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
240                                 needattr = B_TRUE;
241                                 break;
242                         case CONN_MAC_IMPLICIT:
243                                 mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
244                                 needattr = B_TRUE;
245                                 break;
246                         }
247                         if (connp->conn_ixa->ixa_tsl != NULL) {
248                                 ts_label_t *tsl;

250                                 tsl = connp->conn_ixa->ixa_tsl;
251                                 mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
252                                 mlp.tme_doi = label2doi(tsl);
253                                 mlp.tme_label = *label2bslabel(tsl);
254                                 needattr = B_TRUE;
255                         }

257                         /* Create a message to report on IPv6 entries */
258                         if (connp->conn_ipversion == IPV6_VERSION) {
```

```
259                         tce6.tcp6ConnLocalAddress = connp->conn_laddr_v6;
260                         tce6.tcp6ConnRemAddress = connp->conn_faddr_v6;
261                         tce6.tcp6ConnLocalPort = ntohs(connp->conn_lport);
262                         tce6.tcp6ConnRemPort = ntohs(connp->conn_fport);
263                         if (connp->conn_ixa->ixa_flags & IXAF_SCOPEID_SET) {
264                                 tce6.tcp6ConnIfIndex =
265                                     connp->conn_ixa->ixa_scopeid;
266                         } else {
267                                 tce6.tcp6ConnIfIndex = connp->conn_bound_if;
268                         }
269                         /* Don't want just anybody seeing these... */
270                         if (ispriv) {
271                                 tce6.tcp6ConnEntryInfo.ce_snxt =
272                                     tcp->tcp_snxt;
273                                 tce6.tcp6ConnEntryInfo.ce_suna =
274                                     tcp->tcp_suna;
275                                 tce6.tcp6ConnEntryInfo.ce_rnxt =
276                                     tcp->tcp_rnxt;
277                                 tce6.tcp6ConnEntryInfo.ce_rack =
278                                     tcp->tcp_rack;
279                         } else {
280                                 /*
281                                  * Netstat, unfortunately, uses this to
282                                  * get send/receive queue sizes.  How to fix?
283                                  * Why not compute the difference only?
284                                  */
285                                 tce6.tcp6ConnEntryInfo.ce_snxt =
286                                     tcp->tcp_snxt - tcp->tcp_suna;
287                                 tce6.tcp6ConnEntryInfo.ce_suna = 0;
288                                 tce6.tcp6ConnEntryInfo.ce_rnxt =
289                                     tcp->tcp_rnxt - tcp->tcp_rack;
290                                 tce6.tcp6ConnEntryInfo.ce_rack = 0;
291                         }

293                         tce6.tcp6ConnEntryInfo.ce_swnd = tcp->tcp_swnd;
294                         tce6.tcp6ConnEntryInfo.ce_rwnd = tcp->tcp_rwnd;
295                         tce6.tcp6ConnEntryInfo.ce_rto =  tcp->tcp_rto;
296                         tce6.tcp6ConnEntryInfo.ce_mss =  tcp->tcp_mss;
297                         tce6.tcp6ConnEntryInfo.ce_state = tcp->tcp_state;

299                         tce6.tcp6ConnCreationProcess =
300                             (connp->conn_cpid < 0) ? MIB2_UNKNOWN_PROCESS :
301                             connp->conn_cpid;
302                         tce6.tcp6ConnCreationTime = connp->conn_open_time;

304                         (void) snmp_append_data2(mp6_conn_ctl->b_cont,
305                             &mp6_conn_tail, (char *)&tce6, tce6_size);

307                         (void) snmp_append_data2(mp6_pidnode_ctl->b_cont,
308                                 &mp6_pidnode_tail, (char *)&tce6, tce6_size);

310                         (void) snmp_append_mblk2(mp6_pidnode_ctl->b_cont,
311                             &mp6_pidnode_tail, conn_get_pid_mblk(connp));

313 #endif /* ! codereview */
314                         mlp.tme_connidx = v6_conn_idx++;
315                         if (needattr)
316                                 (void) snmp_append_data2(mp6_attr_ctl->b_cont,
317                                     &mp6_attr_tail, (char *)&mlp, sizeof (mlp));
318                         }
319                         /*
320                          * Create an IPv4 table entry for IPv4 entries and also
321                          * for IPv6 entries which are bound to in6addr_any
322                          * but don't have IPV6_V6ONLY set.
323                          * (i.e. anything an IPv4 peer could connect to)
324                          */
```



```
325                         if (connp->conn_ipversion == IPV4_VERSION ||
326                             (tcp->tcp_state <= TCPS_LISTEN &&
327                             !connp->conn_ipv6_v6only &&
328                             IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6))) {
329                                 if (connp->conn_ipversion == IPV6_VERSION) {
330                                         tce.tcpConnRemAddress = INADDR_ANY;
331                                         tce.tcpConnLocalAddress = INADDR_ANY;
332                                 } else {
333                                         tce.tcpConnRemAddress =
334                                             connp->conn_faddr_v4;
335                                         tce.tcpConnLocalAddress =
336                                             connp->conn_laddr_v4;
337                                 }
338                                 tce.tcpConnLocalPort = ntohs(connp->conn_lport);
339                                 tce.tcpConnRemPort = ntohs(connp->conn_fport);
340                                 /* Don't want just anybody seeing these... */
341                                 if (ispriv) {
342                                         tce.tcpConnEntryInfo.ce_snxt =
343                                             tcp->tcp_snxt;
344                                         tce.tcpConnEntryInfo.ce_suna =
345                                             tcp->tcp_suna;
346                                         tce.tcpConnEntryInfo.ce_rnxt =
347                                             tcp->tcp_rnxt;
348                                         tce.tcpConnEntryInfo.ce_rack =
349                                             tcp->tcp_rack;
350                                 } else {
351                                         /*
352                                          * Netstat, unfortunately, uses this to
353                                          * get send/receive queue sizes.  How
354                                          * to fix?
355                                          * Why not compute the difference only?
356                                          */
357                                         tce.tcpConnEntryInfo.ce_snxt =
358                                             tcp->tcp_snxt - tcp->tcp_suna;
359                                         tce.tcpConnEntryInfo.ce_suna = 0;
360                                         tce.tcpConnEntryInfo.ce_rnxt =
361                                             tcp->tcp_rnxt - tcp->tcp_rack;
362                                         tce.tcpConnEntryInfo.ce_rack = 0;
363                                 }

365                                 tce.tcpConnEntryInfo.ce_swnd = tcp->tcp_swnd;
366                                 tce.tcpConnEntryInfo.ce_rwnd = tcp->tcp_rwnd;
367                                 tce.tcpConnEntryInfo.ce_rto =  tcp->tcp_rto;
368                                 tce.tcpConnEntryInfo.ce_mss =  tcp->tcp_mss;
369                                 tce.tcpConnEntryInfo.ce_state =
370                                     tcp->tcp_state;

372                                 tce.tcpConnCreationProcess =
373                                     (connp->conn_cpid < 0) ?
374                                     MIB2_UNKNOWN_PROCESS :
375                                     connp->conn_cpid;
376                                 tce.tcpConnCreationTime = connp->conn_open_time;

378                                 (void) snmp_append_data2(mp_conn_ctl->b_cont,
379                                     &mp_conn_tail, (char *)&tce, tce_size);

381                                 (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
382                                     &mp_pidnode_tail, (char *)&tce, tce_size);

384                                 (void) snmp_append_mblk2(mp_pidnode_ctl->b_cont,
385                                     &mp_pidnode_tail, conn_get_pid_mblk(connp));

387 #endif /* ! codereview */
388                                 mlp.tme_connidx = v4_conn_idx++;
389                                 if (needattr)
390                                         (void) snmp_append_data2(
```

```
391                                         mp_attr_ctl->b_cont,
392                                         &mp_attr_tail, (char *)&mlp,
393                                         sizeof (mlp));
394                         }
395                 }
396         }

398         tcp_sum_mib(tcps, &tcp_mib);

400         /* Fixed length structure for IPv4 and IPv6 counters */
401         SET_MIB(tcp_mib.tcpConnTableSize, tce_size);
402         SET_MIB(tcp_mib.tcp6ConnTableSize, tce6_size);

404         /*
405          * Synchronize 32- and 64-bit counters.  Note that tcpInSegs and
406          * tcpOutSegs are not updated anywhere in TCP.  The new 64 bits
407          * counters are used.  Hence the old counters' values in tcp_sc_mib
408          * are always 0.
409          */
410         SYNC32_MIB(&tcp_mib, tcpInSegs, tcpHCInSegs);
411         SYNC32_MIB(&tcp_mib, tcpOutSegs, tcpHCOutSegs);

413         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
414         optp->level = MIB2_TCP;
415         optp->name = 0;
416         (void) snmp_append_data(mpdata, (char *)&tcp_mib, tcp_mib_size);
417         optp->len = msgdsize(mpdata);
418         qreply(q, mpctl);

420         /* table of connections... */
421         optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
422             sizeof (struct T_optmgmt_ack)];
423         optp->level = MIB2_TCP;
424         optp->name = MIB2_TCP_CONN;
425         optp->len = msgdsize(mp_conn_ctl->b_cont);
426         qreply(q, mp_conn_ctl);

428         /* table of MLP attributes... */
429         optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
430             sizeof (struct T_optmgmt_ack)];
431         optp->level = MIB2_TCP;
432         optp->name = EXPER_XPORT_MLP;
433         optp->len = msgdsize(mp_attr_ctl->b_cont);
434         if (optp->len == 0)
435                 freemsg(mp_attr_ctl);
436         else
437                 qreply(q, mp_attr_ctl);

439         /* table of IPv6 connections... */
440         optp = (struct opthdr *)&mp6_conn_ctl->b_rptr[
441             sizeof (struct T_optmgmt_ack)];
442         optp->level = MIB2_TCP6;
443         optp->name = MIB2_TCP6_CONN;
444         optp->len = msgdsize(mp6_conn_ctl->b_cont);
445         qreply(q, mp6_conn_ctl);

447         /* table of IPv6 MLP attributes... */
448         optp = (struct opthdr *)&mp6_attr_ctl->b_rptr[
449             sizeof (struct T_optmgmt_ack)];
450         optp->level = MIB2_TCP6;
451         optp->name = EXPER_XPORT_MLP;
452         optp->len = msgdsize(mp6_attr_ctl->b_cont);
453         if (optp->len == 0)
454                 freemsg(mp6_attr_ctl);
455         else
456                 qreply(q, mp6_attr_ctl);
```



```
458         /* table of EXPER_XPORT_PROC_INFO  ipv4 */
459         optp = (struct opthdr *)&mp_pidnode_ctl->b_rptr[
460                 sizeof (struct T_optmgmt_ack)];
461         optp->level = MIB2_TCP;
462         optp->name = EXPER_XPORT_PROC_INFO;
463         optp->len = msgdsize(mp_pidnode_ctl->b_cont);
464         if (optp->len == 0)
465                 freemsg(mp_pidnode_ctl);
466         else
467                 qreply(q, mp_pidnode_ctl);

469         /* table of EXPER_XPORT_PROC_INFO  ipv6 */
470         optp = (struct opthdr *)&mp6_pidnode_ctl->b_rptr[
471                 sizeof (struct T_optmgmt_ack)];
472         optp->level = MIB2_TCP6;
473         optp->name = EXPER_XPORT_PROC_INFO;
474         optp->len = msgdsize(mp6_pidnode_ctl->b_cont);
475         if (optp->len == 0)
476                 freemsg(mp6_pidnode_ctl);
477         else
478                 qreply(q, mp6_pidnode_ctl);

480 #endif /* ! codereview */
481         return (mp2ctl);
482 }

484 /* Return 0 if invalid set request, 1 otherwise, including non-tcp requests  */
485 /* ARGSUSED */
486 int
487 tcp_snmp_set(queue_t *q, int level, int name, uchar_t *ptr, int len)
488 {
489         mib2_tcpConnEntry_t     *tce = (mib2_tcpConnEntry_t *)ptr;

491         switch (level) {
492         case MIB2_TCP:
493                 switch (name) {
494                 case 13:
495                         if (tce->tcpConnState != MIB2_TCP_deleteTCB)
496                                 return (0);
497                         /* TODO: delete entry defined by tce */
498                         return (1);
499                 default:
500                         return (0);
501                 }
502         default:
503                 return (1);
504         }
505 }

507 /*
508  * TCP Kstats implementation
509  */
510 void *
511 tcp_kstat_init(netstackid_t stackid)
512 {
513         kstat_t *ksp;

515         tcp_named_kstat_t template = {
516                 { "rtoAlgorithm",       KSTAT_DATA_INT32, 0 },
517                 { "rtoMin",             KSTAT_DATA_INT32, 0 },
518                 { "rtoMax",             KSTAT_DATA_INT32, 0 },
519                 { "maxConn",            KSTAT_DATA_INT32, 0 },
520                 { "activeOpens",        KSTAT_DATA_UINT32, 0 },
521                 { "passiveOpens",       KSTAT_DATA_UINT32, 0 },
522                 { "attemptFails",       KSTAT_DATA_UINT32, 0 },
```

```
523                 { "estabResets",        KSTAT_DATA_UINT32, 0 },
524                 { "currEstab",          KSTAT_DATA_UINT32, 0 },
525                 { "inSegs",             KSTAT_DATA_UINT64, 0 },
526                 { "outSegs",            KSTAT_DATA_UINT64, 0 },
527                 { "retransSegs",        KSTAT_DATA_UINT32, 0 },
528                 { "connTableSize",      KSTAT_DATA_INT32, 0 },
529                 { "outRsts",            KSTAT_DATA_UINT32, 0 },
530                 { "outDataSegs",        KSTAT_DATA_UINT32, 0 },
531                 { "outDataBytes",       KSTAT_DATA_UINT32, 0 },
532                 { "retransBytes",       KSTAT_DATA_UINT32, 0 },
533                 { "outAck",             KSTAT_DATA_UINT32, 0 },
534                 { "outAckDelayed",      KSTAT_DATA_UINT32, 0 },
535                 { "outUrg",             KSTAT_DATA_UINT32, 0 },
536                 { "outWinUpdate",       KSTAT_DATA_UINT32, 0 },
537                 { "outWinProbe",        KSTAT_DATA_UINT32, 0 },
538                 { "outControl",         KSTAT_DATA_UINT32, 0 },
539                 { "outFastRetrans",     KSTAT_DATA_UINT32, 0 },
540                 { "inAckSegs",          KSTAT_DATA_UINT32, 0 },
541                 { "inAckBytes",         KSTAT_DATA_UINT32, 0 },
542                 { "inDupAck",           KSTAT_DATA_UINT32, 0 },
543                 { "inAckUnsent",        KSTAT_DATA_UINT32, 0 },
544                 { "inDataInorderSegs",  KSTAT_DATA_UINT32, 0 },
545                 { "inDataInorderBytes", KSTAT_DATA_UINT32, 0 },
546                 { "inDataUnorderSegs",  KSTAT_DATA_UINT32, 0 },
547                 { "inDataUnorderBytes", KSTAT_DATA_UINT32, 0 },
548                 { "inDataDupSegs",      KSTAT_DATA_UINT32, 0 },
549                 { "inDataDupBytes",     KSTAT_DATA_UINT32, 0 },
550                 { "inDataPartDupSegs",  KSTAT_DATA_UINT32, 0 },
551                 { "inDataPartDupBytes", KSTAT_DATA_UINT32, 0 },
552                 { "inDataPastWinSegs",  KSTAT_DATA_UINT32, 0 },
553                 { "inDataPastWinBytes", KSTAT_DATA_UINT32, 0 },
554                 { "inWinProbe",         KSTAT_DATA_UINT32, 0 },
555                 { "inWinUpdate",        KSTAT_DATA_UINT32, 0 },
556                 { "inClosed",           KSTAT_DATA_UINT32, 0 },
557                 { "rttUpdate",          KSTAT_DATA_UINT32, 0 },
558                 { "rttNoUpdate",        KSTAT_DATA_UINT32, 0 },
559                 { "timRetrans",         KSTAT_DATA_UINT32, 0 },
560                 { "timRetransDrop",     KSTAT_DATA_UINT32, 0 },
561                 { "timKeepalive",       KSTAT_DATA_UINT32, 0 },
562                 { "timKeepaliveProbe",  KSTAT_DATA_UINT32, 0 },
563                 { "timKeepaliveDrop",   KSTAT_DATA_UINT32, 0 },
564                 { "listenDrop",         KSTAT_DATA_UINT32, 0 },
565                 { "listenDropQ0",       KSTAT_DATA_UINT32, 0 },
566                 { "halfOpenDrop",       KSTAT_DATA_UINT32, 0 },
567                 { "outSackRetransSegs", KSTAT_DATA_UINT32, 0 },
568                 { "connTableSize6",     KSTAT_DATA_INT32, 0 }
569         };

571         ksp = kstat_create_netstack(TCP_MOD_NAME, stackid, TCP_MOD_NAME, "mib2",
572             KSTAT_TYPE_NAMED, NUM_OF_FIELDS(tcp_named_kstat_t), 0, stackid);

574         if (ksp == NULL)
575                 return (NULL);

577         template.rtoAlgorithm.value.ui32 = 4;
578         template.maxConn.value.i32 = -1;

580         bcopy(&template, ksp->ks_data, sizeof (template));
581         ksp->ks_update = tcp_kstat_update;
582         ksp->ks_private = (void *)(uintptr_t)stackid;

584         /*
585          * If this is an exclusive netstack for a local zone, the global zone
586          * should still be able to read the kstat.
587          */
588         if (stackid != GLOBAL_NETSTACKID)
```



```
589                 kstat_zone_add(ksp, GLOBAL_ZONEID);

591         kstat_install(ksp);
592         return (ksp);
593 }

595 void
596 tcp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
597 {
598         if (ksp != NULL) {
599                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
600                 kstat_delete_netstack(ksp, stackid);
601         }
602 }

604 static int
605 tcp_kstat_update(kstat_t *kp, int rw)
606 {
607         tcp_named_kstat_t *tcpkp;
608         tcp_t           *tcp;
609         connf_t         *connfp;
610         conn_t          *connp;
611         int             i;
612         netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
613         netstack_t      *ns;
614         tcp_stack_t     *tcps;
615         ip_stack_t      *ipst;
616         mib2_tcp_t      tcp_mib;

618         if (rw == KSTAT_WRITE)
619                 return (EACCES);

621         ns = netstack_find_by_stackid(stackid);
622         if (ns == NULL)
623                 return (-1);
624         tcps = ns->netstack_tcp;
625         if (tcps == NULL) {
626                 netstack_rele(ns);
627                 return (-1);
628         }

630         tcpkp = (tcp_named_kstat_t *)kp->ks_data;

632         tcpkp->currEstab.value.ui32 = 0;
633         tcpkp->rtoMin.value.ui32 = tcps->tcps_rexmit_interval_min;
634         tcpkp->rtoMax.value.ui32 = tcps->tcps_rexmit_interval_max;

636         ipst = ns->netstack_ip;

638         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
639                 connfp = &ipst->ips_ipcl_globalhash_fanout[i];
640                 connp = NULL;
641                 while ((connp =
642                     ipcl_get_next_conn(connfp, connp, IPCL_TCPCONN)) != NULL) {
643                         tcp = connp->conn_tcp;
644                         switch (tcp_snmp_state(tcp)) {
645                         case MIB2_TCP_established:
646                         case MIB2_TCP_closeWait:
647                                 tcpkp->currEstab.value.ui32++;
648                                 break;
649                         }
650                 }
651         }
652         bzero(&tcp_mib, sizeof (tcp_mib));
653         tcp_sum_mib(tcps, &tcp_mib);
```

```
655         /* Fixed length structure for IPv4 and IPv6 counters */
656         SET_MIB(tcp_mib.tcpConnTableSize, sizeof (mib2_tcpConnEntry_t));
657         SET_MIB(tcp_mib.tcp6ConnTableSize, sizeof (mib2_tcp6ConnEntry_t));

659         tcpkp->activeOpens.value.ui32 = tcp_mib.tcpActiveOpens;
660         tcpkp->passiveOpens.value.ui32 = tcp_mib.tcpPassiveOpens;
661         tcpkp->attemptFails.value.ui32 = tcp_mib.tcpAttemptFails;
662         tcpkp->estabResets.value.ui32 = tcp_mib.tcpEstabResets;
663         tcpkp->inSegs.value.ui64 = tcp_mib.tcpHCInSegs;
664         tcpkp->outSegs.value.ui64 = tcp_mib.tcpHCOutSegs;
665         tcpkp->retransSegs.value.ui32 = tcp_mib.tcpRetransSegs;
666         tcpkp->connTableSize.value.i32 = tcp_mib.tcpConnTableSize;
667         tcpkp->outRsts.value.ui32 = tcp_mib.tcpOutRsts;
668         tcpkp->outDataSegs.value.ui32 = tcp_mib.tcpOutDataSegs;
669         tcpkp->outDataBytes.value.ui32 = tcp_mib.tcpOutDataBytes;
670         tcpkp->retransBytes.value.ui32 = tcp_mib.tcpRetransBytes;
671         tcpkp->outAck.value.ui32 = tcp_mib.tcpOutAck;
672         tcpkp->outAckDelayed.value.ui32 = tcp_mib.tcpOutAckDelayed;
673         tcpkp->outUrg.value.ui32 = tcp_mib.tcpOutUrg;
674         tcpkp->outWinUpdate.value.ui32 = tcp_mib.tcpOutWinUpdate;
675         tcpkp->outWinProbe.value.ui32 = tcp_mib.tcpOutWinProbe;
676         tcpkp->outControl.value.ui32 = tcp_mib.tcpOutControl;
677         tcpkp->outFastRetrans.value.ui32 = tcp_mib.tcpOutFastRetrans;
678         tcpkp->inAckSegs.value.ui32 = tcp_mib.tcpInAckSegs;
679         tcpkp->inAckBytes.value.ui32 = tcp_mib.tcpInAckBytes;
680         tcpkp->inDupAck.value.ui32 = tcp_mib.tcpInDupAck;
681         tcpkp->inAckUnsent.value.ui32 = tcp_mib.tcpInAckUnsent;
682         tcpkp->inDataInorderSegs.value.ui32 = tcp_mib.tcpInDataInorderSegs;
683         tcpkp->inDataInorderBytes.value.ui32 = tcp_mib.tcpInDataInorderBytes;
684         tcpkp->inDataUnorderSegs.value.ui32 = tcp_mib.tcpInDataUnorderSegs;
685         tcpkp->inDataUnorderBytes.value.ui32 = tcp_mib.tcpInDataUnorderBytes;
686         tcpkp->inDataDupSegs.value.ui32 = tcp_mib.tcpInDataDupSegs;
687         tcpkp->inDataDupBytes.value.ui32 = tcp_mib.tcpInDataDupBytes;
688         tcpkp->inDataPartDupSegs.value.ui32 = tcp_mib.tcpInDataPartDupSegs;
689         tcpkp->inDataPartDupBytes.value.ui32 = tcp_mib.tcpInDataPartDupBytes;
690         tcpkp->inDataPastWinSegs.value.ui32 = tcp_mib.tcpInDataPastWinSegs;
691         tcpkp->inDataPastWinBytes.value.ui32 = tcp_mib.tcpInDataPastWinBytes;
692         tcpkp->inWinProbe.value.ui32 = tcp_mib.tcpInWinProbe;
693         tcpkp->inWinUpdate.value.ui32 = tcp_mib.tcpInWinUpdate;
694         tcpkp->inClosed.value.ui32 = tcp_mib.tcpInClosed;
695         tcpkp->rttNoUpdate.value.ui32 = tcp_mib.tcpRttNoUpdate;
696         tcpkp->rttUpdate.value.ui32 = tcp_mib.tcpRttUpdate;
697         tcpkp->timRetrans.value.ui32 = tcp_mib.tcpTimRetrans;
698         tcpkp->timRetransDrop.value.ui32 = tcp_mib.tcpTimRetransDrop;
699         tcpkp->timKeepalive.value.ui32 = tcp_mib.tcpTimKeepalive;
700         tcpkp->timKeepaliveProbe.value.ui32 = tcp_mib.tcpTimKeepaliveProbe;
701         tcpkp->timKeepaliveDrop.value.ui32 = tcp_mib.tcpTimKeepaliveDrop;
702         tcpkp->listenDrop.value.ui32 = tcp_mib.tcpListenDrop;
703         tcpkp->listenDropQ0.value.ui32 = tcp_mib.tcpListenDropQ0;
704         tcpkp->halfOpenDrop.value.ui32 = tcp_mib.tcpHalfOpenDrop;
705         tcpkp->outSackRetransSegs.value.ui32 = tcp_mib.tcpOutSackRetransSegs;
706         tcpkp->connTableSize6.value.i32 = tcp_mib.tcp6ConnTableSize;

708         netstack_rele(ns);
709         return (0);
710 }

712 /*
713  * kstats related to squeues i.e. not per IP instance
714  */
715 void *
716 tcp_g_kstat_init(tcp_g_stat_t *tcp_g_statp)
717 {
718         kstat_t *ksp;

720         tcp_g_stat_t template = {
```



```
721                 { "tcp_timermp_alloced",        KSTAT_DATA_UINT64 },
722                 { "tcp_timermp_allocfail",      KSTAT_DATA_UINT64 },
723                 { "tcp_timermp_allocdblfail",   KSTAT_DATA_UINT64 },
724                 { "tcp_freelist_cleanup",       KSTAT_DATA_UINT64 },
725         };

727         ksp = kstat_create(TCP_MOD_NAME, 0, "tcpstat_g", "net",
728             KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t),
729             KSTAT_FLAG_VIRTUAL);

731         if (ksp == NULL)
732                 return (NULL);

734         bcopy(&template, tcp_g_statp, sizeof (template));
735         ksp->ks_data = (void *)tcp_g_statp;

737         kstat_install(ksp);
738         return (ksp);
739 }

741 void
742 tcp_g_kstat_fini(kstat_t *ksp)
743 {
744         if (ksp != NULL) {
745                 kstat_delete(ksp);
746         }
747 }

749 void *
750 tcp_kstat2_init(netstackid_t stackid)
751 {
752         kstat_t *ksp;

754         tcp_stat_t template = {
755                 { "tcp_time_wait_syn_success",  KSTAT_DATA_UINT64, 0 },
756                 { "tcp_clean_death_nondetached",        KSTAT_DATA_UINT64, 0 },
757                 { "tcp_eager_blowoff_q",        KSTAT_DATA_UINT64, 0 },
758                 { "tcp_eager_blowoff_q0",       KSTAT_DATA_UINT64, 0 },
759                 { "tcp_no_listener",            KSTAT_DATA_UINT64, 0 },
760                 { "tcp_listendrop",             KSTAT_DATA_UINT64, 0 },
761                 { "tcp_listendropq0",           KSTAT_DATA_UINT64, 0 },
762                 { "tcp_wsrv_called",            KSTAT_DATA_UINT64, 0 },
763                 { "tcp_flwctl_on",              KSTAT_DATA_UINT64, 0 },
764                 { "tcp_timer_fire_early",       KSTAT_DATA_UINT64, 0 },
765                 { "tcp_timer_fire_miss",        KSTAT_DATA_UINT64, 0 },
766                 { "tcp_zcopy_on",               KSTAT_DATA_UINT64, 0 },
767                 { "tcp_zcopy_off",              KSTAT_DATA_UINT64, 0 },
768                 { "tcp_zcopy_backoff",          KSTAT_DATA_UINT64, 0 },
769                 { "tcp_fusion_flowctl",         KSTAT_DATA_UINT64, 0 },
770                 { "tcp_fusion_backenabled",     KSTAT_DATA_UINT64, 0 },
771                 { "tcp_fusion_urg",             KSTAT_DATA_UINT64, 0 },
772                 { "tcp_fusion_putnext",         KSTAT_DATA_UINT64, 0 },
773                 { "tcp_fusion_unfusable",       KSTAT_DATA_UINT64, 0 },
774                 { "tcp_fusion_aborted",         KSTAT_DATA_UINT64, 0 },
775                 { "tcp_fusion_unqualified",     KSTAT_DATA_UINT64, 0 },
776                 { "tcp_fusion_rrw_busy",        KSTAT_DATA_UINT64, 0 },
777                 { "tcp_fusion_rrw_msgcnt",      KSTAT_DATA_UINT64, 0 },
778                 { "tcp_fusion_rrw_plugged",     KSTAT_DATA_UINT64, 0 },
779                 { "tcp_in_ack_unsent_drop",     KSTAT_DATA_UINT64, 0 },
780                 { "tcp_sock_fallback",          KSTAT_DATA_UINT64, 0 },
781                 { "tcp_lso_enabled",            KSTAT_DATA_UINT64, 0 },
782                 { "tcp_lso_disabled",           KSTAT_DATA_UINT64, 0 },
783                 { "tcp_lso_times",              KSTAT_DATA_UINT64, 0 },
784                 { "tcp_lso_pkt_out",            KSTAT_DATA_UINT64, 0 },
785                 { "tcp_listen_cnt_drop",        KSTAT_DATA_UINT64, 0 },
786                 { "tcp_listen_mem_drop",        KSTAT_DATA_UINT64, 0 },
```

```
787                 { "tcp_zwin_mem_drop",          KSTAT_DATA_UINT64, 0 },
788                 { "tcp_zwin_ack_syn",           KSTAT_DATA_UINT64, 0 },
789                 { "tcp_rst_unsent",             KSTAT_DATA_UINT64, 0 },
790                 { "tcp_reclaim_cnt",            KSTAT_DATA_UINT64, 0 },
791                 { "tcp_reass_timeout",          KSTAT_DATA_UINT64, 0 },
792 #ifdef TCP_DEBUG_COUNTER
793                 { "tcp_time_wait",              KSTAT_DATA_UINT64, 0 },
794                 { "tcp_rput_time_wait",         KSTAT_DATA_UINT64, 0 },
795                 { "tcp_detach_time_wait",       KSTAT_DATA_UINT64, 0 },
796                 { "tcp_timeout_calls",          KSTAT_DATA_UINT64, 0 },
797                 { "tcp_timeout_cached_alloc",   KSTAT_DATA_UINT64, 0 },
798                 { "tcp_timeout_cancel_reqs",    KSTAT_DATA_UINT64, 0 },
799                 { "tcp_timeout_canceled",       KSTAT_DATA_UINT64, 0 },
800                 { "tcp_timermp_freed",          KSTAT_DATA_UINT64, 0 },
801                 { "tcp_push_timer_cnt",         KSTAT_DATA_UINT64, 0 },
802                 { "tcp_ack_timer_cnt",          KSTAT_DATA_UINT64, 0 },
803 #endif
804         };

806         ksp = kstat_create_netstack(TCP_MOD_NAME, stackid, "tcpstat", "net",
807             KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t), 0,
808             stackid);

810         if (ksp == NULL)
811                 return (NULL);

813         bcopy(&template, ksp->ks_data, sizeof (template));
814         ksp->ks_private = (void *)(uintptr_t)stackid;
815         ksp->ks_update = tcp_kstat2_update;

817         /*
818          * If this is an exclusive netstack for a local zone, the global zone
819          * should still be able to read the kstat.
820          */
821         if (stackid != GLOBAL_NETSTACKID)
822                 kstat_zone_add(ksp, GLOBAL_ZONEID);

824         kstat_install(ksp);
825         return (ksp);
826 }

828 void
829 tcp_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
830 {
831         if (ksp != NULL) {
832                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
833                 kstat_delete_netstack(ksp, stackid);
834         }
835 }

837 /*
838  * Sum up all per CPU tcp_stat_t kstat counters.
839  */
840 static int
841 tcp_kstat2_update(kstat_t *kp, int rw)
842 {
843         netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
844         netstack_t      *ns;
845         tcp_stack_t     *tcps;
846         tcp_stat_t      *stats;
847         int             i;
848         int             cnt;

850         if (rw == KSTAT_WRITE)
851                 return (EACCES);
```



```
853         ns = netstack_find_by_stackid(stackid);
854         if (ns == NULL)
855                 return (-1);
856         tcps = ns->netstack_tcp;
857         if (tcps == NULL) {
858                 netstack_rele(ns);
859                 return (-1);
860         }

862         stats = (tcp_stat_t *)kp->ks_data;
863         tcp_clr_stats(stats);

865         /*
866          * tcps_sc_cnt may change in the middle of the loop.  It is better
867          * to get its value first.
868          */
869         cnt = tcps->tcps_sc_cnt;
870         for (i = 0; i < cnt; i++)
871                 tcp_add_stats(&tcps->tcps_sc[i]->tcp_sc_stats, stats);

873         netstack_rele(ns);
874         return (0);
875 }

877 /*
878  * To add stats from one mib2_tcp_t to another.  Static fields are not added.
879  * The caller should set them up propertly.
880  */
881 static void
882 tcp_add_mib(mib2_tcp_t *from, mib2_tcp_t *to)
883 {
884         to->tcpActiveOpens += from->tcpActiveOpens;
885         to->tcpPassiveOpens += from->tcpPassiveOpens;
886         to->tcpAttemptFails += from->tcpAttemptFails;
887         to->tcpEstabResets += from->tcpEstabResets;
888         to->tcpInSegs += from->tcpInSegs;
889         to->tcpOutSegs += from->tcpOutSegs;
890         to->tcpRetransSegs += from->tcpRetransSegs;
891         to->tcpOutRsts += from->tcpOutRsts;

893         to->tcpOutDataSegs += from->tcpOutDataSegs;
894         to->tcpOutDataBytes += from->tcpOutDataBytes;
895         to->tcpRetransBytes += from->tcpRetransBytes;
896         to->tcpOutAck += from->tcpOutAck;
897         to->tcpOutAckDelayed += from->tcpOutAckDelayed;
898         to->tcpOutUrg += from->tcpOutUrg;
899         to->tcpOutWinUpdate += from->tcpOutWinUpdate;
900         to->tcpOutWinProbe += from->tcpOutWinProbe;
901         to->tcpOutControl += from->tcpOutControl;
902         to->tcpOutFastRetrans += from->tcpOutFastRetrans;

904         to->tcpInAckBytes += from->tcpInAckBytes;
905         to->tcpInDupAck += from->tcpInDupAck;
906         to->tcpInAckUnsent += from->tcpInAckUnsent;
907         to->tcpInDataInorderSegs += from->tcpInDataInorderSegs;
908         to->tcpInDataInorderBytes += from->tcpInDataInorderBytes;
909         to->tcpInDataUnorderSegs += from->tcpInDataUnorderSegs;
910         to->tcpInDataUnorderBytes += from->tcpInDataUnorderBytes;
911         to->tcpInDataDupSegs += from->tcpInDataDupSegs;
912         to->tcpInDataDupBytes += from->tcpInDataDupBytes;
913         to->tcpInDataPartDupSegs += from->tcpInDataPartDupSegs;
914         to->tcpInDataPartDupBytes += from->tcpInDataPartDupBytes;
915         to->tcpInDataPastWinSegs += from->tcpInDataPastWinSegs;
916         to->tcpInDataPastWinBytes += from->tcpInDataPastWinBytes;
917         to->tcpInWinProbe += from->tcpInWinProbe;
918         to->tcpInWinUpdate += from->tcpInWinUpdate;
```

```
        to->tcpInClosed += from->tcpInClosed;

        to->tcpRttNoUpdate += from->tcpRttNoUpdate;
        to->tcpRttUpdate += from->tcpRttUpdate;
        to->tcpTimRetrans += from->tcpTimRetrans;
        to->tcpTimRetransDrop += from->tcpTimRetransDrop;
        to->tcpTimKeepalive += from->tcpTimKeepalive;
        to->tcpTimKeepaliveProbe += from->tcpTimKeepaliveProbe;
        to->tcpTimKeepaliveDrop += from->tcpTimKeepaliveDrop;
        to->tcpListenDrop += from->tcpListenDrop;
        to->tcpListenDropQ0 += from->tcpListenDropQ0;
        to->tcpHalfOpenDrop += from->tcpHalfOpenDrop;
        to->tcpOutSackRetransSegs += from->tcpOutSackRetransSegs;
        to->tcpHCInSegs += from->tcpHCInSegs;
        to->tcpHCOutSegs += from->tcpHCOutSegs;
}

/*
 * To sum up all MIB2 stats for a tcp_stack_t from all per CPU stats.  The
 * caller should initialize the target mib2_tcp_t properly as this function
 * just adds up all the per CPU stats.
 */
static void
tcp_sum_mib(tcp_stack_t *tcps, mib2_tcp_t *tcp_mib)
{
        int i;
        int cnt;

        /*
         * tcps_sc_cnt may change in the middle of the loop.  It is better
         * to get its value first.
         */
        cnt = tcps->tcps_sc_cnt;
        for (i = 0; i < cnt; i++)
                tcp_add_mib(&tcps->tcps_sc[i]->tcp_sc_mib, tcp_mib);
}

/*
 * To set all tcp_stat_t counters to 0.
 */
static void
tcp_clr_stats(tcp_stat_t *stats)
{
        stats->tcp_time_wait_syn_success.value.ui64 = 0;
        stats->tcp_clean_death_nondetached.value.ui64 = 0;
        stats->tcp_eager_blowoff_q.value.ui64 = 0;
        stats->tcp_eager_blowoff_q0.value.ui64 = 0;
        stats->tcp_no_listener.value.ui64 = 0;
        stats->tcp_listendrop.value.ui64 = 0;
        stats->tcp_listendropq0.value.ui64 = 0;
        stats->tcp_wsrv_called.value.ui64 = 0;
        stats->tcp_flwctl_on.value.ui64 = 0;
        stats->tcp_timer_fire_early.value.ui64 = 0;
        stats->tcp_timer_fire_miss.value.ui64 = 0;
        stats->tcp_zcopy_on.value.ui64 = 0;
        stats->tcp_zcopy_off.value.ui64 = 0;
        stats->tcp_zcopy_backoff.value.ui64 = 0;
        stats->tcp_fusion_flowctl.value.ui64 = 0;
        stats->tcp_fusion_backenabled.value.ui64 = 0;
        stats->tcp_fusion_urg.value.ui64 = 0;
        stats->tcp_fusion_putnext.value.ui64 = 0;
        stats->tcp_fusion_unfusable.value.ui64 = 0;
        stats->tcp_fusion_aborted.value.ui64 = 0;
        stats->tcp_fusion_unqualified.value.ui64 = 0;
        stats->tcp_fusion_rrw_busy.value.ui64 = 0;
        stats->tcp_fusion_rrw_msgcnt.value.ui64 = 0;
```



```
        stats->tcp_fusion_rrw_plugged.value.ui64 = 0;
        stats->tcp_in_ack_unsent_drop.value.ui64 = 0;
        stats->tcp_sock_fallback.value.ui64 = 0;
        stats->tcp_lso_enabled.value.ui64 = 0;
        stats->tcp_lso_disabled.value.ui64 = 0;
        stats->tcp_lso_times.value.ui64 = 0;
        stats->tcp_lso_pkt_out.value.ui64 = 0;
        stats->tcp_listen_cnt_drop.value.ui64 = 0;
        stats->tcp_listen_mem_drop.value.ui64 = 0;
        stats->tcp_zwin_mem_drop.value.ui64 = 0;
        stats->tcp_zwin_ack_syn.value.ui64 = 0;
        stats->tcp_rst_unsent.value.ui64 = 0;
        stats->tcp_reclaim_cnt.value.ui64 = 0;
        stats->tcp_reass_timeout.value.ui64 = 0;

#ifdef TCP_DEBUG_COUNTER
        stats->tcp_time_wait.value.ui64 = 0;
        stats->tcp_rput_time_wait.value.ui64 = 0;
        stats->tcp_detach_time_wait.value.ui64 = 0;
        stats->tcp_timeout_calls.value.ui64 = 0;
        stats->tcp_timeout_cached_alloc.value.ui64 = 0;
        stats->tcp_timeout_cancel_reqs.value.ui64 = 0;
        stats->tcp_timeout_canceled.value.ui64 = 0;
        stats->tcp_timermp_freed.value.ui64 = 0;
        stats->tcp_push_timer_cnt.value.ui64 = 0;
        stats->tcp_ack_timer_cnt.value.ui64 = 0;
#endif
}

/*
 * To add counters from the per CPU tcp_stat_counter_t to the stack
 * tcp_stat_t.
 */
static void
tcp_add_stats(tcp_stat_counter_t *from, tcp_stat_t *to)
{
        to->tcp_time_wait_syn_success.value.ui64 +=
            from->tcp_time_wait_syn_success;
        to->tcp_clean_death_nondetached.value.ui64 +=
            from->tcp_clean_death_nondetached;
        to->tcp_eager_blowoff_q.value.ui64 +=
            from->tcp_eager_blowoff_q;
        to->tcp_eager_blowoff_q0.value.ui64 +=
            from->tcp_eager_blowoff_q0;
        to->tcp_no_listener.value.ui64 +=
            from->tcp_no_listener;
        to->tcp_listendrop.value.ui64 +=
            from->tcp_listendrop;
        to->tcp_listendropq0.value.ui64 +=
            from->tcp_listendropq0;
        to->tcp_wsrv_called.value.ui64 +=
            from->tcp_wsrv_called;
        to->tcp_flwctl_on.value.ui64 +=
            from->tcp_flwctl_on;
        to->tcp_timer_fire_early.value.ui64 +=
            from->tcp_timer_fire_early;
        to->tcp_timer_fire_miss.value.ui64 +=
            from->tcp_timer_fire_miss;
        to->tcp_zcopy_on.value.ui64 +=
            from->tcp_zcopy_on;
        to->tcp_zcopy_off.value.ui64 +=
            from->tcp_zcopy_off;
        to->tcp_zcopy_backoff.value.ui64 +=
            from->tcp_zcopy_backoff;
        to->tcp_fusion_flowctl.value.ui64 +=
            from->tcp_fusion_flowctl;
```

```
1051         to->tcp_fusion_backenabled.value.ui64 +=
1052             from->tcp_fusion_backenabled;
1053         to->tcp_fusion_urg.value.ui64 +=
1054             from->tcp_fusion_urg;
1055         to->tcp_fusion_putnext.value.ui64 +=
1056             from->tcp_fusion_putnext;
1057         to->tcp_fusion_unfusable.value.ui64 +=
1058             from->tcp_fusion_unfusable;
1059         to->tcp_fusion_aborted.value.ui64 +=
1060             from->tcp_fusion_aborted;
1061         to->tcp_fusion_unqualified.value.ui64 +=
1062             from->tcp_fusion_unqualified;
1063         to->tcp_fusion_rrw_busy.value.ui64 +=
1064             from->tcp_fusion_rrw_busy;
1065         to->tcp_fusion_rrw_msgcnt.value.ui64 +=
1066             from->tcp_fusion_rrw_msgcnt;
1067         to->tcp_fusion_rrw_plugged.value.ui64 +=
1068             from->tcp_fusion_rrw_plugged;
1069         to->tcp_in_ack_unsent_drop.value.ui64 +=
1070             from->tcp_in_ack_unsent_drop;
1071         to->tcp_sock_fallback.value.ui64 +=
1072             from->tcp_sock_fallback;
1073         to->tcp_lso_enabled.value.ui64 +=
1074             from->tcp_lso_enabled;
1075         to->tcp_lso_disabled.value.ui64 +=
1076             from->tcp_lso_disabled;
1077         to->tcp_lso_times.value.ui64 +=
1078             from->tcp_lso_times;
1079         to->tcp_lso_pkt_out.value.ui64 +=
1080             from->tcp_lso_pkt_out;
1081         to->tcp_listen_cnt_drop.value.ui64 +=
1082             from->tcp_listen_cnt_drop;
1083         to->tcp_listen_mem_drop.value.ui64 +=
1084             from->tcp_listen_mem_drop;
1085         to->tcp_zwin_mem_drop.value.ui64 +=
1086             from->tcp_zwin_mem_drop;
1087         to->tcp_zwin_ack_syn.value.ui64 +=
1088             from->tcp_zwin_ack_syn;
1089         to->tcp_rst_unsent.value.ui64 +=
1090             from->tcp_rst_unsent;
1091         to->tcp_reclaim_cnt.value.ui64 +=
1092             from->tcp_reclaim_cnt;
1093         to->tcp_reass_timeout.value.ui64 +=
1094             from->tcp_reass_timeout;

1096 #ifdef TCP_DEBUG_COUNTER
1097         to->tcp_time_wait.value.ui64 +=
1098             from->tcp_time_wait;
1099         to->tcp_rput_time_wait.value.ui64 +=
1100             from->tcp_rput_time_wait;
1101         to->tcp_detach_time_wait.value.ui64 +=
1102             from->tcp_detach_time_wait;
1103         to->tcp_timeout_calls.value.ui64 +=
1104             from->tcp_timeout_calls;
1105         to->tcp_timeout_cached_alloc.value.ui64 +=
1106             from->tcp_timeout_cached_alloc;
1107         to->tcp_timeout_cancel_reqs.value.ui64 +=
1108             from->tcp_timeout_cancel_reqs;
1109         to->tcp_timeout_canceled.value.ui64 +=
1110             from->tcp_timeout_canceled;
1111         to->tcp_timermp_freed.value.ui64 +=
1112             from->tcp_timermp_freed;
1113         to->tcp_push_timer_cnt.value.ui64 +=
1114             from->tcp_push_timer_cnt;
1115         to->tcp_ack_timer_cnt.value.ui64 +=
1116             from->tcp_ack_timer_cnt;
```



```
1117 #endif
1118 }
```

```
**********************************************************
   17786 Mon Aug 17 21:08:06 2015
new/usr/src/uts/common/inet/udp/udp_stats.c
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/types.h>
#include <sys/tihdr.h>
#include <sys/policy.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/kstatcom.h>
#include <inet/snmpcom.h>
#include <inet/mib2.h>
#include <inet/optcom.h>
#include <inet/snmpcom.h>
#include <inet/kstatcom.h>
#include <inet/udp_impl.h>

static int      udp_kstat_update(kstat_t *, int);
static int      udp_kstat2_update(kstat_t *, int);
static void     udp_sum_mib(udp_stack_t *, mib2_udp_t *);
static void     udp_clr_stats(udp_stat_t *);
static void     udp_add_stats(udp_stat_counter_t *, udp_stat_t *);
static void     udp_add_mib(mib2_udp_t *, mib2_udp_t *);
/*
 * return SNMP stuff in buffer in mpdata. We don't hold any lock and report
 * information that can be changing beneath us.
 */
mblk_t *
udp_snmp_get(queue_t *q, mblk_t *mpctl, boolean_t legacy_req)
{
        mblk_t                  *mpdata;
        mblk_t                  *mp_conn_ctl;
        mblk_t                  *mp_attr_ctl;
        mblk_t                  *mp_pidnode_ctl;
#endif /* ! codereview */
        mblk_t                  *mp6_conn_ctl;
        mblk_t                  *mp6_attr_ctl;
        mblk_t                  *mp6_pidnode_ctl;
#endif /* ! codereview */
```



```
        mblk_t                  *mp_conn_tail;
        mblk_t                  *mp_attr_tail;
        mblk_t                  *mp_pidnode_tail;
#endif /* ! codereview */
        mblk_t                  *mp6_conn_tail;
        mblk_t                  *mp6_attr_tail;
        mblk_t                  *mp6_pidnode_tail;
#endif /* ! codereview */
        struct opthdr           *optp;
        mib2_udpEntry_t         ude;
        mib2_udp6Entry_t        ude6;
        mib2_transportMLPEntry_t mlp;
        int                     state;
        zoneid_t                zoneid;
        int                     i;
        connf_t                 *connfp;
        conn_t                  *connp = Q_TO_CONN(q);
        int                     v4_conn_idx;
        int                     v6_conn_idx;
        boolean_t               needattr;
        udp_t                   *udp;
        ip_stack_t              *ipst = connp->conn_netstack->netstack_ip;
        udp_stack_t             *us = connp->conn_netstack->netstack_udp;
        mblk_t                  *mp2ctl;
        mib2_udp_t              udp_mib;
        size_t                  udp_mib_size, ude_size, ude6_size;


        /*
         * make a copy of the original message
         */
        mp2ctl = copymsg(mpctl);

        mp_conn_ctl = mp_attr_ctl = mp6_conn_ctl = NULL;
        if (mpctl == NULL ||
            (mpdata = mpctl->b_cont) == NULL ||
            (mp_conn_ctl = copymsg(mpctl)) == NULL ||
            (mp_attr_ctl = copymsg(mpctl)) == NULL ||
            (mp_pidnode_ctl = copymsg(mpctl)) == NULL ||
#endif /* ! codereview */
            (mp6_conn_ctl = copymsg(mpctl)) == NULL ||
            (mp6_attr_ctl = copymsg(mpctl)) == NULL ||
            (mp6_pidnode_ctl = copymsg(mpctl)) == NULL) {
 67         (mp6_attr_ctl = copymsg(mpctl)) == NULL) {
                freemsg(mp_conn_ctl);
                freemsg(mp_attr_ctl);
                freemsg(mp_pidnode_ctl);
#endif /* ! codereview */
                freemsg(mp6_conn_ctl);
                freemsg(mp6_attr_ctl);
                freemsg(mp6_pidnode_ctl);
#endif /* ! codereview */
                freemsg(mpctl);
                freemsg(mp2ctl);
                return (0);
        }

        zoneid = connp->conn_zoneid;

        if (legacy_req) {
                udp_mib_size = LEGACY_MIB_SIZE(&udp_mib, mib2_udp_t);
                ude_size = LEGACY_MIB_SIZE(&ude, mib2_udpEntry_t);
                ude6_size = LEGACY_MIB_SIZE(&ude6, mib2_udp6Entry_t);
        } else {
                udp_mib_size = sizeof (mib2_udp_t);
                ude_size = sizeof (mib2_udpEntry_t);
```

```
126                 ude6_size = sizeof (mib2_udp6Entry_t);
127         }

129         bzero(&udp_mib, sizeof (udp_mib));
130         /* fixed length structure for IPv4 and IPv6 counters */
131         SET_MIB(udp_mib.udpEntrySize, ude_size);
132         SET_MIB(udp_mib.udp6EntrySize, ude6_size);

134         udp_sum_mib(us, &udp_mib);

136         /*
137          * Synchronize 32- and 64-bit counters.  Note that udpInDatagrams and
138          * udpOutDatagrams are not updated anywhere in UDP.  The new 64 bits
139          * counters are used.  Hence the old counters' values in us_sc_mib
140          * are always 0.
141          */
142         SYNC32_MIB(&udp_mib, udpInDatagrams, udpHCInDatagrams);
143         SYNC32_MIB(&udp_mib, udpOutDatagrams, udpHCOutDatagrams);

145         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
146         optp->level = MIB2_UDP;
147         optp->name = 0;
148         (void) snmp_append_data(mpdata, (char *)&udp_mib, udp_mib_size);
149         optp->len = msgdsize(mpdata);
150         qreply(q, mpctl);

152         mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;
153         mp_pidnode_tail = mp6_pidnode_tail = NULL;
154 #endif /* ! codereview */
155         v4_conn_idx = v6_conn_idx = 0;

157         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
158                 connfp = &ipst->ips_ipcl_globalhash_fanout[i];
159                 connp = NULL;

161                 while ((connp = ipcl_get_next_conn(connfp, connp,
162                     IPCL_UDPCONN))) {
163                         udp = connp->conn_udp;
164                         if (zoneid != connp->conn_zoneid)
165                                 continue;

167                         /*
168                          * Note that the port numbers are sent in
169                          * host byte order
170                          */

172                         if (udp->udp_state == TS_UNBND)
173                                 state = MIB2_UDP_unbound;
174                         else if (udp->udp_state == TS_IDLE)
175                                 state = MIB2_UDP_idle;
176                         else if (udp->udp_state == TS_DATA_XFER)
177                                 state = MIB2_UDP_connected;
178                         else
179                                 state = MIB2_UDP_unknown;

181                         needattr = B_FALSE;
182                         bzero(&mlp, sizeof (mlp));
183                         if (connp->conn_mlp_type != mlptSingle) {
184                                 if (connp->conn_mlp_type == mlptShared ||
185                                     connp->conn_mlp_type == mlptBoth)
186                                         mlp.tme_flags |= MIB2_TMEF_SHARED;
187                                 if (connp->conn_mlp_type == mlptPrivate ||
188                                     connp->conn_mlp_type == mlptBoth)
189                                         mlp.tme_flags |= MIB2_TMEF_PRIVATE;
190                                 needattr = B_TRUE;
191                         }
```



```
192                         if (connp->conn_anon_mlp) {
193                                 mlp.tme_flags |= MIB2_TMEF_ANONMLP;
194                                 needattr = B_TRUE;
195                         }
196                         switch (connp->conn_mac_mode) {
197                         case CONN_MAC_DEFAULT:
198                                 break;
199                         case CONN_MAC_AWARE:
200                                 mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
201                                 needattr = B_TRUE;
202                                 break;
203                         case CONN_MAC_IMPLICIT:
204                                 mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
205                                 needattr = B_TRUE;
206                                 break;
207                         }
208                         mutex_enter(&connp->conn_lock);
209                         if (udp->udp_state == TS_DATA_XFER &&
210                             connp->conn_ixa->ixa_tsl != NULL) {
211                                 ts_label_t *tsl;

213                                 tsl = connp->conn_ixa->ixa_tsl;
214                                 mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
215                                 mlp.tme_doi = label2doi(tsl);
216                                 mlp.tme_label = *label2bslabel(tsl);
217                                 needattr = B_TRUE;
218                         }
219                         mutex_exit(&connp->conn_lock);

221                         /*
222                          * Create an IPv4 table entry for IPv4 entries and also
223                          * any IPv6 entries which are bound to in6addr_any
224                          * (i.e. anything a IPv4 peer could connect/send to).
225                          */
226                         if (connp->conn_ipversion == IPV4_VERSION ||
227                             (udp->udp_state <= TS_IDLE &&
228                             IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6))) {
229                                 ude.udpEntryInfo.ue_state = state;
230                                 /*
231                                  * If in6addr_any this will set it to
232                                  * INADDR_ANY
233                                  */
234                                 ude.udpLocalAddress = connp->conn_laddr_v4;
235                                 ude.udpLocalPort = ntohs(connp->conn_lport);
236                                 if (udp->udp_state == TS_DATA_XFER) {
237                                         /*
238                                          * Can potentially get here for
239                                          * v6 socket if another process
240                                          * (say, ping) has just done a
241                                          * sendto(), changing the state
242                                          * from the TS_IDLE above to
243                                          * TS_DATA_XFER by the time we hit
244                                          * this part of the code.
245                                          */
246                                         ude.udpEntryInfo.ue_RemoteAddress =
247                                             connp->conn_faddr_v4;
248                                         ude.udpEntryInfo.ue_RemotePort =
249                                             ntohs(connp->conn_fport);
250                                 } else {
251                                         ude.udpEntryInfo.ue_RemoteAddress = 0;
252                                         ude.udpEntryInfo.ue_RemotePort = 0;
253                                 }

255                                 /*
256                                  * We make the assumption that all udp_t
257                                  * structs will be created within an address
```

```
258                                  * region no larger than 32-bits.
259                                  */
260                                 ude.udpInstance = (uint32_t)(uintptr_t)udp;
261                                 ude.udpCreationProcess =
262                                     (connp->conn_cpid < 0) ?
263                                     MIB2_UNKNOWN_PROCESS :
264                                     connp->conn_cpid;
265                                 ude.udpCreationTime = connp->conn_open_time;

267                                 (void) snmp_append_data2(mp_conn_ctl->b_cont,
268                                     &mp_conn_tail, (char *)&ude, ude_size);

270                                 (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
271                                     &mp_pidnode_tail, (char *)&ude, ude_size);

273                                 (void) snmp_append_mblk2(mp_pidnode_ctl->b_cont,
274                                     &mp_pidnode_tail, conn_get_pid_mblk(connp));

276 #endif /* ! codereview */
277                                 mlp.tme_connidx = v4_conn_idx++;
278                                 if (needattr)
279                                         (void) snmp_append_data2(
280                                             mp_attr_ctl->b_cont, &mp_attr_tail,
281                                             (char *)&mlp, sizeof (mlp));
282                         }
283                         if (connp->conn_ipversion == IPV6_VERSION) {
284                                 ude6.udp6EntryInfo.ue_state  = state;
285                                 ude6.udp6LocalAddress = connp->conn_laddr_v6;
286                                 ude6.udp6LocalPort = ntohs(connp->conn_lport);
287                                 mutex_enter(&connp->conn_lock);
288                                 if (connp->conn_ixa->ixa_flags &
289                                     IXAF_SCOPEID_SET) {
290                                         ude6.udp6IfIndex =
291                                             connp->conn_ixa->ixa_scopeid;
292                                 } else {
293                                         ude6.udp6IfIndex = connp->conn_bound_if;
294                                 }
295                                 mutex_exit(&connp->conn_lock);
296                                 if (udp->udp_state == TS_DATA_XFER) {
297                                         ude6.udp6EntryInfo.ue_RemoteAddress =
298                                             connp->conn_faddr_v6;
299                                         ude6.udp6EntryInfo.ue_RemotePort =
300                                             ntohs(connp->conn_fport);
301                                 } else {
302                                         ude6.udp6EntryInfo.ue_RemoteAddress =
303                                             sin6_null.sin6_addr;
304                                         ude6.udp6EntryInfo.ue_RemotePort = 0;
305                                 }
306                                 /*
307                                  * We make the assumption that all udp_t
308                                  * structs will be created within an address
309                                  * region no larger than 32-bits.
310                                  */
311                                 ude6.udp6Instance = (uint32_t)(uintptr_t)udp;
312                                 ude6.udp6CreationProcess =
313                                     (connp->conn_cpid < 0) ?
314                                     MIB2_UNKNOWN_PROCESS :
315                                     connp->conn_cpid;
316                                 ude6.udp6CreationTime = connp->conn_open_time;

318                                 (void) snmp_append_data2(mp6_conn_ctl->b_cont,
319                                     &mp6_conn_tail, (char *)&ude6, ude6_size);

321                                 (void) snmp_append_data2(
322                                     mp6_pidnode_ctl->b_cont, &mp6_pidnode_tail,
323                                     (char *)&ude6, ude6_size);
```



```
325                                 (void) snmp_append_mblk2(
326                                     mp6_pidnode_ctl->b_cont, &mp6_pidnode_tail,
327                                     conn_get_pid_mblk(connp));

329 #endif /* ! codereview */
330                                 mlp.tme_connidx = v6_conn_idx++;
331                                 if (needattr)
332                                         (void) snmp_append_data2(
333                                             mp6_attr_ctl->b_cont,
334                                             &mp6_attr_tail, (char *)&mlp,
335                                             sizeof (mlp));
336                         }
337                 }
338         }

340         /* IPv4 UDP endpoints */
341         optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
342             sizeof (struct T_optmgmt_ack)];
343         optp->level = MIB2_UDP;
344         optp->name = MIB2_UDP_ENTRY;
345         optp->len = msgdsize(mp_conn_ctl->b_cont);
346         qreply(q, mp_conn_ctl);

348         /* table of MLP attributes... */
349         optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
350             sizeof (struct T_optmgmt_ack)];
351         optp->level = MIB2_UDP;
352         optp->name = EXPER_XPORT_MLP;
353         optp->len = msgdsize(mp_attr_ctl->b_cont);
354         if (optp->len == 0)
355                 freemsg(mp_attr_ctl);
356         else
357                 qreply(q, mp_attr_ctl);

359         /* table of EXPER_XPORT_PROC_INFO ipv4 */
360         optp = (struct opthdr *)&mp_pidnode_ctl->b_rptr[
361             sizeof (struct T_optmgmt_ack)];
362         optp->level = MIB2_UDP;
363         optp->name = EXPER_XPORT_PROC_INFO;
364         optp->len = msgdsize(mp_pidnode_ctl->b_cont);
365         if (optp->len == 0)
366                 freemsg(mp_pidnode_ctl);
367         else
368                 qreply(q, mp_pidnode_ctl);

370 #endif /* ! codereview */
371         /* IPv6 UDP endpoints */
372         optp = (struct opthdr *)&mp6_conn_ctl->b_rptr[
373             sizeof (struct T_optmgmt_ack)];
374         optp->level = MIB2_UDP6;
375         optp->name = MIB2_UDP6_ENTRY;
376         optp->len = msgdsize(mp6_conn_ctl->b_cont);
377         qreply(q, mp6_conn_ctl);

379         /* table of MLP attributes... */
380         optp = (struct opthdr *)&mp6_attr_ctl->b_rptr[
381             sizeof (struct T_optmgmt_ack)];
382         optp->level = MIB2_UDP6;
383         optp->name = EXPER_XPORT_MLP;
384         optp->len = msgdsize(mp6_attr_ctl->b_cont);
385         if (optp->len == 0)
386                 freemsg(mp6_attr_ctl);
387         else
388                 qreply(q, mp6_attr_ctl);
```

```
390         /* table of EXPER_XPORT_PROC_INFO ipv6 */
391         optp = (struct opthdr *)&mp6_pidnode_ctl->b_rptr[
392             sizeof (struct T_optmgmt_ack)];
393         optp->level = MIB2_UDP6;
394         optp->name = EXPER_XPORT_PROC_INFO;
395         optp->len = msgdsize(mp6_pidnode_ctl->b_cont);
396         if (optp->len == 0)
397                 freemsg(mp6_pidnode_ctl);
398         else
399                 qreply(q, mp6_pidnode_ctl);
400 #endif /* ! codereview */

402         return (mp2ctl);
403 }

405 /*
406  * Return 0 if invalid set request, 1 otherwise, including non-udp requests.
407  * NOTE: Per MIB-II, UDP has no writable data.
408  * TODO:  If this ever actually tries to set anything, it needs to be
409  * to do the appropriate locking.
410  */
411 /* ARGSUSED */
412 int
413 udp_snmp_set(queue_t *q, t_scalar_t level, t_scalar_t name,
414     uchar_t *ptr, int len)
415 {
416         switch (level) {
417         case MIB2_UDP:
418                 return (0);
419         default:
420                 return (1);
421         }
422 }

424 void
425 udp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
426 {
427         if (ksp != NULL) {
428                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
429                 kstat_delete_netstack(ksp, stackid);
430         }
431 }

433 /*
434  * To add stats from one mib2_udp_t to another.  Static fields are not added.
435  * The caller should set them up propertly.
436  */
437 static void
438 udp_add_mib(mib2_udp_t *from, mib2_udp_t *to)
439 {
440         to->udpHCInDatagrams += from->udpHCInDatagrams;
441         to->udpInErrors += from->udpInErrors;
442         to->udpHCOutDatagrams += from->udpHCOutDatagrams;
443         to->udpOutErrors += from->udpOutErrors;
444 }


447 void *
448 udp_kstat2_init(netstackid_t stackid)
449 {
450         kstat_t *ksp;

452         udp_stat_t template = {
453                 { "udp_sock_fallback",          KSTAT_DATA_UINT64 },
454                 { "udp_out_opt",                KSTAT_DATA_UINT64 },
455                 { "udp_out_err_notconn",        KSTAT_DATA_UINT64 },
```



```
456                 { "udp_out_err_output",         KSTAT_DATA_UINT64 },
457                 { "udp_out_err_tudr",           KSTAT_DATA_UINT64 },
458 #ifdef DEBUG
459                 { "udp_data_conn",              KSTAT_DATA_UINT64 },
460                 { "udp_data_notconn",           KSTAT_DATA_UINT64 },
461                 { "udp_out_lastdst",            KSTAT_DATA_UINT64 },
462                 { "udp_out_diffdst",            KSTAT_DATA_UINT64 },
463                 { "udp_out_ipv6",               KSTAT_DATA_UINT64 },
464                 { "udp_out_mapped",             KSTAT_DATA_UINT64 },
465                 { "udp_out_ipv4",               KSTAT_DATA_UINT64 },
466 #endif
467         };

469         ksp = kstat_create_netstack(UDP_MOD_NAME, 0, "udpstat", "net",
470             KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t),
471             0, stackid);

473         if (ksp == NULL)
474                 return (NULL);

476         bcopy(&template, ksp->ks_data, sizeof (template));
477         ksp->ks_update = udp_kstat2_update;
478         ksp->ks_private = (void *)(uintptr_t)stackid;

480         kstat_install(ksp);
481         return (ksp);
482 }

484 void
485 udp_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
486 {
487         if (ksp != NULL) {
488                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
489                 kstat_delete_netstack(ksp, stackid);
490         }
491 }

493 /*
494  * To copy counters from the per CPU udpp_stat_counter_t to the stack
495  * udp_stat_t.
496  */
497 static void
498 udp_add_stats(udp_stat_counter_t *from, udp_stat_t *to)
499 {
500         to->udp_sock_fallback.value.ui64 += from->udp_sock_fallback;
501         to->udp_out_opt.value.ui64 += from->udp_out_opt;
502         to->udp_out_err_notconn.value.ui64 += from->udp_out_err_notconn;
503         to->udp_out_err_output.value.ui64 += from->udp_out_err_output;
504         to->udp_out_err_tudr.value.ui64 += from->udp_out_err_tudr;
505 #ifdef DEBUG
506         to->udp_data_conn.value.ui64 += from->udp_data_conn;
507         to->udp_data_notconn.value.ui64 += from->udp_data_notconn;
508         to->udp_out_lastdst.value.ui64 += from->udp_out_lastdst;
509         to->udp_out_diffdst.value.ui64 += from->udp_out_diffdst;
510         to->udp_out_ipv6.value.ui64 += from->udp_out_ipv6;
511         to->udp_out_mapped.value.ui64 += from->udp_out_mapped;
512         to->udp_out_ipv4.value.ui64 += from->udp_out_ipv4;
513 #endif
514 }

516 /*
517  * To set all udp_stat_t counters to 0.
518  */
519 static void
520 udp_clr_stats(udp_stat_t *stats)
521 {
```

```
522         stats->udp_sock_fallback.value.ui64 = 0;
523         stats->udp_out_opt.value.ui64 = 0;
524         stats->udp_out_err_notconn.value.ui64 = 0;
525         stats->udp_out_err_output.value.ui64 = 0;
526         stats->udp_out_err_tudr.value.ui64 = 0;
527 #ifdef DEBUG
528         stats->udp_data_conn.value.ui64 = 0;
529         stats->udp_data_notconn.value.ui64 = 0;
530         stats->udp_out_lastdst.value.ui64 = 0;
531         stats->udp_out_diffdst.value.ui64 = 0;
532         stats->udp_out_ipv6.value.ui64 = 0;
533         stats->udp_out_mapped.value.ui64 = 0;
534         stats->udp_out_ipv4.value.ui64 = 0;
535 #endif
536 }

538 int
539 udp_kstat2_update(kstat_t *kp, int rw)
540 {
541         udp_stat_t      *stats;
542         netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
543         netstack_t      *ns;
544         udp_stack_t     *us;
545         int             i;
546         int             cnt;

548         if (rw == KSTAT_WRITE)
549                 return (EACCES);

551         ns = netstack_find_by_stackid(stackid);
552         if (ns == NULL)
553                 return (-1);
554         us = ns->netstack_udp;
555         if (us == NULL) {
556                 netstack_rele(ns);
557                 return (-1);
558         }
559         stats = (udp_stat_t *)kp->ks_data;
560         udp_clr_stats(stats);

562         cnt = us->us_sc_cnt;
563         for (i = 0; i < cnt; i++)
564                 udp_add_stats(&us->us_sc[i]->udp_sc_stats, stats);

566         netstack_rele(ns);
567         return (0);
568 }

570 void *
571 udp_kstat_init(netstackid_t stackid)
572 {
573         kstat_t *ksp;

575         udp_named_kstat_t template = {
576                 { "inDatagrams",        KSTAT_DATA_UINT64, 0 },
577                 { "inErrors",           KSTAT_DATA_UINT32, 0 },
578                 { "outDatagrams",       KSTAT_DATA_UINT64, 0 },
579                 { "entrySize",          KSTAT_DATA_INT32, 0 },
580                 { "entry6Size",         KSTAT_DATA_INT32, 0 },
581                 { "outErrors",          KSTAT_DATA_UINT32, 0 },
582         };

584         ksp = kstat_create_netstack(UDP_MOD_NAME, 0, UDP_MOD_NAME, "mib2",
585             KSTAT_TYPE_NAMED, NUM_OF_FIELDS(udp_named_kstat_t), 0, stackid);

587         if (ksp == NULL)
```



```
588                 return (NULL);

590         template.entrySize.value.ui32 = sizeof (mib2_udpEntry_t);
591         template.entry6Size.value.ui32 = sizeof (mib2_udp6Entry_t);

593         bcopy(&template, ksp->ks_data, sizeof (template));
594         ksp->ks_update = udp_kstat_update;
595         ksp->ks_private = (void *)(uintptr_t)stackid;

597         kstat_install(ksp);
598         return (ksp);
599 }

601 /*
602  * To sum up all MIB2 stats for a udp_stack_t from all per CPU stats.  The
603  * caller should initialize the target mib2_udp_t properly as this function
604  * just adds up all the per CPU stats.
605  */
606 static void
607 udp_sum_mib(udp_stack_t *us, mib2_udp_t *udp_mib)
608 {
609         int i;
610         int cnt;

612         cnt = us->us_sc_cnt;
613         for (i = 0; i < cnt; i++)
614                 udp_add_mib(&us->us_sc[i]->udp_sc_mib, udp_mib);
615 }

617 static int
618 udp_kstat_update(kstat_t *kp, int rw)
619 {
620         udp_named_kstat_t *udpkp;
621         netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
622         netstack_t      *ns;
623         udp_stack_t     *us;
624         mib2_udp_t      udp_mib;

626         if (rw == KSTAT_WRITE)
627                 return (EACCES);

629         ns = netstack_find_by_stackid(stackid);
630         if (ns == NULL)
631                 return (-1);
632         us = ns->netstack_udp;
633         if (us == NULL) {
634                 netstack_rele(ns);
635                 return (-1);
636         }
637         udpkp = (udp_named_kstat_t *)kp->ks_data;

639         bzero(&udp_mib, sizeof (udp_mib));
640         udp_sum_mib(us, &udp_mib);

642         udpkp->inDatagrams.value.ui64 = udp_mib.udpHCInDatagrams;
643         udpkp->inErrors.value.ui32 =    udp_mib.udpInErrors;
644         udpkp->outDatagrams.value.ui64 = udp_mib.udpHCOutDatagrams;
645         udpkp->outErrors.value.ui32 =   udp_mib.udpOutErrors;
646         netstack_rele(ns);
647         return (0);
648 }
```

```
**********************************************************
   47146 Mon Aug 17 21:08:07 2015
new/usr/src/uts/common/os/fio.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 836 /*
 837  * Duplicate all file descriptors across a fork.
 838  */
 839 void
 840 flist_fork(proc_t *pp, proc_t *cp)
 840 flist_fork(uf_info_t *pfip, uf_info_t *cfip)
 841 {
 842         int fd, nfiles;
 843         uf_entry_t *pufp, *cufp;

 845         uf_info_t *pfip = P_FINFO(pp);
 846         uf_info_t *cfip = P_FINFO(cp);

 848 #endif /* ! codereview */
 849         mutex_init(&cfip->fi_lock, NULL, MUTEX_DEFAULT, NULL);
 850         cfip->fi_rlist = NULL;

 852         /*
 853          * We don't need to hold fi_lock because all other lwp's in the
 854          * parent have been held.
 855          */
 856         cfip->fi_nfiles = nfiles = flist_minsize(pfip);

 858         cfip->fi_list = kmem_zalloc(nfiles * sizeof (uf_entry_t), KM_SLEEP);

 860         for (fd = 0, pufp = pfip->fi_list, cufp = cfip->fi_list; fd < nfiles;
 861             fd++, pufp++, cufp++) {
 862                 cufp->uf_file = pufp->uf_file;
 863                 cufp->uf_alloc = pufp->uf_alloc;
 864                 cufp->uf_flag = pufp->uf_flag;
 865                 cufp->uf_busy = pufp->uf_busy;

 867                 if (cufp->uf_file != NULL && cufp->uf_file->f_vnode != NULL) {
 868                         (void) VOP_IOCTL(cufp->uf_file->f_vnode, F_ASSOCI_PID,
 869                             (intptr_t)cp->p_pidp->pid_id, FKIOCTL, kcred,
 870                             NULL, NULL);
 871                 }

 873 #endif /* ! codereview */
 874                 if (pufp->uf_file == NULL) {
 875                         ASSERT(pufp->uf_flag == 0);
 876                         if (pufp->uf_busy) {
 877                                 /*
 878                                  * Grab locks to appease ASSERTs in fd_reserve
 879                                  */
 880                                 mutex_enter(&cfip->fi_lock);
 881                                 mutex_enter(&cufp->uf_lock);
 882                                 fd_reserve(cfip, fd, -1);
 883                                 mutex_exit(&cufp->uf_lock);
 884                                 mutex_exit(&cfip->fi_lock);
 885                         }
 886                 }
 887         }
 888 }

 890 /*
 891  * Close all open file descriptors for the current process.
 892  * This is only called from exit(), which is single-threaded,
 893  * so we don't need any locking.
```



```
 894  */
 895 void
 896 closeall(uf_info_t *fip)
 897 {
 898         int fd;
 899         file_t *fp;
 900         uf_entry_t *ufp;

 902         ufp = fip->fi_list;
 903         for (fd = 0; fd < fip->fi_nfiles; fd++, ufp++) {
 904                 if ((fp = ufp->uf_file) != NULL) {
 905                         ufp->uf_file = NULL;
 906                         if (ufp->uf_portfd != NULL) {
 907                                 portfd_t *pfd;
 908                                 /* remove event port association */
 909                                 pfd = ufp->uf_portfd;
 910                                 ufp->uf_portfd = NULL;
 911                                 port_close_fd(pfd);
 912                         }
 913                         ASSERT(ufp->uf_fpollinfo == NULL);
 914                         (void) closef(fp);
 915                 }
 916         }

 918         kmem_free(fip->fi_list, fip->fi_nfiles * sizeof (uf_entry_t));
 919         fip->fi_list = NULL;
 920         fip->fi_nfiles = 0;
 921         while (fip->fi_rlist != NULL) {
 922                 uf_rlist_t *urp = fip->fi_rlist;
 923                 fip->fi_rlist = urp->ur_next;
 924                 kmem_free(urp->ur_list, urp->ur_nfiles * sizeof (uf_entry_t));
 925                 kmem_free(urp, sizeof (uf_rlist_t));
 926         }
 927 }

 929 /*
 930  * Internal form of close.  Decrement reference count on file
 931  * structure.  Decrement reference count on the vnode following
 932  * removal of the referencing file structure.
 933  */
 934 int
 935 closef(file_t *fp)
 936 {
 937         vnode_t *vp;
 938         int error;
 939         int count;
 940         int flag;
 941         offset_t offset;

 943         /*
 944          * audit close of file (may be exit)
 945          */
 946         if (AU_AUDITING())
 947                 audit_closef(fp);
 948         ASSERT(MUTEX_NOT_HELD(&P_FINFO(curproc)->fi_lock));

 950         mutex_enter(&fp->f_tlock);

 952         ASSERT(fp->f_count > 0);

 954         count = fp->f_count--;
 955         flag = fp->f_flag;
 956         offset = fp->f_offset;

 958         vp = fp->f_vnode;
 959         if (vp != NULL) {
```

```
960                 (void) VOP_IOCTL(vp, F_DASSOC_PID,
961                     (intptr_t)(ttoproc(curthread)->p_pidp->pid_id), FKIOCTL,
962                     kcred, NULL, NULL);
963         }
964 #endif /* ! codereview */

966         error = VOP_CLOSE(vp, flag, count, offset, fp->f_cred, NULL);

968         if (count > 1) {
969                 mutex_exit(&fp->f_tlock);
970                 return (error);
971         }
972         ASSERT(fp->f_count == 0);
973         mutex_exit(&fp->f_tlock);

975         /*
976          * If DTrace has getf() subroutines active, it will set dtrace_closef
977          * to point to code that implements a barrier with respect to probe
978          * context.  This must be called before the file_t is freed (and the
979          * vnode that it refers to is released) -- but it must be after the
980          * file_t has been removed from the uf_entry_t.  That is, there must
981          * be no way for a racing getf() in probe context to yield the fp that
982          * we're operating upon.
983          */
984         if (dtrace_closef != NULL)
985                 (*dtrace_closef)();

987         VN_RELE(vp);
988         /*
989          * deallocate resources to audit_data
990          */
991         if (audit_active)
992                 audit_unfalloc(fp);
993         crfree(fp->f_cred);
994         kmem_cache_free(file_cache, fp);
995         return (error);
996 }

998 /*
999  * This is a combination of ufalloc() and setf().
1000  */
1001 int
1002 ufalloc_file(int start, file_t *fp)
1003 {
1004         proc_t *p = curproc;
1005         uf_info_t *fip = P_FINFO(p);
1006         int filelimit;
1007         uf_entry_t *ufp;
1008         int nfiles;
1009         int fd;

1011         /*
1012          * Assertion is to convince the correctness of the following
1013          * assignment for filelimit after casting to int.
1014          */
1015         ASSERT(p->p_fno_ctl <= INT_MAX);
1016         filelimit = (int)p->p_fno_ctl;

1018         for (;;) {
1019                 mutex_enter(&fip->fi_lock);
1020                 fd = fd_find(fip, start);
1021                 if (fd >= 0 && fd == fip->fi_badfd) {
1022                         start = fd + 1;
1023                         mutex_exit(&fip->fi_lock);
1024                         continue;
1025                 }
```



```
1026                 if ((uint_t)fd < filelimit)
1027                         break;
1028                 if (fd >= filelimit) {
1029                         mutex_exit(&fip->fi_lock);
1030                         mutex_enter(&p->p_lock);
1031                         (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1032                             p->p_rctls, p, RCA_SAFE);
1033                         mutex_exit(&p->p_lock);
1034                         return (-1);
1035                 }
1036                 /* fd_find() returned -1 */
1037                 nfiles = fip->fi_nfiles;
1038                 mutex_exit(&fip->fi_lock);
1039                 flist_grow(MAX(start, nfiles));
1040         }

1042         UF_ENTER(ufp, fip, fd);
1043         fd_reserve(fip, fd, 1);
1044         ASSERT(ufp->uf_file == NULL);
1045         ufp->uf_file = fp;
1046         UF_EXIT(ufp);
1047         mutex_exit(&fip->fi_lock);
1048         return (fd);
1049 }

1051 /*
1052  * Allocate a user file descriptor greater than or equal to "start".
1053  */
1054 int
1055 ufalloc(int start)
1056 {
1057         return (ufalloc_file(start, NULL));
1058 }

1060 /*
1061  * Check that a future allocation of count fds on proc p has a good
1062  * chance of succeeding.  If not, do rctl processing as if we'd failed
1063  * the allocation.
1064  *
1065  * Our caller must guarantee that p cannot disappear underneath us.
1066  */
1067 int
1068 ufcanalloc(proc_t *p, uint_t count)
1069 {
1070         uf_info_t *fip = P_FINFO(p);
1071         int filelimit;
1072         int current;

1074         if (count == 0)
1075                 return (1);

1077         ASSERT(p->p_fno_ctl <= INT_MAX);
1078         filelimit = (int)p->p_fno_ctl;

1080         mutex_enter(&fip->fi_lock);
1081         current = flist_nalloc(fip);            /* # of in-use descriptors */
1082         mutex_exit(&fip->fi_lock);

1084         /*
1085          * If count is a positive integer, the worst that can happen is
1086          * an overflow to a negative value, which is caught by the >= 0 check.
1087          */
1088         current += count;
1089         if (count <= INT_MAX && current >= 0 && current <= filelimit)
1090                 return (1);
```

```
1092         mutex_enter(&p->p_lock);
1093         (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1094             p->p_rctls, p, RCA_SAFE);
1095         mutex_exit(&p->p_lock);
1096         return (0);
1097 }

1099 /*
1100  * Allocate a user file descriptor and a file structure.
1101  * Initialize the descriptor to point at the file structure.
1102  * If fdp is NULL, the user file descriptor will not be allocated.
1103  */
1104 int
1105 falloc(vnode_t *vp, int flag, file_t **fpp, int *fdp)
1106 {
1107         file_t *fp;
1108         int fd;

1110         if (fdp) {
1111                 if ((fd = ufalloc(0)) == -1)
1112                         return (EMFILE);
1113         }
1114         fp = kmem_cache_alloc(file_cache, KM_SLEEP);
1115         /*
1116          * Note: falloc returns the fp locked
1117          */
1118         mutex_enter(&fp->f_tlock);
1119         fp->f_count = 1;
1120         fp->f_flag = (ushort_t)flag;
1121         fp->f_flag2 = (flag & (FSEARCH|FEXEC)) >> 16;
1122         fp->f_vnode = vp;
1123         fp->f_offset = 0;
1124         fp->f_audit_data = 0;
1125         crhold(fp->f_cred = CRED());
1126         /*
1127          * allocate resources to audit_data
1128          */
1129         if (audit_active)
1130                 audit_falloc(fp);
1131         *fpp = fp;
1132         if (fdp)
1133                 *fdp = fd;
1134         return (0);
1135 }

1137 /*ARGSUSED*/
1138 static int
1139 file_cache_constructor(void *buf, void *cdrarg, int kmflags)
1140 {
1141         file_t *fp = buf;

1143         mutex_init(&fp->f_tlock, NULL, MUTEX_DEFAULT, NULL);
1144         return (0);
1145 }

1147 /*ARGSUSED*/
1148 static void
1149 file_cache_destructor(void *buf, void *cdrarg)
1150 {
1151         file_t *fp = buf;

1153         mutex_destroy(&fp->f_tlock);
1154 }

1156 void
1157 finit()
```



```
1158 {
1159         file_cache = kmem_cache_create("file_cache", sizeof (file_t), 0,
1160             file_cache_constructor, file_cache_destructor, NULL, NULL, NULL, 0);
1161 }

1163 void
1164 unfalloc(file_t *fp)
1165 {
1166         ASSERT(MUTEX_HELD(&fp->f_tlock));
1167         if (--fp->f_count <= 0) {
1168                 /*
1169                  * deallocate resources to audit_data
1170                  */
1171                 if (audit_active)
1172                         audit_unfalloc(fp);
1173                 crfree(fp->f_cred);
1174                 mutex_exit(&fp->f_tlock);
1175                 kmem_cache_free(file_cache, fp);
1176         } else
1177                 mutex_exit(&fp->f_tlock);
1178 }

1180 /*
1181  * Given a file descriptor, set the user's
1182  * file pointer to the given parameter.
1183  */
1184 void
1185 setf(int fd, file_t *fp)
1186 {
1187         uf_info_t *fip = P_FINFO(curproc);
1188         uf_entry_t *ufp;

1190         if (AU_AUDITING())
1191                 audit_setf(fp, fd);

1193         if (fp == NULL) {
1194                 mutex_enter(&fip->fi_lock);
1195                 UF_ENTER(ufp, fip, fd);
1196                 fd_reserve(fip, fd, -1);
1197                 mutex_exit(&fip->fi_lock);
1198         } else {
1199                 UF_ENTER(ufp, fip, fd);
1200                 ASSERT(ufp->uf_busy);
1201         }
1202         ASSERT(ufp->uf_fpollinfo == NULL);
1203         ASSERT(ufp->uf_flag == 0);
1204         ufp->uf_file = fp;
1205         cv_broadcast(&ufp->uf_wanted_cv);
1206         UF_EXIT(ufp);
1207 }

1209 /*
1210  * Given a file descriptor, return the file table flags, plus,
1211  * if this is a socket in asynchronous mode, the FASYNC flag.
1212  * getf() may or may not have been called before calling f_getfl().
1213  */
1214 int
1215 f_getfl(int fd, int *flagp)
1216 {
1217         uf_info_t *fip = P_FINFO(curproc);
1218         uf_entry_t *ufp;
1219         file_t *fp;
1220         int error;

1222         if ((uint_t)fd >= fip->fi_nfiles)
1223                 error = EBADF;
```

```
1224         else {
1225                 UF_ENTER(ufp, fip, fd);
1226                 if ((fp = ufp->uf_file) == NULL)
1227                         error = EBADF;
1228                 else {
1229                         vnode_t *vp = fp->f_vnode;
1230                         int flag = fp->f_flag | (fp->f_flag2 << 16);

1232                         /*
1233                          * BSD fcntl() FASYNC compatibility.
1234                          */
1235                         if (vp->v_type == VSOCK)
1236                                 flag |= sock_getfasync(vp);
1237                         *flagp = flag;
1238                         error = 0;
1239                 }
1240                 UF_EXIT(ufp);
1241         }

1243         return (error);
1244 }

1246 /*
1247  * Given a file descriptor, return the user's file flags.
1248  * Force the FD_CLOEXEC flag for writable self-open /proc files.
1249  * getf() may or may not have been called before calling f_getfd_error().
1250  */
1251 int
1252 f_getfd_error(int fd, int *flagp)
1253 {
1254         uf_info_t *fip = P_FINFO(curproc);
1255         uf_entry_t *ufp;
1256         file_t *fp;
1257         int flag;
1258         int error;

1260         if ((uint_t)fd >= fip->fi_nfiles)
1261                 error = EBADF;
1262         else {
1263                 UF_ENTER(ufp, fip, fd);
1264                 if ((fp = ufp->uf_file) == NULL)
1265                         error = EBADF;
1266                 else {
1267                         flag = ufp->uf_flag;
1268                         if ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode))
1269                                 flag |= FD_CLOEXEC;
1270                         *flagp = flag;
1271                         error = 0;
1272                 }
1273                 UF_EXIT(ufp);
1274         }

1276         return (error);
1277 }

1279 /*
1280  * getf() must have been called before calling f_getfd().
1281  */
1282 char
1283 f_getfd(int fd)
1284 {
1285         int flag = 0;
1286         (void) f_getfd_error(fd, &flag);
1287         return ((char)flag);
1288 }
```



```
1290 /*
1291  * Given a file descriptor and file flags, set the user's file flags.
1292  * At present, the only valid flag is FD_CLOEXEC.
1293  * getf() may or may not have been called before calling f_setfd_error().
1294  */
1295 int
1296 f_setfd_error(int fd, int flags)
1297 {
1298         uf_info_t *fip = P_FINFO(curproc);
1299         uf_entry_t *ufp;
1300         int error;

1302         if ((uint_t)fd >= fip->fi_nfiles)
1303                 error = EBADF;
1304         else {
1305                 UF_ENTER(ufp, fip, fd);
1306                 if (ufp->uf_file == NULL)
1307                         error = EBADF;
1308                 else {
1309                         ufp->uf_flag = flags & FD_CLOEXEC;
1310                         error = 0;
1311                 }
1312                 UF_EXIT(ufp);
1313         }
1314         return (error);
1315 }

1317 void
1318 f_setfd(int fd, char flags)
1319 {
1320         (void) f_setfd_error(fd, flags);
1321 }

1323 #define BADFD_MIN       3
1324 #define BADFD_MAX       255

1326 /*
1327  * Attempt to allocate a file descriptor which is bad and which
1328  * is "poison" to the application.  It cannot be closed (except
1329  * on exec), allocated for a different use, etc.
1330  */
1331 int
1332 f_badfd(int start, int *fdp, int action)
1333 {
1334         int fdr;
1335         int badfd;
1336         uf_info_t *fip = P_FINFO(curproc);

1338 #ifdef _LP64
1339         /* No restrictions on 64 bit _file */
1340         if (get_udatamodel() != DATAMODEL_ILP32)
1341                 return (EINVAL);
1342 #endif

1344         if (start > BADFD_MAX || start < BADFD_MIN)
1345                 return (EINVAL);

1347         if (action >= NSIG || action < 0)
1348                 return (EINVAL);

1350         mutex_enter(&fip->fi_lock);
1351         badfd = fip->fi_badfd;
1352         mutex_exit(&fip->fi_lock);

1354         if (badfd != -1)
1355                 return (EAGAIN);
```

```
1357         fdr = ufalloc(start);

1359         if (fdr > BADFD_MAX) {
1360                 setf(fdr, NULL);
1361                 return (EMFILE);
1362         }
1363         if (fdr < 0)
1364                 return (EMFILE);

1366         mutex_enter(&fip->fi_lock);
1367         if (fip->fi_badfd != -1) {
1368                 /* Lost race */
1369                 mutex_exit(&fip->fi_lock);
1370                 setf(fdr, NULL);
1371                 return (EAGAIN);
1372         }
1373         fip->fi_action = action;
1374         fip->fi_badfd = fdr;
1375         mutex_exit(&fip->fi_lock);
1376         setf(fdr, NULL);

1378         *fdp = fdr;

1380         return (0);
1381 }

1383 /*
1384  * Allocate a file descriptor and assign it to the vnode "*vpp",
1385  * performing the usual open protocol upon it and returning the
1386  * file descriptor allocated.  It is the responsibility of the
1387  * caller to dispose of "*vpp" if any error occurs.
1388  */
1389 int
1390 fassign(vnode_t **vpp, int mode, int *fdp)
1391 {
1392         file_t *fp;
1393         int error;
1394         int fd;

1396         if (error = falloc((vnode_t *)NULL, mode, &fp, &fd))
1397                 return (error);
1398         if (error = VOP_OPEN(vpp, mode, fp->f_cred, NULL)) {
1399                 setf(fd, NULL);
1400                 unfalloc(fp);
1401                 return (error);
1402         }
1403         fp->f_vnode = *vpp;
1404         mutex_exit(&fp->f_tlock);
1405         /*
1406          * Fill in the slot falloc reserved.
1407          */
1408         setf(fd, fp);
1409         *fdp = fd;
1410         return (0);
1411 }

1413 /*
1414  * When a process forks it must increment the f_count of all file pointers
1415  * since there is a new process pointing at them.  fcnt_add(fip, 1) does this.
1416  * Since we are called when there is only 1 active lwp we don't need to
1417  * hold fi_lock or any uf_lock.  If the fork fails, fork_fail() calls
1418  * fcnt_add(fip, -1) to restore the counts.
1419  */
1420 void
1421 fcnt_add(uf_info_t *fip, int incr)
```



```
1422 {
1423         int i;
1424         uf_entry_t *ufp;
1425         file_t *fp;

1427         ufp = fip->fi_list;
1428         for (i = 0; i < fip->fi_nfiles; i++, ufp++) {
1429                 if ((fp = ufp->uf_file) != NULL) {
1430                         mutex_enter(&fp->f_tlock);
1431                         ASSERT((incr == 1 && fp->f_count >= 1) ||
1432                             (incr == -1 && fp->f_count >= 2));
1433                         fp->f_count += incr;
1434                         mutex_exit(&fp->f_tlock);
1435                 }
1436         }
1437 }

1439 /*
1440  * This is called from exec to close all fd's that have the FD_CLOEXEC flag
1441  * set and also to close all self-open for write /proc file descriptors.
1442  */
1443 void
1444 close_exec(uf_info_t *fip)
1445 {
1446         int fd;
1447         file_t *fp;
1448         fpollinfo_t *fpip;
1449         uf_entry_t *ufp;
1450         portfd_t *pfd;

1452         ufp = fip->fi_list;
1453         for (fd = 0; fd < fip->fi_nfiles; fd++, ufp++) {
1454                 if ((fp = ufp->uf_file) != NULL &&
1455                     ((ufp->uf_flag & FD_CLOEXEC) ||
1456                     ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode)))) {
1457                         fpip = ufp->uf_fpollinfo;
1458                         mutex_enter(&fip->fi_lock);
1459                         mutex_enter(&ufp->uf_lock);
1460                         fd_reserve(fip, fd, -1);
1461                         mutex_exit(&fip->fi_lock);
1462                         ufp->uf_file = NULL;
1463                         ufp->uf_fpollinfo = NULL;
1464                         ufp->uf_flag = 0;
1465                         /*
1466                          * We may need to cleanup some cached poll states
1467                          * in t_pollstate before the fd can be reused. It
1468                          * is important that we don't access a stale thread
1469                          * structure. We will do the cleanup in two
1470                          * phases to avoid deadlock and holding uf_lock for
1471                          * too long. In phase 1, hold the uf_lock and call
1472                          * pollblockexit() to set state in t_pollstate struct
1473                          * so that a thread does not exit on us. In phase 2,
1474                          * we drop the uf_lock and call pollcacheclean().
1475                          */
1476                         pfd = ufp->uf_portfd;
1477                         ufp->uf_portfd = NULL;
1478                         if (fpip != NULL)
1479                                 pollblockexit(fpip);
1480                         mutex_exit(&ufp->uf_lock);
1481                         if (fpip != NULL)
1482                                 pollcacheclean(fpip, fd);
1483                         if (pfd)
1484                                 port_close_fd(pfd);
1485                         (void) closef(fp);
1486                 }
1487         }
```

```
1489         /* Reset bad fd */
1490         fip->fi_badfd = -1;
1491         fip->fi_action = -1;
1492 }

1494 /*
1495  * Utility function called by most of the *at() system call interfaces.
1496  *
1497  * Generate a starting vnode pointer for an (fd, path) pair where 'fd'
1498  * is an open file descriptor for a directory to be used as the starting
1499  * point for the lookup of the relative pathname 'path' (or, if path is
1500  * NULL, generate a vnode pointer for the direct target of the operation).
1501  *
1502  * If we successfully return a non-NULL startvp, it has been the target
1503  * of VN_HOLD() and the caller must call VN_RELE() on it.
1504  */
1505 int
1506 fgetstartvp(int fd, char *path, vnode_t **startvpp)
1507 {
1508         vnode_t         *startvp;
1509         file_t          *startfp;
1510         char            startchar;

1512         if (fd == AT_FDCWD && path == NULL)
1513                 return (EFAULT);

1515         if (fd == AT_FDCWD) {
1516                 /*
1517                  * Start from the current working directory.
1518                  */
1519                 startvp = NULL;
1520         } else {
1521                 if (path == NULL)
1522                         startchar = '\0';
1523                 else if (copyin(path, &startchar, sizeof (char)))
1524                         return (EFAULT);

1526                 if (startchar == '/') {
1527                         /*
1528                          * 'path' is an absolute pathname.
1529                          */
1530                         startvp = NULL;
1531                 } else {
1532                         /*
1533                          * 'path' is a relative pathname or we will
1534                          * be applying the operation to 'fd' itself.
1535                          */
1536                         if ((startfp = getf(fd)) == NULL)
1537                                 return (EBADF);
1538                         startvp = startfp->f_vnode;
1539                         VN_HOLD(startvp);
1540                         releasef(fd);
1541                 }
1542         }
1543         *startvpp = startvp;
1544         return (0);
1545 }

1547 /*
1548  * Called from fchownat() and fchmodat() to set ownership and mode.
1549  * The contents of *vap must be set before calling here.
1550  */
1551 int
1552 fsetattrat(int fd, char *path, int flags, struct vattr *vap)
1553 {
```



```
1554         vnode_t         *startvp;
1555         vnode_t         *vp;
1556         int             error;

1558         /*
1559          * Since we are never called to set the size of a file, we don't
1560          * need to check for non-blocking locks (via nbl_need_check(vp)).
1561          */
1562         ASSERT(!(vap->va_mask & AT_SIZE));

1564         if ((error = fgetstartvp(fd, path, &startvp)) != 0)
1565                 return (error);
1566         if (AU_AUDITING() && startvp != NULL)
1567                 audit_setfsat_path(1);

1569         /*
1570          * Do lookup for fchownat/fchmodat when path not NULL
1571          */
1572         if (path != NULL) {
1573                 if (error = lookupnameat(path, UIO_USERSPACE,
1574                     (flags == AT_SYMLINK_NOFOLLOW) ?
1575                     NO_FOLLOW : FOLLOW,
1576                     NULLVPP, &vp, startvp)) {
1577                         if (startvp != NULL)
1578                                 VN_RELE(startvp);
1579                         return (error);
1580                 }
1581         } else {
1582                 vp = startvp;
1583                 ASSERT(vp);
1584                 VN_HOLD(vp);
1585         }

1587         if (vn_is_readonly(vp)) {
1588                 error = EROFS;
1589         } else {
1590                 error = VOP_SETATTR(vp, vap, 0, CRED(), NULL);
1591         }

1593         if (startvp != NULL)
1594                 VN_RELE(startvp);
1595         VN_RELE(vp);

1597         return (error);
1598 }

1600 /*
1601  * Return true if the given vnode is referenced by any
1602  * entry in the current process's file descriptor table.
1603  */
1604 int
1605 fisopen(vnode_t *vp)
1606 {
1607         int fd;
1608         file_t *fp;
1609         vnode_t *ovp;
1610         uf_info_t *fip = P_FINFO(curproc);
1611         uf_entry_t *ufp;

1613         mutex_enter(&fip->fi_lock);
1614         for (fd = 0; fd < fip->fi_nfiles; fd++) {
1615                 UF_ENTER(ufp, fip, fd);
1616                 if ((fp = ufp->uf_file) != NULL &&
1617                     (ovp = fp->f_vnode) != NULL && VN_CMP(vp, ovp)) {
1618                         UF_EXIT(ufp);
1619                         mutex_exit(&fip->fi_lock);
```

```
1620                        return (1);
1621                }
1622                UF_EXIT(ufp);
1623        }
1624        mutex_exit(&fip->fi_lock);
1625        return (0);
1626 }

1628 /*
1629  * Return zero if at least one file currently open (by curproc) shouldn't be
1630  * allowed to change zones.
1631  */
1632 int
1633 files_can_change_zones(void)
1634 {
1635        int fd;
1636        file_t *fp;
1637        uf_info_t *fip = P_FINFO(curproc);
1638        uf_entry_t *ufp;

1640        mutex_enter(&fip->fi_lock);
1641        for (fd = 0; fd < fip->fi_nfiles; fd++) {
1642                UF_ENTER(ufp, fip, fd);
1643                if ((fp = ufp->uf_file) != NULL &&
1644                    !vn_can_change_zones(fp->f_vnode)) {
1645                        UF_EXIT(ufp);
1646                        mutex_exit(&fip->fi_lock);
1647                        return (0);
1648                }
1649                UF_EXIT(ufp);
1650        }
1651        mutex_exit(&fip->fi_lock);
1652        return (1);
1653 }

1655 #ifdef DEBUG

1657 /*
1658  * The following functions are only used in ASSERT()s elsewhere.
1659  * They do not modify the state of the system.
1660  */

1662 /*
1663  * Return true (1) if the current thread is in the fpollinfo
1664  * list for this file descriptor, else false (0).
1665  */
1666 static int
1667 curthread_in_plist(uf_entry_t *ufp)
1668 {
1669        fpollinfo_t *fpip;

1671        ASSERT(MUTEX_HELD(&ufp->uf_lock));
1672        for (fpip = ufp->uf_fpollinfo; fpip; fpip = fpip->fp_next)
1673                if (fpip->fp_thread == curthread)
1674                        return (1);
1675        return (0);
1676 }

1678 /*
1679  * Sanity check to make sure that after lwp_exit(),
1680  * curthread does not appear on any fd's fpollinfo list.
1681  */
1682 void
1683 checkfpollinfo(void)
1684 {
1685        int fd;
```



```
1686        uf_info_t *fip = P_FINFO(curproc);
1687        uf_entry_t *ufp;

1689        mutex_enter(&fip->fi_lock);
1690        for (fd = 0; fd < fip->fi_nfiles; fd++) {
1691                UF_ENTER(ufp, fip, fd);
1692                ASSERT(!curthread_in_plist(ufp));
1693                UF_EXIT(ufp);
1694        }
1695        mutex_exit(&fip->fi_lock);
1696 }

1698 /*
1699  * Return true (1) if the current thread is in the fpollinfo
1700  * list for this file descriptor, else false (0).
1701  * This is the same as curthread_in_plist(),
1702  * but is called w/o holding uf_lock.
1703  */
1704 int
1705 infpollinfo(int fd)
1706 {
1707        uf_info_t *fip = P_FINFO(curproc);
1708        uf_entry_t *ufp;
1709        int rc;

1711        UF_ENTER(ufp, fip, fd);
1712        rc = curthread_in_plist(ufp);
1713        UF_EXIT(ufp);
1714        return (rc);
1715 }

1717 #endif  /* DEBUG */

1719 /*
1720  * Add the curthread to fpollinfo list, meaning this fd is currently in the
1721  * thread's poll cache. Each lwp polling this file descriptor should call
1722  * this routine once.
1723  */
1724 void
1725 addfpollinfo(int fd)
1726 {
1727        struct uf_entry *ufp;
1728        fpollinfo_t *fpip;
1729        uf_info_t *fip = P_FINFO(curproc);

1731        fpip = kmem_zalloc(sizeof (fpollinfo_t), KM_SLEEP);
1732        fpip->fp_thread = curthread;
1733        UF_ENTER(ufp, fip, fd);
1734        /*
1735         * Assert we are not already on the list, that is, that
1736         * this lwp did not call addfpollinfo twice for the same fd.
1737         */
1738        ASSERT(!curthread_in_plist(ufp));
1739        /*
1740         * addfpollinfo is always done inside the getf/releasef pair.
1741         */
1742        ASSERT(ufp->uf_refcnt >= 1);
1743        fpip->fp_next = ufp->uf_fpollinfo;
1744        ufp->uf_fpollinfo = fpip;
1745        UF_EXIT(ufp);
1746 }

1748 /*
1749  * Delete curthread from fpollinfo list if it is there.
1750  */
1751 void
```

```
1752 delfpollinfo(int fd)
1753 {
1754         struct uf_entry *ufp;
1755         struct fpollinfo *fpip;
1756         struct fpollinfo **fpipp;
1757         uf_info_t *fip = P_FINFO(curproc);

1759         UF_ENTER(ufp, fip, fd);
1760         for (fpipp = &ufp->uf_fpollinfo;
1761             (fpip = *fpipp) != NULL;
1762             fpipp = &fpip->fp_next) {
1763                 if (fpip->fp_thread == curthread) {
1764                         *fpipp = fpip->fp_next;
1765                         kmem_free(fpip, sizeof (fpollinfo_t));
1766                         break;
1767                 }
1768         }
1769         /*
1770          * Assert that we are not still on the list, that is, that
1771          * this lwp did not call addfpollinfo twice for the same fd.
1772          */
1773         ASSERT(!curthread_in_plist(ufp));
1774         UF_EXIT(ufp);
1775 }

1777 /*
1778  * fd is associated with a port. pfd is a pointer to the fd entry in the
1779  * cache of the port.
1780  */

1782 void
1783 addfd_port(int fd, portfd_t *pfd)
1784 {
1785         struct uf_entry *ufp;
1786         uf_info_t *fip = P_FINFO(curproc);

1788         UF_ENTER(ufp, fip, fd);
1789         /*
1790          * addfd_port is always done inside the getf/releasef pair.
1791          */
1792         ASSERT(ufp->uf_refcnt >= 1);
1793         if (ufp->uf_portfd == NULL) {
1794                 /* first entry */
1795                 ufp->uf_portfd = pfd;
1796                 pfd->pfd_next = NULL;
1797         } else {
1798                 pfd->pfd_next = ufp->uf_portfd;
1799                 ufp->uf_portfd = pfd;
1800                 pfd->pfd_next->pfd_prev = pfd;
1801         }
1802         UF_EXIT(ufp);
1803 }

1805 void
1806 delfd_port(int fd, portfd_t *pfd)
1807 {
1808         struct uf_entry *ufp;
1809         uf_info_t *fip = P_FINFO(curproc);

1811         UF_ENTER(ufp, fip, fd);
1812         /*
1813          * delfd_port is always done inside the getf/releasef pair.
1814          */
1815         ASSERT(ufp->uf_refcnt >= 1);
1816         if (ufp->uf_portfd == pfd) {
1817                 /* remove first entry */
```



```
1818                 ufp->uf_portfd = pfd->pfd_next;
1819         } else {
1820                 pfd->pfd_prev->pfd_next = pfd->pfd_next;
1821                 if (pfd->pfd_next != NULL)
1822                         pfd->pfd_next->pfd_prev = pfd->pfd_prev;
1823         }
1824         UF_EXIT(ufp);
1825 }

1827 static void
1828 port_close_fd(portfd_t *pfd)
1829 {
1830         portfd_t        *pfdn;

1832         /*
1833          * At this point, no other thread should access
1834          * the portfd_t list for this fd. The uf_file, uf_portfd
1835          * pointers in the uf_entry_t struct for this fd would
1836          * be set to NULL.
1837          */
1838         for (; pfd != NULL; pfd = pfdn) {
1839                 pfdn = pfd->pfd_next;
1840                 port_close_pfd(pfd);
1841         }
1842 }
```

```
**********************************************************
   37125 Mon Aug 17 21:08:07 2015
new/usr/src/uts/common/os/fork.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 925 /*
 926  * create a child proc struct.
 927  */
 928 static int
 929 getproc(proc_t **cpp, pid_t pid, uint_t flags)
 930 {
 931         proc_t          *pp, *cp;
 932         pid_t           newpid;
 933         struct user     *uarea;
 934         extern uint_t   nproc;
 935         struct cred     *cr;
 936         uid_t           ruid;
 937         zoneid_t        zoneid;
 938         task_t          *task;
 939         kproject_t      *proj;
 940         zone_t          *zone;
 941         int             rctlfail = 0;

 943         if (zone_status_get(curproc->p_zone) >= ZONE_IS_SHUTTING_DOWN)
 944                 return (-1);    /* no point in starting new processes */

 946         pp = (flags & GETPROC_KERNEL) ? &p0 : curproc;
 947         task = pp->p_task;
 948         proj = task->tk_proj;
 949         zone = pp->p_zone;

 951         mutex_enter(&pp->p_lock);
 952         mutex_enter(&zone->zone_nlwps_lock);
 953         if (proj != proj0p) {
 954                 if (task->tk_nprocs >= task->tk_nprocs_ctl)
 955                         if (rctl_test(rc_task_nprocs, task->tk_rctls,
 956                             pp, 1, 0) & RCT_DENY)
 957                                 rctlfail = 1;

 959                 if (proj->kpj_nprocs >= proj->kpj_nprocs_ctl)
 960                         if (rctl_test(rc_project_nprocs, proj->kpj_rctls,
 961                             pp, 1, 0) & RCT_DENY)
 962                                 rctlfail = 1;

 964                 if (zone->zone_nprocs >= zone->zone_nprocs_ctl)
 965                         if (rctl_test(rc_zone_nprocs, zone->zone_rctls,
 966                             pp, 1, 0) & RCT_DENY)
 967                                 rctlfail = 1;

 969                 if (rctlfail) {
 970                         mutex_exit(&zone->zone_nlwps_lock);
 971                         mutex_exit(&pp->p_lock);
 972                         atomic_inc_32(&zone->zone_ffcap);
 973                         goto punish;
 974                 }
 975         }
 976         task->tk_nprocs++;
 977         proj->kpj_nprocs++;
 978         zone->zone_nprocs++;
 979         mutex_exit(&zone->zone_nlwps_lock);
 980         mutex_exit(&pp->p_lock);

 982         cp = kmem_cache_alloc(process_cache, KM_SLEEP);
 983         bzero(cp, sizeof (proc_t));
```



```
 985         /*
 986          * Make proc entry for child process
 987          */
 988         mutex_init(&cp->p_splock, NULL, MUTEX_DEFAULT, NULL);
 989         mutex_init(&cp->p_crlock, NULL, MUTEX_DEFAULT, NULL);
 990         mutex_init(&cp->p_pflock, NULL, MUTEX_DEFAULT, NULL);
 991 #if defined(__x86)
 992         mutex_init(&cp->p_ldtlock, NULL, MUTEX_DEFAULT, NULL);
 993 #endif
 994         mutex_init(&cp->p_maplock, NULL, MUTEX_DEFAULT, NULL);
 995         cp->p_stat = SIDL;
 996         cp->p_mstart = gethrtime();
 997         cp->p_as = &kas;
 998         /*
 999          * p_zone must be set before we call pid_allocate since the process
1000          * will be visible after that and code such as prfind_zone will
1001          * look at the p_zone field.
1002          */
1003         cp->p_zone = pp->p_zone;
1004         cp->p_t1_lgrpid = LGRP_NONE;
1005         cp->p_tr_lgrpid = LGRP_NONE;

1007         if ((newpid = pid_allocate(cp, pid, PID_ALLOC_PROC)) == -1) {
1008                 if (nproc == v.v_proc) {
1009                         CPU_STATS_ADDQ(CPU, sys, procovf, 1);
1010                         cmn_err(CE_WARN, "out of processes");
1011                 }
1012                 goto bad;
1013         }

1015         mutex_enter(&pp->p_lock);
1016         cp->p_exec = pp->p_exec;
1017         cp->p_execdir = pp->p_execdir;
1018         mutex_exit(&pp->p_lock);

1020         if (cp->p_exec) {
1021                 VN_HOLD(cp->p_exec);
1022                 /*
1023                  * Each VOP_OPEN() must be paired with a corresponding
1024                  * VOP_CLOSE(). In this case, the executable will be
1025                  * closed for the child in either proc_exit() or gexec().
1026                  */
1027                 if (VOP_OPEN(&cp->p_exec, FREAD, CRED(), NULL) != 0) {
1028                         VN_RELE(cp->p_exec);
1029                         cp->p_exec = NULLVP;
1030                         cp->p_execdir = NULLVP;
1031                         goto bad;
1032                 }
1033         }
1034         if (cp->p_execdir)
1035                 VN_HOLD(cp->p_execdir);

1037         /*
1038          * If not privileged make sure that this user hasn't exceeded
1039          * v.v_maxup processes, and that users collectively haven't
1040          * exceeded v.v_maxupttl processes.
1041          */
1042         mutex_enter(&pidlock);
1043         ASSERT(nproc < v.v_proc);       /* otherwise how'd we get our pid? */
1044         cr = CRED();
1045         ruid = crgetruid(cr);
1046         zoneid = crgetzoneid(cr);
1047         if (nproc >= v.v_maxup &&       /* short-circuit; usually false */
1048             (nproc >= v.v_maxupttl ||
1049             upcount_get(ruid, zoneid) >= v.v_maxup) &&
```

```
1050            secpolicy_newproc(cr) != 0) {
1051                mutex_exit(&pidlock);
1052                zcmn_err(zoneid, CE_NOTE,
1053                    "out of per-user processes for uid %d", ruid);
1054                goto bad;
1055        }

1057        /*
1058         * Everything is cool, put the new proc on the active process list.
1059         * It is already on the pid list and in /proc.
1060         * Increment the per uid process count (upcount).
1061         */
1062        nproc++;
1063        upcount_inc(ruid, zoneid);

1065        cp->p_next = practive;
1066        practive->p_prev = cp;
1067        practive = cp;

1069        cp->p_ignore = pp->p_ignore;
1070        cp->p_siginfo = pp->p_siginfo;
1071        cp->p_flag = pp->p_flag & (SJCTL|SNOWAIT|SNOCD);
1072        cp->p_sessp = pp->p_sessp;
1073        sess_hold(pp);
1074        cp->p_brand = pp->p_brand;
1075        if (PROC_IS_BRANDED(pp))
1076                BROP(pp)->b_copy_procdata(cp, pp);
1077        cp->p_bssbase = pp->p_bssbase;
1078        cp->p_brkbase = pp->p_brkbase;
1079        cp->p_brksize = pp->p_brksize;
1080        cp->p_brkpageszc = pp->p_brkpageszc;
1081        cp->p_stksize = pp->p_stksize;
1082        cp->p_stkpageszc = pp->p_stkpageszc;
1083        cp->p_stkprot = pp->p_stkprot;
1084        cp->p_datprot = pp->p_datprot;
1085        cp->p_usrstack = pp->p_usrstack;
1086        cp->p_model = pp->p_model;
1087        cp->p_ppid = pp->p_pid;
1088        cp->p_ancpid = pp->p_pid;
1089        cp->p_portcnt = pp->p_portcnt;

1091        /*
1092         * Initialize watchpoint structures
1093         */
1094        avl_create(&cp->p_warea, wa_compare, sizeof (struct watched_area),
1095            offsetof(struct watched_area, wa_link));

1097        /*
1098         * Initialize immediate resource control values.
1099         */
1100        cp->p_stk_ctl = pp->p_stk_ctl;
1101        cp->p_fsz_ctl = pp->p_fsz_ctl;
1102        cp->p_vmem_ctl = pp->p_vmem_ctl;
1103        cp->p_fno_ctl = pp->p_fno_ctl;

1105        /*
1106         * Link up to parent-child-sibling chain.  No need to lock
1107         * in general since only a call to freeproc() (done by the
1108         * same parent as newproc()) diddles with the child chain.
1109         */
1110        cp->p_sibling = pp->p_child;
1111        if (pp->p_child)
1112                pp->p_child->p_psibling = cp;

1114        cp->p_parent = pp;
1115        pp->p_child = cp;
```



```
1117        cp->p_child_ns = NULL;
1118        cp->p_sibling_ns = NULL;

1120        cp->p_nextorph = pp->p_orphan;
1121        cp->p_nextofkin = pp;
1122        pp->p_orphan = cp;

1124        /*
1125         * Inherit profiling state; do not inherit REALPROF profiling state.
1126         */
1127        cp->p_prof = pp->p_prof;
1128        cp->p_rprof_cyclic = CYCLIC_NONE;

1130        /*
1131         * Inherit pool pointer from the parent.  Kernel processes are
1132         * always bound to the default pool.
1133         */
1134        mutex_enter(&pp->p_lock);
1135        if (flags & GETPROC_KERNEL) {
1136                cp->p_pool = pool_default;
1137                cp->p_flag |= SSYS;
1138        } else {
1139                cp->p_pool = pp->p_pool;
1140        }
1141        atomic_inc_32(&cp->p_pool->pool_ref);
1142        mutex_exit(&pp->p_lock);

1144        /*
1145         * Add the child process to the current task.  Kernel processes
1146         * are always attached to task0.
1147         */
1148        mutex_enter(&cp->p_lock);
1149        if (flags & GETPROC_KERNEL)
1150                task_attach(task0p, cp);
1151        else
1152                task_attach(pp->p_task, cp);
1153        mutex_exit(&cp->p_lock);
1154        mutex_exit(&pidlock);

1156        avl_create(&cp->p_ct_held, contract_compar, sizeof (contract_t),
1157            offsetof(contract_t, ct_ctlist));

1159        /*
1160         * Duplicate any audit information kept in the process table
1161         */
1162        if (audit_active)       /* copy audit data to cp */
1163                audit_newproc(cp);

1165        crhold(cp->p_cred = cr);

1167        /*
1168         * Bump up the counts on the file structures pointed at by the
1169         * parent's file table since the child will point at them too.
1170         */
1171        fcnt_add(P_FINFO(pp), 1);

1173        if (PTOU(pp)->u_cdir) {
1174                VN_HOLD(PTOU(pp)->u_cdir);
1175        } else {
1176                ASSERT(pp == &p0);
1177                /*
1178                 * We must be at or before vfs_mountroot(); it will take care of
1179                 * assigning our current directory.
1180                 */
1181        }
```

```
1182         if (PTOU(pp)->u_rdir)
1183                 VN_HOLD(PTOU(pp)->u_rdir);
1184         if (PTOU(pp)->u_cwd)
1185                 refstr_hold(PTOU(pp)->u_cwd);

1187         /*
1188          * copy the parent's uarea.
1189          */
1190         uarea = PTOU(cp);
1191         bcopy(PTOU(pp), uarea, sizeof (*uarea));
1192         flist_fork(pp, cp);
1192         flist_fork(P_FINFO(pp), P_FINFO(cp));

1194         gethrestime(&uarea->u_start);
1195         uarea->u_ticks = ddi_get_lbolt();
1196         uarea->u_mem = rm_asrss(pp->p_as);
1197         uarea->u_acflag = AFORK;

1199         /*
1200          * If inherit-on-fork, copy /proc tracing flags to child.
1201          */
1202         if ((pp->p_proc_flag & P_PR_FORK) != 0) {
1203                 cp->p_proc_flag |= pp->p_proc_flag & (P_PR_TRACE|P_PR_FORK);
1204                 cp->p_sigmask = pp->p_sigmask;
1205                 cp->p_fltmask = pp->p_fltmask;
1206         } else {
1207                 sigemptyset(&cp->p_sigmask);
1208                 premptyset(&cp->p_fltmask);
1209                 uarea->u_systrap = 0;
1210                 premptyset(&uarea->u_entrymask);
1211                 premptyset(&uarea->u_exitmask);
1212         }
1213         /*
1214          * If microstate accounting is being inherited, mark child
1215          */
1216         if ((pp->p_flag & SMSFORK) != 0)
1217                 cp->p_flag |= pp->p_flag & (SMSFORK|SMSACCT);

1219         /*
1220          * Inherit fixalignment flag from the parent
1221          */
1222         cp->p_fixalignment = pp->p_fixalignment;

1224         *cpp = cp;
1225         return (0);

1227 bad:
1228         ASSERT(MUTEX_NOT_HELD(&pidlock));

1230         mutex_destroy(&cp->p_crlock);
1231         mutex_destroy(&cp->p_pflock);
1232 #if defined(__x86)
1233         mutex_destroy(&cp->p_ldtlock);
1234 #endif
1235         if (newpid != -1) {
1236                 proc_entry_free(cp->p_pidp);
1237                 (void) pid_rele(cp->p_pidp);
1238         }
1239         kmem_cache_free(process_cache, cp);

1241         mutex_enter(&zone->zone_nlwps_lock);
1242         task->tk_nprocs--;
1243         proj->kpj_nprocs--;
1244         zone->zone_nprocs--;
1245         mutex_exit(&zone->zone_nlwps_lock);
1246         atomic_inc_32(&zone->zone_ffnoproc);
```



```
1248 punish:
1249         /*
1250          * We most likely got into this situation because some process is
1251          * forking out of control.  As punishment, put it to sleep for a
1252          * bit so it can't eat the machine alive.  Sleep interval is chosen
1253          * to allow no more than one fork failure per cpu per clock tick
1254          * on average (yes, I just made this up).  This has two desirable
1255          * properties: (1) it sets a constant limit on the fork failure
1256          * rate, and (2) the busier the system is, the harsher the penalty
1257          * for abusing it becomes.
1258          */
1259         INCR_COUNT(&fork_fail_pending, &pidlock);
1260         delay(fork_fail_pending / ncpus + 1);
1261         DECR_COUNT(&fork_fail_pending, &pidlock);

1263         return (-1); /* out of memory or proc slots */
1264 }
______unchanged_portion_omitted_
```

```
**********************************************************
  218647 Mon Aug 17 21:08:07 2015
new/usr/src/uts/common/os/streamio.c
XXXX adding PID information to netstat output
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
  22 /*        All Rights Reserved   */


  25 /*
  26  * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
  27  */

  29 #include <sys/types.h>
  30 #include <sys/sysmacros.h>
  31 #include <sys/param.h>
  32 #include <sys/errno.h>
  33 #include <sys/signal.h>
  34 #include <sys/stat.h>
  35 #include <sys/proc.h>
  36 #include <sys/cred.h>
  37 #include <sys/user.h>
  38 #include <sys/vnode.h>
  39 #include <sys/file.h>
  40 #include <sys/stream.h>
  41 #include <sys/strsubr.h>
  42 #include <sys/stropts.h>
  43 #include <sys/tihdr.h>
  44 #include <sys/var.h>
  45 #include <sys/poll.h>
  46 #include <sys/termio.h>
  47 #include <sys/ttold.h>
  48 #include <sys/systm.h>
  49 #include <sys/uio.h>
  50 #include <sys/cmn_err.h>
  51 #include <sys/sad.h>
  52 #include <sys/netstack.h>
  53 #include <sys/priocntl.h>
  54 #include <sys/jioctl.h>
  55 #include <sys/procset.h>
  56 #include <sys/session.h>
  57 #include <sys/kmem.h>
  58 #include <sys/filio.h>
  59 #include <sys/vtrace.h>
  60 #include <sys/debug.h>
  61 #include <sys/strredir.h>
```



```
  62 #include <sys/fs/fifonode.h>
  63 #include <sys/fs/snode.h>
  64 #include <sys/strlog.h>
  65 #include <sys/strsun.h>
  66 #include <sys/project.h>
  67 #include <sys/kbio.h>
  68 #include <sys/msio.h>
  69 #include <sys/tty.h>
  70 #include <sys/ptyvar.h>
  71 #include <sys/vuid_event.h>
  72 #include <sys/modctl.h>
  73 #include <sys/sunddi.h>
  74 #include <sys/sunldi_impl.h>
  75 #include <sys/autoconf.h>
  76 #include <sys/policy.h>
  77 #include <sys/dld.h>
  78 #include <sys/zone.h>
  79 #include <c2/audit.h>
  80 #include <sys/fcntl.h>
  81 #endif /* ! codereview */

  83 /*
  84  * This define helps improve the readability of streams code while
  85  * still maintaining a very old streams performance enhancement.  The
  86  * performance enhancement basically involved having all callers
  87  * of straccess() perform the first check that straccess() will do
  88  * locally before actually calling straccess().  (There by reducing
  89  * the number of unnecessary calls to straccess().)
  90  */
  91 #define i_straccess(x, y)       ((stp->sd_sidp == NULL) ? 0 : \
  92                                     (stp->sd_vnode->v_type == VFIFO) ? 0 : \
  93                                     straccess((x), (y)))

  95 /*
  96  * what is mblk_pull_len?
  97  *
  98  * If a streams message consists of many short messages,
  99  * a performance degradation occurs from copyout overhead.
 100  * To decrease the per mblk overhead, messages that are
 101  * likely to consist of many small mblks are pulled up into
 102  * one continuous chunk of memory.
 103  *
 104  * To avoid the processing overhead of examining every
 105  * mblk, a quick heuristic is used. If the first mblk in
 106  * the message is shorter than mblk_pull_len, it is likely
 107  * that the rest of the mblk will be short.
 108  *
 109  * This heuristic was decided upon after performance tests
 110  * indicated that anything more complex slowed down the main
 111  * code path.
 112  */
 113 #define MBLK_PULL_LEN 64
 114 uint32_t mblk_pull_len = MBLK_PULL_LEN;

 116 /*
 117  * The sgttyb_handling flag controls the handling of the old BSD
 118  * TIOCGETP, TIOCSETP, and TIOCSETN ioctls as follows:
 119  *
 120  * 0 - Emit no warnings at all and retain old, broken behavior.
 121  * 1 - Emit no warnings and silently handle new semantics.
 122  * 2 - Send cmn_err(CE_NOTE) when either TIOCSETP or TIOCSETN is used
 123  *     (once per system invocation).  Handle with new semantics.
 124  * 3 - Send SIGSYS when any TIOCGETP, TIOCSETP, or TIOCSETN call is
 125  *     made (so that offenders drop core and are easy to debug).
 126  *
 127  * The "new semantics" are that TIOCGETP returns B38400 for
```

```
 * sg_[io]speed if the corresponding value is over B38400, and that
 * TIOCSET[PN] accept B38400 in these cases to mean "retain current
 * bit rate."
 */
int sgttyb_handling = 1;
static boolean_t sgttyb_complaint;

/* don't push drcompat module by default on Style-2 streams */
static int push_drcompat = 0;

/*
 * id value used to distinguish between different ioctl messages
 */
static uint32_t ioc_id;

static void putback(struct stdata *, queue_t *, mblk_t *, int);
static void strcleanall(struct vnode *);
static int strwsrv(queue_t *);
static int strdocmd(struct stdata *, struct strcmd *, cred_t *);
static boolean_t is_xti_str(const struct stdata *);
#endif /* ! codereview */

/*
 * qinit and module_info structures for stream head read and write queues
 */
struct module_info strm_info = { 0, "strrhead", 0, INFPSZ, STRHIGH, STRLOW };
struct module_info stwm_info = { 0, "strwhead", 0, 0, 0, 0 };
struct qinit strdata = { strrput, NULL, NULL, NULL, &strm_info };
struct qinit stwdata = { NULL, strwsrv, NULL, NULL, NULL, &stwm_info };
struct module_info fiform_info = { 0, "fifostrrhead", 0, PIPE_BUF, FIFOHIWAT,
    FIFOLOWAT };
struct module_info fifowm_info = { 0, "fifostrwhead", 0, 0, 0, 0 };
struct qinit fifo_strdata = { strrput, NULL, NULL, NULL, NULL, &fiform_info };
struct qinit fifo_stwdata = { NULL, strwsrv, NULL, NULL, NULL, &fifowm_info };

extern kmutex_t strresources;   /* protects global resources */
extern kmutex_t muxifier;       /* single-threads multiplexor creation */

static boolean_t msghasdata(mblk_t *bp);
#define msgnodata(bp) (!msghasdata(bp))

/*
 * Stream head locking notes:
 *      There are four monitors associated with the stream head:
 *      1. v_stream monitor: in stropen() and strclose() v_lock
 *              is held while the association of vnode and stream
 *              head is established or tested for.
 *      2. open/close/push/pop monitor: sd_lock is held while each
 *              thread bids for exclusive access to this monitor
 *              for opening or closing a stream.  In addition, this
 *              monitor is entered during pushes and pops.  This
 *              guarantees that during plumbing operations there
 *              is only one thread trying to change the plumbing.
 *              Any other threads present in the stream are only
 *              using the plumbing.
 *      3. read/write monitor: in the case of read, a thread holds
 *              sd_lock while trying to get data from the stream
 *              head queue.  if there is none to fulfill a read
 *              request, it sets RSLEEP and calls cv_wait_sig() down
 *              in strwaitq() to await the arrival of new data.
 *              when new data arrives in strrput(), sd_lock is acquired
 *              before testing for RSLEEP and calling cv_broadcast().
 *              the behavior of strwrite(), strwsrv(), and WSLEEP
 *              mirror this.
 *      4. ioctl monitor: sd_lock is gotten to ensure that only one
 *              thread is doing an ioctl at a time.
```



```
 */

static int
push_mod(queue_t *qp, dev_t *devp, struct stdata *stp, const char *name,
    int anchor, cred_t *crp, uint_t anchor_zoneid)
{
        int error;
        fmodsw_impl_t *fp;

        if (stp->sd_flag & (STRHUP|STRDERR|STWRERR)) {
                error = (stp->sd_flag & STRHUP) ? ENXIO : EIO;
                return (error);
        }
        if (stp->sd_pushcnt >= nstrpush) {
                return (EINVAL);
        }

        if ((fp = fmodsw_find(name, FMODSW_HOLD | FMODSW_LOAD)) == NULL) {
                stp->sd_flag |= STREOPENFAIL;
                return (EINVAL);
        }

        /*
         * push new module and call its open routine via qattach
         */
        if ((error = qattach(qp, devp, 0, crp, fp, B_FALSE)) != 0)
                return (error);

        /*
         * Check to see if caller wants a STREAMS anchor
         * put at this place in the stream, and add if so.
         */
        mutex_enter(&stp->sd_lock);
        if (anchor == stp->sd_pushcnt) {
                stp->sd_anchor = stp->sd_pushcnt;
                stp->sd_anchorzone = anchor_zoneid;
        }
        mutex_exit(&stp->sd_lock);

        return (0);
}

/*
 * Open a stream device.
 */
int
stropen(vnode_t *vp, dev_t *devp, int flag, cred_t *crp)
{
        struct stdata *stp;
        queue_t *qp;
        int s;
        dev_t dummydev, savedev;
        struct autopush *ap;
        struct dlautopush dlap;
        int error = 0;
        ssize_t rmin, rmax;
        int cloneopen;
        queue_t *brq;
        major_t major;
        str_stack_t *ss;
        zoneid_t zoneid;
        uint_t anchor;

        /*
         * If the stream already exists, wait for any open in progress
         * to complete, then call the open function of each module and
```

```
260          * driver in the stream.  Otherwise create the stream.
261          */
262         TRACE_1(TR_FAC_STREAMS_FR, TR_STROPEN, "stropen:%p", vp);
263 retry:
264         mutex_enter(&vp->v_lock);
265         if ((stp = vp->v_stream) != NULL) {
267                 /*
268                  * Waiting for stream to be created to device
269                  * due to another open.
270                  */
271                 mutex_exit(&vp->v_lock);
273                 if (STRMATED(stp)) {
274                         struct stdata *strmatep = stp->sd_mate;
276                         STRLOCKMATES(stp);
277                         if (strmatep->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
278                                 if (flag & (FNDELAY|FNONBLOCK)) {
279                                         error = EAGAIN;
280                                         mutex_exit(&strmatep->sd_lock);
281                                         goto ckreturn;
282                                 }
283                                 mutex_exit(&stp->sd_lock);
284                                 if (!cv_wait_sig(&strmatep->sd_monitor,
285                                     &strmatep->sd_lock)) {
286                                         error = EINTR;
287                                         mutex_exit(&strmatep->sd_lock);
288                                         mutex_enter(&stp->sd_lock);
289                                         goto ckreturn;
290                                 }
291                                 mutex_exit(&strmatep->sd_lock);
292                                 goto retry;
293                         }
294                         if (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
295                                 if (flag & (FNDELAY|FNONBLOCK)) {
296                                         error = EAGAIN;
297                                         mutex_exit(&strmatep->sd_lock);
298                                         goto ckreturn;
299                                 }
300                                 mutex_exit(&strmatep->sd_lock);
301                                 if (!cv_wait_sig(&stp->sd_monitor,
302                                     &stp->sd_lock)) {
303                                         error = EINTR;
304                                         goto ckreturn;
305                                 }
306                                 mutex_exit(&stp->sd_lock);
307                                 goto retry;
308                         }
310                         if (stp->sd_flag & (STRDERR|STWRERR)) {
311                                 error = EIO;
312                                 mutex_exit(&strmatep->sd_lock);
313                                 goto ckreturn;
314                         }
316                         stp->sd_flag |= STWOPEN;
317                         STRUNLOCKMATES(stp);
318                 } else {
319                         mutex_enter(&stp->sd_lock);
320                         if (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
321                                 if (flag & (FNDELAY|FNONBLOCK)) {
322                                         error = EAGAIN;
323                                         goto ckreturn;
324                                 }
325                                 if (!cv_wait_sig(&stp->sd_monitor,
```



```
326                                     &stp->sd_lock)) {
327                                         error = EINTR;
328                                         goto ckreturn;
329                                 }
330                                 mutex_exit(&stp->sd_lock);
331                                 goto retry;  /* could be clone! */
332                         }
334                         if (stp->sd_flag & (STRDERR|STWRERR)) {
335                                 error = EIO;
336                                 goto ckreturn;
337                         }
339                         stp->sd_flag |= STWOPEN;
340                         mutex_exit(&stp->sd_lock);
341                 }
343                 /*
344                  * Open all modules and devices down stream to notify
345                  * that another user is streaming.  For modules, set the
346                  * last argument to MODOPEN and do not pass any open flags.
347                  * Ignore dummydev since this is not the first open.
348                  */
349                 claimstr(stp->sd_wrq);
350                 qp = stp->sd_wrq;
351                 while (_SAMESTR(qp)) {
352                         qp = qp->q_next;
353                         if ((error = qreopen(_RD(qp), devp, flag, crp)) != 0)
354                                 break;
355                 }
356                 releasestr(stp->sd_wrq);
357                 mutex_enter(&stp->sd_lock);
358                 stp->sd_flag &= ~(STRHUP|STWOPEN|STRDERR|STWRERR);
359                 stp->sd_rerror = 0;
360                 stp->sd_werror = 0;
361 ckreturn:
362                 cv_broadcast(&stp->sd_monitor);
363                 mutex_exit(&stp->sd_lock);
364                 return (error);
365         }
367         /*
368          * This vnode isn't streaming.  SPECFS already
369          * checked for multiple vnodes pointing to the
370          * same stream, so create a stream to the driver.
371          */
372         qp = allocq();
373         stp = shalloc(qp);
375         /*
376          * Initialize stream head.  shalloc() has given us
377          * exclusive access, and we have the vnode locked;
378          * we can do whatever we want with stp.
379          */
380         stp->sd_flag = STWOPEN;
381         stp->sd_siglist = NULL;
382         stp->sd_pollist.ph_list = NULL;
383         stp->sd_sigflags = 0;
384         stp->sd_mark = NULL;
385         stp->sd_closetime = STRTIMOUT;
386         stp->sd_sidp = NULL;
387         stp->sd_pgidp = NULL;
388         stp->sd_vnode = vp;
389         stp->sd_rerror = 0;
390         stp->sd_werror = 0;
391         stp->sd_wroff = 0;
```

```
392         stp->sd_tail = 0;
393         stp->sd_iocblk = NULL;
394         stp->sd_cmdblk = NULL;
395         stp->sd_pushcnt = 0;
396         stp->sd_qn_minpsz = 0;
397         stp->sd_qn_maxpsz = INFPSZ - 1; /* used to check for initialization */
398         stp->sd_maxblk = INFPSZ;
399         qp->q_ptr = _WR(qp)->q_ptr = stp;
400         STREAM(qp) = STREAM(_WR(qp)) = stp;
401         vp->v_stream = stp;
402         mutex_exit(&vp->v_lock);

404         /*
405          * If this is not a system process, then add it to
406          * the list associated with the stream head.
407          */
408         if (!(curproc->p_flag & SSYS) && is_xti_str(stp))
409                 sh_insert_pid(stp, curproc->p_pidp->pid_id);

411 #endif /* ! codereview */
412         if (vp->v_type == VFIFO) {
413                 stp->sd_flag |= OLDNDELAY;
414                 /*
415                  * This means, both for pipes and fifos
416                  * strwrite will send SIGPIPE if the other
417                  * end is closed. For putmsg it depends
418                  * on whether it is a XPG4_2 application
419                  * or not
420                  */
421                 stp->sd_wput_opt = SW_SIGPIPE;

423                 /* setq might sleep in kmem_alloc - avoid holding locks. */
424                 setq(qp, &fifo_strdata, &fifo_stwdata, NULL, QMTSAFE,
425                     SQ_CI|SQ_CO, B_FALSE);

427                 set_qend(qp);
428                 stp->sd_strtab = fifo_getinfo();
429                 _WR(qp)->q_nfsrv = _WR(qp);
430                 qp->q_nfsrv = qp;
431                 /*
432                  * Wake up others that are waiting for stream to be created.
433                  */
434                 mutex_enter(&stp->sd_lock);
435                 /*
436                  * nothing is be pushed on stream yet, so
437                  * optimized stream head packetsizes are just that
438                  * of the read queue
439                  */
440                 stp->sd_qn_minpsz = qp->q_minpsz;
441                 stp->sd_qn_maxpsz = qp->q_maxpsz;
442                 stp->sd_flag &= ~STWOPEN;
443                 goto fifo_opendone;
444         }
445         /* setq might sleep in kmem_alloc - avoid holding locks. */
446         setq(qp, &strdata, &stwdata, NULL, QMTSAFE, SQ_CI|SQ_CO, B_FALSE);

448         set_qend(qp);

450         /*
451          * Open driver and create stream to it (via qattach).
452          */
453         savedev = *devp;
454         cloneopen = (getmajor(*devp) == clone_major);
455         if ((error = qattach(qp, devp, flag, crp, NULL, B_FALSE)) != 0) {
456                 mutex_enter(&vp->v_lock);
457                 vp->v_stream = NULL;
```



```
458                 mutex_exit(&vp->v_lock);
459                 mutex_enter(&stp->sd_lock);
460                 cv_broadcast(&stp->sd_monitor);
461                 mutex_exit(&stp->sd_lock);
462                 freeq(_RD(qp));
463                 shfree(stp);
464                 return (error);
465         }
466         /*
467          * Set sd_strtab after open in order to handle clonable drivers
468          */
469         stp->sd_strtab = STREAMSTAB(getmajor(*devp));

471         /*
472          * Historical note: dummydev used to be be prior to the initial
473          * open (via qattach above), which made the value seen
474          * inconsistent between an I_PUSH and an autopush of a module.
475          */
476         dummydev = *devp;

478         /*
479          * For clone open of old style (Q not associated) network driver,
480          * push DRMODNAME module to handle DL_ATTACH/DL_DETACH
481          */
482         brq = _RD(_WR(qp)->q_next);
483         major = getmajor(*devp);
484         if (push_drcompat && cloneopen && NETWORK_DRV(major) &&
485             ((brq->q_flag & _QASSOCIATED) == 0)) {
486                 if (push_mod(qp, &dummydev, stp, DRMODNAME, 0, crp, 0) != 0)
487                         cmn_err(CE_WARN, "cannot push " DRMODNAME
488                             " streams module");
489         }

491         if (!NETWORK_DRV(major)) {
492                 savedev = *devp;
493         } else {
494                 /*
495                  * For network devices, process differently based on the
496                  * return value from dld_autopush():
497                  *
498                  *   0: the passed-in device points to a GLDv3 datalink with
499                  *   per-link autopush configuration; use that configuration
500                  *   and ignore any per-driver autopush configuration.
501                  *
502                  *   1: the passed-in device points to a physical GLDv3
503                  *   datalink without per-link autopush configuration.  The
504                  *   passed in device was changed to refer to the actual
505                  *   physical device (if it's not already); we use that new
506                  *   device to look up any per-driver autopush configuration.
507                  *
508                  *   -1: neither of the above cases applied; use the initial
509                  *   device to look up any per-driver autopush configuration.
510                  */
511                 switch (dld_autopush(&savedev, &dlap)) {
512                 case 0:
513                         zoneid = crgetzoneid(crp);
514                         for (s = 0; s < dlap.dap_npush; s++) {
515                                 error = push_mod(qp, &dummydev, stp,
516                                     dlap.dap_aplist[s], dlap.dap_anchor, crp,
517                                     zoneid);
518                                 if (error != 0)
519                                         break;
520                         }
521                         goto opendone;
522                 case 1:
523                         break;
```

```
524                 case -1:
525                         savedev = *devp;
526                         break;
527                 }
528         }
529         /*
530          * Find the autopush configuration based on "savedev". Start with the
531          * global zone. If not found check in the local zone.
532          */
533         zoneid = GLOBAL_ZONEID;
534 retryap:
535         ss = netstack_find_by_stackid(zoneid_to_netstackid(zoneid))->
536             netstack_str;
537         if ((ap = sad_ap_find_by_dev(savedev, ss)) == NULL) {
538                 netstack_rele(ss->ss_netstack);
539                 if (zoneid == GLOBAL_ZONEID) {
540                         /*
541                          * None found. Also look in the zone's autopush table.
542                          */
543                         zoneid = crgetzoneid(crp);
544                         if (zoneid != GLOBAL_ZONEID)
545                                 goto retryap;
546                 }
547                 goto opendone;
548         }
549         anchor = ap->ap_anchor;
550         zoneid = crgetzoneid(crp);
551         for (s = 0; s < ap->ap_npush; s++) {
552                 error = push_mod(qp, &dummydev, stp, ap->ap_list[s],
553                     anchor, crp, zoneid);
554                 if (error != 0)
555                         break;
556         }
557         sad_ap_rele(ap, ss);
558         netstack_rele(ss->ss_netstack);

560 opendone:

562         /*
563          * let specfs know that open failed part way through
564          */
565         if (error) {
566                 mutex_enter(&stp->sd_lock);
567                 stp->sd_flag |= STREOPENFAIL;
568                 mutex_exit(&stp->sd_lock);
569         }

571         /*
572          * Wake up others that are waiting for stream to be created.
573          */
574         mutex_enter(&stp->sd_lock);
575         stp->sd_flag &= ~STWOPEN;

577         /*
578          * As a performance concern we are caching the values of
579          * q_minpsz and q_maxpsz of the module below the stream
580          * head in the stream head.
581          */
582         mutex_enter(QLOCK(stp->sd_wrq->q_next));
583         rmin = stp->sd_wrq->q_next->q_minpsz;
584         rmax = stp->sd_wrq->q_next->q_maxpsz;
585         mutex_exit(QLOCK(stp->sd_wrq->q_next));

587         /* do this processing here as a performance concern */
588         if (strmsgsz != 0) {
589                 if (rmax == INFPSZ)
```



```
590                         rmax = strmsgsz;
591                 else
592                         rmax = MIN(strmsgsz, rmax);
593         }

595         mutex_enter(QLOCK(stp->sd_wrq));
596         stp->sd_qn_minpsz = rmin;
597         stp->sd_qn_maxpsz = rmax;
598         mutex_exit(QLOCK(stp->sd_wrq));

600 fifo_opendone:
601         cv_broadcast(&stp->sd_monitor);
602         mutex_exit(&stp->sd_lock);
603         return (error);
604 }

606 static int strsink(queue_t *, mblk_t *);
607 static struct qinit deadrend = {
608         strsink, NULL, NULL, NULL, NULL, &strm_info, NULL
609 };
610 static struct qinit deadwend = {
611         NULL, NULL, NULL, NULL, NULL, &stwm_info, NULL
612 };

614 /*
615  * Close a stream.
616  * This is called from closef() on the last close of an open stream.
617  * Strclean() will already have removed the siglist and pollist
618  * information, so all that remains is to remove all multiplexor links
619  * for the stream, pop all the modules (and the driver), and free the
620  * stream structure.
621  */

623 int
624 strclose(struct vnode *vp, int flag, cred_t *crp)
625 {
626         struct stdata *stp;
627         queue_t *qp;
628         int rval;
629         int freestp = 1;
630         queue_t *rmq;

632         TRACE_1(TR_FAC_STREAMS_FR,
633             TR_STRCLOSE, "strclose:%p", vp);
634         ASSERT(vp->v_stream);

636         stp = vp->v_stream;
637         ASSERT(!(stp->sd_flag & STPLEX));
638         qp = stp->sd_wrq;

640         /*
641          * Needed so that strpoll will return non-zero for this fd.
642          * Note that with POLLNOERR STRHUP does still cause POLLHUP.
643          */
644         mutex_enter(&stp->sd_lock);
645         stp->sd_flag |= STRHUP;
646         mutex_exit(&stp->sd_lock);

648         /*
649          * If the registered process or process group did not have an
650          * open instance of this stream then strclean would not be
651          * called. Thus at the time of closing all remaining siglist entries
652          * are removed.
653          */
654         if (stp->sd_siglist != NULL)
655                 strcleanall(vp);
```

```
657         ASSERT(stp->sd_siglist == NULL);
658         ASSERT(stp->sd_sigflags == 0);
659
660         if (STRMATED(stp)) {
661                 struct stdata *strmatep = stp->sd_mate;
662                 int waited = 1;
663
664                 STRLOCKMATES(stp);
665                 while (waited) {
666                         waited = 0;
667                         while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
668                                 mutex_exit(&strmatep->sd_lock);
669                                 cv_wait(&stp->sd_monitor, &stp->sd_lock);
670                                 mutex_exit(&stp->sd_lock);
671                                 STRLOCKMATES(stp);
672                                 waited = 1;
673                         }
674                         while (strmatep->sd_flag &
675                             (STWOPEN|STRCLOSE|STRPLUMB)) {
676                                 mutex_exit(&stp->sd_lock);
677                                 cv_wait(&strmatep->sd_monitor,
678                                     &strmatep->sd_lock);
679                                 mutex_exit(&strmatep->sd_lock);
680                                 STRLOCKMATES(stp);
681                                 waited = 1;
682                         }
683                 }
684                 stp->sd_flag |= STRCLOSE;
685                 STRUNLOCKMATES(stp);
686         } else {
687                 mutex_enter(&stp->sd_lock);
688                 stp->sd_flag |= STRCLOSE;
689                 mutex_exit(&stp->sd_lock);
690         }
691
692         ASSERT(qp->q_first == NULL);    /* No more delayed write */
693
694         /* Check if an I_LINK was ever done on this stream */
695         if (stp->sd_flag & STRHASLINKS) {
696                 netstack_t *ns;
697                 str_stack_t *ss;
698
699                 ns = netstack_find_by_cred(crp);
700                 ASSERT(ns != NULL);
701                 ss = ns->netstack_str;
702                 ASSERT(ss != NULL);
703
704                 (void) munlinkall(stp, LINKCLOSE|LINKNORMAL, crp, &rval, ss);
705                 netstack_rele(ss->ss_netstack);
706         }
707
708         while (_SAMESTR(qp)) {
709                 /*
710                  * Holding sd_lock prevents q_next from changing in
711                  * this stream.
712                  */
713                 mutex_enter(&stp->sd_lock);
714                 if (!(flag & (FNDELAY|FNONBLOCK)) && (stp->sd_closetime > 0)) {
715
716                         /*
717                          * sleep until awakened by strwsrv() or timeout
718                          */
719                         for (;;) {
720                                 mutex_enter(QLOCK(qp->q_next));
721                                 if (!(qp->q_next->q_mblkcnt)) {
```



```
722                                         mutex_exit(QLOCK(qp->q_next));
723                                         break;
724                                 }
725                                 stp->sd_flag |= WSLEEP;
726
727                                 /* ensure strwsrv gets enabled */
728                                 qp->q_next->q_flag |= QWANTW;
729                                 mutex_exit(QLOCK(qp->q_next));
730                                 /* get out if we timed out or recv'd a signal */
731                                 if (str_cv_wait(&qp->q_wait, &stp->sd_lock,
732                                     stp->sd_closetime, 0) <= 0) {
733                                         break;
734                                 }
735                         }
736                         stp->sd_flag &= ~WSLEEP;
737                 }
738                 mutex_exit(&stp->sd_lock);
739
740                 rmq = qp->q_next;
741                 if (rmq->q_flag & QISDRV) {
742                         ASSERT(!_SAMESTR(rmq));
743                         wait_sq_svc(_RD(qp)->q_syncq);
744                 }
745
746                 qdetach(_RD(rmq), 1, flag, crp, B_FALSE);
747         }
748
749         /*
750          * Since we call pollwakeup in close() now, the poll list should
751          * be empty in most cases. The only exception is the layered devices
752          * (e.g. the console drivers with redirection modules pushed on top
753          * of it).  We have to do this after calling qdetach() because
754          * the redirection module won't have torn down the console
755          * redirection until after qdetach() has been invoked.
756          */
757         if (stp->sd_pollist.ph_list != NULL) {
758                 pollwakeup(&stp->sd_pollist, POLLERR);
759                 pollhead_clean(&stp->sd_pollist);
760         }
761         ASSERT(stp->sd_pollist.ph_list == NULL);
762         ASSERT(stp->sd_sidp == NULL);
763         ASSERT(stp->sd_pgidp == NULL);
764
765         /* Prevent qenable from re-enabling the stream head queue */
766         disable_svc(_RD(qp));
767
768         /*
769          * Wait until service procedure of each queue is
770          * run, if QINSERVICE is set.
771          */
772         wait_svc(_RD(qp));
773
774         /*
775          * Now, flush both queues.
776          */
777         flushq(_RD(qp), FLUSHALL);
778         flushq(qp, FLUSHALL);
779
780         /*
781          * If the write queue of the stream head is pointing to a
782          * read queue, we have a twisted stream.  If the read queue
783          * is alive, convert the stream head queues into a dead end.
784          * If the read queue is dead, free the dead pair.
785          */
786         if (qp->q_next && !_SAMESTR(qp)) {
787                 if (qp->q_next->q_qinfo == &deadrend) { /* half-closed pipe */
```

```
788                         flushq(qp->q_next, FLUSHALL); /* ensure no message */
789                         shfree(qp->q_next->q_stream);
790                         freeq(qp->q_next);
791                         freeq(_RD(qp));
792                 } else if (qp->q_next == _RD(qp)) {     /* fifo */
793                         freeq(_RD(qp));
794                 } else {                                /* pipe */
795                         freestp = 0;
796                         /*
797                          * The q_info pointers are never accessed when
798                          * SQLOCK is held.
799                          */
800                         ASSERT(qp->q_syncq == _RD(qp)->q_syncq);
801                         mutex_enter(SQLOCK(qp->q_syncq));
802                         qp->q_qinfo = &deadwend;
803                         _RD(qp)->q_qinfo = &deadrend;
804                         mutex_exit(SQLOCK(qp->q_syncq));
805                 }
806         } else {
807                 freeq(_RD(qp)); /* free stream head queue pair */
808         }

810         mutex_enter(&vp->v_lock);
811         if (stp->sd_iocblk) {
812                 if (stp->sd_iocblk != (mblk_t *)-1) {
813                         freemsg(stp->sd_iocblk);
814                 }
815                 stp->sd_iocblk = NULL;
816         }
817         stp->sd_vnode = NULL;
818         vp->v_stream = NULL;
819         mutex_exit(&vp->v_lock);
820         mutex_enter(&stp->sd_lock);
821         freemsg(stp->sd_cmdblk);
822         stp->sd_cmdblk = NULL;
823         stp->sd_flag &= ~STRCLOSE;
824         cv_broadcast(&stp->sd_monitor);
825         mutex_exit(&stp->sd_lock);

827         if (freestp)
828                 shfree(stp);
829         return (0);
830 }

832 static int
833 strsink(queue_t *q, mblk_t *bp)
834 {
835         struct copyresp *resp;

837         switch (bp->b_datap->db_type) {
838         case M_FLUSH:
839                 if ((*bp->b_rptr & FLUSHW) && !(bp->b_flag & MSGNOLOOP)) {
840                         *bp->b_rptr &= ~FLUSHR;
841                         bp->b_flag |= MSGNOLOOP;
842                         /*
843                          * Protect against the driver passing up
844                          * messages after it has done a qprocsoff.
845                          */
846                         if (_OTHERQ(q)->q_next == NULL)
847                                 freemsg(bp);
848                         else
849                                 qreply(q, bp);
850                 } else {
851                         freemsg(bp);
852                 }
853                 break;
```



```
855         case M_COPYIN:
856         case M_COPYOUT:
857                 if (bp->b_cont) {
858                         freemsg(bp->b_cont);
859                         bp->b_cont = NULL;
860                 }
861                 bp->b_datap->db_type = M_IOCDATA;
862                 bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
863                 resp = (struct copyresp *)bp->b_rptr;
864                 resp->cp_rval = (caddr_t)1;     /* failure */
865                 /*
866                  * Protect against the driver passing up
867                  * messages after it has done a qprocsoff.
868                  */
869                 if (_OTHERQ(q)->q_next == NULL)
870                         freemsg(bp);
871                 else
872                         qreply(q, bp);
873                 break;

875         case M_IOCTL:
876                 if (bp->b_cont) {
877                         freemsg(bp->b_cont);
878                         bp->b_cont = NULL;
879                 }
880                 bp->b_datap->db_type = M_IOCNAK;
881                 /*
882                  * Protect against the driver passing up
883                  * messages after it has done a qprocsoff.
884                  */
885                 if (_OTHERQ(q)->q_next == NULL)
886                         freemsg(bp);
887                 else
888                         qreply(q, bp);
889                 break;

891         default:
892                 freemsg(bp);
893                 break;
894         }

896         return (0);
897 }

899 /*
900  * Clean up after a process when it closes a stream.  This is called
901  * from closef for all closes, whereas strclose is called only for the
902  * last close on a stream.  The siglist is scanned for entries for the
903  * current process, and these are removed.
904  */
905 void
906 strclean(struct vnode *vp)
907 {
908         strsig_t *ssp, *pssp, *tssp;
909         stdata_t *stp;
910         int update = 0;

912         TRACE_1(TR_FAC_STREAMS_FR,
913             TR_STRCLEAN, "strclean:%p", vp);
914         stp = vp->v_stream;
915         pssp = NULL;
916         mutex_enter(&stp->sd_lock);
917         ssp = stp->sd_siglist;
918         while (ssp) {
919                 if (ssp->ss_pidp == curproc->p_pidp) {
```

```
920                         tssp = ssp->ss_next;
921                         if (pssp)
922                                 pssp->ss_next = tssp;
923                         else
924                                 stp->sd_siglist = tssp;
925                         mutex_enter(&pidlock);
926                         PID_RELE(ssp->ss_pidp);
927                         mutex_exit(&pidlock);
928                         kmem_free(ssp, sizeof (strsig_t));
929                         update = 1;
930                         ssp = tssp;
931                 } else {
932                         pssp = ssp;
933                         ssp = ssp->ss_next;
934                 }
935         }
936         if (update) {
937                 stp->sd_sigflags = 0;
938                 for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
939                         stp->sd_sigflags |= ssp->ss_events;
940         }
941         mutex_exit(&stp->sd_lock);
942 }

944 /*
945  * Used on the last close to remove any remaining items on the siglist.
946  * These could be present on the siglist due to I_ESETSIG calls that
947  * use process groups or processed that do not have an open file descriptor
948  * for this stream (Such entries would not be removed by strclean).
949  */
950 static void
951 strcleanall(struct vnode *vp)
952 {
953         strsig_t *ssp, *nssp;
954         stdata_t *stp;

956         stp = vp->v_stream;
957         mutex_enter(&stp->sd_lock);
958         ssp = stp->sd_siglist;
959         stp->sd_siglist = NULL;
960         while (ssp) {
961                 nssp = ssp->ss_next;
962                 mutex_enter(&pidlock);
963                 PID_RELE(ssp->ss_pidp);
964                 mutex_exit(&pidlock);
965                 kmem_free(ssp, sizeof (strsig_t));
966                 ssp = nssp;
967         }
968         stp->sd_sigflags = 0;
969         mutex_exit(&stp->sd_lock);
970 }

972 /*
973  * Retrieve the next message from the logical stream head read queue
974  * using either rwnext (if sync stream) or getq_noenab.
975  * It is the callers responsibility to call qbackenable after
976  * it is finished with the message. The caller should not call
977  * qbackenable until after any putback calls to avoid spurious backenabling.
978  */
979 mblk_t *
980 strget(struct stdata *stp, queue_t *q, struct uio *uiop, int first,
981     int *errorp)
982 {
983         mblk_t *bp;
984         int error;
985         ssize_t rbytes = 0;
```



```
987         /* Holding sd_lock prevents the read queue from changing  */
988         ASSERT(MUTEX_HELD(&stp->sd_lock));

990         if (uiop != NULL && stp->sd_struiordq != NULL &&
991             q->q_first == NULL &&
992             (!first || (stp->sd_wakeq & RSLEEP))) {
993                 /*
994                  * Stream supports rwnext() for the read side.
995                  * If this is the first time we're called by e.g. strread
996                  * only do the downcall if there is a deferred wakeup
997                  * (registered in sd_wakeq).
998                  */
999                 struiod_t uiod;

1001                if (first)
1002                        stp->sd_wakeq &= ~RSLEEP;

1004                (void) uiodup(uiop, &uiod.d_uio, uiod.d_iov,
1005                    sizeof (uiod.d_iov) / sizeof (*uiod.d_iov));
1006                uiod.d_mp = 0;
1007                /*
1008                 * Mark that a thread is in rwnext on the read side
1009                 * to prevent strrput from nacking ioctls immediately.
1010                 * When the last concurrent rwnext returns
1011                 * the ioctls are nack'ed.
1012                 */
1013                ASSERT(MUTEX_HELD(&stp->sd_lock));
1014                stp->sd_struiodnak++;
1015                /*
1016                 * Note: rwnext will drop sd_lock.
1017                 */
1018                error = rwnext(q, &uiod);
1019                ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));
1020                mutex_enter(&stp->sd_lock);
1021                stp->sd_struiodnak--;
1022                while (stp->sd_struiodnak == 0 &&
1023                    ((bp = stp->sd_struionak) != NULL)) {
1024                        stp->sd_struionak = bp->b_next;
1025                        bp->b_next = NULL;
1026                        bp->b_datap->db_type = M_IOCNAK;
1027                        /*
1028                         * Protect against the driver passing up
1029                         * messages after it has done a qprocsoff.
1030                         */
1031                        if (_OTHERQ(q)->q_next == NULL)
1032                                freemsg(bp);
1033                        else {
1034                                mutex_exit(&stp->sd_lock);
1035                                qreply(q, bp);
1036                                mutex_enter(&stp->sd_lock);
1037                        }
1038                }
1039                ASSERT(MUTEX_HELD(&stp->sd_lock));
1040                if (error == 0 || error == EWOULDBLOCK) {
1041                        if ((bp = uiod.d_mp) != NULL) {
1042                                *errorp = 0;
1043                                ASSERT(MUTEX_HELD(&stp->sd_lock));
1044                                return (bp);
1045                        }
1046                        error = 0;
1047                } else if (error == EINVAL) {
1048                        /*
1049                         * The stream plumbing must have
1050                         * changed while we were away, so
1051                         * just turn off rwnext()s.
```

```
1052                          */
1053                         error = 0;
1054                 } else if (error == EBUSY) {
1055                         /*
1056                          * The module might have data in transit using putnext
1057                          * Fall back on waiting + getq.
1058                          */
1059                         error = 0;
1060                 } else {
1061                         *errorp = error;
1062                         ASSERT(MUTEX_HELD(&stp->sd_lock));
1063                         return (NULL);
1064                 }
1065                 /*
1066                  * Try a getq in case a rwnext() generated mblk
1067                  * has bubbled up via strrput().
1068                  */
1069         }
1070         *errorp = 0;
1071         ASSERT(MUTEX_HELD(&stp->sd_lock));

1073         /*
1074          * If we have a valid uio, try and use this as a guide for how
1075          * many bytes to retrieve from the queue via getq_noenab().
1076          * Doing this can avoid unneccesary counting of overlong
1077          * messages in putback(). We currently only do this for sockets
1078          * and only if there is no sd_rputdatafunc hook.
1079          *
1080          * The sd_rputdatafunc hook transforms the entire message
1081          * before any bytes in it can be given to a client. So, rbytes
1082          * must be 0 if there is a hook.
1083          */
1084         if ((uiop != NULL) && (stp->sd_vnode->v_type == VSOCK) &&
1085             (stp->sd_rputdatafunc == NULL))
1086                 rbytes = uiop->uio_resid;

1088         return (getq_noenab(q, rbytes));
1089 }

1091 /*
1092  * Copy out the message pointed to by 'bp' into the uio pointed to by 'uiop'.
1093  * If the message does not fit in the uio the remainder of it is returned;
1094  * otherwise NULL is returned.  Any embedded zero-length mblk_t's are
1095  * consumed, even if uio_resid reaches zero.  On error, '*errorp' is set to
1096  * the error code, the message is consumed, and NULL is returned.
1097  */
1098 static mblk_t *
1099 struiocopyout(mblk_t *bp, struct uio *uiop, int *errorp)
1100 {
1101         int error;
1102         ptrdiff_t n;
1103         mblk_t *nbp;

1105         ASSERT(bp->b_wptr >= bp->b_rptr);

1107         do {
1108                 if ((n = MIN(uiop->uio_resid, MBLKL(bp))) != 0) {
1109                         ASSERT(n > 0);

1111                         error = uiomove(bp->b_rptr, n, UIO_READ, uiop);
1112                         if (error != 0) {
1113                                 freemsg(bp);
1114                                 *errorp = error;
1115                                 return (NULL);
1116                         }
1117                 }
```



```
1119                 bp->b_rptr += n;
1120                 while (bp != NULL && (bp->b_rptr >= bp->b_wptr)) {
1121                         nbp = bp;
1122                         bp = bp->b_cont;
1123                         freeb(nbp);
1124                 }
1125         } while (bp != NULL && uiop->uio_resid > 0);

1127         *errorp = 0;
1128         return (bp);
1129 }

1131 /*
1132  * Read a stream according to the mode flags in sd_flag:
1133  *
1134  * (default mode)               - Byte stream, msg boundaries are ignored
1135  * RD_MSGDIS (msg discard)      - Read on msg boundaries and throw away
1136  *                              any data remaining in msg
1137  * RD_MSGNODIS (msg non-discard) - Read on msg boundaries and put back
1138  *                              any remaining data on head of read queue
1139  *
1140  * Consume readable messages on the front of the queue until
1141  * ttolwp(curthread)->lwp_count
1142  * is satisfied, the readable messages are exhausted, or a message
1143  * boundary is reached in a message mode.  If no data was read and
1144  * the stream was not opened with the NDELAY flag, block until data arrives.
1145  * Otherwise return the data read and update the count.
1146  *
1147  * In default mode a 0 length message signifies end-of-file and terminates
1148  * a read in progress.  The 0 length message is removed from the queue
1149  * only if it is the only message read (no data is read).
1150  *
1151  * An attempt to read an M_PROTO or M_PCPROTO message results in an
1152  * EBADMSG error return, unless either RD_PROTDAT or RD_PROTDIS are set.
1153  * If RD_PROTDAT is set, M_PROTO and M_PCPROTO messages are read as data.
1154  * If RD_PROTDIS is set, the M_PROTO and M_PCPROTO parts of the message
1155  * are unlinked from and M_DATA blocks in the message, the protos are
1156  * thrown away, and the data is read.
1157  */
1158 /* ARGSUSED */
1159 int
1160 strread(struct vnode *vp, struct uio *uiop, cred_t *crp)
1161 {
1162         struct stdata *stp;
1163         mblk_t *bp, *nbp;
1164         queue_t *q;
1165         int error = 0;
1166         uint_t old_sd_flag;
1167         int first;
1168         char rflg;
1169         uint_t mark;            /* Contains MSG*MARK and _LASTMARK */
1170 #define _LASTMARK       0x8000  /* Distinct from MSG*MARK */
1171         short delim;
1172         unsigned char pri = 0;
1173         char waitflag;
1174         unsigned char type;

1176         TRACE_1(TR_FAC_STREAMS_FR,
1177             TR_STRREAD_ENTER, "strread:%p", vp);
1178         ASSERT(vp->v_stream);
1179         stp = vp->v_stream;

1181         mutex_enter(&stp->sd_lock);

1183         if ((error = i_straccess(stp, JCREAD)) != 0) {
```

```
1184                mutex_exit(&stp->sd_lock);
1185                return (error);
1186        }

1188        if (stp->sd_flag & (STRDERR|STPLEX)) {
1189                error = strgeterr(stp, STRDERR|STPLEX, 0);
1190                if (error != 0) {
1191                        mutex_exit(&stp->sd_lock);
1192                        return (error);
1193                }
1194        }

1196        /*
1197         * Loop terminates when uiop->uio_resid == 0.
1198         */
1199        rflg = 0;
1200        waitflag = READWAIT;
1201        q = _RD(stp->sd_wrq);
1202        for (;;) {
1203                ASSERT(MUTEX_HELD(&stp->sd_lock));
1204                old_sd_flag = stp->sd_flag;
1205                mark = 0;
1206                delim = 0;
1207                first = 1;
1208                while ((bp = strget(stp, q, uiop, first, &error)) == NULL) {
1209                        int done = 0;

1211                        ASSERT(MUTEX_HELD(&stp->sd_lock));

1213                        if (error != 0)
1214                                goto oops;

1216                        if (stp->sd_flag & (STRHUP|STREOF)) {
1217                                goto oops;
1218                        }
1219                        if (rflg && !(stp->sd_flag & STRDELIM)) {
1220                                goto oops;
1221                        }
1222                        /*
1223                         * If a read(fd,buf,0) has been done, there is no
1224                         * need to sleep. We always have zero bytes to
1225                         * return.
1226                         */
1227                        if (uiop->uio_resid == 0) {
1228                                goto oops;
1229                        }

1231                        qbackenable(q, 0);

1233                        TRACE_3(TR_FAC_STREAMS_FR, TR_STRREAD_WAIT,
1234                            "strread calls strwaitq:%p, %p, %p",
1235                            vp, uiop, crp);
1236                        if ((error = strwaitq(stp, waitflag, uiop->uio_resid,
1237                            uiop->uio_fmode, -1, &done)) != 0 || done) {
1238                                TRACE_3(TR_FAC_STREAMS_FR, TR_STRREAD_DONE,
1239                                    "strread error or done:%p, %p, %p",
1240                                    vp, uiop, crp);
1241                                if ((uiop->uio_fmode & FNDELAY) &&
1242                                    (stp->sd_flag & OLDNDELAY) &&
1243                                    (error == EAGAIN))
1244                                        error = 0;
1245                                goto oops;
1246                        }
1247                        TRACE_3(TR_FAC_STREAMS_FR, TR_STRREAD_AWAKE,
1248                            "strread awakes:%p, %p, %p", vp, uiop, crp);
1249                        if ((error = i_straccess(stp, JCREAD)) != 0) {
```



```
1250                                goto oops;
1251                        }
1252                        first = 0;
1253                }

1255                ASSERT(MUTEX_HELD(&stp->sd_lock));
1256                ASSERT(bp);
1257                pri = bp->b_band;
1258                /*
1259                 * Extract any mark information. If the message is not
1260                 * completely consumed this information will be put in the mblk
1261                 * that is putback.
1262                 * If MSGMARKNEXT is set and the message is completely consumed
1263                 * the STRATMARK flag will be set below. Likewise, if
1264                 * MSGNOTMARKNEXT is set and the message is
1265                 * completely consumed STRNOTATMARK will be set.
1266                 *
1267                 * For some unknown reason strread only breaks the read at the
1268                 * last mark.
1269                 */
1270                mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
1271                ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
1272                    (MSGMARKNEXT|MSGNOTMARKNEXT));
1273                if (mark != 0 && bp == stp->sd_mark) {
1274                        if (rflg) {
1275                                putback(stp, q, bp, pri);
1276                                goto oops;
1277                        }
1278                        mark |= _LASTMARK;
1279                        stp->sd_mark = NULL;
1280                }
1281                if ((stp->sd_flag & STRDELIM) && (bp->b_flag & MSGDELIM))
1282                        delim = 1;
1283                mutex_exit(&stp->sd_lock);

1285                if (STREAM_NEEDSERVICE(stp))
1286                        stream_runservice(stp);

1288                type = bp->b_datap->db_type;

1290                switch (type) {

1292                case M_DATA:
1293        ismdata:
1294                        if (msgnodata(bp)) {
1295                                if (mark || delim) {
1296                                        freemsg(bp);
1297                                } else if (rflg) {

1299                                        /*
1300                                         * If already read data put zero
1301                                         * length message back on queue else
1302                                         * free msg and return 0.
1303                                         */
1304                                        bp->b_band = pri;
1305                                        mutex_enter(&stp->sd_lock);
1306                                        putback(stp, q, bp, pri);
1307                                        mutex_exit(&stp->sd_lock);
1308                                } else {
1309                                        freemsg(bp);
1310                                }
1311                                error =  0;
1312                                goto oops1;
1313                        }

1315                        rflg = 1;
```

```
1316                        waitflag |= NOINTR;
1317                        bp = struiocopyout(bp, uiop, &error);
1318                        if (error != 0)
1319                                goto oops1;

1321                        mutex_enter(&stp->sd_lock);
1322                        if (bp) {
1323                                /*
1324                                 * Have remaining data in message.
1325                                 * Free msg if in discard mode.
1326                                 */
1327                                if (stp->sd_read_opt & RD_MSGDIS) {
1328                                        freemsg(bp);
1329                                } else {
1330                                        bp->b_band = pri;
1331                                        if ((mark & _LASTMARK) &&
1332                                            (stp->sd_mark == NULL))
1333                                                stp->sd_mark = bp;
1334                                        bp->b_flag |= mark & ~_LASTMARK;
1335                                        if (delim)
1336                                                bp->b_flag |= MSGDELIM;
1337                                        if (msgnodata(bp))
1338                                                freemsg(bp);
1339                                        else
1340                                                putback(stp, q, bp, pri);
1341                                }
1342                        } else {
1343                                /*
1344                                 * Consumed the complete message.
1345                                 * Move the MSG*MARKNEXT information
1346                                 * to the stream head just in case
1347                                 * the read queue becomes empty.
1348                                 *
1349                                 * If the stream head was at the mark
1350                                 * (STRATMARK) before we dropped sd_lock above
1351                                 * and some data was consumed then we have
1352                                 * moved past the mark thus STRATMARK is
1353                                 * cleared. However, if a message arrived in
1354                                 * strrput during the copyout above causing
1355                                 * STRATMARK to be set we can not clear that
1356                                 * flag.
1357                                 */
1358                                if (mark &
1359                                    (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
1360                                        if (mark & MSGMARKNEXT) {
1361                                                stp->sd_flag &= ~STRNOTATMARK;
1362                                                stp->sd_flag |= STRATMARK;
1363                                        } else if (mark & MSGNOTMARKNEXT) {
1364                                                stp->sd_flag &= ~STRATMARK;
1365                                                stp->sd_flag |= STRNOTATMARK;
1366                                        } else {
1367                                                stp->sd_flag &=
1368                                                    ~(STRATMARK|STRNOTATMARK);
1369                                        }
1370                                } else if (rflg && (old_sd_flag & STRATMARK)) {
1371                                        stp->sd_flag &= ~STRATMARK;
1372                                }
1373                        }

1375                        /*
1376                         * Check for signal messages at the front of the read
1377                         * queue and generate the signal(s) if appropriate.
1378                         * The only signal that can be on queue is M_SIG at
1379                         * this point.
1380                         */
1381                        while ((((bp = q->q_first)) != NULL) &&
```



```
1382                            (bp->b_datap->db_type == M_SIG)) {
1383                                bp = getq_noenab(q, 0);
1384                                /*
1385                                 * sd_lock is held so the content of the
1386                                 * read queue can not change.
1387                                 */
1388                                ASSERT(bp != NULL && DB_TYPE(bp) == M_SIG);
1389                                strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
1390                                mutex_exit(&stp->sd_lock);
1391                                freemsg(bp);
1392                                if (STREAM_NEEDSERVICE(stp))
1393                                        stream_runservice(stp);
1394                                mutex_enter(&stp->sd_lock);
1395                        }

1397                        if ((uiop->uio_resid == 0) || (mark & _LASTMARK) ||
1398                            delim ||
1399                            (stp->sd_read_opt & (RD_MSGDIS|RD_MSGNODIS))) {
1400                                goto oops;
1401                        }
1402                        continue;

1404                case M_SIG:
1405                        strsignal(stp, *bp->b_rptr, (int32_t)bp->b_band);
1406                        freemsg(bp);
1407                        mutex_enter(&stp->sd_lock);
1408                        continue;

1410                case M_PROTO:
1411                case M_PCPROTO:
1412                        /*
1413                         * Only data messages are readable.
1414                         * Any others generate an error, unless
1415                         * RD_PROTDIS or RD_PROTDAT is set.
1416                         */
1417                        if (stp->sd_read_opt & RD_PROTDAT) {
1418                                for (nbp = bp; nbp; nbp = nbp->b_next) {
1419                                        if ((nbp->b_datap->db_type ==
1420                                            M_PROTO) ||
1421                                            (nbp->b_datap->db_type ==
1422                                            M_PCPROTO)) {
1423                                                nbp->b_datap->db_type = M_DATA;
1424                                        } else {
1425                                                break;
1426                                        }
1427                                }
1428                                /*
1429                                 * clear stream head hi pri flag based on
1430                                 * first message
1431                                 */
1432                                if (type == M_PCPROTO) {
1433                                        mutex_enter(&stp->sd_lock);
1434                                        stp->sd_flag &= ~STRPRI;
1435                                        mutex_exit(&stp->sd_lock);
1436                                }
1437                                goto ismdata;
1438                        } else if (stp->sd_read_opt & RD_PROTDIS) {
1439                                /*
1440                                 * discard non-data messages
1441                                 */
1442                                while (bp &&
1443                                    ((bp->b_datap->db_type == M_PROTO) ||
1444                                    (bp->b_datap->db_type == M_PCPROTO))) {
1445                                        nbp = unlinkb(bp);
1446                                        freeb(bp);
1447                                        bp = nbp;
```

```
1448                                }
1449                                /*
1450                                 * clear stream head hi pri flag based on
1451                                 * first message
1452                                 */
1453                                if (type == M_PCPROTO) {
1454                                        mutex_enter(&stp->sd_lock);
1455                                        stp->sd_flag &= ~STRPRI;
1456                                        mutex_exit(&stp->sd_lock);
1457                                }
1458                                if (bp) {
1459                                        bp->b_band = pri;
1460                                        goto ismdata;
1461                                } else {
1462                                        break;
1463                                }
1464                        }
1465                        /* FALLTHRU */
1466                case M_PASSFP:
1467                        if ((bp->b_datap->db_type == M_PASSFP) &&
1468                            (stp->sd_read_opt & RD_PROTDIS)) {
1469                                freemsg(bp);
1470                                break;
1471                        }
1472                        mutex_enter(&stp->sd_lock);
1473                        putback(stp, q, bp, pri);
1474                        mutex_exit(&stp->sd_lock);
1475                        if (rflg == 0)
1476                                error = EBADMSG;
1477                        goto oops1;

1479                default:
1480                        /*
1481                         * Garbage on stream head read queue.
1482                         */
1483                        cmn_err(CE_WARN, "bad %x found at stream head\n",
1484                            bp->b_datap->db_type);
1485                        freemsg(bp);
1486                        goto oops1;
1487                }
1488                mutex_enter(&stp->sd_lock);
1489        }
1490 oops:
1491        mutex_exit(&stp->sd_lock);
1492 oops1:
1493        qbackenable(q, pri);
1494        return (error);
1495 #undef  _LASTMARK
1496 }

1498 /*
1499  * Default processing of M_PROTO/M_PCPROTO messages.
1500  * Determine which wakeups and signals are needed.
1501  * This can be replaced by a user-specified procedure for kernel users
1502  * of STREAMS.
1503  */
1504 /* ARGSUSED */
1505 mblk_t *
1506 strrput_proto(vnode_t *vp, mblk_t *mp,
1507     strwakeup_t *wakeups, strsigset_t *firstmsgsigs,
1508     strsigset_t *allmsgsigs, strpollset_t *pollwakeups)
1509 {
1510        *wakeups = RSLEEP;
1511        *allmsgsigs = 0;

1513        switch (mp->b_datap->db_type) {
```



```
1514        case M_PROTO:
1515                if (mp->b_band == 0) {
1516                        *firstmsgsigs = S_INPUT | S_RDNORM;
1517                        *pollwakeups = POLLIN | POLLRDNORM;
1518                } else {
1519                        *firstmsgsigs = S_INPUT | S_RDBAND;
1520                        *pollwakeups = POLLIN | POLLRDBAND;
1521                }
1522                break;
1523        case M_PCPROTO:
1524                *firstmsgsigs = S_HIPRI;
1525                *pollwakeups = POLLPRI;
1526                break;
1527        }
1528        return (mp);
1529 }

1531 /*
1532  * Default processing of everything but M_DATA, M_PROTO, M_PCPROTO and
1533  * M_PASSFP messages.
1534  * Determine which wakeups and signals are needed.
1535  * This can be replaced by a user-specified procedure for kernel users
1536  * of STREAMS.
1537  */
1538 /* ARGSUSED */
1539 mblk_t *
1540 strrput_misc(vnode_t *vp, mblk_t *mp,
1541     strwakeup_t *wakeups, strsigset_t *firstmsgsigs,
1542     strsigset_t *allmsgsigs, strpollset_t *pollwakeups)
1543 {
1544        *wakeups = 0;
1545        *firstmsgsigs = 0;
1546        *allmsgsigs = 0;
1547        *pollwakeups = 0;
1548        return (mp);
1549 }

1551 /*
1552  * Stream read put procedure.  Called from downstream driver/module
1553  * with messages for the stream head.  Data, protocol, and in-stream
1554  * signal messages are placed on the queue, others are handled directly.
1555  */
1556 int
1557 strrput(queue_t *q, mblk_t *bp)
1558 {
1559        struct stdata   *stp;
1560        ulong_t         rput_opt;
1561        strwakeup_t     wakeups;
1562        strsigset_t     firstmsgsigs;   /* Signals if first message on queue */
1563        strsigset_t     allmsgsigs;     /* Signals for all messages */
1564        strsigset_t     signals;        /* Signals events to generate */
1565        strpollset_t    pollwakeups;
1566        mblk_t          *nextbp;
1567        uchar_t         band = 0;
1568        int             hipri_sig;

1570        stp = (struct stdata *)q->q_ptr;
1571        /*
1572         * Use rput_opt for optimized access to the SR_ flags except
1573         * SR_POLLIN. That flag has to be checked under sd_lock since it
1574         * is modified by strpoll().
1575         */
1576        rput_opt = stp->sd_rput_opt;

1578        ASSERT(qclaimed(q));
1579        TRACE_2(TR_FAC_STREAMS_FR, TR_STRRPUT_ENTER,
```

```
1580             "strrput called with message type:q %p bp %p", q, bp);

1582         /*
1583          * Perform initial processing and pass to the parameterized functions.
1584          */
1585         ASSERT(bp->b_next == NULL);

1587         switch (bp->b_datap->db_type) {
1588         case M_DATA:
1589                 /*
1590                  * sockfs is the only consumer of STREOF and when it is set,
1591                  * it implies that the receiver is not interested in receiving
1592                  * any more data, hence the mblk is freed to prevent unnecessary
1593                  * message queueing at the stream head.
1594                  */
1595                 if (stp->sd_flag == STREOF) {
1596                         freemsg(bp);
1597                         return (0);
1598                 }
1599                 if ((rput_opt & SR_IGN_ZEROLEN) &&
1600                     bp->b_rptr == bp->b_wptr && msgnodata(bp)) {
1601                         /*
1602                          * Ignore zero-length M_DATA messages. These might be
1603                          * generated by some transports.
1604                          * The zero-length M_DATA messages, even if they
1605                          * are ignored, should effect the atmark tracking and
1606                          * should wake up a thread sleeping in strwaitmark.
1607                          */
1608                         mutex_enter(&stp->sd_lock);
1609                         if (bp->b_flag & MSGMARKNEXT) {
1610                                 /*
1611                                  * Record the position of the mark either
1612                                  * in q_last or in STRATMARK.
1613                                  */
1614                                 if (q->q_last != NULL) {
1615                                         q->q_last->b_flag &= ~MSGNOTMARKNEXT;
1616                                         q->q_last->b_flag |= MSGMARKNEXT;
1617                                 } else {
1618                                         stp->sd_flag &= ~STRNOTATMARK;
1619                                         stp->sd_flag |= STRATMARK;
1620                                 }
1621                         } else if (bp->b_flag & MSGNOTMARKNEXT) {
1622                                 /*
1623                                  * Record that this is not the position of
1624                                  * the mark either in q_last or in
1625                                  * STRNOTATMARK.
1626                                  */
1627                                 if (q->q_last != NULL) {
1628                                         q->q_last->b_flag &= ~MSGMARKNEXT;
1629                                         q->q_last->b_flag |= MSGNOTMARKNEXT;
1630                                 } else {
1631                                         stp->sd_flag &= ~STRATMARK;
1632                                         stp->sd_flag |= STRNOTATMARK;
1633                                 }
1634                         }
1635                         if (stp->sd_flag & RSLEEP) {
1636                                 stp->sd_flag &= ~RSLEEP;
1637                                 cv_broadcast(&q->q_wait);
1638                         }
1639                         mutex_exit(&stp->sd_lock);
1640                         freemsg(bp);
1641                         return (0);
1642                 }
1643                 wakeups = RSLEEP;
1644                 if (bp->b_band == 0) {
1645                         firstmsgsigs = S_INPUT | S_RDNORM;
```



```
1646                         pollwakeups = POLLIN | POLLRDNORM;
1647                 } else {
1648                         firstmsgsigs = S_INPUT | S_RDBAND;
1649                         pollwakeups = POLLIN | POLLRDBAND;
1650                 }
1651                 if (rput_opt & SR_SIGALLDATA)
1652                         allmsgsigs = firstmsgsigs;
1653                 else
1654                         allmsgsigs = 0;

1656                 mutex_enter(&stp->sd_lock);
1657                 if ((rput_opt & SR_CONSOL_DATA) &&
1658                     (q->q_last != NULL) &&
1659                     (bp->b_flag & (MSGMARK|MSGDELIM)) == 0) {
1660                         /*
1661                          * Consolidate an M_DATA message onto an M_DATA,
1662                          * M_PROTO, or M_PCPROTO by merging it with q_last.
1663                          * The consolidation does not take place if
1664                          * the old message is marked with either of the
1665                          * marks or the delim flag or if the new
1666                          * message is marked with MSGMARK. The MSGMARK
1667                          * check is needed to handle the odd semantics of
1668                          * MSGMARK where essentially the whole message
1669                          * is to be treated as marked.
1670                          * Carry any MSGMARKNEXT  and MSGNOTMARKNEXT from the
1671                          * new message to the front of the b_cont chain.
1672                          */
1673                         mblk_t *lbp = q->q_last;
1674                         unsigned char db_type = lbp->b_datap->db_type;

1676                         if ((db_type == M_DATA || db_type == M_PROTO ||
1677                             db_type == M_PCPROTO) &&
1678                             !(lbp->b_flag & (MSGDELIM|MSGMARK|MSGMARKNEXT))) {
1679                                 rmvq_noenab(q, lbp);
1680                                 /*
1681                                  * The first message in the b_cont list
1682                                  * tracks MSGMARKNEXT and MSGNOTMARKNEXT.
1683                                  * We need to handle the case where we
1684                                  * are appending:
1685                                  *
1686                                  * 1) a MSGMARKNEXT to a MSGNOTMARKNEXT.
1687                                  * 2) a MSGMARKNEXT to a plain message.
1688                                  * 3) a MSGNOTMARKNEXT to a plain message
1689                                  * 4) a MSGNOTMARKNEXT to a MSGNOTMARKNEXT
1690                                  *    message.
1691                                  *
1692                                  * Thus we never append a MSGMARKNEXT or
1693                                  * MSGNOTMARKNEXT to a MSGMARKNEXT message.
1694                                  */
1695                                 if (bp->b_flag & MSGMARKNEXT) {
1696                                         lbp->b_flag |= MSGMARKNEXT;
1697                                         lbp->b_flag &= ~MSGNOTMARKNEXT;
1698                                         bp->b_flag &= ~MSGMARKNEXT;
1699                                 } else if (bp->b_flag & MSGNOTMARKNEXT) {
1700                                         lbp->b_flag |= MSGNOTMARKNEXT;
1701                                         bp->b_flag &= ~MSGNOTMARKNEXT;
1702                                 }

1704                                 linkb(lbp, bp);
1705                                 bp = lbp;
1706                                 /*
1707                                  * The new message logically isn't the first
1708                                  * even though the q_first check below thinks
1709                                  * it is. Clear the firstmsgsigs to make it
1710                                  * not appear to be first.
1711                                  */
```

```
1712                                    firstmsgsigs = 0;
1713                            }
1714                    }
1715                    break;

1717            case M_PASSFP:
1718                    wakeups = RSLEEP;
1719                    allmsgsigs = 0;
1720                    if (bp->b_band == 0) {
1721                            firstmsgsigs = S_INPUT | S_RDNORM;
1722                            pollwakeups = POLLIN | POLLRDNORM;
1723                    } else {
1724                            firstmsgsigs = S_INPUT | S_RDBAND;
1725                            pollwakeups = POLLIN | POLLRDBAND;
1726                    }
1727                    mutex_enter(&stp->sd_lock);
1728                    break;

1730            case M_PROTO:
1731            case M_PCPROTO:
1732                    ASSERT(stp->sd_rprotofunc != NULL);
1733                    bp = (stp->sd_rprotofunc)(stp->sd_vnode, bp,
1734                        &wakeups, &firstmsgsigs, &allmsgsigs, &pollwakeups);
1735 #define ALLSIG  (S_INPUT|S_HIPRI|S_OUTPUT|S_MSG|S_ERROR|S_HANGUP|S_RDNORM|\
1736                  S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)
1737 #define ALLPOLL (POLLIN|POLLPRI|POLLOUT|POLLRDNORM|POLLWRNORM|POLLRDBAND|\
1738                  POLLWRBAND)

1740                    ASSERT((wakeups & ~(RSLEEP|WSLEEP)) == 0);
1741                    ASSERT((firstmsgsigs & ~ALLSIG) == 0);
1742                    ASSERT((allmsgsigs & ~ALLSIG) == 0);
1743                    ASSERT((pollwakeups & ~ALLPOLL) == 0);

1745                    mutex_enter(&stp->sd_lock);
1746                    break;

1748            default:
1749                    ASSERT(stp->sd_rmiscfunc != NULL);
1750                    bp = (stp->sd_rmiscfunc)(stp->sd_vnode, bp,
1751                        &wakeups, &firstmsgsigs, &allmsgsigs, &pollwakeups);
1752                    ASSERT((wakeups & ~(RSLEEP|WSLEEP)) == 0);
1753                    ASSERT((firstmsgsigs & ~ALLSIG) == 0);
1754                    ASSERT((allmsgsigs & ~ALLSIG) == 0);
1755                    ASSERT((pollwakeups & ~ALLPOLL) == 0);
1756 #undef  ALLSIG
1757 #undef  ALLPOLL
1758                    mutex_enter(&stp->sd_lock);
1759                    break;
1760            }
1761            ASSERT(MUTEX_HELD(&stp->sd_lock));

1763            /* By default generate superset of signals */
1764            signals = (firstmsgsigs | allmsgsigs);

1766            /*
1767             * The  proto and misc functions can return multiple messages
1768             * as a b_next chain. Such messages are processed separately.
1769             */
1770 one_more:
1771            hipri_sig = 0;
1772            if (bp == NULL) {
1773                    nextbp = NULL;
1774            } else {
1775                    nextbp = bp->b_next;
1776                    bp->b_next = NULL;
```



```
1778                    switch (bp->b_datap->db_type) {
1779                    case M_PCPROTO:
1780                            /*
1781                             * Only one priority protocol message is allowed at the
1782                             * stream head at a time.
1783                             */
1784                            if (stp->sd_flag & STRPRI) {
1785                                    TRACE_0(TR_FAC_STREAMS_FR, TR_STRRPUT_PROTERR,
1786                                        "M_PCPROTO already at head");
1787                                    freemsg(bp);
1788                                    mutex_exit(&stp->sd_lock);
1789                                    goto done;
1790                            }
1791                            stp->sd_flag |= STRPRI;
1792                            hipri_sig = 1;
1793                            /* FALLTHRU */
1794                    case M_DATA:
1795                    case M_PROTO:
1796                    case M_PASSFP:
1797                            band = bp->b_band;
1798                            /*
1799                             * Marking doesn't work well when messages
1800                             * are marked in more than one band.  We only
1801                             * remember the last message received, even if
1802                             * it is placed on the queue ahead of other
1803                             * marked messages.
1804                             */
1805                            if (bp->b_flag & MSGMARK)
1806                                    stp->sd_mark = bp;
1807                            (void) putq(q, bp);

1809                            /*
1810                             * If message is a PCPROTO message, always use
1811                             * firstmsgsigs to determine if a signal should be
1812                             * sent as strrput is the only place to send
1813                             * signals for PCPROTO. Other messages are based on
1814                             * the STRGETINPROG flag. The flag determines if
1815                             * strrput or (k)strgetmsg will be responsible for
1816                             * sending the signals, in the firstmsgsigs case.
1817                             */
1818                            if ((hipri_sig == 1) ||
1819                                (((stp->sd_flag & STRGETINPROG) == 0) &&
1820                                (q->q_first == bp)))
1821                                    signals = (firstmsgsigs | allmsgsigs);
1822                            else
1823                                    signals = allmsgsigs;
1824                            break;

1826                    default:
1827                            mutex_exit(&stp->sd_lock);
1828                            (void) strrput_nondata(q, bp);
1829                            mutex_enter(&stp->sd_lock);
1830                            break;
1831                    }
1832            }
1833            ASSERT(MUTEX_HELD(&stp->sd_lock));
1834            /*
1835             * Wake sleeping read/getmsg and cancel deferred wakeup
1836             */
1837            if (wakeups & RSLEEP)
1838                    stp->sd_wakeq &= ~RSLEEP;

1840            wakeups &= stp->sd_flag;
1841            if (wakeups & RSLEEP) {
1842                    stp->sd_flag &= ~RSLEEP;
1843                    cv_broadcast(&q->q_wait);
```

```
1844         }
1845         if (wakeups & WSLEEP) {
1846                 stp->sd_flag &= ~WSLEEP;
1847                 cv_broadcast(&_WR(q)->q_wait);
1848         }

1850         if (pollwakeups != 0) {
1851                 if (pollwakeups == (POLLIN | POLLRDNORM)) {
1852                         /*
1853                          * Can't use rput_opt since it was not
1854                          * read when sd_lock was held and SR_POLLIN is changed
1855                          * by strpoll() under sd_lock.
1856                          */
1857                         if (!(stp->sd_rput_opt & SR_POLLIN))
1858                                 goto no_pollwake;
1859                         stp->sd_rput_opt &= ~SR_POLLIN;
1860                 }
1861                 mutex_exit(&stp->sd_lock);
1862                 pollwakeup(&stp->sd_pollist, pollwakeups);
1863                 mutex_enter(&stp->sd_lock);
1864         }
1865 no_pollwake:

1867         /*
1868          * strsendsig can handle multiple signals with a
1869          * single call.
1870          */
1871         if (stp->sd_sigflags & signals)
1872                 strsendsig(stp->sd_siglist, signals, band, 0);
1873         mutex_exit(&stp->sd_lock);


1876 done:
1877         if (nextbp == NULL)
1878                 return (0);

1880         /*
1881          * Any signals were handled the first time.
1882          * Wakeups and pollwakeups are redone to avoid any race
1883          * conditions - all the messages are not queued until the
1884          * last message has been processed by strrput.
1885          */
1886         bp = nextbp;
1887         signals = firstmsgsigs = allmsgsigs = 0;
1888         mutex_enter(&stp->sd_lock);
1889         goto one_more;
1890 }

1892 static void
1893 log_dupioc(queue_t *rq, mblk_t *bp)
1894 {
1895         queue_t *wq, *qp;
1896         char *modnames, *mnp, *dname;
1897         size_t maxmodstr;
1898         boolean_t islast;

1900         /*
1901          * Allocate a buffer large enough to hold the names of nstrpush modules
1902          * and one driver, with spaces between and NUL terminator.  If we can't
1903          * get memory, then we'll just log the driver name.
1904          */
1905         maxmodstr = nstrpush * (FMNAMESZ + 1);
1906         mnp = modnames = kmem_alloc(maxmodstr, KM_NOSLEEP);

1908         /* march down write side to print log message down to the driver */
1909         wq = WR(rq);
```



```
1911         /* make sure q_next doesn't shift around while we're grabbing data */
1912         claimstr(wq);
1913         qp = wq->q_next;
1914         do {
1915                 dname = Q2NAME(qp);
1916                 islast = !SAMESTR(qp) || qp->q_next == NULL;
1917                 if (modnames == NULL) {
1918                         /*
1919                          * If we don't have memory, then get the driver name in
1920                          * the log where we can see it.  Note that memory
1921                          * pressure is a possible cause of these sorts of bugs.
1922                          */
1923                         if (islast) {
1924                                 modnames = dname;
1925                                 maxmodstr = 0;
1926                         }
1927                 } else {
1928                         mnp += snprintf(mnp, FMNAMESZ + 1, "%s", dname);
1929                         if (!islast)
1930                                 *mnp++ = ' ';
1931                 }
1932                 qp = qp->q_next;
1933         } while (!islast);
1934         releasestr(wq);
1935         /* Cannot happen unless stream head is corrupt. */
1936         ASSERT(modnames != NULL);
1937         (void) strlog(rq->q_qinfo->qi_minfo->mi_idnum, 0, 1,
1938             SL_CONSOLE|SL_TRACE|SL_ERROR,
1939             "Warning: stream %p received duplicate %X M_IOC%s; module list: %s",
1940             rq->q_ptr, ((struct iocblk *)bp->b_rptr)->ioc_cmd,
1941             (DB_TYPE(bp) == M_IOCACK ? "ACK" : "NAK"), modnames);
1942         if (maxmodstr != 0)
1943                 kmem_free(modnames, maxmodstr);
1944 }

1946 int
1947 strrput_nondata(queue_t *q, mblk_t *bp)
1948 {
1949         struct stdata *stp;
1950         struct iocblk *iocbp;
1951         struct stroptions *sop;
1952         struct copyreq *reqp;
1953         struct copyresp *resp;
1954         unsigned char bpri;
1955         unsigned char  flushed_already = 0;

1957         stp = (struct stdata *)q->q_ptr;

1959         ASSERT(!(stp->sd_flag & STPLEX));
1960         ASSERT(qclaimed(q));

1962         switch (bp->b_datap->db_type) {
1963         case M_ERROR:
1964                 /*
1965                  * An error has occurred downstream, the errno is in the first
1966                  * bytes of the message.
1967                  */
1968                 if ((bp->b_wptr - bp->b_rptr) == 2) {   /* New flavor */
1969                         unsigned char rw = 0;

1971                         mutex_enter(&stp->sd_lock);
1972                         if (*bp->b_rptr != NOERROR) {   /* read error */
1973                                 if (*bp->b_rptr != 0) {
1974                                         if (stp->sd_flag & STRDERR)
1975                                                 flushed_already |= FLUSHR;
```

```
1976                                    stp->sd_flag |= STRDERR;
1977                                    rw |= FLUSHR;
1978                            } else {
1979                                    stp->sd_flag &= ~STRDERR;
1980                            }
1981                            stp->sd_rerror = *bp->b_rptr;
1982                    }
1983                    bp->b_rptr++;
1984                    if (*bp->b_rptr != NOERROR) {   /* write error */
1985                            if (*bp->b_rptr != 0) {
1986                                    if (stp->sd_flag & STWRERR)
1987                                            flushed_already |= FLUSHW;
1988                                    stp->sd_flag |= STWRERR;
1989                                    rw |= FLUSHW;
1990                            } else {
1991                                    stp->sd_flag &= ~STWRERR;
1992                            }
1993                            stp->sd_werror = *bp->b_rptr;
1994                    }
1995                    if (rw) {
1996                            TRACE_2(TR_FAC_STREAMS_FR, TR_STRRPUT_WAKE,
1997                                "strrput cv_broadcast:q %p, bp %p",
1998                                q, bp);
1999                            cv_broadcast(&q->q_wait); /* readers */
2000                            cv_broadcast(&_WR(q)->q_wait); /* writers */
2001                            cv_broadcast(&stp->sd_monitor); /* ioctllers */

2003                            mutex_exit(&stp->sd_lock);
2004                            pollwakeup(&stp->sd_pollist, POLLERR);
2005                            mutex_enter(&stp->sd_lock);

2007                            if (stp->sd_sigflags & S_ERROR)
2008                                    strsendsig(stp->sd_siglist, S_ERROR, 0,
2009                                        ((rw & FLUSHR) ? stp->sd_rerror :
2010                                        stp->sd_werror));
2011                            mutex_exit(&stp->sd_lock);
2012                            /*
2013                             * Send the M_FLUSH only
2014                             * for the first M_ERROR
2015                             * message on the stream
2016                             */
2017                            if (flushed_already == rw) {
2018                                    freemsg(bp);
2019                                    return (0);
2020                            }

2022                            bp->b_datap->db_type = M_FLUSH;
2023                            *bp->b_rptr = rw;
2024                            bp->b_wptr = bp->b_rptr + 1;
2025                            /*
2026                             * Protect against the driver
2027                             * passing up messages after
2028                             * it has done a qprocsoff
2029                             */
2030                            if (_OTHERQ(q)->q_next == NULL)
2031                                    freemsg(bp);
2032                            else
2033                                    qreply(q, bp);
2034                            return (0);
2035                    } else
2036                            mutex_exit(&stp->sd_lock);
2037            } else if (*bp->b_rptr != 0) {          /* Old flavor */
2038                            if (stp->sd_flag & (STRDERR|STWRERR))
2039                                    flushed_already = FLUSHRW;
2040                            mutex_enter(&stp->sd_lock);
2041                            stp->sd_flag |= (STRDERR|STWRERR);
```



```
2042                            stp->sd_rerror = *bp->b_rptr;
2043                            stp->sd_werror = *bp->b_rptr;
2044                            TRACE_2(TR_FAC_STREAMS_FR,
2045                                TR_STRRPUT_WAKE2,
2046                                "strrput wakeup #2:q %p, bp %p", q, bp);
2047                            cv_broadcast(&q->q_wait); /* the readers */
2048                            cv_broadcast(&_WR(q)->q_wait); /* the writers */
2049                            cv_broadcast(&stp->sd_monitor); /* ioctllers */

2051                            mutex_exit(&stp->sd_lock);
2052                            pollwakeup(&stp->sd_pollist, POLLERR);
2053                            mutex_enter(&stp->sd_lock);

2055                            if (stp->sd_sigflags & S_ERROR)
2056                                    strsendsig(stp->sd_siglist, S_ERROR, 0,
2057                                        (stp->sd_werror ? stp->sd_werror :
2058                                        stp->sd_rerror));
2059                            mutex_exit(&stp->sd_lock);

2061                            /*
2062                             * Send the M_FLUSH only
2063                             * for the first M_ERROR
2064                             * message on the stream
2065                             */
2066                            if (flushed_already != FLUSHRW) {
2067                                    bp->b_datap->db_type = M_FLUSH;
2068                                    *bp->b_rptr = FLUSHRW;
2069                                    /*
2070                                     * Protect against the driver passing up
2071                                     * messages after it has done a
2072                                     * qprocsoff.
2073                                     */
2074                            if (_OTHERQ(q)->q_next == NULL)
2075                                    freemsg(bp);
2076                            else
2077                                    qreply(q, bp);
2078                            return (0);
2079                            }
2080                    }
2081                    freemsg(bp);
2082                    return (0);

2084            case M_HANGUP:

2086                    freemsg(bp);
2087                    mutex_enter(&stp->sd_lock);
2088                    stp->sd_werror = ENXIO;
2089                    stp->sd_flag |= STRHUP;
2090                    stp->sd_flag &= ~(WSLEEP|RSLEEP);

2092                    /*
2093                     * send signal if controlling tty
2094                     */

2096                    if (stp->sd_sidp) {
2097                            prsignal(stp->sd_sidp, SIGHUP);
2098                            if (stp->sd_sidp != stp->sd_pgidp)
2099                                    pgsignal(stp->sd_pgidp, SIGTSTP);
2100                    }

2102                    /*
2103                     * wake up read, write, and exception pollers and
2104                     * reset wakeup mechanism.
2105                     */
2106                    cv_broadcast(&q->q_wait);       /* the readers */
2107                    cv_broadcast(&_WR(q)->q_wait);  /* the writers */
```

```
2108                cv_broadcast(&stp->sd_monitor); /* the ioctllers */
2109                strhup(stp);
2110                mutex_exit(&stp->sd_lock);
2111                return (0);

2113        case M_UNHANGUP:
2114                freemsg(bp);
2115                mutex_enter(&stp->sd_lock);
2116                stp->sd_werror = 0;
2117                stp->sd_flag &= ~STRHUP;
2118                mutex_exit(&stp->sd_lock);
2119                return (0);

2121        case M_SIG:
2122                /*
2123                 * Someone downstream wants to post a signal.  The
2124                 * signal to post is contained in the first byte of the
2125                 * message.  If the message would go on the front of
2126                 * the queue, send a signal to the process group
2127                 * (if not SIGPOLL) or to the siglist processes
2128                 * (SIGPOLL).  If something is already on the queue,
2129                 * OR if we are delivering a delayed suspend (*sigh*
2130                 * another "tty" hack) and there's no one sleeping already,
2131                 * just enqueue the message.
2132                 */
2133                mutex_enter(&stp->sd_lock);
2134                if (q->q_first || (*bp->b_rptr == SIGTSTP &&
2135                    !(stp->sd_flag & RSLEEP))) {
2136                        (void) putq(q, bp);
2137                        mutex_exit(&stp->sd_lock);
2138                        return (0);
2139                }
2140                mutex_exit(&stp->sd_lock);
2141                /* FALLTHRU */

2143        case M_PCSIG:
2144                /*
2145                 * Don't enqueue, just post the signal.
2146                 */
2147                strsignal(stp, *bp->b_rptr, 0L);
2148                freemsg(bp);
2149                return (0);

2151        case M_CMD:
2152                if (MBLKL(bp) != sizeof (cmdblk_t)) {
2153                        freemsg(bp);
2154                        return (0);
2155                }

2157                mutex_enter(&stp->sd_lock);
2158                if (stp->sd_flag & STRCMDWAIT) {
2159                        ASSERT(stp->sd_cmdblk == NULL);
2160                        stp->sd_cmdblk = bp;
2161                        cv_broadcast(&stp->sd_monitor);
2162                        mutex_exit(&stp->sd_lock);
2163                } else {
2164                        mutex_exit(&stp->sd_lock);
2165                        freemsg(bp);
2166                }
2167                return (0);

2169        case M_FLUSH:
2170                /*
2171                 * Flush queues.  The indication of which queues to flush
2172                 * is in the first byte of the message.  If the read queue
2173                 * is specified, then flush it.  If FLUSHBAND is set, just
```



```
2174                 * flush the band specified by the second byte of the message.
2175                 *
2176                 * If a module has issued a M_SETOPT to not flush hi
2177                 * priority messages off of the stream head, then pass this
2178                 * flag into the flushq code to preserve such messages.
2179                 */

2181                if (*bp->b_rptr & FLUSHR) {
2182                        mutex_enter(&stp->sd_lock);
2183                        if (*bp->b_rptr & FLUSHBAND) {
2184                                ASSERT((bp->b_wptr - bp->b_rptr) >= 2);
2185                                flushband(q, *(bp->b_rptr + 1), FLUSHALL);
2186                        } else
2187                                flushq_common(q, FLUSHALL,
2188                                    stp->sd_read_opt & RFLUSHPCPROT);
2189                        if ((q->q_first == NULL) ||
2190                            (q->q_first->b_datap->db_type < QPCTL))
2191                                stp->sd_flag &= ~STRPRI;
2192                        else {
2193                                ASSERT(stp->sd_flag & STRPRI);
2194                        }
2195                        mutex_exit(&stp->sd_lock);
2196                }
2197                if ((*bp->b_rptr & FLUSHW) && !(bp->b_flag & MSGNOLOOP)) {
2198                        *bp->b_rptr &= ~FLUSHR;
2199                        bp->b_flag |= MSGNOLOOP;
2200                        /*
2201                         * Protect against the driver passing up
2202                         * messages after it has done a qprocsoff.
2203                         */
2204                        if (_OTHERQ(q)->q_next == NULL)
2205                                freemsg(bp);
2206                        else
2207                                qreply(q, bp);
2208                        return (0);
2209                }
2210                freemsg(bp);
2211                return (0);

2213        case M_IOCACK:
2214        case M_IOCNAK:
2215                iocbp = (struct iocblk *)bp->b_rptr;
2216                /*
2217                 * If not waiting for ACK or NAK then just free msg.
2218                 * If incorrect id sequence number then just free msg.
2219                 * If already have ACK or NAK for user then this is a
2220                 *    duplicate, display a warning and free the msg.
2221                 */
2222                mutex_enter(&stp->sd_lock);
2223                if ((stp->sd_flag & IOCWAIT) == 0 || stp->sd_iocblk ||
2224                    (stp->sd_iocid != iocbp->ioc_id)) {
2225                        /*
2226                         * If the ACK/NAK is a dup, display a message
2227                         * Dup is when sd_iocid == ioc_id, and
2228                         * sd_iocblk == <valid ptr> or -1 (the former
2229                         * is when an ioctl has been put on the stream
2230                         * head, but has not yet been consumed, the
2231                         * later is when it has been consumed).
2232                         */
2233                        if ((stp->sd_iocid == iocbp->ioc_id) &&
2234                            (stp->sd_iocblk != NULL)) {
2235                                log_dupioc(q, bp);
2236                        }
2237                        freemsg(bp);
2238                        mutex_exit(&stp->sd_lock);
2239                        return (0);
```

```
2240                }

2242                /*
2243                 * Assign ACK or NAK to user and wake up.
2244                 */
2245                stp->sd_iocblk = bp;
2246                cv_broadcast(&stp->sd_monitor);
2247                mutex_exit(&stp->sd_lock);
2248                return (0);

2250        case M_COPYIN:
2251        case M_COPYOUT:
2252                reqp = (struct copyreq *)bp->b_rptr;

2254                /*
2255                 * If not waiting for ACK or NAK then just fail request.
2256                 * If already have ACK, NAK, or copy request, then just
2257                 * fail request.
2258                 * If incorrect id sequence number then just fail request.
2259                 */
2260                mutex_enter(&stp->sd_lock);
2261                if ((stp->sd_flag & IOCWAIT) == 0 || stp->sd_iocblk ||
2262                    (stp->sd_iocid != reqp->cq_id)) {
2263                        if (bp->b_cont) {
2264                                freemsg(bp->b_cont);
2265                                bp->b_cont = NULL;
2266                        }
2267                        bp->b_datap->db_type = M_IOCDATA;
2268                        bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
2269                        resp = (struct copyresp *)bp->b_rptr;
2270                        resp->cp_rval = (caddr_t)1;     /* failure */
2271                        mutex_exit(&stp->sd_lock);
2272                        putnext(stp->sd_wrq, bp);
2273                        return (0);
2274                }

2276                /*
2277                 * Assign copy request to user and wake up.
2278                 */
2279                stp->sd_iocblk = bp;
2280                cv_broadcast(&stp->sd_monitor);
2281                mutex_exit(&stp->sd_lock);
2282                return (0);

2284        case M_SETOPTS:
2285                /*
2286                 * Set stream head options (read option, write offset,
2287                 * min/max packet size, and/or high/low water marks for
2288                 * the read side only).
2289                 */

2291                bpri = 0;
2292                sop = (struct stroptions *)bp->b_rptr;
2293                mutex_enter(&stp->sd_lock);
2294                if (sop->so_flags & SO_READOPT) {
2295                        switch (sop->so_readopt & RMODEMASK) {
2296                        case RNORM:
2297                                stp->sd_read_opt &= ~(RD_MSGDIS | RD_MSGNODIS);
2298                                break;

2300                        case RMSGD:
2301                                stp->sd_read_opt =
2302                                    ((stp->sd_read_opt & ~RD_MSGNODIS) |
2303                                    RD_MSGDIS);
2304                                break;
```



```
2306                        case RMSGN:
2307                                stp->sd_read_opt =
2308                                    ((stp->sd_read_opt & ~RD_MSGDIS) |
2309                                    RD_MSGNODIS);
2310                                break;
2311                        }
2312                        switch (sop->so_readopt & RPROTMASK) {
2313                        case RPROTNORM:
2314                                stp->sd_read_opt &= ~(RD_PROTDAT | RD_PROTDIS);
2315                                break;

2317                        case RPROTDAT:
2318                                stp->sd_read_opt =
2319                                    ((stp->sd_read_opt & ~RD_PROTDIS) |
2320                                    RD_PROTDAT);
2321                                break;

2323                        case RPROTDIS:
2324                                stp->sd_read_opt =
2325                                    ((stp->sd_read_opt & ~RD_PROTDAT) |
2326                                    RD_PROTDIS);
2327                                break;
2328                        }
2329                        switch (sop->so_readopt & RFLUSHMASK) {
2330                        case RFLUSHPCPROT:
2331                                /*
2332                                 * This sets the stream head to NOT flush
2333                                 * M_PCPROTO messages.
2334                                 */
2335                                stp->sd_read_opt |= RFLUSHPCPROT;
2336                                break;
2337                        }
2338                }
2339                if (sop->so_flags & SO_ERROPT) {
2340                        switch (sop->so_erropt & RERRMASK) {
2341                        case RERRNORM:
2342                                stp->sd_flag &= ~STRDERRNONPERSIST;
2343                                break;
2344                        case RERRNONPERSIST:
2345                                stp->sd_flag |= STRDERRNONPERSIST;
2346                                break;
2347                        }
2348                        switch (sop->so_erropt & WERRMASK) {
2349                        case WERRNORM:
2350                                stp->sd_flag &= ~STWRERRNONPERSIST;
2351                                break;
2352                        case WERRNONPERSIST:
2353                                stp->sd_flag |= STWRERRNONPERSIST;
2354                                break;
2355                        }
2356                }
2357                if (sop->so_flags & SO_COPYOPT) {
2358                        if (sop->so_copyopt & ZCVMSAFE) {
2359                                stp->sd_copyflag |= STZCVMSAFE;
2360                                stp->sd_copyflag &= ~STZCVMUNSAFE;
2361                        } else if (sop->so_copyopt & ZCVMUNSAFE) {
2362                                stp->sd_copyflag |= STZCVMUNSAFE;
2363                                stp->sd_copyflag &= ~STZCVMSAFE;
2364                        }

2366                        if (sop->so_copyopt & COPYCACHED) {
2367                                stp->sd_copyflag |= STRCOPYCACHED;
2368                        }
2369                }
2370                if (sop->so_flags & SO_WROFF)
2371                        stp->sd_wroff = sop->so_wroff;
```

```
2372                if (sop->so_flags & SO_TAIL)
2373                        stp->sd_tail = sop->so_tail;
2374                if (sop->so_flags & SO_MINPSZ)
2375                        q->q_minpsz = sop->so_minpsz;
2376                if (sop->so_flags & SO_MAXPSZ)
2377                        q->q_maxpsz = sop->so_maxpsz;
2378                if (sop->so_flags & SO_MAXBLK)
2379                        stp->sd_maxblk = sop->so_maxblk;
2380                if (sop->so_flags & SO_HIWAT) {
2381                        if (sop->so_flags & SO_BAND) {
2382                                if (strqset(q, QHIWAT,
2383                                    sop->so_band, sop->so_hiwat)) {
2384                                        cmn_err(CE_WARN, "strrput: could not "
2385                                            "allocate qband\n");
2386                                } else {
2387                                        bpri = sop->so_band;
2388                                }
2389                        } else {
2390                                q->q_hiwat = sop->so_hiwat;
2391                        }
2392                }
2393                if (sop->so_flags & SO_LOWAT) {
2394                        if (sop->so_flags & SO_BAND) {
2395                                if (strqset(q, QLOWAT,
2396                                    sop->so_band, sop->so_lowat)) {
2397                                        cmn_err(CE_WARN, "strrput: could not "
2398                                            "allocate qband\n");
2399                                } else {
2400                                        bpri = sop->so_band;
2401                                }
2402                        } else {
2403                                q->q_lowat = sop->so_lowat;
2404                        }
2405                }
2406                if (sop->so_flags & SO_MREADON)
2407                        stp->sd_flag |= SNDMREAD;
2408                if (sop->so_flags & SO_MREADOFF)
2409                        stp->sd_flag &= ~SNDMREAD;
2410                if (sop->so_flags & SO_NDELON)
2411                        stp->sd_flag |= OLDNDELAY;
2412                if (sop->so_flags & SO_NDELOFF)
2413                        stp->sd_flag &= ~OLDNDELAY;
2414                if (sop->so_flags & SO_ISTTY)
2415                        stp->sd_flag |= STRISTTY;
2416                if (sop->so_flags & SO_ISNTTY)
2417                        stp->sd_flag &= ~STRISTTY;
2418                if (sop->so_flags & SO_TOSTOP)
2419                        stp->sd_flag |= STRTOSTOP;
2420                if (sop->so_flags & SO_TONSTOP)
2421                        stp->sd_flag &= ~STRTOSTOP;
2422                if (sop->so_flags & SO_DELIM)
2423                        stp->sd_flag |= STRDELIM;
2424                if (sop->so_flags & SO_NODELIM)
2425                        stp->sd_flag &= ~STRDELIM;

2427                mutex_exit(&stp->sd_lock);
2428                freemsg(bp);

2430                /* Check backenable in case the water marks changed */
2431                qbackenable(q, bpri);
2432                return (0);

2434        /*
2435         * The following set of cases deal with situations where two stream
2436         * heads are connected to each other (twisted streams).  These messages
2437         * have no meaning at the stream head.
```



```
2438         */
2439        case M_BREAK:
2440        case M_CTL:
2441        case M_DELAY:
2442        case M_START:
2443        case M_STOP:
2444        case M_IOCDATA:
2445        case M_STARTI:
2446        case M_STOPI:
2447                freemsg(bp);
2448                return (0);

2450        case M_IOCTL:
2451                /*
2452                 * Always NAK this condition
2453                 * (makes no sense)
2454                 * If there is one or more threads in the read side
2455                 * rwnext we have to defer the nacking until that thread
2456                 * returns (in strget).
2457                 */
2458                mutex_enter(&stp->sd_lock);
2459                if (stp->sd_struiodnak != 0) {
2460                        /*
2461                         * Defer NAK to the streamhead. Queue at the end
2462                         * the list.
2463                         */
2464                        mblk_t *mp = stp->sd_struionak;

2466                        while (mp && mp->b_next)
2467                                mp = mp->b_next;
2468                        if (mp)
2469                                mp->b_next = bp;
2470                        else
2471                                stp->sd_struionak = bp;
2472                        bp->b_next = NULL;
2473                        mutex_exit(&stp->sd_lock);
2474                        return (0);
2475                }
2476                mutex_exit(&stp->sd_lock);

2478                bp->b_datap->db_type = M_IOCNAK;
2479                /*
2480                 * Protect against the driver passing up
2481                 * messages after it has done a qprocsoff.
2482                 */
2483                if (_OTHERQ(q)->q_next == NULL)
2484                        freemsg(bp);
2485                else
2486                        qreply(q, bp);
2487                return (0);

2489        default:
2490 #ifdef DEBUG
2491                cmn_err(CE_WARN,
2492                    "bad message type %x received at stream head\n",
2493                    bp->b_datap->db_type);
2494 #endif
2495                freemsg(bp);
2496                return (0);
2497        }

2499        /* NOTREACHED */
2500 }

2502 /*
2503  * Check if the stream pointed to by ‘stp’ can be written to, and return an
```

```
 * error code if not.  If 'eiohup' is set, then return EIO if STRHUP is set.
 * If 'sigpipeok' is set and the SW_SIGPIPE option is enabled on the stream,
 * then always return EPIPE and send a SIGPIPE to the invoking thread.
 */
static int
strwriteable(struct stdata *stp, boolean_t eiohup, boolean_t sigpipeok)
{
        int error;

        ASSERT(MUTEX_HELD(&stp->sd_lock));

        /*
         * For modem support, POSIX states that on writes, EIO should
         * be returned if the stream has been hung up.
         */
        if (eiohup && (stp->sd_flag & (STPLEX|STRHUP)) == STRHUP)
                error = EIO;
        else
                error = strgeterr(stp, STRHUP|STPLEX|STWRERR, 0);

        if (error != 0) {
                if (!(stp->sd_flag & STPLEX) &&
                    (stp->sd_wput_opt & SW_SIGPIPE) && sigpipeok) {
                        tsignal(curthread, SIGPIPE);
                        error = EPIPE;
                }
        }

        return (error);
}

/*
 * Copyin and send data down a stream.
 * The caller will allocate and copyin any control part that precedes the
 * message and pass that in as mctl.
 *
 * Caller should *not* hold sd_lock.
 * When EWOULDBLOCK is returned the caller has to redo the canputnext
 * under sd_lock in order to avoid missing a backenabling wakeup.
 *
 * Use iosize = -1 to not send any M_DATA. iosize = 0 sends zero-length M_DATA.
 *
 * Set MSG_IGNFLOW in flags to ignore flow control for hipri messages.
 * For sync streams we can only ignore flow control by reverting to using
 * putnext.
 *
 * If sd_maxblk is less than *iosize this routine might return without
 * transferring all of *iosize. In all cases, on return *iosize will contain
 * the amount of data that was transferred.
 */
static int
strput(struct stdata *stp, mblk_t *mctl, struct uio *uiop, ssize_t *iosize,
    int b_flag, int pri, int flags)
{
        struiod_t uiod;
        mblk_t *mp;
        queue_t *wqp = stp->sd_wrq;
        int error = 0;
        ssize_t count = *iosize;

        ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));

        if (uiop != NULL && count >= 0)
                flags |= stp->sd_struiowrq ? STRUIO_POSTPONE : 0;

        if (!(flags & STRUIO_POSTPONE)) {
```



```
                /*
                 * Use regular canputnext, strmakedata, putnext sequence.
                 */
                if (pri == 0) {
                        if (!canputnext(wqp) && !(flags & MSG_IGNFLOW)) {
                                freemsg(mctl);
                                return (EWOULDBLOCK);
                        }
                } else {
                        if (!(flags & MSG_IGNFLOW) && !bcanputnext(wqp, pri)) {
                                freemsg(mctl);
                                return (EWOULDBLOCK);
                        }
                }

                if ((error = strmakedata(iosize, uiop, stp, flags,
                    &mp)) != 0) {
                        freemsg(mctl);
                        /*
                         * need to change return code to ENOMEM
                         * so that this is not confused with
                         * flow control, EAGAIN.
                         */

                        if (error == EAGAIN)
                                return (ENOMEM);
                        else
                                return (error);
                }
                if (mctl != NULL) {
                        if (mctl->b_cont == NULL)
                                mctl->b_cont = mp;
                        else if (mp != NULL)
                                linkb(mctl, mp);
                        mp = mctl;
                } else if (mp == NULL)
                        return (0);

                mp->b_flag |= b_flag;
                mp->b_band = (uchar_t)pri;

                if (flags & MSG_IGNFLOW) {
                        /*
                         * XXX Hack: Don't get stuck running service
                         * procedures. This is needed for sockfs when
                         * sending the unbind message out of the rput
                         * procedure - we don't want a put procedure
                         * to run service procedures.
                         */
                        putnext(wqp, mp);
                } else {
                        stream_willservice(stp);
                        putnext(wqp, mp);
                        stream_runservice(stp);
                }
                return (0);
        }
        /*
         * Stream supports rwnext() for the write side.
         */
        if ((error = strmakedata(iosize, uiop, stp, flags, &mp)) != 0) {
                freemsg(mctl);
                /*
                 * map EAGAIN to ENOMEM since EAGAIN means "flow controlled".
                 */
                return (error == EAGAIN ? ENOMEM : error);
```

```
2636         }
2637         if (mctl != NULL) {
2638                 if (mctl->b_cont == NULL)
2639                         mctl->b_cont = mp;
2640                 else if (mp != NULL)
2641                         linkb(mctl, mp);
2642                 mp = mctl;
2643         } else if (mp == NULL) {
2644                 return (0);
2645         }

2647         mp->b_flag |= b_flag;
2648         mp->b_band = (uchar_t)pri;

2650         (void) uiodup(uiop, &uiod.d_uio, uiod.d_iov,
2651             sizeof (uiod.d_iov) / sizeof (*uiod.d_iov));
2652         uiod.d_uio.uio_offset = 0;
2653         uiod.d_mp = mp;
2654         error = rwnext(wqp, &uiod);
2655         if (! uiod.d_mp) {
2656                 uioskip(uiop, *iosize);
2657                 return (error);
2658         }
2659         ASSERT(mp == uiod.d_mp);
2660         if (error == EINVAL) {
2661                 /*
2662                  * The stream plumbing must have changed while
2663                  * we were away, so just turn off rwnext()s.
2664                  */
2665                 error = 0;
2666         } else if (error == EBUSY || error == EWOULDBLOCK) {
2667                 /*
2668                  * Couldn't enter a perimeter or took a page fault,
2669                  * so fall-back to putnext().
2670                  */
2671                 error = 0;
2672         } else {
2673                 freemsg(mp);
2674                 return (error);
2675         }
2676         /* Have to check canput before consuming data from the uio */
2677         if (pri == 0) {
2678                 if (!canputnext(wqp) && !(flags & MSG_IGNFLOW)) {
2679                         freemsg(mp);
2680                         return (EWOULDBLOCK);
2681                 }
2682         } else {
2683                 if (!bcanputnext(wqp, pri) && !(flags & MSG_IGNFLOW)) {
2684                         freemsg(mp);
2685                         return (EWOULDBLOCK);
2686                 }
2687         }
2688         ASSERT(mp == uiod.d_mp);
2689         /* Copyin data from the uio */
2690         if ((error = struioget(wqp, mp, &uiod, 0)) != 0) {
2691                 freemsg(mp);
2692                 return (error);
2693         }
2694         uioskip(uiop, *iosize);
2695         if (flags & MSG_IGNFLOW) {
2696                 /*
2697                  * XXX Hack: Don't get stuck running service procedures.
2698                  * This is needed for sockfs when sending the unbind message
2699                  * out of the rput procedure - we don't want a put procedure
2700                  * to run service procedures.
2701                  */
```



```
2702                 putnext(wqp, mp);
2703         } else {
2704                 stream_willservice(stp);
2705                 putnext(wqp, mp);
2706                 stream_runservice(stp);
2707         }
2708         return (0);
2709 }

2711 /*
2712  * Write attempts to break the write request into messages conforming
2713  * with the minimum and maximum packet sizes set downstream.
2714  *
2715  * Write will not block if downstream queue is full and
2716  * O_NDELAY is set, otherwise it will block waiting for the queue to get room.
2717  *
2718  * A write of zero bytes gets packaged into a zero length message and sent
2719  * downstream like any other message.
2720  *
2721  * If buffers of the requested sizes are not available, the write will
2722  * sleep until the buffers become available.
2723  *
2724  * Write (if specified) will supply a write offset in a message if it
2725  * makes sense. This can be specified by downstream modules as part of
2726  * a M_SETOPTS message.  Write will not supply the write offset if it
2727  * cannot supply any data in a buffer.  In other words, write will never
2728  * send down an empty packet due to a write offset.
2729  */
2730 /* ARGSUSED2 */
2731 int
2732 strwrite(struct vnode *vp, struct uio *uiop, cred_t *crp)
2733 {
2734         return (strwrite_common(vp, uiop, crp, 0));
2735 }

2737 /* ARGSUSED2 */
2738 int
2739 strwrite_common(struct vnode *vp, struct uio *uiop, cred_t *crp, int wflag)
2740 {
2741         struct stdata *stp;
2742         struct queue *wqp;
2743         ssize_t rmin, rmax;
2744         ssize_t iosize;
2745         int waitflag;
2746         int tempmode;
2747         int error = 0;
2748         int b_flag;

2750         ASSERT(vp->v_stream);
2751         stp = vp->v_stream;

2753         mutex_enter(&stp->sd_lock);

2755         if ((error = i_straccess(stp, JCWRITE)) != 0) {
2756                 mutex_exit(&stp->sd_lock);
2757                 return (error);
2758         }

2760         if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
2761                 error = strwriteable(stp, B_TRUE, B_TRUE);
2762                 if (error != 0) {
2763                         mutex_exit(&stp->sd_lock);
2764                         return (error);
2765                 }
2766         }
```

```
2768         mutex_exit(&stp->sd_lock);

2770         wqp = stp->sd_wrq;

2772         /* get these values from them cached in the stream head */
2773         rmin = stp->sd_qn_minpsz;
2774         rmax = stp->sd_qn_maxpsz;

2776         /*
2777          * Check the min/max packet size constraints.  If min packet size
2778          * is non-zero, the write cannot be split into multiple messages
2779          * and still guarantee the size constraints.
2780          */
2781         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_IN, "strwrite in:q %p", wqp);

2783         ASSERT((rmax >= 0) || (rmax == INFPSZ));
2784         if (rmax == 0) {
2785                 return (0);
2786         }
2787         if (rmin > 0) {
2788                 if (uiop->uio_resid < rmin) {
2789                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2790                             "strwrite out:q %p out %d error %d",
2791                             wqp, 0, ERANGE);
2792                         return (ERANGE);
2793                 }
2794                 if ((rmax != INFPSZ) && (uiop->uio_resid > rmax)) {
2795                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2796                             "strwrite out:q %p out %d error %d",
2797                             wqp, 1, ERANGE);
2798                         return (ERANGE);
2799                 }
2800         }

2802         /*
2803          * Do until count satisfied or error.
2804          */
2805         waitflag = WRITEWAIT | wflag;
2806         if (stp->sd_flag & OLDNDELAY)
2807                 tempmode = uiop->uio_fmode & ~FNDELAY;
2808         else
2809                 tempmode = uiop->uio_fmode;

2811         if (rmax == INFPSZ)
2812                 rmax = uiop->uio_resid;

2814         /*
2815          * Note that tempmode does not get used in strput/strmakedata
2816          * but only in strwaitq. The other routines use uio_fmode
2817          * unmodified.
2818          */

2820         /* LINTED: constant in conditional context */
2821         while (1) {     /* breaks when uio_resid reaches zero */
2822                 /*
2823                  * Determine the size of the next message to be
2824                  * packaged.  May have to break write into several
2825                  * messages based on max packet size.
2826                  */
2827                 iosize = MIN(uiop->uio_resid, rmax);

2829                 /*
2830                  * Put block downstream when flow control allows it.
2831                  */
2832                 if ((stp->sd_flag & STRDELIM) && (uiop->uio_resid == iosize))
2833                         b_flag = MSGDELIM;
```



```
2834                 else
2835                         b_flag = 0;

2837                 for (;;) {
2838                         int done = 0;

2840                         error = strput(stp, NULL, uiop, &iosize, b_flag, 0, 0);
2841                         if (error == 0)
2842                                 break;
2843                         if (error != EWOULDBLOCK)
2844                                 goto out;

2846                         mutex_enter(&stp->sd_lock);
2847                         /*
2848                          * Check for a missed wakeup.
2849                          * Needed since strput did not hold sd_lock across
2850                          * the canputnext.
2851                          */
2852                         if (canputnext(wqp)) {
2853                                 /* Try again */
2854                                 mutex_exit(&stp->sd_lock);
2855                                 continue;
2856                         }
2857                         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_WAIT,
2858                             "strwrite wait:q %p wait", wqp);
2859                         if ((error = strwaitq(stp, waitflag, (ssize_t)0,
2860                             tempmode, -1, &done)) != 0 || done) {
2861                                 mutex_exit(&stp->sd_lock);
2862                                 if ((vp->v_type == VFIFO) &&
2863                                     (uiop->uio_fmode & FNDELAY) &&
2864                                     (error == EAGAIN))
2865                                         error = 0;
2866                                 goto out;
2867                         }
2868                         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_WAKE,
2869                             "strwrite wake:q %p awakes", wqp);
2870                         if ((error = i_straccess(stp, JCWRITE)) != 0) {
2871                                 mutex_exit(&stp->sd_lock);
2872                                 goto out;
2873                         }
2874                         mutex_exit(&stp->sd_lock);
2875                 }
2876                 waitflag |= NOINTR;
2877                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRWRITE_RESID,
2878                     "strwrite resid:q %p uiop %p", wqp, uiop);
2879                 if (uiop->uio_resid) {
2880                         /* Recheck for errors - needed for sockets */
2881                         if ((stp->sd_wput_opt & SW_RECHECK_ERR) &&
2882                             (stp->sd_flag & (STWRERR|STRHUP|STPLEX))) {
2883                                 mutex_enter(&stp->sd_lock);
2884                                 error = strwriteable(stp, B_FALSE, B_TRUE);
2885                                 mutex_exit(&stp->sd_lock);
2886                                 if (error != 0)
2887                                         return (error);
2888                         }
2889                         continue;
2890                 }
2891                 break;
2892         }
2893 out:
2894         /*
2895          * For historical reasons, applications expect EAGAIN when a data
2896          * mblk_t cannot be allocated, so change ENOMEM back to EAGAIN.
2897          */
2898         if (error == ENOMEM)
2899                 error = EAGAIN;
```

```
2900         TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2901             "strwrite out:q %p out %d error %d", wqp, 2, error);
2902         return (error);
2903 }

2905 /*
2906  * Stream head write service routine.
2907  * Its job is to wake up any sleeping writers when a queue
2908  * downstream needs data (part of the flow control in putq and getq).
2909  * It also must wake anyone sleeping on a poll().
2910  * For stream head right below mux module, it must also invoke put procedure
2911  * of next downstream module.
2912  */
2913 int
2914 strwsrv(queue_t *q)
2915 {
2916         struct stdata *stp;
2917         queue_t *tq;
2918         qband_t *qbp;
2919         int i;
2920         qband_t *myqbp;
2921         int isevent;
2922         unsigned char   qbf[NBAND];     /* band flushing backenable flags */

2924         TRACE_1(TR_FAC_STREAMS_FR,
2925             TR_STRWSRV, "strwsrv:q %p", q);
2926         stp = (struct stdata *)q->q_ptr;
2927         ASSERT(qclaimed(q));
2928         mutex_enter(&stp->sd_lock);
2929         ASSERT(!(stp->sd_flag & STPLEX));

2931         if (stp->sd_flag & WSLEEP) {
2932                 stp->sd_flag &= ~WSLEEP;
2933                 cv_broadcast(&q->q_wait);
2934         }
2935         mutex_exit(&stp->sd_lock);

2937         /* The other end of a stream pipe went away. */
2938         if ((tq = q->q_next) == NULL) {
2939                 return (0);
2940         }

2942         /* Find the next module forward that has a service procedure */
2943         claimstr(q);
2944         tq = q->q_nfsrv;
2945         ASSERT(tq != NULL);

2947         if ((q->q_flag & QBACK)) {
2948                 if ((tq->q_flag & QFULL)) {
2949                         mutex_enter(QLOCK(tq));
2950                         if (!(tq->q_flag & QFULL)) {
2951                                 mutex_exit(QLOCK(tq));
2952                                 goto wakeup;
2953                         }
2954                         /*
2955                          * The queue must have become full again. Set QWANTW
2956                          * again so strwsrv will be back enabled when
2957                          * the queue becomes non-full next time.
2958                          */
2959                         tq->q_flag |= QWANTW;
2960                         mutex_exit(QLOCK(tq));
2961                 } else {
2962                 wakeup:
2963                         pollwakeup(&stp->sd_pollist, POLLWRNORM);
2964                         mutex_enter(&stp->sd_lock);
2965                         if (stp->sd_sigflags & S_WRNORM)
```



```
2966                                 strsendsig(stp->sd_siglist, S_WRNORM, 0, 0);
2967                         mutex_exit(&stp->sd_lock);
2968                 }
2969         }

2971         isevent = 0;
2972         i = 1;
2973         bzero((caddr_t)qbf, NBAND);
2974         mutex_enter(QLOCK(tq));
2975         if ((myqbp = q->q_bandp) != NULL)
2976                 for (qbp = tq->q_bandp; qbp && myqbp; qbp = qbp->qb_next) {
2977                         ASSERT(myqbp);
2978                         if ((myqbp->qb_flag & QB_BACK)) {
2979                                 if (qbp->qb_flag & QB_FULL) {
2980                                         /*
2981                                          * The band must have become full again.
2982                                          * Set QB_WANTW again so strwsrv will
2983                                          * be back enabled when the band becomes
2984                                          * non-full next time.
2985                                          */
2986                                         qbp->qb_flag |= QB_WANTW;
2987                                 } else {
2988                                         isevent = 1;
2989                                         qbf[i] = 1;
2990                                 }
2991                         }
2992                         myqbp = myqbp->qb_next;
2993                         i++;
2994                 }
2995         mutex_exit(QLOCK(tq));

2997         if (isevent) {
2998                 for (i = tq->q_nband; i; i--) {
2999                         if (qbf[i]) {
3000                                 pollwakeup(&stp->sd_pollist, POLLWRBAND);
3001                                 mutex_enter(&stp->sd_lock);
3002                                 if (stp->sd_sigflags & S_WRBAND)
3003                                         strsendsig(stp->sd_siglist, S_WRBAND,
3004                                             (uchar_t)i, 0);
3005                                 mutex_exit(&stp->sd_lock);
3006                         }
3007                 }
3008         }

3010         releasestr(q);
3011         return (0);
3012 }

3014 /*
3015  * Special case of strcopyin/strcopyout for copying
3016  * struct strioctl that can deal with both data
3017  * models.
3018  */

3020 #ifdef  _LP64

3022 static int
3023 strcopyin_strioctl(void *from, void *to, int flag, int copyflag)
3024 {
3025         struct  strioctl32 strioc32;
3026         struct  strioctl *striocp;

3028         if (copyflag & U_TO_K) {
3029                 ASSERT((copyflag & K_TO_K) == 0);

3031                 if ((flag & FMODELS) == DATAMODEL_ILP32) {
```

```
3032                        if (copyin(from, &strioc32, sizeof (strioc32)))
3033                                return (EFAULT);

3035                        striocp = (struct strioctl *)to;
3036                        striocp->ic_cmd = strioc32.ic_cmd;
3037                        striocp->ic_timout = strioc32.ic_timout;
3038                        striocp->ic_len = strioc32.ic_len;
3039                        striocp->ic_dp  = (char *)(uintptr_t)strioc32.ic_dp;

3041                } else { /* NATIVE data model */
3042                        if (copyin(from, to, sizeof (struct strioctl))) {
3043                                return (EFAULT);
3044                        } else {
3045                                return (0);
3046                        }
3047                }
3048        } else {
3049                ASSERT(copyflag & K_TO_K);
3050                bcopy(from, to, sizeof (struct strioctl));
3051        }
3052        return (0);
3053 }

3055 static int
3056 strcopyout_strioctl(void *from, void *to, int flag, int copyflag)
3057 {
3058        struct  strioctl32 strioc32;
3059        struct  strioctl *striocp;

3061        if (copyflag & U_TO_K) {
3062                ASSERT((copyflag & K_TO_K) == 0);

3064                if ((flag & FMODELS) == DATAMODEL_ILP32) {
3065                        striocp = (struct strioctl *)from;
3066                        strioc32.ic_cmd = striocp->ic_cmd;
3067                        strioc32.ic_timout = striocp->ic_timout;
3068                        strioc32.ic_len = striocp->ic_len;
3069                        strioc32.ic_dp  = (caddr32_t)(uintptr_t)striocp->ic_dp;
3070                        ASSERT((char *)(uintptr_t)strioc32.ic_dp ==
3071                            striocp->ic_dp);

3073                        if (copyout(&strioc32, to, sizeof (strioc32)))
3074                                return (EFAULT);

3076                } else { /* NATIVE data model */
3077                        if (copyout(from, to, sizeof (struct strioctl))) {
3078                                return (EFAULT);
3079                        } else {
3080                                return (0);
3081                        }
3082                }
3083        } else {
3084                ASSERT(copyflag & K_TO_K);
3085                bcopy(from, to, sizeof (struct strioctl));
3086        }
3087        return (0);
3088 }

3090 #else   /* ! _LP64 */

3092 /* ARGSUSED2 */
3093 static int
3094 strcopyin_strioctl(void *from, void *to, int flag, int copyflag)
3095 {
3096        return (strcopyin(from, to, sizeof (struct strioctl), copyflag));
3097 }
```



```
3099 /* ARGSUSED2 */
3100 static int
3101 strcopyout_strioctl(void *from, void *to, int flag, int copyflag)
3102 {
3103        return (strcopyout(from, to, sizeof (struct strioctl), copyflag));
3104 }

3106 #endif  /* _LP64 */

3108 /*
3109  * Determine type of job control semantics expected by user.  The
3110  * possibilities are:
3111  *      JCREAD  - Behaves like read() on fd; send SIGTTIN
3112  *      JCWRITE - Behaves like write() on fd; send SIGTTOU if TOSTOP set
3113  *      JCSETP  - Sets a value in the stream; send SIGTTOU, ignore TOSTOP
3114  *      JCGETP  - Gets a value in the stream; no signals.
3115  * See straccess in strsubr.c for usage of these values.
3116  *
3117  * This routine also returns -1 for I_STR as a special case; the
3118  * caller must call again with the real ioctl number for
3119  * classification.
3120  */
3121 static int
3122 job_control_type(int cmd)
3123 {
3124        switch (cmd) {
3125        case I_STR:
3126                return (-1);

3128        case I_RECVFD:
3129        case I_E_RECVFD:
3130                return (JCREAD);

3132        case I_FDINSERT:
3133        case I_SENDFD:
3134                return (JCWRITE);

3136        case TCSETA:
3137        case TCSETAW:
3138        case TCSETAF:
3139        case TCSBRK:
3140        case TCXONC:
3141        case TCFLSH:
3142        case TCDSET:    /* Obsolete */
3143        case TIOCSWINSZ:
3144        case TCSETS:
3145        case TCSETSW:
3146        case TCSETSF:
3147        case TIOCSETD:
3148        case TIOCHPCL:
3149        case TIOCSETP:
3150        case TIOCSETN:
3151        case TIOCEXCL:
3152        case TIOCNXCL:
3153        case TIOCFLUSH:
3154        case TIOCSETC:
3155        case TIOCLBIS:
3156        case TIOCLBIC:
3157        case TIOCLSET:
3158        case TIOCSBRK:
3159        case TIOCCBRK:
3160        case TIOCSDTR:
3161        case TIOCCDTR:
3162        case TIOCSLTC:
3163        case TIOCSTOP:
```

```
3164         case TIOCSTART:
3165         case TIOCSTI:
3166         case TIOCSPGRP:
3167         case TIOCMSET:
3168         case TIOCMBIS:
3169         case TIOCMBIC:
3170         case TIOCREMOTE:
3171         case TIOCSIGNAL:
3172         case LDSETT:
3173         case LDSMAP:    /* Obsolete */
3174         case DIOCSETP:
3175         case I_FLUSH:
3176         case I_SRDOPT:
3177         case I_SETSIG:
3178         case I_SWROPT:
3179         case I_FLUSHBAND:
3180         case I_SETCLTIME:
3181         case I_SERROPT:
3182         case I_ESETSIG:
3183         case FIONBIO:
3184         case FIOASYNC:
3185         case FIOSETOWN:
3186         case JBOOT:     /* Obsolete */
3187         case JTERM:     /* Obsolete */
3188         case JTIMOM:    /* Obsolete */
3189         case JZOMBOOT:  /* Obsolete */
3190         case JAGENT:    /* Obsolete */
3191         case JTRUN:     /* Obsolete */
3192         case JXTPROTO:  /* Obsolete */
3193                 return (JCSETP);
3194         }

3196         return (JCGETP);
3197 }

3199 /*
3200  * ioctl for streams
3201  */
3202 int
3203 strioctl(struct vnode *vp, int cmd, intptr_t arg, int flag, int copyflag,
3204     cred_t *crp, int *rvalp)
3205 {
3206         struct stdata *stp;
3207         struct strcmd *scp;
3208         struct strioctl strioc;
3209         struct uio uio;
3210         struct iovec iov;
3211         int access;
3212         mblk_t *mp;
3213         int error = 0;
3214         int done = 0;
3215         ssize_t rmin, rmax;
3216         queue_t *wrq;
3217         queue_t *rdq;
3218         boolean_t kioctl = B_FALSE;
3219         uint32_t auditing = AU_AUDITING();

3221         if (flag & FKIOCTL) {
3222                 copyflag = K_TO_K;
3223                 kioctl = B_TRUE;
3224         }
3225         ASSERT(vp->v_stream);
3226         ASSERT(copyflag == U_TO_K || copyflag == K_TO_K);
3227         stp = vp->v_stream;

3229         TRACE_3(TR_FAC_STREAMS_FR, TR_IOCTL_ENTER,
```



```
3230             "strioctl:stp %p cmd %X arg %lX", stp, cmd, arg);

3232         /*
3233          * If the copy is kernel to kernel, make sure that the FNATIVE
3234          * flag is set.  After this it would be a serious error to have
3235          * no model flag.
3236          */
3237         if (copyflag == K_TO_K)
3238                 flag = (flag & ~FMODELS) | FNATIVE;

3240         ASSERT((flag & FMODELS) != 0);

3242         wrq = stp->sd_wrq;
3243         rdq = _RD(wrq);

3245         access = job_control_type(cmd);

3247         /* We should never see these here, should be handled by iwscn */
3248         if (cmd == SRIOCSREDIR || cmd == SRIOCISREDIR)
3249                 return (EINVAL);

3251         mutex_enter(&stp->sd_lock);
3252         if ((access != -1) && ((error = i_straccess(stp, access)) != 0)) {
3253                 mutex_exit(&stp->sd_lock);
3254                 return (error);
3255         }
3256         mutex_exit(&stp->sd_lock);

3258         /*
3259          * Check for sgttyb-related ioctls first, and complain as
3260          * necessary.
3261          */
3262         switch (cmd) {
3263         case TIOCGETP:
3264         case TIOCSETP:
3265         case TIOCSETN:
3266                 if (sgttyb_handling >= 2 && !sgttyb_complaint) {
3267                         sgttyb_complaint = B_TRUE;
3268                         cmn_err(CE_NOTE,
3269                             "application used obsolete TIOC[GS]ET");
3270                 }
3271                 if (sgttyb_handling >= 3) {
3272                         tsignal(curthread, SIGSYS);
3273                         return (EIO);
3274                 }
3275                 break;
3276         }

3278         mutex_enter(&stp->sd_lock);

3280         switch (cmd) {
3281         case I_RECVFD:
3282         case I_E_RECVFD:
3283         case I_PEEK:
3284         case I_NREAD:
3285         case FIONREAD:
3286         case FIORDCHK:
3287         case I_ATMARK:
3288         case FIONBIO:
3289         case FIOASYNC:
3290                 if (stp->sd_flag & (STRDERR|STPLEX)) {
3291                         error = strgeterr(stp, STRDERR|STPLEX, 0);
3292                         if (error != 0) {
3293                                 mutex_exit(&stp->sd_lock);
3294                                 return (error);
3295                         }
```

```
3296                }
3297                break;

3299        default:
3300                if (stp->sd_flag & (STRDERR|STWRERR|STPLEX)) {
3301                        error = strgeterr(stp, STRDERR|STWRERR|STPLEX, 0);
3302                        if (error != 0) {
3303                                mutex_exit(&stp->sd_lock);
3304                                return (error);
3305                        }
3306                }
3307        }

3309        mutex_exit(&stp->sd_lock);

3311        switch (cmd) {
3312        default:
3313                /*
3314                 * The stream head has hardcoded knowledge of a
3315                 * miscellaneous collection of terminal-, keyboard- and
3316                 * mouse-related ioctls, enumerated below.  This hardcoded
3317                 * knowledge allows the stream head to automatically
3318                 * convert transparent ioctl requests made by userland
3319                 * programs into I_STR ioctls which many old STREAMS
3320                 * modules and drivers require.
3321                 *
3322                 * No new ioctls should ever be added to this list.
3323                 * Instead, the STREAMS module or driver should be written
3324                 * to either handle transparent ioctls or require any
3325                 * userland programs to use I_STR ioctls (by returning
3326                 * EINVAL to any transparent ioctl requests).
3327                 *
3328                 * More importantly, removing ioctls from this list should
3329                 * be done with the utmost care, since our STREAMS modules
3330                 * and drivers *count* on the stream head performing this
3331                 * conversion, and thus may panic while processing
3332                 * transparent ioctl request for one of these ioctls (keep
3333                 * in mind that third party modules and drivers may have
3334                 * similar problems).
3335                 */
3336                if (((cmd & IOCTYPE) == LDIOC) ||
3337                    ((cmd & IOCTYPE) == tIOC) ||
3338                    ((cmd & IOCTYPE) == TIOC) ||
3339                    ((cmd & IOCTYPE) == KIOC) ||
3340                    ((cmd & IOCTYPE) == MSIOC) ||
3341                    ((cmd & IOCTYPE) == VUIOC)) {
3342                        /*
3343                         * The ioctl is a tty ioctl - set up strioc buffer
3344                         * and call strdoioctl() to do the work.
3345                         */
3346                        if (stp->sd_flag & STRHUP)
3347                                return (ENXIO);
3348                        strioc.ic_cmd = cmd;
3349                        strioc.ic_timout = INFTIM;

3351                        switch (cmd) {

3353                        case TCXONC:
3354                        case TCSBRK:
3355                        case TCFLSH:
3356                        case TCDSET:
3357                                {
3358                                int native_arg = (int)arg;
3359                                strioc.ic_len = sizeof (int);
3360                                strioc.ic_dp = (char *)&native_arg;
3361                                return (strdoioctl(stp, &strioc, flag,
```



```
3362                                    K_TO_K, crp, rvalp));
3363                        }

3365                    case TCSETA:
3366                    case TCSETAW:
3367                    case TCSETAF:
3368                            strioc.ic_len = sizeof (struct termio);
3369                            strioc.ic_dp = (char *)arg;
3370                            return (strdoioctl(stp, &strioc, flag,
3371                                copyflag, crp, rvalp));

3373                    case TCSETS:
3374                    case TCSETSW:
3375                    case TCSETSF:
3376                            strioc.ic_len = sizeof (struct termios);
3377                            strioc.ic_dp = (char *)arg;
3378                            return (strdoioctl(stp, &strioc, flag,
3379                                copyflag, crp, rvalp));

3381                    case LDSETT:
3382                            strioc.ic_len = sizeof (struct termcb);
3383                            strioc.ic_dp = (char *)arg;
3384                            return (strdoioctl(stp, &strioc, flag,
3385                                copyflag, crp, rvalp));

3387                    case TIOCSETP:
3388                            strioc.ic_len = sizeof (struct sgttyb);
3389                            strioc.ic_dp = (char *)arg;
3390                            return (strdoioctl(stp, &strioc, flag,
3391                                copyflag, crp, rvalp));

3393                    case TIOCSTI:
3394                            if ((flag & FREAD) == 0 &&
3395                                secpolicy_sti(crp) != 0) {
3396                                    return (EPERM);
3397                            }
3398                            mutex_enter(&stp->sd_lock);
3399                            mutex_enter(&curproc->p_splock);
3400                            if (stp->sd_sidp != curproc->p_sessp->s_sidp &&
3401                                secpolicy_sti(crp) != 0) {
3402                                    mutex_exit(&curproc->p_splock);
3403                                    mutex_exit(&stp->sd_lock);
3404                                    return (EACCES);
3405                            }
3406                            mutex_exit(&curproc->p_splock);
3407                            mutex_exit(&stp->sd_lock);

3409                            strioc.ic_len = sizeof (char);
3410                            strioc.ic_dp = (char *)arg;
3411                            return (strdoioctl(stp, &strioc, flag,
3412                                copyflag, crp, rvalp));

3414                    case TIOCSWINSZ:
3415                            strioc.ic_len = sizeof (struct winsize);
3416                            strioc.ic_dp = (char *)arg;
3417                            return (strdoioctl(stp, &strioc, flag,
3418                                copyflag, crp, rvalp));

3420                    case TIOCSSIZE:
3421                            strioc.ic_len = sizeof (struct ttysize);
3422                            strioc.ic_dp = (char *)arg;
3423                            return (strdoioctl(stp, &strioc, flag,
3424                                copyflag, crp, rvalp));

3426                    case TIOCSSOFTCAR:
3427                    case KIOCTRANS:
```

```
3428                        case KIOCTRANSABLE:
3429                        case KIOCCMD:
3430                        case KIOCSDIRECT:
3431                        case KIOCSCOMPAT:
3432                        case KIOCSKABORTEN:
3433                        case KIOCSRPTDELAY:
3434                        case KIOCSRPTRATE:
3435                        case VUIDSFORMAT:
3436                        case TIOCSPPS:
3437                                strioc.ic_len = sizeof (int);
3438                                strioc.ic_dp = (char *)arg;
3439                                return (strdoioctl(stp, &strioc, flag,
3440                                    copyflag, crp, rvalp));

3442                        case KIOCSETKEY:
3443                        case KIOCGETKEY:
3444                                strioc.ic_len = sizeof (struct kiockey);
3445                                strioc.ic_dp = (char *)arg;
3446                                return (strdoioctl(stp, &strioc, flag,
3447                                    copyflag, crp, rvalp));

3449                        case KIOCSKEY:
3450                        case KIOCGKEY:
3451                                strioc.ic_len = sizeof (struct kiockeymap);
3452                                strioc.ic_dp = (char *)arg;
3453                                return (strdoioctl(stp, &strioc, flag,
3454                                    copyflag, crp, rvalp));

3456                        case KIOCSLED:
3457                                /* arg is a pointer to char */
3458                                strioc.ic_len = sizeof (char);
3459                                strioc.ic_dp = (char *)arg;
3460                                return (strdoioctl(stp, &strioc, flag,
3461                                    copyflag, crp, rvalp));

3463                        case MSIOSETPARMS:
3464                                strioc.ic_len = sizeof (Ms_parms);
3465                                strioc.ic_dp = (char *)arg;
3466                                return (strdoioctl(stp, &strioc, flag,
3467                                    copyflag, crp, rvalp));

3469                        case VUIDSADDR:
3470                        case VUIDGADDR:
3471                                strioc.ic_len = sizeof (struct vuid_addr_probe);
3472                                strioc.ic_dp = (char *)arg;
3473                                return (strdoioctl(stp, &strioc, flag,
3474                                    copyflag, crp, rvalp));

3476                        /*
3477                         * These M_IOCTL's don't require any data to be sent
3478                         * downstream, and the driver will allocate and link
3479                         * on its own mblk_t upon M_IOCACK -- thus we set
3480                         * ic_len to zero and set ic_dp to arg so we know
3481                         * where to copyout to later.
3482                         */
3483                        case TIOCGSOFTCAR:
3484                        case TIOCGWINSZ:
3485                        case TIOCGSIZE:
3486                        case KIOCGTRANS:
3487                        case KIOCGTRANSABLE:
3488                        case KIOCTYPE:
3489                        case KIOCGDIRECT:
3490                        case KIOCGCOMPAT:
3491                        case KIOCLAYOUT:
3492                        case KIOCGLED:
3493                        case MSIOGETPARMS:
```



```
3494                        case MSIOBUTTONS:
3495                        case VUIDGFORMAT:
3496                        case TIOCGPPS:
3497                        case TIOCGPPSEV:
3498                        case TCGETA:
3499                        case TCGETS:
3500                        case LDGETT:
3501                        case TIOCGETP:
3502                        case KIOCGRPTDELAY:
3503                        case KIOCGRPTRATE:
3504                                strioc.ic_len = 0;
3505                                strioc.ic_dp = (char *)arg;
3506                                return (strdoioctl(stp, &strioc, flag,
3507                                    copyflag, crp, rvalp));
3508                        }
3509                }

3511                /*
3512                 * Unknown cmd - send it down as a transparent ioctl.
3513                 */
3514                strioc.ic_cmd = cmd;
3515                strioc.ic_timout = INFTIM;
3516                strioc.ic_len = TRANSPARENT;
3517                strioc.ic_dp = (char *)&arg;

3519                return (strdoioctl(stp, &strioc, flag, copyflag, crp, rvalp));

3521        case I_STR:
3522                /*
3523                 * Stream ioctl.  Read in an strioctl buffer from the user
3524                 * along with any data specified and send it downstream.
3525                 * Strdoioctl will wait allow only one ioctl message at
3526                 * a time, and waits for the acknowledgement.
3527                 */

3529                if (stp->sd_flag & STRHUP)
3530                        return (ENXIO);

3532                error = strcopyin_strioctl((void *)arg, &strioc, flag,
3533                    copyflag);
3534                if (error != 0)
3535                        return (error);

3537                if ((strioc.ic_len < 0) || (strioc.ic_timout < -1))
3538                        return (EINVAL);

3540                access = job_control_type(strioc.ic_cmd);
3541                mutex_enter(&stp->sd_lock);
3542                if ((access != -1) &&
3543                    ((error = i_straccess(stp, access)) != 0)) {
3544                        mutex_exit(&stp->sd_lock);
3545                        return (error);
3546                }
3547                mutex_exit(&stp->sd_lock);

3549                /*
3550                 * The I_STR facility provides a trap door for malicious
3551                 * code to send down bogus streamio(7I) ioctl commands to
3552                 * unsuspecting STREAMS modules and drivers which expect to
3553                 * only get these messages from the stream head.
3554                 * Explicitly prohibit any streamio ioctls which can be
3555                 * passed downstream by the stream head.  Note that we do
3556                 * not block all streamio ioctls because the ioctl
3557                 * numberspace is not well managed and thus it's possible
3558                 * that a module or driver's ioctl numbers may accidentally
3559                 * collide with them.
```

```
3560                 */
3561                switch (strioc.ic_cmd) {
3562                case I_LINK:
3563                case I_PLINK:
3564                case I_UNLINK:
3565                case I_PUNLINK:
3566                case _I_GETPEERCRED:
3567                case _I_PLINK_LH:
3568                        return (EINVAL);
3569                }

3571                error = strdoioctl(stp, &strioc, flag, copyflag, crp, rvalp);
3572                if (error == 0) {
3573                        error = strcopyout_strioctl(&strioc, (void *)arg,
3574                            flag, copyflag);
3575                }
3576                return (error);

3578        case _I_CMD:
3579                /*
3580                 * Like I_STR, but without using M_IOC* messages and without
3581                 * copyins/copyouts beyond the passed-in argument.
3582                 */
3583                if (stp->sd_flag & STRHUP)
3584                        return (ENXIO);

3586                if ((scp = kmem_alloc(sizeof (strcmd_t), KM_NOSLEEP)) == NULL)
3587                        return (ENOMEM);

3589                if (copyin((void *)arg, scp, sizeof (strcmd_t))) {
3590                        kmem_free(scp, sizeof (strcmd_t));
3591                        return (EFAULT);
3592                }

3594                access = job_control_type(scp->sc_cmd);
3595                mutex_enter(&stp->sd_lock);
3596                if (access != -1 && (error = i_straccess(stp, access)) != 0) {
3597                        mutex_exit(&stp->sd_lock);
3598                        kmem_free(scp, sizeof (strcmd_t));
3599                        return (error);
3600                }
3601                mutex_exit(&stp->sd_lock);

3603                *rvalp = 0;
3604                if ((error = strdocmd(stp, scp, crp)) == 0) {
3605                        if (copyout(scp, (void *)arg, sizeof (strcmd_t)))
3606                                error = EFAULT;
3607                }
3608                kmem_free(scp, sizeof (strcmd_t));
3609                return (error);

3611        case I_NREAD:
3612                /*
3613                 * Return number of bytes of data in first message
3614                 * in queue in "arg" and return the number of messages
3615                 * in queue in return value.
3616                 */
3617        {
3618                size_t  size;
3619                int     retval;
3620                int     count = 0;

3622                mutex_enter(QLOCK(rdq));

3624                size = msgdsize(rdq->q_first);
3625                for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
```



```
3626                        count++;

3628                mutex_exit(QLOCK(rdq));
3629                if (stp->sd_struiordq) {
3630                        infod_t infod;

3632                        infod.d_cmd = INFOD_COUNT;
3633                        infod.d_count = 0;
3634                        if (count == 0) {
3635                                infod.d_cmd |= INFOD_FIRSTBYTES;
3636                                infod.d_bytes = 0;
3637                        }
3638                        infod.d_res = 0;
3639                        (void) infonext(rdq, &infod);
3640                        count += infod.d_count;
3641                        if (infod.d_res & INFOD_FIRSTBYTES)
3642                                size = infod.d_bytes;
3643                }

3645                /*
3646                 * Drop down from size_t to the "int" required by the
3647                 * interface.  Cap at INT_MAX.
3648                 */
3649                retval = MIN(size, INT_MAX);
3650                error = strcopyout(&retval, (void *)arg, sizeof (retval),
3651                    copyflag);
3652                if (!error)
3653                        *rvalp = count;
3654                return (error);
3655        }

3657        case FIONREAD:
3658                /*
3659                 * Return number of bytes of data in all data messages
3660                 * in queue in "arg".
3661                 */
3662        {
3663                size_t  size = 0;
3664                int     retval;

3666                mutex_enter(QLOCK(rdq));
3667                for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3668                        size += msgdsize(mp);
3669                mutex_exit(QLOCK(rdq));

3671                if (stp->sd_struiordq) {
3672                        infod_t infod;

3674                        infod.d_cmd = INFOD_BYTES;
3675                        infod.d_res = 0;
3676                        infod.d_bytes = 0;
3677                        (void) infonext(rdq, &infod);
3678                        size += infod.d_bytes;
3679                }

3681                /*
3682                 * Drop down from size_t to the "int" required by the
3683                 * interface.  Cap at INT_MAX.
3684                 */
3685                retval = MIN(size, INT_MAX);
3686                error = strcopyout(&retval, (void *)arg, sizeof (retval),
3687                    copyflag);

3689                *rvalp = 0;
3690                return (error);
3691        }
```

```
3692         case FIORDCHK:
3693                 /*
3694                  * FIORDCHK does not use arg value (like FIONREAD),
3695                  * instead a count is returned. I_NREAD value may
3696                  * not be accurate but safe. The real thing to do is
3697                  * to add the msgdsizes of all data  messages until
3698                  * a non-data message.
3699                  */
3700         {
3701                 size_t size = 0;

3703                 mutex_enter(QLOCK(rdq));
3704                 for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3705                         size += msgdsize(mp);
3706                 mutex_exit(QLOCK(rdq));

3708                 if (stp->sd_struiordq) {
3709                         infod_t infod;

3711                         infod.d_cmd = INFOD_BYTES;
3712                         infod.d_res = 0;
3713                         infod.d_bytes = 0;
3714                         (void) infonext(rdq, &infod);
3715                         size += infod.d_bytes;
3716                 }

3718                 /*
3719                  * Since ioctl returns an int, and memory sizes under
3720                  * LP64 may not fit, we return INT_MAX if the count was
3721                  * actually greater.
3722                  */
3723                 *rvalp = MIN(size, INT_MAX);
3724                 return (0);
3725         }

3727         case I_FIND:
3728                 /*
3729                  * Get module name.
3730                  */
3731         {
3732                 char mname[FMNAMESZ + 1];
3733                 queue_t *q;

3735                 error = (copyflag & U_TO_K ? copyinstr : copystr)((void *)arg,
3736                     mname, FMNAMESZ + 1, NULL);
3737                 if (error)
3738                         return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

3740                 /*
3741                  * Return EINVAL if we're handed a bogus module name.
3742                  */
3743                 if (fmodsw_find(mname, FMODSW_LOAD) == NULL) {
3744                         TRACE_0(TR_FAC_STREAMS_FR,
3745                             TR_I_CANT_FIND, "couldn't I_FIND");
3746                         return (EINVAL);
3747                 }

3749                 *rvalp = 0;

3751                 /* Look downstream to see if module is there. */
3752                 claimstr(stp->sd_wrq);
3753                 for (q = stp->sd_wrq->q_next; q; q = q->q_next) {
3754                         if (q->q_flag & QREADR) {
3755                                 q = NULL;
3756                                 break;
3757                         }
```



```
3758                         if (strcmp(mname, Q2NAME(q)) == 0)
3759                                 break;
3760                 }
3761                 releasestr(stp->sd_wrq);

3763                 *rvalp = (q ? 1 : 0);
3764                 return (error);
3765         }

3767         case I_PUSH:
3768         case __I_PUSH_NOCTTY:
3769                 /*
3770                  * Push a module.
3771                  * For the case __I_PUSH_NOCTTY push a module but
3772                  * do not allocate controlling tty. See bugid 4025044
3773                  */

3775         {
3776                 char mname[FMNAMESZ + 1];
3777                 fmodsw_impl_t *fp;
3778                 dev_t dummydev;

3780                 if (stp->sd_flag & STRHUP)
3781                         return (ENXIO);

3783                 /*
3784                  * Get module name and look up in fmodsw.
3785                  */
3786                 error = (copyflag & U_TO_K ? copyinstr : copystr)((void *)arg,
3787                     mname, FMNAMESZ + 1, NULL);
3788                 if (error)
3789                         return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

3791                 if ((fp = fmodsw_find(mname, FMODSW_HOLD | FMODSW_LOAD)) ==
3792                     NULL)
3793                         return (EINVAL);

3795                 TRACE_2(TR_FAC_STREAMS_FR, TR_I_PUSH,
3796                     "I_PUSH:fp %p stp %p", fp, stp);

3798                 if (error = strstartplumb(stp, flag, cmd)) {
3799                         fmodsw_rele(fp);
3800                         return (error);
3801                 }

3803                 /*
3804                  * See if any more modules can be pushed on this stream.
3805                  * Note that this check must be done after strstartplumb()
3806                  * since otherwise multiple threads issuing I_PUSHes on
3807                  * the same stream will be able to exceed nstrpush.
3808                  */
3809                 mutex_enter(&stp->sd_lock);
3810                 if (stp->sd_pushcnt >= nstrpush) {
3811                         fmodsw_rele(fp);
3812                         strendplumb(stp);
3813                         mutex_exit(&stp->sd_lock);
3814                         return (EINVAL);
3815                 }
3816                 mutex_exit(&stp->sd_lock);

3818                 /*
3819                  * Push new module and call its open routine
3820                  * via qattach().  Modules don't change device
3821                  * numbers, so just ignore dummydev here.
3822                  */
3823                 dummydev = vp->v_rdev;
```

```
3824                if ((error = qattach(rdq, &dummydev, 0, crp, fp,
3825                    B_FALSE)) == 0) {
3826                        if (vp->v_type == VCHR && /* sorry, no pipes allowed */
3827                            (cmd == I_PUSH) && (stp->sd_flag & STRISTTY)) {
3828                                /*
3829                                 * try to allocate it as a controlling terminal
3830                                 */
3831                                (void) strctty(stp);
3832                        }
3833                }

3835                mutex_enter(&stp->sd_lock);

3837                /*
3838                 * As a performance concern we are caching the values of
3839                 * q_minpsz and q_maxpsz of the module below the stream
3840                 * head in the stream head.
3841                 */
3842                mutex_enter(QLOCK(stp->sd_wrq->q_next));
3843                rmin = stp->sd_wrq->q_next->q_minpsz;
3844                rmax = stp->sd_wrq->q_next->q_maxpsz;
3845                mutex_exit(QLOCK(stp->sd_wrq->q_next));

3847                /* Do this processing here as a performance concern */
3848                if (strmsgsz != 0) {
3849                        if (rmax == INFPSZ)
3850                                rmax = strmsgsz;
3851                        else  {
3852                                if (vp->v_type == VFIFO)
3853                                        rmax = MIN(PIPE_BUF, rmax);
3854                                else    rmax = MIN(strmsgsz, rmax);
3855                        }
3856                }

3858                mutex_enter(QLOCK(wrq));
3859                stp->sd_qn_minpsz = rmin;
3860                stp->sd_qn_maxpsz = rmax;
3861                mutex_exit(QLOCK(wrq));

3863                strendplumb(stp);
3864                mutex_exit(&stp->sd_lock);
3865                return (error);
3866        }

3868        case I_POP:
3869        {
3870                queue_t *q;

3872                if (stp->sd_flag & STRHUP)
3873                        return (ENXIO);
3874                if (!wrq->q_next)       /* for broken pipes */
3875                        return (EINVAL);

3877                if (error = strstartplumb(stp, flag, cmd))
3878                        return (error);

3880                /*
3881                 * If there is an anchor on this stream and popping
3882                 * the current module would attempt to pop through the
3883                 * anchor, then disallow the pop unless we have sufficient
3884                 * privileges; take the cheapest (non-locking) check
3885                 * first.
3886                 */
3887                if (secpolicy_ip_config(crp, B_TRUE) != 0 ||
3888                    (stp->sd_anchorzone != crgetzoneid(crp))) {
3889                        mutex_enter(&stp->sd_lock);
```



```
3890                        /*
3891                         * Anchors only apply if there's at least one
3892                         * module on the stream (sd_pushcnt > 0).
3893                         */
3894                        if (stp->sd_pushcnt > 0 &&
3895                            stp->sd_pushcnt == stp->sd_anchor &&
3896                            stp->sd_vnode->v_type != VFIFO) {
3897                                strendplumb(stp);
3898                                mutex_exit(&stp->sd_lock);
3899                                if (stp->sd_anchorzone != crgetzoneid(crp))
3900                                        return (EINVAL);
3901                                /* Audit and report error */
3902                                return (secpolicy_ip_config(crp, B_FALSE));
3903                        }
3904                        mutex_exit(&stp->sd_lock);
3905                }

3907                q = wrq->q_next;
3908                TRACE_2(TR_FAC_STREAMS_FR, TR_I_POP,
3909                    "I_POP:%p from %p", q, stp);
3910                if (q->q_next == NULL || (q->q_flag & (QREADR|QISDRV))) {
3911                        error = EINVAL;
3912                } else {
3913                        qdetach(_RD(q), 1, flag, crp, B_FALSE);
3914                        error = 0;
3915                }
3916                mutex_enter(&stp->sd_lock);

3918                /*
3919                 * As a performance concern we are caching the values of
3920                 * q_minpsz and q_maxpsz of the module below the stream
3921                 * head in the stream head.
3922                 */
3923                mutex_enter(QLOCK(wrq->q_next));
3924                rmin = wrq->q_next->q_minpsz;
3925                rmax = wrq->q_next->q_maxpsz;
3926                mutex_exit(QLOCK(wrq->q_next));

3928                /* Do this processing here as a performance concern */
3929                if (strmsgsz != 0) {
3930                        if (rmax == INFPSZ)
3931                                rmax = strmsgsz;
3932                        else  {
3933                                if (vp->v_type == VFIFO)
3934                                        rmax = MIN(PIPE_BUF, rmax);
3935                                else    rmax = MIN(strmsgsz, rmax);
3936                        }
3937                }

3939                mutex_enter(QLOCK(wrq));
3940                stp->sd_qn_minpsz = rmin;
3941                stp->sd_qn_maxpsz = rmax;
3942                mutex_exit(QLOCK(wrq));

3944                /* If we popped through the anchor, then reset the anchor. */
3945                if (stp->sd_pushcnt < stp->sd_anchor) {
3946                        stp->sd_anchor = 0;
3947                        stp->sd_anchorzone = 0;
3948                }
3949                strendplumb(stp);
3950                mutex_exit(&stp->sd_lock);
3951                return (error);
3952        }

3954        case _I_MUXID2FD:
3955        {
```

```
3956                /*
3957                 * Create a fd for a I_PLINK'ed lower stream with a given
3958                 * muxid.  With the fd, application can send down ioctls,
3959                 * like I_LIST, to the previously I_PLINK'ed stream.  Note
3960                 * that after getting the fd, the application has to do an
3961                 * I_PUNLINK on the muxid before it can do any operation
3962                 * on the lower stream.  This is required by spec1170.
3963                 *
3964                 * The fd used to do this ioctl should point to the same
3965                 * controlling device used to do the I_PLINK.  If it uses
3966                 * a different stream or an invalid muxid, I_MUXID2FD will
3967                 * fail.  The error code is set to EINVAL.
3968                 *
3969                 * The intended use of this interface is the following.
3970                 * An application I_PLINK'ed a stream and exits.  The fd
3971                 * to the lower stream is gone.  Another application
3972                 * wants to get a fd to the lower stream, it uses I_MUXID2FD.
3973                 */
3974                int muxid = (int)arg;
3975                int fd;
3976                linkinfo_t *linkp;
3977                struct file *fp;
3978                netstack_t *ns;
3979                str_stack_t *ss;

3981                /*
3982                 * Do not allow the wildcard muxid.  This ioctl is not
3983                 * intended to find arbitrary link.
3984                 */
3985                if (muxid == 0) {
3986                        return (EINVAL);
3987                }

3989                ns = netstack_find_by_cred(crp);
3990                ASSERT(ns != NULL);
3991                ss = ns->netstack_str;
3992                ASSERT(ss != NULL);

3994                mutex_enter(&muxifier);
3995                linkp = findlinks(vp->v_stream, muxid, LINKPERSIST, ss);
3996                if (linkp == NULL) {
3997                        mutex_exit(&muxifier);
3998                        netstack_rele(ss->ss_netstack);
3999                        return (EINVAL);
4000                }

4002                if ((fd = ufalloc(0)) == -1) {
4003                        mutex_exit(&muxifier);
4004                        netstack_rele(ss->ss_netstack);
4005                        return (EMFILE);
4006                }
4007                fp = linkp->li_fpdown;
4008                mutex_enter(&fp->f_tlock);
4009                fp->f_count++;
4010                mutex_exit(&fp->f_tlock);
4011                mutex_exit(&muxifier);
4012                setf(fd, fp);
4013                *rvalp = fd;
4014                netstack_rele(ss->ss_netstack);
4015                return (0);
4016        }

4018        case _I_INSERT:
4019        {
4020                /*
4021                 * To insert a module to a given position in a stream.
```



```
4022                 * In the first release, only allow privileged user
4023                 * to use this ioctl. Furthermore, the insert is only allowed
4024                 * below an anchor if the zoneid is the same as the zoneid
4025                 * which created the anchor.
4026                 *
4027                 * Note that we do not plan to support this ioctl
4028                 * on pipes in the first release.  We want to learn more
4029                 * about the implications of these ioctls before extending
4030                 * their support.  And we do not think these features are
4031                 * valuable for pipes.
4032                 */
4033                STRUCT_DECL(strmodconf, strmodinsert);
4034                char mod_name[FMNAMESZ + 1];
4035                fmodsw_impl_t *fp;
4036                dev_t dummydev;
4037                queue_t *tmp_wrq;
4038                int pos;
4039                boolean_t is_insert;

4041                STRUCT_INIT(strmodinsert, flag);
4042                if (stp->sd_flag & STRHUP)
4043                        return (ENXIO);
4044                if (STRMATED(stp))
4045                        return (EINVAL);
4046                if ((error = secpolicy_net_config(crp, B_FALSE)) != 0)
4047                        return (error);
4048                if (stp->sd_anchor != 0 &&
4049                    stp->sd_anchorzone != crgetzoneid(crp))
4050                        return (EINVAL);

4052                error = strcopyin((void *)arg, STRUCT_BUF(strmodinsert),
4053                    STRUCT_SIZE(strmodinsert), copyflag);
4054                if (error)
4055                        return (error);

4057                /*
4058                 * Get module name and look up in fmodsw.
4059                 */
4060                error = (copyflag & U_TO_K ? copyinstr :
4061                    copystr)(STRUCT_FGETP(strmodinsert, mod_name),
4062                    mod_name, FMNAMESZ + 1, NULL);
4063                if (error)
4064                        return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

4066                if ((fp = fmodsw_find(mod_name, FMODSW_HOLD | FMODSW_LOAD)) ==
4067                    NULL)
4068                        return (EINVAL);

4070                if (error = strstartplumb(stp, flag, cmd)) {
4071                        fmodsw_rele(fp);
4072                        return (error);
4073                }

4075                /*
4076                 * Is this _I_INSERT just like an I_PUSH?  We need to know
4077                 * this because we do some optimizations if this is a
4078                 * module being pushed.
4079                 */
4080                pos = STRUCT_FGET(strmodinsert, pos);
4081                is_insert = (pos != 0);

4083                /*
4084                 * Make sure pos is valid.  Even though it is not an I_PUSH,
4085                 * we impose the same limit on the number of modules in a
4086                 * stream.
4087                 */
```

```
4088                mutex_enter(&stp->sd_lock);
4089                if (stp->sd_pushcnt >= nstrpush || pos < 0 ||
4090                    pos > stp->sd_pushcnt) {
4091                        fmodsw_rele(fp);
4092                        strendplumb(stp);
4093                        mutex_exit(&stp->sd_lock);
4094                        return (EINVAL);
4095                }
4096                if (stp->sd_anchor != 0) {
4097                        /*
4098                         * Is this insert below the anchor?
4099                         * Pushcnt hasn't been increased yet hence
4100                         * we test for greater than here, and greater or
4101                         * equal after qattach.
4102                         */
4103                        if (pos > (stp->sd_pushcnt - stp->sd_anchor) &&
4104                            stp->sd_anchorzone != crgetzoneid(crp)) {
4105                                fmodsw_rele(fp);
4106                                strendplumb(stp);
4107                                mutex_exit(&stp->sd_lock);
4108                                return (EPERM);
4109                        }
4110                }

4112                mutex_exit(&stp->sd_lock);

4114                /*
4115                 * First find the correct position this module to
4116                 * be inserted.  We don't need to call claimstr()
4117                 * as the stream should not be changing at this point.
4118                 *
4119                 * Insert new module and call its open routine
4120                 * via qattach().  Modules don't change device
4121                 * numbers, so just ignore dummydev here.
4122                 */
4123                for (tmp_wrq = stp->sd_wrq; pos > 0;
4124                    tmp_wrq = tmp_wrq->q_next, pos--) {
4125                        ASSERT(SAMESTR(tmp_wrq));
4126                }
4127                dummydev = vp->v_rdev;
4128                if ((error = qattach(_RD(tmp_wrq), &dummydev, 0, crp,
4129                    fp, is_insert)) != 0) {
4130                        mutex_enter(&stp->sd_lock);
4131                        strendplumb(stp);
4132                        mutex_exit(&stp->sd_lock);
4133                        return (error);
4134                }

4136                mutex_enter(&stp->sd_lock);

4138                /*
4139                 * As a performance concern we are caching the values of
4140                 * q_minpsz and q_maxpsz of the module below the stream
4141                 * head in the stream head.
4142                 */
4143                if (!is_insert) {
4144                        mutex_enter(QLOCK(stp->sd_wrq->q_next));
4145                        rmin = stp->sd_wrq->q_next->q_minpsz;
4146                        rmax = stp->sd_wrq->q_next->q_maxpsz;
4147                        mutex_exit(QLOCK(stp->sd_wrq->q_next));

4149                        /* Do this processing here as a performance concern */
4150                        if (strmsgsz != 0) {
4151                                if (rmax == INFPSZ) {
4152                                        rmax = strmsgsz;
4153                                } else  {
```



```
4154                                        rmax = MIN(strmsgsz, rmax);
4155                                }
4156                        }

4158                        mutex_enter(QLOCK(wrq));
4159                        stp->sd_qn_minpsz = rmin;
4160                        stp->sd_qn_maxpsz = rmax;
4161                        mutex_exit(QLOCK(wrq));
4162                }

4164                /*
4165                 * Need to update the anchor value if this module is
4166                 * inserted below the anchor point.
4167                 */
4168                if (stp->sd_anchor != 0) {
4169                        pos = STRUCT_FGET(strmodinsert, pos);
4170                        if (pos >= (stp->sd_pushcnt - stp->sd_anchor))
4171                                stp->sd_anchor++;
4172                }

4174                strendplumb(stp);
4175                mutex_exit(&stp->sd_lock);
4176                return (0);
4177        }

4179        case _I_REMOVE:
4180        {
4181                /*
4182                 * To remove a module with a given name in a stream.  The
4183                 * caller of this ioctl needs to provide both the name and
4184                 * the position of the module to be removed.  This eliminates
4185                 * the ambiguity of removal if a module is inserted/pushed
4186                 * multiple times in a stream.  In the first release, only
4187                 * allow privileged user to use this ioctl.
4188                 * Furthermore, the remove is only allowed
4189                 * below an anchor if the zoneid is the same as the zoneid
4190                 * which created the anchor.
4191                 *
4192                 * Note that we do not plan to support this ioctl
4193                 * on pipes in the first release.  We want to learn more
4194                 * about the implications of these ioctls before extending
4195                 * their support.  And we do not think these features are
4196                 * valuable for pipes.
4197                 *
4198                 * Also note that _I_REMOVE cannot be used to remove a
4199                 * driver or the stream head.
4200                 */
4201                STRUCT_DECL(strmodconf, strmodremove);
4202                queue_t *q;
4203                int pos;
4204                char mod_name[FMNAMESZ + 1];
4205                boolean_t is_remove;

4207                STRUCT_INIT(strmodremove, flag);
4208                if (stp->sd_flag & STRHUP)
4209                        return (ENXIO);
4210                if (STRMATED(stp))
4211                        return (EINVAL);
4212                if ((error = secpolicy_net_config(crp, B_FALSE)) != 0)
4213                        return (error);
4214                if (stp->sd_anchor != 0 &&
4215                    stp->sd_anchorzone != crgetzoneid(crp))
4216                        return (EINVAL);

4218                error = strcopyin((void *)arg, STRUCT_BUF(strmodremove),
4219                    STRUCT_SIZE(strmodremove), copyflag);
```

```
4220                if (error)
4221                        return (error);

4223                error = (copyflag & U_TO_K ? copyinstr :
4224                    copystr)(STRUCT_FGETP(strmodremove, mod_name),
4225                    mod_name, FMNAMESZ + 1, NULL);
4226                if (error)
4227                        return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

4229                if ((error = strstartplumb(stp, flag, cmd)) != 0)
4230                        return (error);

4232                /*
4233                 * Match the name of given module to the name of module at
4234                 * the given position.
4235                 */
4236                pos = STRUCT_FGET(strmodremove, pos);

4238                is_remove = (pos != 0);
4239                for (q = stp->sd_wrq->q_next; SAMESTR(q) && pos > 0;
4240                    q = q->q_next, pos--)
4241                        ;
4242                if (pos > 0 || !SAMESTR(q) ||
4243                    strcmp(Q2NAME(q), mod_name) != 0) {
4244                        mutex_enter(&stp->sd_lock);
4245                        strendplumb(stp);
4246                        mutex_exit(&stp->sd_lock);
4247                        return (EINVAL);
4248                }

4250                /*
4251                 * If the position is at or below an anchor, then the zoneid
4252                 * must match the zoneid that created the anchor.
4253                 */
4254                if (stp->sd_anchor != 0) {
4255                        pos = STRUCT_FGET(strmodremove, pos);
4256                        if (pos >= (stp->sd_pushcnt - stp->sd_anchor) &&
4257                            stp->sd_anchorzone != crgetzoneid(crp)) {
4258                                mutex_enter(&stp->sd_lock);
4259                                strendplumb(stp);
4260                                mutex_exit(&stp->sd_lock);
4261                                return (EPERM);
4262                        }
4263                }


4266                ASSERT(!(q->q_flag & QREADR));
4267                qdetach(_RD(q), 1, flag, crp, is_remove);

4269                mutex_enter(&stp->sd_lock);

4271                /*
4272                 * As a performance concern we are caching the values of
4273                 * q_minpsz and q_maxpsz of the module below the stream
4274                 * head in the stream head.
4275                 */
4276                if (!is_remove) {
4277                        mutex_enter(QLOCK(wrq->q_next));
4278                        rmin = wrq->q_next->q_minpsz;
4279                        rmax = wrq->q_next->q_maxpsz;
4280                        mutex_exit(QLOCK(wrq->q_next));

4282                        /* Do this processing here as a performance concern */
4283                        if (strmsgsz != 0) {
4284                                if (rmax == INFPSZ)
4285                                        rmax = strmsgsz;
```



```
4286                                else  {
4287                                        if (vp->v_type == VFIFO)
4288                                                rmax = MIN(PIPE_BUF, rmax);
4289                                        else    rmax = MIN(strmsgsz, rmax);
4290                                }
4291                        }

4293                        mutex_enter(QLOCK(wrq));
4294                        stp->sd_qn_minpsz = rmin;
4295                        stp->sd_qn_maxpsz = rmax;
4296                        mutex_exit(QLOCK(wrq));
4297                }

4299                /*
4300                 * Need to update the anchor value if this module is removed
4301                 * at or below the anchor point.  If the removed module is at
4302                 * the anchor point, remove the anchor for this stream if
4303                 * there is no module above the anchor point.  Otherwise, if
4304                 * the removed module is below the anchor point, decrement the
4305                 * anchor point by 1.
4306                 */
4307                if (stp->sd_anchor != 0) {
4308                        pos = STRUCT_FGET(strmodremove, pos);
4309                        if (pos == stp->sd_pushcnt - stp->sd_anchor + 1)
4310                                stp->sd_anchor = 0;
4311                        else if (pos > (stp->sd_pushcnt - stp->sd_anchor + 1))
4312                                stp->sd_anchor--;
4313                }

4315                strendplumb(stp);
4316                mutex_exit(&stp->sd_lock);
4317                return (0);
4318        }

4320        case I_ANCHOR:
4321                /*
4322                 * Set the anchor position on the stream to reside at
4323                 * the top module (in other words, the top module
4324                 * cannot be popped).  Anchors with a FIFO make no
4325                 * obvious sense, so they're not allowed.
4326                 */
4327                mutex_enter(&stp->sd_lock);

4329                if (stp->sd_vnode->v_type == VFIFO) {
4330                        mutex_exit(&stp->sd_lock);
4331                        return (EINVAL);
4332                }
4333                /* Only allow the same zoneid to update the anchor */
4334                if (stp->sd_anchor != 0 &&
4335                    stp->sd_anchorzone != crgetzoneid(crp)) {
4336                        mutex_exit(&stp->sd_lock);
4337                        return (EINVAL);
4338                }
4339                stp->sd_anchor = stp->sd_pushcnt;
4340                stp->sd_anchorzone = crgetzoneid(crp);
4341                mutex_exit(&stp->sd_lock);
4342                return (0);

4344        case I_LOOK:
4345                /*
4346                 * Get name of first module downstream.
4347                 * If no module, return an error.
4348                 */
4349                claimstr(wrq);
4350                if (_SAMESTR(wrq) && wrq->q_next->q_next != NULL) {
4351                        char *name = Q2NAME(wrq->q_next);
```

```
4353                        error = strcopyout(name, (void *)arg, strlen(name) + 1,
4354                            copyflag);
4355                        releasestr(wrq);
4356                        return (error);
4357                }
4358                releasestr(wrq);
4359                return (EINVAL);

4361        case I_LINK:
4362        case I_PLINK:
4363                /*
4364                 * Link a multiplexor.
4365                 */
4366                return (mlink(vp, cmd, (int)arg, crp, rvalp, 0));

4368        case _I_PLINK_LH:
4369                /*
4370                 * Link a multiplexor: Call must originate from kernel.
4371                 */
4372                if (kioctl)
4373                        return (ldi_mlink_lh(vp, cmd, arg, crp, rvalp));

4375                return (EINVAL);
4376        case I_UNLINK:
4377        case I_PUNLINK:
4378                /*
4379                 * Unlink a multiplexor.
4380                 * If arg is -1, unlink all links for which this is the
4381                 * controlling stream.  Otherwise, arg is an index number
4382                 * for a link to be removed.
4383                 */
4384        {
4385                struct linkinfo *linkp;
4386                int native_arg = (int)arg;
4387                int type;
4388                netstack_t *ns;
4389                str_stack_t *ss;

4391                TRACE_1(TR_FAC_STREAMS_FR,
4392                    TR_I_UNLINK, "I_UNLINK/I_PUNLINK:%p", stp);
4393                if (vp->v_type == VFIFO) {
4394                        return (EINVAL);
4395                }
4396                if (cmd == I_UNLINK)
4397                        type = LINKNORMAL;
4398                else    /* I_PUNLINK */
4399                        type = LINKPERSIST;
4400                if (native_arg == 0) {
4401                        return (EINVAL);
4402                }
4403                ns = netstack_find_by_cred(crp);
4404                ASSERT(ns != NULL);
4405                ss = ns->netstack_str;
4406                ASSERT(ss != NULL);

4408                if (native_arg == MUXID_ALL)
4409                        error = munlinkall(stp, type, crp, rvalp, ss);
4410                else {
4411                        mutex_enter(&muxifier);
4412                        if (!(linkp = findlinks(stp, (int)arg, type, ss))) {
4413                                /* invalid user supplied index number */
4414                                mutex_exit(&muxifier);
4415                                netstack_rele(ss->ss_netstack);
4416                                return (EINVAL);
4417                        }
```



```
4418                        /* munlink drops the muxifier lock */
4419                        error = munlink(stp, linkp, type, crp, rvalp, ss);
4420                }
4421                netstack_rele(ss->ss_netstack);
4422                return (error);
4423        }

4425        case I_FLUSH:
4426                /*
4427                 * send a flush message downstream
4428                 * flush message can indicate
4429                 * FLUSHR - flush read queue
4430                 * FLUSHW - flush write queue
4431                 * FLUSHRW - flush read/write queue
4432                 */
4433                if (stp->sd_flag & STRHUP)
4434                        return (ENXIO);
4435                if (arg & ~FLUSHRW)
4436                        return (EINVAL);

4438                for (;;) {
4439                        if (putnextctl1(stp->sd_wrq, M_FLUSH, (int)arg)) {
4440                                break;
4441                        }
4442                        if (error = strwaitbuf(1, BPRI_HI)) {
4443                                return (error);
4444                        }
4445                }

4447                /*
4448                 * Send down an unsupported ioctl and wait for the nack
4449                 * in order to allow the M_FLUSH to propagate back
4450                 * up to the stream head.
4451                 * Replaces if (qready()) runqueues();
4452                 */
4453                strioc.ic_cmd = -1;     /* The unsupported ioctl */
4454                strioc.ic_timout = 0;
4455                strioc.ic_len = 0;
4456                strioc.ic_dp = NULL;
4457                (void) strdoioctl(stp, &strioc, flag, K_TO_K, crp, rvalp);
4458                *rvalp = 0;
4459                return (0);

4461        case I_FLUSHBAND:
4462        {
4463                struct bandinfo binfo;

4465                error = strcopyin((void *)arg, &binfo, sizeof (binfo),
4466                    copyflag);
4467                if (error)
4468                        return (error);
4469                if (stp->sd_flag & STRHUP)
4470                        return (ENXIO);
4471                if (binfo.bi_flag & ~FLUSHRW)
4472                        return (EINVAL);
4473                while (!(mp = allocb(2, BPRI_HI))) {
4474                        if (error = strwaitbuf(2, BPRI_HI))
4475                                return (error);
4476                }
4477                mp->b_datap->db_type = M_FLUSH;
4478                *mp->b_wptr++ = binfo.bi_flag | FLUSHBAND;
4479                *mp->b_wptr++ = binfo.bi_pri;
4480                putnext(stp->sd_wrq, mp);
4481                /*
4482                 * Send down an unsupported ioctl and wait for the nack
4483                 * in order to allow the M_FLUSH to propagate back
```

```
4484                 * up to the stream head.
4485                 * Replaces if (qready()) runqueues();
4486                 */
4487                strioc.ic_cmd = -1;     /* The unsupported ioctl */
4488                strioc.ic_timout = 0;
4489                strioc.ic_len = 0;
4490                strioc.ic_dp = NULL;
4491                (void) strdoioctl(stp, &strioc, flag, K_TO_K, crp, rvalp);
4492                *rvalp = 0;
4493                return (0);
4494        }

4496        case I_SRDOPT:
4497                /*
4498                 * Set read options
4499                 *
4500                 * RNORM - default stream mode
4501                 * RMSGN - message no discard
4502                 * RMSGD - message discard
4503                 * RPROTNORM - fail read with EBADMSG for M_[PC]PROTOs
4504                 * RPROTDAT - convert M_[PC]PROTOs to M_DATAs
4505                 * RPROTDIS - discard M_[PC]PROTOs and retain M_DATAs
4506                 */
4507                if (arg & ~(RMODEMASK | RPROTMASK))
4508                        return (EINVAL);

4510                if ((arg & (RMSGD|RMSGN)) == (RMSGD|RMSGN))
4511                        return (EINVAL);

4513                mutex_enter(&stp->sd_lock);
4514                switch (arg & RMODEMASK) {
4515                case RNORM:
4516                        stp->sd_read_opt &= ~(RD_MSGDIS | RD_MSGNODIS);
4517                        break;
4518                case RMSGD:
4519                        stp->sd_read_opt = (stp->sd_read_opt & ~RD_MSGNODIS) |
4520                            RD_MSGDIS;
4521                        break;
4522                case RMSGN:
4523                        stp->sd_read_opt = (stp->sd_read_opt & ~RD_MSGDIS) |
4524                            RD_MSGNODIS;
4525                        break;
4526                }

4528                switch (arg & RPROTMASK) {
4529                case RPROTNORM:
4530                        stp->sd_read_opt &= ~(RD_PROTDAT | RD_PROTDIS);
4531                        break;

4533                case RPROTDAT:
4534                        stp->sd_read_opt = ((stp->sd_read_opt & ~RD_PROTDIS) |
4535                            RD_PROTDAT);
4536                        break;

4538                case RPROTDIS:
4539                        stp->sd_read_opt = ((stp->sd_read_opt & ~RD_PROTDAT) |
4540                            RD_PROTDIS);
4541                        break;
4542                }
4543                mutex_exit(&stp->sd_lock);
4544                return (0);

4546        case I_GRDOPT:
4547                /*
4548                 * Get read option and return the value
4549                 * to spot pointed to by arg
```



```
4550                 */
4551        {
4552                int rdopt;

4554                rdopt = ((stp->sd_read_opt & RD_MSGDIS) ? RMSGD :
4555                    ((stp->sd_read_opt & RD_MSGNODIS) ? RMSGN : RNORM));
4556                rdopt |= ((stp->sd_read_opt & RD_PROTDAT) ? RPROTDAT :
4557                    ((stp->sd_read_opt & RD_PROTDIS) ? RPROTDIS : RPROTNORM));

4559                return (strcopyout(&rdopt, (void *)arg, sizeof (int),
4560                    copyflag));
4561        }

4563        case I_SERROPT:
4564                /*
4565                 * Set error options
4566                 *
4567                 * RERRNORM - persistent read errors
4568                 * RERRNONPERSIST - non-persistent read errors
4569                 * WERRNORM - persistent write errors
4570                 * WERRNONPERSIST - non-persistent write errors
4571                 */
4572                if (arg & ~(RERRMASK | WERRMASK))
4573                        return (EINVAL);

4575                mutex_enter(&stp->sd_lock);
4576                switch (arg & RERRMASK) {
4577                case RERRNORM:
4578                        stp->sd_flag &= ~STRDERRNONPERSIST;
4579                        break;
4580                case RERRNONPERSIST:
4581                        stp->sd_flag |= STRDERRNONPERSIST;
4582                        break;
4583                }
4584                switch (arg & WERRMASK) {
4585                case WERRNORM:
4586                        stp->sd_flag &= ~STWRERRNONPERSIST;
4587                        break;
4588                case WERRNONPERSIST:
4589                        stp->sd_flag |= STWRERRNONPERSIST;
4590                        break;
4591                }
4592                mutex_exit(&stp->sd_lock);
4593                return (0);

4595        case I_GERROPT:
4596                /*
4597                 * Get error option and return the value
4598                 * to spot pointed to by arg
4599                 */
4600        {
4601                int erropt = 0;

4603                erropt |= (stp->sd_flag & STRDERRNONPERSIST) ? RERRNONPERSIST :
4604                    RERRNORM;
4605                erropt |= (stp->sd_flag & STWRERRNONPERSIST) ? WERRNONPERSIST :
4606                    WERRNORM;
4607                return (strcopyout(&erropt, (void *)arg, sizeof (int),
4608                    copyflag));
4609        }

4611        case I_SETSIG:
4612                /*
4613                 * Register the calling proc to receive the SIGPOLL
4614                 * signal based on the events given in arg.  If
4615                 * arg is zero, remove the proc from register list.
```

```
4616                 */
4617        {
4618                strsig_t *ssp, *pssp;
4619                struct pid *pidp;
4620
4621                pssp = NULL;
4622                pidp = curproc->p_pidp;
4623                /*
4624                 * Hold sd_lock to prevent traversal of sd_siglist while
4625                 * it is modified.
4626                 */
4627                mutex_enter(&stp->sd_lock);
4628                for (ssp = stp->sd_siglist; ssp && (ssp->ss_pidp != pidp);
4629                    pssp = ssp, ssp = ssp->ss_next)
4630                        ;
4631
4632                if (arg) {
4633                        if (arg & ~(S_INPUT|S_HIPRI|S_MSG|S_HANGUP|S_ERROR|
4634                            S_RDNORM|S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)) {
4635                                mutex_exit(&stp->sd_lock);
4636                                return (EINVAL);
4637                        }
4638                        if ((arg & S_BANDURG) && !(arg & S_RDBAND)) {
4639                                mutex_exit(&stp->sd_lock);
4640                                return (EINVAL);
4641                        }
4642
4643                        /*
4644                         * If proc not already registered, add it
4645                         * to list.
4646                         */
4647                        if (!ssp) {
4648                                ssp = kmem_alloc(sizeof (strsig_t), KM_SLEEP);
4649                                ssp->ss_pidp = pidp;
4650                                ssp->ss_pid = pidp->pid_id;
4651                                ssp->ss_next = NULL;
4652                                if (pssp)
4653                                        pssp->ss_next = ssp;
4654                                else
4655                                        stp->sd_siglist = ssp;
4656                                mutex_enter(&pidlock);
4657                                PID_HOLD(pidp);
4658                                mutex_exit(&pidlock);
4659                        }
4660
4661                        /*
4662                         * Set events.
4663                         */
4664                        ssp->ss_events = (int)arg;
4665                } else {
4666                        /*
4667                         * Remove proc from register list.
4668                         */
4669                        if (ssp) {
4670                                mutex_enter(&pidlock);
4671                                PID_RELE(pidp);
4672                                mutex_exit(&pidlock);
4673                                if (pssp)
4674                                        pssp->ss_next = ssp->ss_next;
4675                                else
4676                                        stp->sd_siglist = ssp->ss_next;
4677                                kmem_free(ssp, sizeof (strsig_t));
4678                        } else {
4679                                mutex_exit(&stp->sd_lock);
4680                                return (EINVAL);
4681                        }
```



```
4682                }
4683
4684                /*
4685                 * Recalculate OR of sig events.
4686                 */
4687                stp->sd_sigflags = 0;
4688                for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4689                        stp->sd_sigflags |= ssp->ss_events;
4690                mutex_exit(&stp->sd_lock);
4691                return (0);
4692        }
4693
4694        case I_GETSIG:
4695                /*
4696                 * Return (in arg) the current registration of events
4697                 * for which the calling proc is to be signaled.
4698                 */
4699        {
4700                struct strsig *ssp;
4701                struct pid  *pidp;
4702
4703                pidp = curproc->p_pidp;
4704                mutex_enter(&stp->sd_lock);
4705                for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4706                        if (ssp->ss_pidp == pidp) {
4707                                error = strcopyout(&ssp->ss_events, (void *)arg,
4708                                    sizeof (int), copyflag);
4709                                mutex_exit(&stp->sd_lock);
4710                                return (error);
4711                        }
4712                mutex_exit(&stp->sd_lock);
4713                return (EINVAL);
4714        }
4715
4716        case I_ESETSIG:
4717                /*
4718                 * Register the ss_pid to receive the SIGPOLL
4719                 * signal based on the events is ss_events arg.  If
4720                 * ss_events is zero, remove the proc from register list.
4721                 */
4722        {
4723                struct strsig *ssp, *pssp;
4724                struct proc *proc;
4725                struct pid  *pidp;
4726                pid_t pid;
4727                struct strsigset ss;
4728
4729                error = strcopyin((void *)arg, &ss, sizeof (ss), copyflag);
4730                if (error)
4731                        return (error);
4732
4733                pid = ss.ss_pid;
4734
4735                if (ss.ss_events != 0) {
4736                        /*
4737                         * Permissions check by sending signal 0.
4738                         * Note that when kill fails it does a set_errno
4739                         * causing the system call to fail.
4740                         */
4741                        error = kill(pid, 0);
4742                        if (error) {
4743                                return (error);
4744                        }
4745                }
4746                mutex_enter(&pidlock);
4747                if (pid == 0)
```

```
4748                    proc = curproc;
4749            else if (pid < 0)
4750                    proc = pgfind(-pid);
4751            else
4752                    proc = prfind(pid);
4753            if (proc == NULL) {
4754                    mutex_exit(&pidlock);
4755                    return (ESRCH);
4756            }
4757            if (pid < 0)
4758                    pidp = proc->p_pgidp;
4759            else
4760                    pidp = proc->p_pidp;
4761            ASSERT(pidp);
4762            /*
4763             * Get a hold on the pid structure while referencing it.
4764             * There is a separate PID_HOLD should it be inserted
4765             * in the list below.
4766             */
4767            PID_HOLD(pidp);
4768            mutex_exit(&pidlock);

4770            pssp = NULL;
4771            /*
4772             * Hold sd_lock to prevent traversal of sd_siglist while
4773             * it is modified.
4774             */
4775            mutex_enter(&stp->sd_lock);
4776            for (ssp = stp->sd_siglist; ssp && (ssp->ss_pid != pid);
4777                pssp = ssp, ssp = ssp->ss_next)
4778                    ;

4780            if (ss.ss_events) {
4781                    if (ss.ss_events &
4782                        ~(S_INPUT|S_HIPRI|S_MSG|S_HANGUP|S_ERROR|
4783                        S_RDNORM|S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)) {
4784                            mutex_exit(&stp->sd_lock);
4785                            mutex_enter(&pidlock);
4786                            PID_RELE(pidp);
4787                            mutex_exit(&pidlock);
4788                            return (EINVAL);
4789                    }
4790                    if ((ss.ss_events & S_BANDURG) &&
4791                        !(ss.ss_events & S_RDBAND)) {
4792                            mutex_exit(&stp->sd_lock);
4793                            mutex_enter(&pidlock);
4794                            PID_RELE(pidp);
4795                            mutex_exit(&pidlock);
4796                            return (EINVAL);
4797                    }

4799                    /*
4800                     * If proc not already registered, add it
4801                     * to list.
4802                     */
4803                    if (!ssp) {
4804                            ssp = kmem_alloc(sizeof (strsig_t), KM_SLEEP);
4805                            ssp->ss_pidp = pidp;
4806                            ssp->ss_pid = pid;
4807                            ssp->ss_next = NULL;
4808                            if (pssp)
4809                                    pssp->ss_next = ssp;
4810                            else
4811                                    stp->sd_siglist = ssp;
4812                            mutex_enter(&pidlock);
4813                            PID_HOLD(pidp);
```



```
4814                            mutex_exit(&pidlock);
4815                    }

4817                    /*
4818                     * Set events.
4819                     */
4820                    ssp->ss_events = ss.ss_events;
4821            } else {
4822                    /*
4823                     * Remove proc from register list.
4824                     */
4825                    if (ssp) {
4826                            mutex_enter(&pidlock);
4827                            PID_RELE(pidp);
4828                            mutex_exit(&pidlock);
4829                            if (pssp)
4830                                    pssp->ss_next = ssp->ss_next;
4831                            else
4832                                    stp->sd_siglist = ssp->ss_next;
4833                            kmem_free(ssp, sizeof (strsig_t));
4834                    } else {
4835                            mutex_exit(&stp->sd_lock);
4836                            mutex_enter(&pidlock);
4837                            PID_RELE(pidp);
4838                            mutex_exit(&pidlock);
4839                            return (EINVAL);
4840                    }
4841            }

4843            /*
4844             * Recalculate OR of sig events.
4845             */
4846            stp->sd_sigflags = 0;
4847            for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4848                    stp->sd_sigflags |= ssp->ss_events;
4849            mutex_exit(&stp->sd_lock);
4850            mutex_enter(&pidlock);
4851            PID_RELE(pidp);
4852            mutex_exit(&pidlock);
4853            return (0);
4854    }

4856    case I_EGETSIG:
4857            /*
4858             * Return (in arg) the current registration of events
4859             * for which the calling proc is to be signaled.
4860             */
4861    {
4862            struct strsig *ssp;
4863            struct proc *proc;
4864            pid_t pid;
4865            struct pid  *pidp;
4866            struct strsigset ss;

4868            error = strcopyin((void *)arg, &ss, sizeof (ss), copyflag);
4869            if (error)
4870                    return (error);

4872            pid = ss.ss_pid;
4873            mutex_enter(&pidlock);
4874            if (pid == 0)
4875                    proc = curproc;
4876            else if (pid < 0)
4877                    proc = pgfind(-pid);
4878            else
4879                    proc = prfind(pid);
```

```
4880                if (proc == NULL) {
4881                        mutex_exit(&pidlock);
4882                        return (ESRCH);
4883                }
4884                if (pid < 0)
4885                        pidp = proc->p_pgidp;
4886                else
4887                        pidp = proc->p_pidp;

4889                /* Prevent the pidp from being reassigned */
4890                PID_HOLD(pidp);
4891                mutex_exit(&pidlock);

4893                mutex_enter(&stp->sd_lock);
4894                for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4895                        if (ssp->ss_pid == pid) {
4896                                ss.ss_pid = ssp->ss_pid;
4897                                ss.ss_events = ssp->ss_events;
4898                                error = strcopyout(&ss, (void *)arg,
4899                                    sizeof (struct strsigset), copyflag);
4900                                mutex_exit(&stp->sd_lock);
4901                                mutex_enter(&pidlock);
4902                                PID_RELE(pidp);
4903                                mutex_exit(&pidlock);
4904                                return (error);
4905                        }
4906                mutex_exit(&stp->sd_lock);
4907                mutex_enter(&pidlock);
4908                PID_RELE(pidp);
4909                mutex_exit(&pidlock);
4910                return (EINVAL);
4911        }

4913        case I_PEEK:
4914        {
4915                STRUCT_DECL(strpeek, strpeek);
4916                size_t n;
4917                mblk_t *fmp, *tmp_mp = NULL;

4919                STRUCT_INIT(strpeek, flag);

4921                error = strcopyin((void *)arg, STRUCT_BUF(strpeek),
4922                    STRUCT_SIZE(strpeek), copyflag);
4923                if (error)
4924                        return (error);

4926                mutex_enter(QLOCK(rdq));
4927                /*
4928                 * Skip the invalid messages
4929                 */
4930                for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
4931                        if (mp->b_datap->db_type != M_SIG)
4932                                break;

4934                /*
4935                 * If user has requested to peek at a high priority message
4936                 * and first message is not, return 0
4937                 */
4938                if (mp != NULL) {
4939                        if ((STRUCT_FGET(strpeek, flags) & RS_HIPRI) &&
4940                            queclass(mp) == QNORM) {
4941                                *rvalp = 0;
4942                                mutex_exit(QLOCK(rdq));
4943                                return (0);
4944                        }
4945                } else if (stp->sd_struiordq == NULL ||
```



```
4946                    (STRUCT_FGET(strpeek, flags) & RS_HIPRI)) {
4947                        /*
4948                         * No mblks to look at at the streamhead and
4949                         * 1). This isn't a synch stream or
4950                         * 2). This is a synch stream but caller wants high
4951                         *      priority messages which is not supported by
4952                         *      the synch stream. (it only supports QNORM)
4953                         */
4954                        *rvalp = 0;
4955                        mutex_exit(QLOCK(rdq));
4956                        return (0);
4957                }

4959                fmp = mp;

4961                if (mp && mp->b_datap->db_type == M_PASSFP) {
4962                        mutex_exit(QLOCK(rdq));
4963                        return (EBADMSG);
4964                }

4966                ASSERT(mp == NULL || mp->b_datap->db_type == M_PCPROTO ||
4967                    mp->b_datap->db_type == M_PROTO ||
4968                    mp->b_datap->db_type == M_DATA);

4970                if (mp && mp->b_datap->db_type == M_PCPROTO) {
4971                        STRUCT_FSET(strpeek, flags, RS_HIPRI);
4972                } else {
4973                        STRUCT_FSET(strpeek, flags, 0);
4974                }


4977                if (mp && ((tmp_mp = dupmsg(mp)) == NULL)) {
4978                        mutex_exit(QLOCK(rdq));
4979                        return (ENOSR);
4980                }
4981                mutex_exit(QLOCK(rdq));

4983                /*
4984                 * set mp = tmp_mp, so that I_PEEK processing can continue.
4985                 * tmp_mp is used to free the dup'd message.
4986                 */
4987                mp = tmp_mp;

4989                uio.uio_fmode = 0;
4990                uio.uio_extflg = UIO_COPY_CACHED;
4991                uio.uio_segflg = (copyflag == U_TO_K) ? UIO_USERSPACE :
4992                    UIO_SYSSPACE;
4993                uio.uio_limit = 0;
4994                /*
4995                 * First process PROTO blocks, if any.
4996                 * If user doesn't want to get ctl info by setting maxlen <= 0,
4997                 * then set len to -1/0 and skip control blocks part.
4998                 */
4999                if (STRUCT_FGET(strpeek, ctlbuf.maxlen) < 0)
5000                        STRUCT_FSET(strpeek, ctlbuf.len, -1);
5001                else if (STRUCT_FGET(strpeek, ctlbuf.maxlen) == 0)
5002                        STRUCT_FSET(strpeek, ctlbuf.len, 0);
5003                else {
5004                        int     ctl_part = 0;

5006                        iov.iov_base = STRUCT_FGETP(strpeek, ctlbuf.buf);
5007                        iov.iov_len = STRUCT_FGET(strpeek, ctlbuf.maxlen);
5008                        uio.uio_iov = &iov;
5009                        uio.uio_resid = iov.iov_len;
5010                        uio.uio_loffset = 0;
5011                        uio.uio_iovcnt = 1;
```

```
5012                        while (mp && mp->b_datap->db_type != M_DATA &&
5013                            uio.uio_resid >= 0) {
5014                                ASSERT(STRUCT_FGET(strpeek, flags) == 0 ?
5015                                    mp->b_datap->db_type == M_PROTO :
5016                                    mp->b_datap->db_type == M_PCPROTO);

5018                                if ((n = MIN(uio.uio_resid,
5019                                    mp->b_wptr - mp->b_rptr)) != 0 &&
5020                                    (error = uiomove((char *)mp->b_rptr, n,
5021                                    UIO_READ, &uio)) != 0) {
5022                                        freemsg(tmp_mp);
5023                                        return (error);
5024                                }
5025                                ctl_part = 1;
5026                                mp = mp->b_cont;
5027                        }
5028                        /* No ctl message */
5029                        if (ctl_part == 0)
5030                                STRUCT_FSET(strpeek, ctlbuf.len, -1);
5031                        else
5032                                STRUCT_FSET(strpeek, ctlbuf.len,
5033                                    STRUCT_FGET(strpeek, ctlbuf.maxlen) -
5034                                    uio.uio_resid);
5035                }

5037                /*
5038                 * Now process DATA blocks, if any.
5039                 * If user doesn't want to get data info by setting maxlen <= 0,
5040                 * then set len to -1/0 and skip data blocks part.
5041                 */
5042                if (STRUCT_FGET(strpeek, databuf.maxlen) < 0)
5043                        STRUCT_FSET(strpeek, databuf.len, -1);
5044                else if (STRUCT_FGET(strpeek, databuf.maxlen) == 0)
5045                        STRUCT_FSET(strpeek, databuf.len, 0);
5046                else {
5047                        int     data_part = 0;

5049                        iov.iov_base = STRUCT_FGETP(strpeek, databuf.buf);
5050                        iov.iov_len = STRUCT_FGET(strpeek, databuf.maxlen);
5051                        uio.uio_iov = &iov;
5052                        uio.uio_resid = iov.iov_len;
5053                        uio.uio_loffset = 0;
5054                        uio.uio_iovcnt = 1;
5055                        while (mp && uio.uio_resid) {
5056                                if (mp->b_datap->db_type == M_DATA) {
5057                                        if ((n = MIN(uio.uio_resid,
5058                                            mp->b_wptr - mp->b_rptr)) != 0 &&
5059                                            (error = uiomove((char *)mp->b_rptr,
5060                                            n, UIO_READ, &uio)) != 0) {
5061                                                freemsg(tmp_mp);
5062                                                return (error);
5063                                        }
5064                                        data_part = 1;
5065                                }
5066                                ASSERT(data_part == 0 ||
5067                                    mp->b_datap->db_type == M_DATA);
5068                                mp = mp->b_cont;
5069                        }
5070                        /* No data message */
5071                        if (data_part == 0)
5072                                STRUCT_FSET(strpeek, databuf.len, -1);
5073                        else
5074                                STRUCT_FSET(strpeek, databuf.len,
5075                                    STRUCT_FGET(strpeek, databuf.maxlen) -
5076                                    uio.uio_resid);
5077                }
```



```
5078                freemsg(tmp_mp);

5080                /*
5081                 * It is a synch stream and user wants to get
5082                 * data (maxlen > 0).
5083                 * uio setup is done by the codes that process DATA
5084                 * blocks above.
5085                 */
5086                if ((fmp == NULL) && STRUCT_FGET(strpeek, databuf.maxlen) > 0) {
5087                        infod_t infod;

5089                        infod.d_cmd = INFOD_COPYOUT;
5090                        infod.d_res = 0;
5091                        infod.d_uiop = &uio;
5092                        error = infonext(rdq, &infod);
5093                        if (error == EINVAL || error == EBUSY)
5094                                error = 0;
5095                        if (error)
5096                                return (error);
5097                        STRUCT_FSET(strpeek, databuf.len, STRUCT_FGET(strpeek,
5098                            databuf.maxlen) - uio.uio_resid);
5099                        if (STRUCT_FGET(strpeek, databuf.len) == 0) {
5100                                /*
5101                                 * No data found by the infonext().
5102                                 */
5103                                STRUCT_FSET(strpeek, databuf.len, -1);
5104                        }
5105                }
5106                error = strcopyout(STRUCT_BUF(strpeek), (void *)arg,
5107                    STRUCT_SIZE(strpeek), copyflag);
5108                if (error) {
5109                        return (error);
5110                }
5111                /*
5112                 * If there is no message retrieved, set return code to 0
5113                 * otherwise, set it to 1.
5114                 */
5115                if (STRUCT_FGET(strpeek, ctlbuf.len) == -1 &&
5116                    STRUCT_FGET(strpeek, databuf.len) == -1)
5117                        *rvalp = 0;
5118                else
5119                        *rvalp = 1;
5120                return (0);
5121        }

5123        case I_FDINSERT:
5124        {
5125                STRUCT_DECL(strfdinsert, strfdinsert);
5126                struct file *resftp;
5127                struct stdata *resstp;
5128                t_uscalar_t     ival;
5129                ssize_t msgsize;
5130                struct strbuf mctl;

5132                STRUCT_INIT(strfdinsert, flag);
5133                if (stp->sd_flag & STRHUP)
5134                        return (ENXIO);
5135                /*
5136                 * STRDERR, STWRERR and STPLEX tested above.
5137                 */
5138                error = strcopyin((void *)arg, STRUCT_BUF(strfdinsert),
5139                    STRUCT_SIZE(strfdinsert), copyflag);
5140                if (error)
5141                        return (error);

5143                if (STRUCT_FGET(strfdinsert, offset) < 0 ||
```

```
5144                    (STRUCT_FGET(strfdinsert, offset) %
5145                    sizeof (t_uscalar_t)) != 0)
5146                        return (EINVAL);
5147                if ((resftp = getf(STRUCT_FGET(strfdinsert, fildes))) != NULL) {
5148                        if ((resstp = resftp->f_vnode->v_stream) == NULL) {
5149                                releasef(STRUCT_FGET(strfdinsert, fildes));
5150                                return (EINVAL);
5151                        }
5152                } else
5153                        return (EINVAL);

5155                mutex_enter(&resstp->sd_lock);
5156                if (resstp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
5157                        error = strgeterr(resstp,
5158                            STRDERR|STWRERR|STRHUP|STPLEX, 0);
5159                        if (error != 0) {
5160                                mutex_exit(&resstp->sd_lock);
5161                                releasef(STRUCT_FGET(strfdinsert, fildes));
5162                                return (error);
5163                        }
5164                }
5165                mutex_exit(&resstp->sd_lock);

5167 #ifdef  _ILP32
5168                {
5169                        queue_t *q;
5170                        queue_t *mate = NULL;

5172                        /* get read queue of stream terminus */
5173                        claimstr(resstp->sd_wrq);
5174                        for (q = resstp->sd_wrq->q_next; q->q_next != NULL;
5175                            q = q->q_next)
5176                                if (!STRMATED(resstp) && STREAM(q) != resstp &&
5177                                    mate == NULL) {
5178                                        ASSERT(q->q_qinfo->qi_srvp);
5179                                        ASSERT(_OTHERQ(q)->q_qinfo->qi_srvp);
5180                                        claimstr(q);
5181                                        mate = q;
5182                                }
5183                        q = _RD(q);
5184                        if (mate)
5185                                releasestr(mate);
5186                        releasestr(resstp->sd_wrq);
5187                        ival = (t_uscalar_t)q;
5188                }
5189 #else
5190                ival = (t_uscalar_t)getminor(resftp->f_vnode->v_rdev);
5191 #endif  /* _ILP32 */

5193                if (STRUCT_FGET(strfdinsert, ctlbuf.len) <
5194                    STRUCT_FGET(strfdinsert, offset) + sizeof (t_uscalar_t)) {
5195                        releasef(STRUCT_FGET(strfdinsert, fildes));
5196                        return (EINVAL);
5197                }

5199                /*
5200                 * Check for legal flag value.
5201                 */
5202                if (STRUCT_FGET(strfdinsert, flags) & ~RS_HIPRI) {
5203                        releasef(STRUCT_FGET(strfdinsert, fildes));
5204                        return (EINVAL);
5205                }

5207                /* get these values from those cached in the stream head */
5208                mutex_enter(QLOCK(stp->sd_wrq));
5209                rmin = stp->sd_qn_minpsz;
```



```
5210                rmax = stp->sd_qn_maxpsz;
5211                mutex_exit(QLOCK(stp->sd_wrq));

5213                /*
5214                 * Make sure ctl and data sizes together fall within
5215                 * the limits of the max and min receive packet sizes
5216                 * and do not exceed system limit.  A negative data
5217                 * length means that no data part is to be sent.
5218                 */
5219                ASSERT((rmax >= 0) || (rmax == INFPSZ));
5220                if (rmax == 0) {
5221                        releasef(STRUCT_FGET(strfdinsert, fildes));
5222                        return (ERANGE);
5223                }
5224                if ((msgsize = STRUCT_FGET(strfdinsert, databuf.len)) < 0)
5225                        msgsize = 0;
5226                if ((msgsize < rmin) ||
5227                    ((msgsize > rmax) && (rmax != INFPSZ)) ||
5228                    (STRUCT_FGET(strfdinsert, ctlbuf.len) > strctlsz)) {
5229                        releasef(STRUCT_FGET(strfdinsert, fildes));
5230                        return (ERANGE);
5231                }

5233                mutex_enter(&stp->sd_lock);
5234                while (!(STRUCT_FGET(strfdinsert, flags) & RS_HIPRI) &&
5235                    !canputnext(stp->sd_wrq)) {
5236                        if ((error = strwaitq(stp, WRITEWAIT, (ssize_t)0,
5237                            flag, -1, &done)) != 0 || done) {
5238                                mutex_exit(&stp->sd_lock);
5239                                releasef(STRUCT_FGET(strfdinsert, fildes));
5240                                return (error);
5241                        }
5242                        if ((error = i_straccess(stp, access)) != 0) {
5243                                mutex_exit(&stp->sd_lock);
5244                                releasef(
5245                                    STRUCT_FGET(strfdinsert, fildes));
5246                                return (error);
5247                        }
5248                }
5249                mutex_exit(&stp->sd_lock);

5251                /*
5252                 * Copy strfdinsert.ctlbuf into native form of
5253                 * ctlbuf to pass down into strmakemsg().
5254                 */
5255                mctl.maxlen = STRUCT_FGET(strfdinsert, ctlbuf.maxlen);
5256                mctl.len = STRUCT_FGET(strfdinsert, ctlbuf.len);
5257                mctl.buf = STRUCT_FGETP(strfdinsert, ctlbuf.buf);

5259                iov.iov_base = STRUCT_FGETP(strfdinsert, databuf.buf);
5260                iov.iov_len = STRUCT_FGET(strfdinsert, databuf.len);
5261                uio.uio_iov = &iov;
5262                uio.uio_iovcnt = 1;
5263                uio.uio_loffset = 0;
5264                uio.uio_segflg = (copyflag == U_TO_K) ? UIO_USERSPACE :
5265                    UIO_SYSSPACE;
5266                uio.uio_fmode = 0;
5267                uio.uio_extflg = UIO_COPY_CACHED;
5268                uio.uio_resid = iov.iov_len;
5269                if ((error = strmakemsg(&mctl,
5270                    &msgsize, &uio, stp,
5271                    STRUCT_FGET(strfdinsert, flags), &mp)) != 0 || !mp) {
5272                        STRUCT_FSET(strfdinsert, databuf.len, msgsize);
5273                        releasef(STRUCT_FGET(strfdinsert, fildes));
5274                        return (error);
5275                }
```

```
5277                STRUCT_FSET(strfdinsert, databuf.len, msgsize);

5279                /*
5280                 * Place the possibly reencoded queue pointer 'offset' bytes
5281                 * from the start of the control portion of the message.
5282                 */
5283                *((t_uscalar_t *)(mp->b_rptr +
5284                    STRUCT_FGET(strfdinsert, offset))) = ival;

5286                /*
5287                 * Put message downstream.
5288                 */
5289                stream_willservice(stp);
5290                putnext(stp->sd_wrq, mp);
5291                stream_runservice(stp);
5292                releasef(STRUCT_FGET(strfdinsert, fildes));
5293                return (error);
5294        }

5296        case I_SENDFD:
5297        {
5298                struct file *fp;

5300                if ((fp = getf((int)arg)) == NULL)
5301                        return (EBADF);
5302                error = do_sendfp(stp, fp, crp);
5303                if (auditing) {
5304                        audit_fdsend((int)arg, fp, error);
5305                }
5306                releasef((int)arg);
5307                return (error);
5308        }

5310        case I_RECVFD:
5311        case I_E_RECVFD:
5312        {
5313                struct k_strrecvfd *srf;
5314                int i, fd;

5316                mutex_enter(&stp->sd_lock);
5317                while (!(mp = getq(rdq))) {
5318                        if (stp->sd_flag & (STRHUP|STREOF)) {
5319                                mutex_exit(&stp->sd_lock);
5320                                return (ENXIO);
5321                        }
5322                        if ((error = strwaitq(stp, GETWAIT, (ssize_t)0,
5323                            flag, -1, &done)) != 0 || done) {
5324                                mutex_exit(&stp->sd_lock);
5325                                return (error);
5326                        }
5327                        if ((error = i_straccess(stp, access)) != 0) {
5328                                mutex_exit(&stp->sd_lock);
5329                                return (error);
5330                        }
5331                }
5332                if (mp->b_datap->db_type != M_PASSFP) {
5333                        putback(stp, rdq, mp, mp->b_band);
5334                        mutex_exit(&stp->sd_lock);
5335                        return (EBADMSG);
5336                }
5337                mutex_exit(&stp->sd_lock);

5339                srf = (struct k_strrecvfd *)mp->b_rptr;
5340                if ((fd = ufalloc(0)) == -1) {
5341                        mutex_enter(&stp->sd_lock);
```



```
5342                        putback(stp, rdq, mp, mp->b_band);
5343                        mutex_exit(&stp->sd_lock);
5344                        return (EMFILE);
5345                }
5346                if (cmd == I_RECVFD) {
5347                        struct o_strrecvfd      ostrfd;

5349                        /* check to see if uid/gid values are too large. */

5351                        if (srf->uid > (o_uid_t)USHRT_MAX ||
5352                            srf->gid > (o_gid_t)USHRT_MAX) {
5353                                mutex_enter(&stp->sd_lock);
5354                                putback(stp, rdq, mp, mp->b_band);
5355                                mutex_exit(&stp->sd_lock);
5356                                setf(fd, NULL); /* release fd entry */
5357                                return (EOVERFLOW);
5358                        }

5360                        ostrfd.fd = fd;
5361                        ostrfd.uid = (o_uid_t)srf->uid;
5362                        ostrfd.gid = (o_gid_t)srf->gid;

5364                        /* Null the filler bits */
5365                        for (i = 0; i < 8; i++)
5366                                ostrfd.fill[i] = 0;

5368                        error = strcopyout(&ostrfd, (void *)arg,
5369                            sizeof (struct o_strrecvfd), copyflag);
5370                } else {                /* I_E_RECVFD */
5371                        struct strrecvfd        strfd;

5373                        strfd.fd = fd;
5374                        strfd.uid = srf->uid;
5375                        strfd.gid = srf->gid;

5377                        /* null the filler bits */
5378                        for (i = 0; i < 8; i++)
5379                                strfd.fill[i] = 0;

5381                        error = strcopyout(&strfd, (void *)arg,
5382                            sizeof (struct strrecvfd), copyflag);
5383                }

5385                if (error) {
5386                        setf(fd, NULL); /* release fd entry */
5387                        mutex_enter(&stp->sd_lock);
5388                        putback(stp, rdq, mp, mp->b_band);
5389                        mutex_exit(&stp->sd_lock);
5390                        return (error);
5391                }
5392                if (auditing) {
5393                        audit_fdrecv(fd, srf->fp);
5394                }

5396                /*
5397                 * Always increment f_count since the freemsg() below will
5398                 * always call free_passfp() which performs a closef().
5399                 */
5400                mutex_enter(&srf->fp->f_tlock);
5401                srf->fp->f_count++;
5402                mutex_exit(&srf->fp->f_tlock);
5403                setf(fd, srf->fp);
5404                freemsg(mp);
5405                return (0);
5406        }
```

```
5408         case I_SWROPT:
5409                 /*
5410                  * Set/clear the write options. arg is a bit
5411                  * mask with any of the following bits set...
5412                  *      SNDZERO - send zero length message
5413                  *      SNDPIPE - send sigpipe to process if
5414                  *              sd_werror is set and process is
5415                  *              doing a write or putmsg.
5416                  * The new stream head write options should reflect
5417                  * what is in arg.
5418                  */
5419                 if (arg & ~(SNDZERO|SNDPIPE))
5420                         return (EINVAL);

5422                 mutex_enter(&stp->sd_lock);
5423                 stp->sd_wput_opt &= ~(SW_SIGPIPE|SW_SNDZERO);
5424                 if (arg & SNDZERO)
5425                         stp->sd_wput_opt |= SW_SNDZERO;
5426                 if (arg & SNDPIPE)
5427                         stp->sd_wput_opt |= SW_SIGPIPE;
5428                 mutex_exit(&stp->sd_lock);
5429                 return (0);

5431         case I_GWROPT:
5432         {
5433                 int wropt = 0;

5435                 if (stp->sd_wput_opt & SW_SNDZERO)
5436                         wropt |= SNDZERO;
5437                 if (stp->sd_wput_opt & SW_SIGPIPE)
5438                         wropt |= SNDPIPE;
5439                 return (strcopyout(&wropt, (void *)arg, sizeof (wropt),
5440                     copyflag));
5441         }

5443         case I_LIST:
5444                 /*
5445                  * Returns all the modules found on this stream,
5446                  * upto the driver. If argument is NULL, return the
5447                  * number of modules (including driver). If argument
5448                  * is not NULL, copy the names into the structure
5449                  * provided.
5450                  */

5452         {
5453                 queue_t *q;
5454                 char *qname;
5455                 int i, nmods;
5456                 struct str_mlist *mlist;
5457                 STRUCT_DECL(str_list, strlist);

5459                 if (arg == NULL) { /* Return number of modules plus driver */
5460                         if (stp->sd_vnode->v_type == VFIFO)
5461                                 *rvalp = stp->sd_pushcnt;
5462                         else
5463                                 *rvalp = stp->sd_pushcnt + 1;
5464                         return (0);
5465                 }

5467                 STRUCT_INIT(strlist, flag);

5469                 error = strcopyin((void *)arg, STRUCT_BUF(strlist),
5470                     STRUCT_SIZE(strlist), copyflag);
5471                 if (error != 0)
5472                         return (error);
```



```
5474                 mlist = STRUCT_FGETP(strlist, sl_modlist);
5475                 nmods = STRUCT_FGET(strlist, sl_nmods);
5476                 if (nmods <= 0)
5477                         return (EINVAL);

5479                 claimstr(stp->sd_wrq);
5480                 q = stp->sd_wrq;
5481                 for (i = 0; i < nmods && _SAMESTR(q); i++, q = q->q_next) {
5482                         qname = Q2NAME(q->q_next);
5483                         error = strcopyout(qname, &mlist[i], strlen(qname) + 1,
5484                             copyflag);
5485                         if (error != 0) {
5486                                 releasestr(stp->sd_wrq);
5487                                 return (error);
5488                         }
5489                 }
5490                 releasestr(stp->sd_wrq);
5491                 return (strcopyout(&i, (void *)arg, sizeof (int), copyflag));
5492         }

5494         case I_CKBAND:
5495         {
5496                 queue_t *q;
5497                 qband_t *qbp;

5499                 if ((arg < 0) || (arg >= NBAND))
5500                         return (EINVAL);
5501                 q = _RD(stp->sd_wrq);
5502                 mutex_enter(QLOCK(q));
5503                 if (arg > (int)q->q_nband) {
5504                         *rvalp = 0;
5505                 } else {
5506                         if (arg == 0) {
5507                                 if (q->q_first)
5508                                         *rvalp = 1;
5509                                 else
5510                                         *rvalp = 0;
5511                         } else {
5512                                 qbp = q->q_bandp;
5513                                 while (--arg > 0)
5514                                         qbp = qbp->qb_next;
5515                                 if (qbp->qb_first)
5516                                         *rvalp = 1;
5517                                 else
5518                                         *rvalp = 0;
5519                         }
5520                 }
5521                 mutex_exit(QLOCK(q));
5522                 return (0);
5523         }

5525         case I_GETBAND:
5526         {
5527                 int intpri;
5528                 queue_t *q;

5530                 q = _RD(stp->sd_wrq);
5531                 mutex_enter(QLOCK(q));
5532                 mp = q->q_first;
5533                 if (!mp) {
5534                         mutex_exit(QLOCK(q));
5535                         return (ENODATA);
5536                 }
5537                 intpri = (int)mp->b_band;
5538                 error = strcopyout(&intpri, (void *)arg, sizeof (int),
5539                     copyflag);
```

```
5540                mutex_exit(QLOCK(q));
5541                return (error);
5542        }

5544        case I_ATMARK:
5545        {
5546                queue_t *q;

5548                if (arg & ~(ANYMARK|LASTMARK))
5549                        return (EINVAL);
5550                q = _RD(stp->sd_wrq);
5551                mutex_enter(&stp->sd_lock);
5552                if ((stp->sd_flag & STRATMARK) && (arg == ANYMARK)) {
5553                        *rvalp = 1;
5554                } else {
5555                        mutex_enter(QLOCK(q));
5556                        mp = q->q_first;

5558                        if (mp == NULL)
5559                                *rvalp = 0;
5560                        else if ((arg == ANYMARK) && (mp->b_flag & MSGMARK))
5561                                *rvalp = 1;
5562                        else if ((arg == LASTMARK) && (mp == stp->sd_mark))
5563                                *rvalp = 1;
5564                        else
5565                                *rvalp = 0;
5566                        mutex_exit(QLOCK(q));
5567                }
5568                mutex_exit(&stp->sd_lock);
5569                return (0);
5570        }

5572        case I_CANPUT:
5573        {
5574                char band;

5576                if ((arg < 0) || (arg >= NBAND))
5577                        return (EINVAL);
5578                band = (char)arg;
5579                *rvalp = bcanputnext(stp->sd_wrq, band);
5580                return (0);
5581        }

5583        case I_SETCLTIME:
5584        {
5585                int closetime;

5587                error = strcopyin((void *)arg, &closetime, sizeof (int),
5588                    copyflag);
5589                if (error)
5590                        return (error);
5591                if (closetime < 0)
5592                        return (EINVAL);

5594                stp->sd_closetime = closetime;
5595                return (0);
5596        }

5598        case I_GETCLTIME:
5599        {
5600                int closetime;

5602                closetime = stp->sd_closetime;
5603                return (strcopyout(&closetime, (void *)arg, sizeof (int),
5604                    copyflag));
5605        }
```



```
5607        case TIOCGSID:
5608        {
5609                pid_t sid;

5611                mutex_enter(&stp->sd_lock);
5612                if (stp->sd_sidp == NULL) {
5613                        mutex_exit(&stp->sd_lock);
5614                        return (ENOTTY);
5615                }
5616                sid = stp->sd_sidp->pid_id;
5617                mutex_exit(&stp->sd_lock);
5618                return (strcopyout(&sid, (void *)arg, sizeof (pid_t),
5619                    copyflag));
5620        }

5622        case TIOCSPGRP:
5623        {
5624                pid_t pgrp;
5625                proc_t *q;
5626                pid_t   sid, fg_pgid, bg_pgid;

5628                if (error = strcopyin((void *)arg, &pgrp, sizeof (pid_t),
5629                    copyflag))
5630                        return (error);
5631                mutex_enter(&stp->sd_lock);
5632                mutex_enter(&pidlock);
5633                if (stp->sd_sidp != ttoproc(curthread)->p_sessp->s_sidp) {
5634                        mutex_exit(&pidlock);
5635                        mutex_exit(&stp->sd_lock);
5636                        return (ENOTTY);
5637                }
5638                if (pgrp == stp->sd_pgidp->pid_id) {
5639                        mutex_exit(&pidlock);
5640                        mutex_exit(&stp->sd_lock);
5641                        return (0);
5642                }
5643                if (pgrp <= 0 || pgrp >= maxpid) {
5644                        mutex_exit(&pidlock);
5645                        mutex_exit(&stp->sd_lock);
5646                        return (EINVAL);
5647                }
5648                if ((q = pgfind(pgrp)) == NULL ||
5649                    q->p_sessp != ttoproc(curthread)->p_sessp) {
5650                        mutex_exit(&pidlock);
5651                        mutex_exit(&stp->sd_lock);
5652                        return (EPERM);
5653                }
5654                sid = stp->sd_sidp->pid_id;
5655                fg_pgid = q->p_pgrp;
5656                bg_pgid = stp->sd_pgidp->pid_id;
5657                CL_SET_PROCESS_GROUP(curthread, sid, bg_pgid, fg_pgid);
5658                PID_RELE(stp->sd_pgidp);
5659                ctty_clear_sighuped();
5660                stp->sd_pgidp = q->p_pgidp;
5661                PID_HOLD(stp->sd_pgidp);
5662                mutex_exit(&pidlock);
5663                mutex_exit(&stp->sd_lock);
5664                return (0);
5665        }

5667        case TIOCGPGRP:
5668        {
5669                pid_t pgrp;

5671                mutex_enter(&stp->sd_lock);
```

```
5672                 if (stp->sd_sidp == NULL) {
5673                         mutex_exit(&stp->sd_lock);
5674                         return (ENOTTY);
5675                 }
5676                 pgrp = stp->sd_pgidp->pid_id;
5677                 mutex_exit(&stp->sd_lock);
5678                 return (strcopyout(&pgrp, (void *)arg, sizeof (pid_t),
5679                     copyflag));
5680         }

5682         case TIOCSCTTY:
5683         {
5684                 return (strctty(stp));
5685         }

5687         case TIOCNOTTY:
5688         {
5689                 /* freectty() always assumes curproc. */
5690                 if (freectty(B_FALSE) != 0)
5691                         return (0);
5692                 return (ENOTTY);
5693         }

5695         case FIONBIO:
5696         case FIOASYNC:
5697                 return (0);     /* handled by the upper layer */
5698         case F_ASSOCI_PID:
5699         {
5700                 if (crp != kcred)
5701                         return (EPERM);
5702                 if (is_xti_str(stp))
5703                         sh_insert_pid(stp, (pid_t)arg);
5704                 return (0);
5705         }
5706         case F_DASSOC_PID:
5707         {
5708                 if (crp != kcred)
5709                         return (EPERM);
5710                 if (is_xti_str(stp))
5711                         sh_remove_pid(stp, (pid_t)arg);
5712                 return (0);
5713         }
5714 #endif /* ! codereview */
5715         }
5716 }

5718 /*
5719  * Custom free routine used for M_PASSFP messages.
5720  */
5721 static void
5722 free_passfp(struct k_strrecvfd *srf)
5723 {
5724         (void) closef(srf->fp);
5725         kmem_free(srf, sizeof (struct k_strrecvfd) + sizeof (frtn_t));
5726 }

5728 /* ARGSUSED */
5729 int
5730 do_sendfp(struct stdata *stp, struct file *fp, struct cred *cr)
5731 {
5732         queue_t *qp, *nextqp;
5733         struct k_strrecvfd *srf;
5734         mblk_t *mp;
5735         frtn_t *frtnp;
5736         size_t bufsize;
5737         queue_t *mate = NULL;
```



```
5738         syncq_t *sq = NULL;
5739         int retval = 0;

5741         if (stp->sd_flag & STRHUP)
5742                 return (ENXIO);

5744         claimstr(stp->sd_wrq);

5746         /* Fastpath, we have a pipe, and we are already mated, use it. */
5747         if (STRMATED(stp)) {
5748                 qp = _RD(stp->sd_mate->sd_wrq);
5749                 claimstr(qp);
5750                 mate = qp;
5751         } else { /* Not already mated. */

5753                 /*
5754                  * Walk the stream to the end of this one.
5755                  * assumes that the claimstr() will prevent
5756                  * plumbing between the stream head and the
5757                  * driver from changing
5758                  */
5759                 qp = stp->sd_wrq;

5761                 /*
5762                  * Loop until we reach the end of this stream.
5763                  * On completion, qp points to the write queue
5764                  * at the end of the stream, or the read queue
5765                  * at the stream head if this is a fifo.
5766                  */
5767                 while (((qp = qp->q_next) != NULL) && _SAMESTR(qp))
5768                         ;

5770                 /*
5771                  * Just in case we get a q_next which is NULL, but
5772                  * not at the end of the stream.  This is actually
5773                  * broken, so we set an assert to catch it in
5774                  * debug, and set an error and return if not debug.
5775                  */
5776                 ASSERT(qp);
5777                 if (qp == NULL) {
5778                         releasestr(stp->sd_wrq);
5779                         return (EINVAL);
5780                 }

5782                 /*
5783                  * Enter the syncq for the driver, so (hopefully)
5784                  * the queue values will not change on us.
5785                  * XXXX - This will only prevent the race IFF only
5786                  *   the write side modifies the q_next member, and
5787                  *   the put procedure is protected by at least
5788                  *   MT_PERQ.
5789                  */
5790                 if ((sq = qp->q_syncq) != NULL)
5791                         entersq(sq, SQ_PUT);

5793                 /* Now get the q_next value from this qp. */
5794                 nextqp = qp->q_next;

5796                 /*
5797                  * If nextqp exists and the other stream is different
5798                  * from this one claim the stream, set the mate, and
5799                  * get the read queue at the stream head of the other
5800                  * stream.  Assumes that nextqp was at least valid when
5801                  * we got it.  Hopefully the entersq of the driver
5802                  * will prevent it from changing on us.
5803                  */
```

```
5804                if ((nextqp != NULL) && (STREAM(nextqp) != stp)) {
5805                        ASSERT(qp->q_qinfo->qi_srvp);
5806                        ASSERT(_OTHERQ(qp)->q_qinfo->qi_srvp);
5807                        ASSERT(_OTHERQ(qp->q_next)->q_qinfo->qi_srvp);
5808                        claimstr(nextqp);

5810                        /* Make sure we still have a q_next */
5811                        if (nextqp != qp->q_next) {
5812                                releasestr(stp->sd_wrq);
5813                                releasestr(nextqp);
5814                                return (EINVAL);
5815                        }

5817                        qp = _RD(STREAM(nextqp)->sd_wrq);
5818                        mate = qp;
5819                }
5820                /* If we entered the synq above, leave it. */
5821                if (sq != NULL)
5822                        leavesq(sq, SQ_PUT);
5823        } /*  STRMATED(STP)  */

5825        /* XXX prevents substitution of the ops vector */
5826        if (qp->q_qinfo != &strdata && qp->q_qinfo != &fifo_strdata) {
5827                retval = EINVAL;
5828                goto out;
5829        }

5831        if (qp->q_flag & QFULL) {
5832                retval = EAGAIN;
5833                goto out;
5834        }

5836        /*
5837         * Since M_PASSFP messages include a file descriptor, we use
5838         * esballoc() and specify a custom free routine (free_passfp()) that
5839         * will close the descriptor as part of freeing the message.  For
5840         * convenience, we stash the frtn_t right after the data block.
5841         */
5842        bufsize = sizeof (struct k_strrecvfd) + sizeof (frtn_t);
5843        srf = kmem_alloc(bufsize, KM_NOSLEEP);
5844        if (srf == NULL) {
5845                retval = EAGAIN;
5846                goto out;
5847        }

5849        frtnp = (frtn_t *)(srf + 1);
5850        frtnp->free_arg = (caddr_t)srf;
5851        frtnp->free_func = free_passfp;

5853        mp = esballoc((uchar_t *)srf, bufsize, BPRI_MED, frtnp);
5854        if (mp == NULL) {
5855                kmem_free(srf, bufsize);
5856                retval = EAGAIN;
5857                goto out;
5858        }
5859        mp->b_wptr += sizeof (struct k_strrecvfd);
5860        mp->b_datap->db_type = M_PASSFP;

5862        srf->fp = fp;
5863        srf->uid = crgetuid(curthread->t_cred);
5864        srf->gid = crgetgid(curthread->t_cred);
5865        mutex_enter(&fp->f_tlock);
5866        fp->f_count++;
5867        mutex_exit(&fp->f_tlock);

5869        put(qp, mp);
```



```
5870 out:
5871        releasestr(stp->sd_wrq);
5872        if (mate)
5873                releasestr(mate);
5874        return (retval);
5875 }

5877 /*
5878  * Send an ioctl message downstream and wait for acknowledgement.
5879  * flags may be set to either U_TO_K or K_TO_K and a combination
5880  * of STR_NOERROR or STR_NOSIG
5881  * STR_NOSIG: Signals are essentially ignored or held and have
5882  *      no effect for the duration of the call.
5883  * STR_NOERROR: Ignores stream head read, write and hup errors.
5884  *      Additionally, if an existing ioctl times out, it is assumed
5885  *      lost and and this ioctl will continue as if the previous ioctl had
5886  *      finished.  ETIME may be returned if this ioctl times out (i.e.
5887  *      ic_timout is not INFTIM).  Non-stream head errors may be returned if
5888  *      the ioc_error indicates that the driver/module had problems,
5889  *      an EFAULT was found when accessing user data, a lack of
5890  *      resources, etc.
5891  */
5892 int
5893 strdoioctl(
5894        struct stdata *stp,
5895        struct strioctl *strioc,
5896        int fflags,             /* file flags with model info */
5897        int flag,
5898        cred_t *crp,
5899        int *rvalp)
5900 {
5901        mblk_t *bp;
5902        struct iocblk *iocbp;
5903        struct copyreq *reqp;
5904        struct copyresp *resp;
5905        int id;
5906        int transparent = 0;
5907        int error = 0;
5908        int len = 0;
5909        caddr_t taddr;
5910        int copyflag = (flag & (U_TO_K | K_TO_K));
5911        int sigflag = (flag & STR_NOSIG);
5912        int errs;
5913        uint_t waitflags;
5914        boolean_t set_iocwaitne = B_FALSE;

5916        ASSERT(copyflag == U_TO_K || copyflag == K_TO_K);
5917        ASSERT((fflags & FMODELS) != 0);

5919        TRACE_2(TR_FAC_STREAMS_FR,
5920            TR_STRDOIOCTL,
5921            "strdoioctl:stp %p strioc %p", stp, strioc);
5922        if (strioc->ic_len == TRANSPARENT) {    /* send arg in M_DATA block */
5923                transparent = 1;
5924                strioc->ic_len = sizeof (intptr_t);
5925        }

5927        if (strioc->ic_len < 0 || (strmsgsz > 0 && strioc->ic_len > strmsgsz))
5928                return (EINVAL);

5930        if ((bp = allocb_cred_wait(sizeof (union ioctypes), sigflag, &error,
5931            crp, curproc->p_pid)) == NULL)
5932                        return (error);

5934        bzero(bp->b_wptr, sizeof (union ioctypes));
```

```
5936         iocbp = (struct iocblk *)bp->b_wptr;
5937         iocbp->ioc_count = strioc->ic_len;
5938         iocbp->ioc_cmd = strioc->ic_cmd;
5939         iocbp->ioc_flag = (fflags & FMODELS);

5941         crhold(crp);
5942         iocbp->ioc_cr = crp;
5943         DB_TYPE(bp) = M_IOCTL;
5944         bp->b_wptr += sizeof (struct iocblk);

5946         if (flag & STR_NOERROR)
5947                 errs = STPLEX;
5948         else
5949                 errs = STRHUP|STRDERR|STWRERR|STPLEX;

5951         /*
5952          * If there is data to copy into ioctl block, do so.
5953          */
5954         if (iocbp->ioc_count > 0) {
5955                 if (transparent)
5956                         /*
5957                          * Note: STR_NOERROR does not have an effect
5958                          * in putiocd()
5959                          */
5960                         id = K_TO_K | sigflag;
5961                 else
5962                         id = flag;
5963                 if ((error = putiocd(bp, strioc->ic_dp, id, crp)) != 0) {
5964                         freemsg(bp);
5965                         crfree(crp);
5966                         return (error);
5967                 }

5969                 /*
5970                  * We could have slept copying in user pages.
5971                  * Recheck the stream head state (the other end
5972                  * of a pipe could have gone away).
5973                  */
5974                 if (stp->sd_flag & errs) {
5975                         mutex_enter(&stp->sd_lock);
5976                         error = strgeterr(stp, errs, 0);
5977                         mutex_exit(&stp->sd_lock);
5978                         if (error != 0) {
5979                                 freemsg(bp);
5980                                 crfree(crp);
5981                                 return (error);
5982                         }
5983                 }
5984         }
5985         if (transparent)
5986                 iocbp->ioc_count = TRANSPARENT;

5988         /*
5989          * Block for up to STRTIMOUT milliseconds if there is an outstanding
5990          * ioctl for this stream already running.  All processes
5991          * sleeping here will be awakened as a result of an ACK
5992          * or NAK being received for the outstanding ioctl, or
5993          * as a result of the timer expiring on the outstanding
5994          * ioctl (a failure), or as a result of any waiting
5995          * process's timer expiring (also a failure).
5996          */

5998         error = 0;
5999         mutex_enter(&stp->sd_lock);
6000         while ((stp->sd_flag & IOCWAIT) ||
6001             (!set_iocwaitne && (stp->sd_flag & IOCWAITNE))) {
```



```
6002                 clock_t cv_rval;

6004                 TRACE_0(TR_FAC_STREAMS_FR,
6005                     TR_STRDOIOCTL_WAIT,
6006                     "strdoioctl sleeps - IOCWAIT");
6007                 cv_rval = str_cv_wait(&stp->sd_iocmonitor, &stp->sd_lock,
6008                     STRTIMOUT, sigflag);
6009                 if (cv_rval <= 0) {
6010                         if (cv_rval == 0) {
6011                                 error = EINTR;
6012                         } else {
6013                                 if (flag & STR_NOERROR) {
6014                                         /*
6015                                          * Terminating current ioctl in
6016                                          * progress -- assume it got lost and
6017                                          * wake up the other thread so that the
6018                                          * operation completes.
6019                                          */
6020                                         if (!(stp->sd_flag & IOCWAITNE)) {
6021                                                 set_iocwaitne = B_TRUE;
6022                                                 stp->sd_flag |= IOCWAITNE;
6023                                                 cv_broadcast(&stp->sd_monitor);
6024                                         }
6025                                         /*
6026                                          * Otherwise, there's a running
6027                                          * STR_NOERROR -- we have no choice
6028                                          * here but to wait forever (or until
6029                                          * interrupted).
6030                                          */
6031                                 } else {
6032                                         /*
6033                                          * pending ioctl has caused
6034                                          * us to time out
6035                                          */
6036                                         error = ETIME;
6037                                 }
6038                         }
6039                 } else if ((stp->sd_flag & errs)) {
6040                         error = strgeterr(stp, errs, 0);
6041                 }
6042                 if (error) {
6043                         mutex_exit(&stp->sd_lock);
6044                         freemsg(bp);
6045                         crfree(crp);
6046                         return (error);
6047                 }
6048         }

6050         /*
6051          * Have control of ioctl mechanism.
6052          * Send down ioctl packet and wait for response.
6053          */
6054         if (stp->sd_iocblk != (mblk_t *)-1) {
6055                 freemsg(stp->sd_iocblk);
6056         }
6057         stp->sd_iocblk = NULL;

6059         /*
6060          * If this is marked with 'noerror' (internal; mostly
6061          * I_{P,}{UN,}LINK), then make sure nobody else is able to get
6062          * in here by setting IOCWAITNE.
6063          */
6064         waitflags = IOCWAIT;
6065         if (flag & STR_NOERROR)
6066                 waitflags |= IOCWAITNE;
```

```
6068         stp->sd_flag |= waitflags;

6070         /*
6071          * Assign sequence number.
6072          */
6073         iocbp->ioc_id = stp->sd_iocid = getiocseqno();

6075         mutex_exit(&stp->sd_lock);

6077         TRACE_1(TR_FAC_STREAMS_FR,
6078             TR_STRDOIOCTL_PUT, "strdoioctl put: stp %p", stp);
6079         stream_willservice(stp);
6080         putnext(stp->sd_wrq, bp);
6081         stream_runservice(stp);

6083         /*
6084          * Timed wait for acknowledgment.  The wait time is limited by the
6085          * timeout value, which must be a positive integer (number of
6086          * milliseconds) to wait, or 0 (use default value of STRTIMOUT
6087          * milliseconds), or -1 (wait forever).  This will be awakened
6088          * either by an ACK/NAK message arriving, the timer expiring, or
6089          * the timer expiring on another ioctl waiting for control of the
6090          * mechanism.
6091          */
6092 waitioc:
6093         mutex_enter(&stp->sd_lock);


6096         /*
6097          * If the reply has already arrived, don't sleep.  If awakened from
6098          * the sleep, fail only if the reply has not arrived by then.
6099          * Otherwise, process the reply.
6100          */
6101         while (!stp->sd_iocblk) {
6102                 clock_t cv_rval;

6104                 if (stp->sd_flag & errs) {
6105                         error = strgeterr(stp, errs, 0);
6106                         if (error != 0) {
6107                                 stp->sd_flag &= ~waitflags;
6108                                 cv_broadcast(&stp->sd_iocmonitor);
6109                                 mutex_exit(&stp->sd_lock);
6110                                 crfree(crp);
6111                                 return (error);
6112                         }
6113                 }

6115                 TRACE_0(TR_FAC_STREAMS_FR,
6116                     TR_STRDOIOCTL_WAIT2,
6117                     "strdoioctl sleeps awaiting reply");
6118                 ASSERT(error == 0);

6120                 cv_rval = str_cv_wait(&stp->sd_monitor, &stp->sd_lock,
6121                     (strioc->ic_timout ?
6122                     strioc->ic_timout * 1000 : STRTIMOUT), sigflag);

6124                 /*
6125                  * There are four possible cases here: interrupt, timeout,
6126                  * wakeup by IOCWAITNE (above), or wakeup by strrput_nondata (a
6127                  * valid M_IOCTL reply).
6128                  *
6129                  * If we've been awakened by a STR_NOERROR ioctl on some other
6130                  * thread, then sd_iocblk will still be NULL, and IOCWAITNE
6131                  * will be set.  Pretend as if we just timed out.  Note that
6132                  * this other thread waited at least STRTIMOUT before trying to
6133                  * awaken our thread, so this is indistinguishable (even for
```



```
6134                  * INFTIM) from the case where we failed with ETIME waiting on
6135                  * IOCWAIT in the prior loop.
6136                  */
6137                 if (cv_rval > 0 && !(flag & STR_NOERROR) &&
6138                     stp->sd_iocblk == NULL && (stp->sd_flag & IOCWAITNE)) {
6139                         cv_rval = -1;
6140                 }

6142                 /*
6143                  * note: STR_NOERROR does not protect
6144                  * us here.. use ic_timout < 0
6145                  */
6146                 if (cv_rval <= 0) {
6147                         if (cv_rval == 0) {
6148                                 error = EINTR;
6149                         } else {
6150                                 error =  ETIME;
6151                         }
6152                         /*
6153                          * A message could have come in after we were scheduled
6154                          * but before we were actually run.
6155                          */
6156                         bp = stp->sd_iocblk;
6157                         stp->sd_iocblk = NULL;
6158                         if (bp != NULL) {
6159                                 if ((bp->b_datap->db_type == M_COPYIN) ||
6160                                     (bp->b_datap->db_type == M_COPYOUT)) {
6161                                         mutex_exit(&stp->sd_lock);
6162                                         if (bp->b_cont) {
6163                                                 freemsg(bp->b_cont);
6164                                                 bp->b_cont = NULL;
6165                                         }
6166                                         bp->b_datap->db_type = M_IOCDATA;
6167                                         bp->b_wptr = bp->b_rptr +
6168                                             sizeof (struct copyresp);
6169                                         resp = (struct copyresp *)bp->b_rptr;
6170                                         resp->cp_rval =
6171                                             (caddr_t)1; /* failure */
6172                                         stream_willservice(stp);
6173                                         putnext(stp->sd_wrq, bp);
6174                                         stream_runservice(stp);
6175                                         mutex_enter(&stp->sd_lock);
6176                                 } else {
6177                                         freemsg(bp);
6178                                 }
6179                         }
6180                         stp->sd_flag &= ~waitflags;
6181                         cv_broadcast(&stp->sd_iocmonitor);
6182                         mutex_exit(&stp->sd_lock);
6183                         crfree(crp);
6184                         return (error);
6185                 }
6186         }
6187         bp = stp->sd_iocblk;
6188         /*
6189          * Note: it is strictly impossible to get here with sd_iocblk set to
6190          * -1.  This is because the initial loop above doesn't allow any new
6191          * ioctls into the fray until all others have passed this point.
6192          */
6193         ASSERT(bp != NULL && bp != (mblk_t *)-1);
6194         TRACE_1(TR_FAC_STREAMS_FR,
6195             TR_STRDOIOCTL_ACK, "strdoioctl got reply: bp %p", bp);
6196         if ((bp->b_datap->db_type == M_IOCACK) ||
6197             (bp->b_datap->db_type == M_IOCNAK)) {
6198                 /* for detection of duplicate ioctl replies */
6199                 stp->sd_iocblk = (mblk_t *)-1;
```

```
6200                stp->sd_flag &= ~waitflags;
6201                cv_broadcast(&stp->sd_iocmonitor);
6202                mutex_exit(&stp->sd_lock);
6203        } else {
6204                /*
6205                 * flags not cleared here because we're still doing
6206                 * copy in/out for ioctl.
6207                 */
6208                stp->sd_iocblk = NULL;
6209                mutex_exit(&stp->sd_lock);
6210        }


6213        /*
6214         * Have received acknowledgment.
6215         */

6217        switch (bp->b_datap->db_type) {
6218        case M_IOCACK:
6219                /*
6220                 * Positive ack.
6221                 */
6222                iocbp = (struct iocblk *)bp->b_rptr;

6224                /*
6225                 * Set error if indicated.
6226                 */
6227                if (iocbp->ioc_error) {
6228                        error = iocbp->ioc_error;
6229                        break;
6230                }

6232                /*
6233                 * Set return value.
6234                 */
6235                *rvalp = iocbp->ioc_rval;

6237                /*
6238                 * Data may have been returned in ACK message (ioc_count > 0).
6239                 * If so, copy it out to the user's buffer.
6240                 */
6241                if (iocbp->ioc_count && !transparent) {
6242                        if (error = getiocd(bp, strioc->ic_dp, copyflag))
6243                                break;
6244                }
6245                if (!transparent) {
6246                        if (len)        /* an M_COPYOUT was used with I_STR */
6247                                strioc->ic_len = len;
6248                        else
6249                                strioc->ic_len = (int)iocbp->ioc_count;
6250                }
6251                break;

6253        case M_IOCNAK:
6254                /*
6255                 * Negative ack.
6256                 *
6257                 * The only thing to do is set error as specified
6258                 * in neg ack packet.
6259                 */
6260                iocbp = (struct iocblk *)bp->b_rptr;

6262                error = (iocbp->ioc_error ? iocbp->ioc_error : EINVAL);
6263                break;

6265        case M_COPYIN:
```



```
6266                /*
6267                 * Driver or module has requested user ioctl data.
6268                 */
6269                reqp = (struct copyreq *)bp->b_rptr;

6271                /*
6272                 * M_COPYIN should *never* have a message attached, though
6273                 * it's harmless if it does -- thus, panic on a DEBUG
6274                 * kernel and just free it on a non-DEBUG build.
6275                 */
6276                ASSERT(bp->b_cont == NULL);
6277                if (bp->b_cont != NULL) {
6278                        freemsg(bp->b_cont);
6279                        bp->b_cont = NULL;
6280                }

6282                error = putiocd(bp, reqp->cq_addr, flag, crp);
6283                if (error && bp->b_cont) {
6284                        freemsg(bp->b_cont);
6285                        bp->b_cont = NULL;
6286                }

6288                bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
6289                bp->b_datap->db_type = M_IOCDATA;

6291                mblk_setcred(bp, crp, curproc->p_pid);
6292                resp = (struct copyresp *)bp->b_rptr;
6293                resp->cp_rval = (caddr_t)(uintptr_t)error;
6294                resp->cp_flag = (fflags & FMODELS);

6296                stream_willservice(stp);
6297                putnext(stp->sd_wrq, bp);
6298                stream_runservice(stp);

6300                if (error) {
6301                        mutex_enter(&stp->sd_lock);
6302                        stp->sd_flag &= ~waitflags;
6303                        cv_broadcast(&stp->sd_iocmonitor);
6304                        mutex_exit(&stp->sd_lock);
6305                        crfree(crp);
6306                        return (error);
6307                }

6309                goto waitioc;

6311        case M_COPYOUT:
6312                /*
6313                 * Driver or module has ioctl data for a user.
6314                 */
6315                reqp = (struct copyreq *)bp->b_rptr;
6316                ASSERT(bp->b_cont != NULL);

6318                /*
6319                 * Always (transparent or non-transparent )
6320                 * use the address specified in the request
6321                 */
6322                taddr = reqp->cq_addr;
6323                if (!transparent)
6324                        len = (int)reqp->cq_size;

6326                /* copyout data to the provided address */
6327                error = getiocd(bp, taddr, copyflag);

6329                freemsg(bp->b_cont);
6330                bp->b_cont = NULL;
```

```
6332                bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
6333                bp->b_datap->db_type = M_IOCDATA;

6335                mblk_setcred(bp, crp, curproc->p_pid);
6336                resp = (struct copyresp *)bp->b_rptr;
6337                resp->cp_rval = (caddr_t)(uintptr_t)error;
6338                resp->cp_flag = (fflags & FMODELS);

6340                stream_willservice(stp);
6341                putnext(stp->sd_wrq, bp);
6342                stream_runservice(stp);

6344                if (error) {
6345                        mutex_enter(&stp->sd_lock);
6346                        stp->sd_flag &= ~waitflags;
6347                        cv_broadcast(&stp->sd_iocmonitor);
6348                        mutex_exit(&stp->sd_lock);
6349                        crfree(crp);
6350                        return (error);
6351                }
6352                goto waitioc;

6354        default:
6355                ASSERT(0);
6356                mutex_enter(&stp->sd_lock);
6357                stp->sd_flag &= ~waitflags;
6358                cv_broadcast(&stp->sd_iocmonitor);
6359                mutex_exit(&stp->sd_lock);
6360                break;
6361        }

6363        freemsg(bp);
6364        crfree(crp);
6365        return (error);
6366 }

6368 /*
6369  * Send an M_CMD message downstream and wait for a reply.  This is a ptools
6370  * special used to retrieve information from modules/drivers a stream without
6371  * being subjected to flow control or interfering with pending messages on the
6372  * stream (e.g. an ioctl in flight).
6373  */
6374 int
6375 strdocmd(struct stdata *stp, struct strcmd *scp, cred_t *crp)
6376 {
6377        mblk_t *mp;
6378        struct cmdblk *cmdp;
6379        int error = 0;
6380        int errs = STRHUP|STRDERR|STWRERR|STPLEX;
6381        clock_t rval, timeout = STRTIMOUT;

6383        if (scp->sc_len < 0 || scp->sc_len > sizeof (scp->sc_buf) ||
6384            scp->sc_timeout < -1)
6385                return (EINVAL);

6387        if (scp->sc_timeout > 0)
6388                timeout = scp->sc_timeout * MILLISEC;

6390        if ((mp = allocb_cred(sizeof (struct cmdblk), crp,
6391            curproc->p_pid)) == NULL)
6392                return (ENOMEM);

6394        crhold(crp);

6396        cmdp = (struct cmdblk *)mp->b_wptr;
6397        cmdp->cb_cr = crp;
```



```
6398        cmdp->cb_cmd = scp->sc_cmd;
6399        cmdp->cb_len = scp->sc_len;
6400        cmdp->cb_error = 0;
6401        mp->b_wptr += sizeof (struct cmdblk);

6403        DB_TYPE(mp) = M_CMD;
6404        DB_CPID(mp) = curproc->p_pid;

6406        /*
6407         * Copy in the payload.
6408         */
6409        if (cmdp->cb_len > 0) {
6410                mp->b_cont = allocb_cred(sizeof (scp->sc_buf), crp,
6411                    curproc->p_pid);
6412                if (mp->b_cont == NULL) {
6413                        error = ENOMEM;
6414                        goto out;
6415                }

6417                /* cb_len comes from sc_len, which has already been checked */
6418                ASSERT(cmdp->cb_len <= sizeof (scp->sc_buf));
6419                (void) bcopy(scp->sc_buf, mp->b_cont->b_wptr, cmdp->cb_len);
6420                mp->b_cont->b_wptr += cmdp->cb_len;
6421                DB_CPID(mp->b_cont) = curproc->p_pid;
6422        }

6424        /*
6425         * Since this mechanism is strictly for ptools, and since only one
6426         * process can be grabbed at a time, we simply fail if there's
6427         * currently an operation pending.
6428         */
6429        mutex_enter(&stp->sd_lock);
6430        if (stp->sd_flag & STRCMDWAIT) {
6431                mutex_exit(&stp->sd_lock);
6432                error = EBUSY;
6433                goto out;
6434        }
6435        stp->sd_flag |= STRCMDWAIT;
6436        ASSERT(stp->sd_cmdblk == NULL);
6437        mutex_exit(&stp->sd_lock);

6439        putnext(stp->sd_wrq, mp);
6440        mp = NULL;

6442        /*
6443         * Timed wait for acknowledgment.  If the reply has already arrived,
6444         * don't sleep.  If awakened from the sleep, fail only if the reply
6445         * has not arrived by then.  Otherwise, process the reply.
6446         */
6447        mutex_enter(&stp->sd_lock);
6448        while (stp->sd_cmdblk == NULL) {
6449                if (stp->sd_flag & errs) {
6450                        if ((error = strgeterr(stp, errs, 0)) != 0)
6451                                goto waitout;
6452                }

6454                rval = str_cv_wait(&stp->sd_monitor, &stp->sd_lock, timeout, 0);
6455                if (stp->sd_cmdblk != NULL)
6456                        break;

6458                if (rval <= 0) {
6459                        error = (rval == 0) ? EINTR : ETIME;
6460                        goto waitout;
6461                }
6462        }
```

```
6464         /*
6465          * We received a reply.
6466          */
6467         mp = stp->sd_cmdblk;
6468         stp->sd_cmdblk = NULL;
6469         ASSERT(mp != NULL && DB_TYPE(mp) == M_CMD);
6470         ASSERT(stp->sd_flag & STRCMDWAIT);
6471         stp->sd_flag &= ~STRCMDWAIT;
6472         mutex_exit(&stp->sd_lock);

6474         cmdp = (struct cmdblk *)mp->b_rptr;
6475         if ((error = cmdp->cb_error) != 0)
6476                 goto out;

6478         /*
6479          * Data may have been returned in the reply (cb_len > 0).
6480          * If so, copy it out to the user's buffer.
6481          */
6482         if (cmdp->cb_len > 0) {
6483                 if (mp->b_cont == NULL || MBLKL(mp->b_cont) < cmdp->cb_len) {
6484                         error = EPROTO;
6485                         goto out;
6486                 }

6488                 cmdp->cb_len = MIN(cmdp->cb_len, sizeof (scp->sc_buf));
6489                 (void) bcopy(mp->b_cont->b_rptr, scp->sc_buf, cmdp->cb_len);
6490         }
6491         scp->sc_len = cmdp->cb_len;
6492 out:
6493         freemsg(mp);
6494         crfree(crp);
6495         return (error);
6496 waitout:
6497         ASSERT(stp->sd_cmdblk == NULL);
6498         stp->sd_flag &= ~STRCMDWAIT;
6499         mutex_exit(&stp->sd_lock);
6500         crfree(crp);
6501         return (error);
6502 }

6504 /*
6505  * For the SunOS keyboard driver.
6506  * Return the next available "ioctl" sequence number.
6507  * Exported, so that streams modules can send "ioctl" messages
6508  * downstream from their open routine.
6509  */
6510 int
6511 getiocseqno(void)
6512 {
6513         int     i;

6515         mutex_enter(&strresources);
6516         i = ++ioc_id;
6517         mutex_exit(&strresources);
6518         return (i);
6519 }

6521 /*
6522  * Get the next message from the read queue.  If the message is
6523  * priority, STRPRI will have been set by strrput().  This flag
6524  * should be reset only when the entire message at the front of the
6525  * queue as been consumed.
6526  *
6527  * NOTE: strgetmsg and kstrgetmsg have much of the logic in common.
6528  */
6529 int
```



```
6530 strgetmsg(
6531         struct vnode *vp,
6532         struct strbuf *mctl,
6533         struct strbuf *mdata,
6534         unsigned char *prip,
6535         int *flagsp,
6536         int fmode,
6537         rval_t *rvp)
6538 {
6539         struct stdata *stp;
6540         mblk_t *bp, *nbp;
6541         mblk_t *savemp = NULL;
6542         mblk_t *savemptail = NULL;
6543         uint_t old_sd_flag;
6544         int flg;
6545         int more = 0;
6546         int error = 0;
6547         char first = 1;
6548         uint_t mark;            /* Contains MSG*MARK and _LASTMARK */
6549 #define _LASTMARK       0x8000  /* Distinct from MSG*MARK */
6550         unsigned char pri = 0;
6551         queue_t *q;
6552         int     pr = 0;                 /* Partial read successful */
6553         struct uio uios;
6554         struct uio *uiop = &uios;
6555         struct iovec iovs;
6556         unsigned char type;

6558         TRACE_1(TR_FAC_STREAMS_FR, TR_STRGETMSG_ENTER,
6559             "strgetmsg:%p", vp);

6561         ASSERT(vp->v_stream);
6562         stp = vp->v_stream;
6563         rvp->r_val1 = 0;

6565         mutex_enter(&stp->sd_lock);

6567         if ((error = i_straccess(stp, JCREAD)) != 0) {
6568                 mutex_exit(&stp->sd_lock);
6569                 return (error);
6570         }

6572         if (stp->sd_flag & (STRDERR|STPLEX)) {
6573                 error = strgeterr(stp, STRDERR|STPLEX, 0);
6574                 if (error != 0) {
6575                         mutex_exit(&stp->sd_lock);
6576                         return (error);
6577                 }
6578         }
6579         mutex_exit(&stp->sd_lock);

6581         switch (*flagsp) {
6582         case MSG_HIPRI:
6583                 if (*prip != 0)
6584                         return (EINVAL);
6585                 break;

6587         case MSG_ANY:
6588         case MSG_BAND:
6589                 break;

6591         default:
6592                 return (EINVAL);
6593         }
6594         /*
6595          * Setup uio and iov for data part
```

```
6596          */
6597         iovs.iov_base = mdata->buf;
6598         iovs.iov_len = mdata->maxlen;
6599         uios.uio_iov = &iovs;
6600         uios.uio_iovcnt = 1;
6601         uios.uio_loffset = 0;
6602         uios.uio_segflg = UIO_USERSPACE;
6603         uios.uio_fmode = 0;
6604         uios.uio_extflg = UIO_COPY_CACHED;
6605         uios.uio_resid = mdata->maxlen;
6606         uios.uio_offset = 0;

6608         q = _RD(stp->sd_wrq);
6609         mutex_enter(&stp->sd_lock);
6610         old_sd_flag = stp->sd_flag;
6611         mark = 0;
6612         for (;;) {
6613                 int done = 0;
6614                 mblk_t *q_first = q->q_first;

6616                 /*
6617                  * Get the next message of appropriate priority
6618                  * from the stream head.  If the caller is interested
6619                  * in band or hipri messages, then they should already
6620                  * be enqueued at the stream head.  On the other hand
6621                  * if the caller wants normal (band 0) messages, they
6622                  * might be deferred in a synchronous stream and they
6623                  * will need to be pulled up.
6624                  *
6625                  * After we have dequeued a message, we might find that
6626                  * it was a deferred M_SIG that was enqueued at the
6627                  * stream head.  It must now be posted as part of the
6628                  * read by calling strsignal_nolock().
6629                  *
6630                  * Also note that strrput does not enqueue an M_PCSIG,
6631                  * and there cannot be more than one hipri message,
6632                  * so there was no need to have the M_PCSIG case.
6633                  *
6634                  * At some time it might be nice to try and wrap the
6635                  * functionality of kstrgetmsg() and strgetmsg() into
6636                  * a common routine so to reduce the amount of replicated
6637                  * code (since they are extremely similar).
6638                  */
6639                 if (!(*flagsp & (MSG_HIPRI|MSG_BAND))) {
6640                         /* Asking for normal, band0 data */
6641                         bp = strget(stp, q, uiop, first, &error);
6642                         ASSERT(MUTEX_HELD(&stp->sd_lock));
6643                         if (bp != NULL) {
6644                                 if (DB_TYPE(bp) == M_SIG) {
6645                                         strsignal_nolock(stp, *bp->b_rptr,
6646                                             bp->b_band);
6647                                         freemsg(bp);
6648                                         continue;
6649                                 } else {
6650                                         break;
6651                                 }
6652                         }
6653                         if (error != 0)
6654                                 goto getmout;

6656                 /*
6657                  * We can't depend on the value of STRPRI here because
6658                  * the stream head may be in transit. Therefore, we
6659                  * must look at the type of the first message to
6660                  * determine if a high priority messages is waiting
6661                  */
```



```
6662                 } else if ((*flagsp & MSG_HIPRI) && q_first != NULL &&
6663                     DB_TYPE(q_first) >= QPCTL &&
6664                     (bp = getq_noenab(q, 0)) != NULL) {
6665                         /* Asked for HIPRI and got one */
6666                         ASSERT(DB_TYPE(bp) >= QPCTL);
6667                         break;
6668                 } else if ((*flagsp & MSG_BAND) && q_first != NULL &&
6669                     ((q_first->b_band >= *prip) || DB_TYPE(q_first) >= QPCTL) &&
6670                     (bp = getq_noenab(q, 0)) != NULL) {
6671                         /*
6672                          * Asked for at least band "prip" and got either at
6673                          * least that band or a hipri message.
6674                          */
6675                         ASSERT(bp->b_band >= *prip || DB_TYPE(bp) >= QPCTL);
6676                         if (DB_TYPE(bp) == M_SIG) {
6677                                 strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
6678                                 freemsg(bp);
6679                                 continue;
6680                         } else {
6681                                 break;
6682                         }
6683                 }

6685                 /* No data. Time to sleep? */
6686                 qbackenable(q, 0);

6688                 /*
6689                  * If STRHUP or STREOF, return 0 length control and data.
6690                  * If resid is 0, then a read(fd,buf,0) was done. Do not
6691                  * sleep to satisfy this request because by default we have
6692                  * zero bytes to return.
6693                  */
6694                 if ((stp->sd_flag & (STRHUP|STREOF)) || (mctl->maxlen == 0 &&
6695                     mdata->maxlen == 0)) {
6696                         mctl->len = mdata->len = 0;
6697                         *flagsp = 0;
6698                         mutex_exit(&stp->sd_lock);
6699                         return (0);
6700                 }
6701                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_WAIT,
6702                     "strgetmsg calls strwaitq:%p, %p",
6703                     vp, uiop);
6704                 if (((error = strwaitq(stp, GETWAIT, (ssize_t)0, fmode, -1,
6705                     &done)) != 0) || done) {
6706                         TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_DONE,
6707                             "strgetmsg error or done:%p, %p",
6708                             vp, uiop);
6709                         mutex_exit(&stp->sd_lock);
6710                         return (error);
6711                 }
6712                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_AWAKE,
6713                     "strgetmsg awakes:%p, %p", vp, uiop);
6714                 if ((error = i_straccess(stp, JCREAD)) != 0) {
6715                         mutex_exit(&stp->sd_lock);
6716                         return (error);
6717                 }
6718                 first = 0;
6719         }
6720         ASSERT(bp != NULL);
6721         /*
6722          * Extract any mark information. If the message is not completely
6723          * consumed this information will be put in the mblk
6724          * that is putback.
6725          * If MSGMARKNEXT is set and the message is completely consumed
6726          * the STRATMARK flag will be set below. Likewise, if
6727          * MSGNOTMARKNEXT is set and the message is
```

```
6728          * completely consumed STRNOTATMARK will be set.
6729          */
6730         mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
6731         ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
6732             (MSGMARKNEXT|MSGNOTMARKNEXT));
6733         if (mark != 0 && bp == stp->sd_mark) {
6734                 mark |= _LASTMARK;
6735                 stp->sd_mark = NULL;
6736         }
6737         /*
6738          * keep track of the original message type and priority
6739          */
6740         pri = bp->b_band;
6741         type = bp->b_datap->db_type;
6742         if (type == M_PASSFP) {
6743                 if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
6744                         stp->sd_mark = bp;
6745                 bp->b_flag |= mark & ~_LASTMARK;
6746                 putback(stp, q, bp, pri);
6747                 qbackenable(q, pri);
6748                 mutex_exit(&stp->sd_lock);
6749                 return (EBADMSG);
6750         }
6751         ASSERT(type != M_SIG);

6753         /*
6754          * Set this flag so strrput will not generate signals. Need to
6755          * make sure this flag is cleared before leaving this routine
6756          * else signals will stop being sent.
6757          */
6758         stp->sd_flag |= STRGETINPROG;
6759         mutex_exit(&stp->sd_lock);

6761         if (STREAM_NEEDSERVICE(stp))
6762                 stream_runservice(stp);

6764         /*
6765          * Set HIPRI flag if message is priority.
6766          */
6767         if (type >= QPCTL)
6768                 flg = MSG_HIPRI;
6769         else
6770                 flg = MSG_BAND;

6772         /*
6773          * First process PROTO or PCPROTO blocks, if any.
6774          */
6775         if (mctl->maxlen >= 0 && type != M_DATA) {
6776                 size_t  n, bcnt;
6777                 char    *ubuf;

6779                 bcnt = mctl->maxlen;
6780                 ubuf = mctl->buf;
6781                 while (bp != NULL && bp->b_datap->db_type != M_DATA) {
6782                         if ((n = MIN(bcnt, bp->b_wptr - bp->b_rptr)) != 0 &&
6783                             copyout(bp->b_rptr, ubuf, n)) {
6784                                 error = EFAULT;
6785                                 mutex_enter(&stp->sd_lock);
6786                                 /*
6787                                  * clear stream head pri flag based on
6788                                  * first message type
6789                                  */
6790                                 if (type >= QPCTL) {
6791                                         ASSERT(type == M_PCPROTO);
6792                                         stp->sd_flag &= ~STRPRI;
6793                                 }
```



```
6794                                 more = 0;
6795                                 freemsg(bp);
6796                                 goto getmout;
6797                         }
6798                         ubuf += n;
6799                         bp->b_rptr += n;
6800                         if (bp->b_rptr >= bp->b_wptr) {
6801                                 nbp = bp;
6802                                 bp = bp->b_cont;
6803                                 freeb(nbp);
6804                         }
6805                         ASSERT(n <= bcnt);
6806                         bcnt -= n;
6807                         if (bcnt == 0)
6808                                 break;
6809                 }
6810                 mctl->len = mctl->maxlen - bcnt;
6811         } else
6812                 mctl->len = -1;

6814         if (bp && bp->b_datap->db_type != M_DATA) {
6815                 /*
6816                  * More PROTO blocks in msg.
6817                  */
6818                 more |= MORECTL;
6819                 savemp = bp;
6820                 while (bp && bp->b_datap->db_type != M_DATA) {
6821                         savemptail = bp;
6822                         bp = bp->b_cont;
6823                 }
6824                 savemptail->b_cont = NULL;
6825         }

6827         /*
6828          * Now process DATA blocks, if any.
6829          */
6830         if (mdata->maxlen >= 0 && bp) {
6831                 /*
6832                  * struiocopyout will consume a potential zero-length
6833                  * M_DATA even if uio_resid is zero.
6834                  */
6835                 size_t oldresid = uiop->uio_resid;

6837                 bp = struiocopyout(bp, uiop, &error);
6838                 if (error != 0) {
6839                         mutex_enter(&stp->sd_lock);
6840                         /*
6841                          * clear stream head hi pri flag based on
6842                          * first message
6843                          */
6844                         if (type >= QPCTL) {
6845                                 ASSERT(type == M_PCPROTO);
6846                                 stp->sd_flag &= ~STRPRI;
6847                         }
6848                         more = 0;
6849                         freemsg(savemp);
6850                         goto getmout;
6851                 }
6852                 /*
6853                  * (pr == 1) indicates a partial read.
6854                  */
6855                 if (oldresid > uiop->uio_resid)
6856                         pr = 1;
6857                 mdata->len = mdata->maxlen - uiop->uio_resid;
6858         } else
6859                 mdata->len = -1;
```

```
6861         if (bp) {                       /* more data blocks in msg */
6862                 more |= MOREDATA;
6863                 if (savemp)
6864                         savemptail->b_cont = bp;
6865                 else
6866                         savemp = bp;
6867         }
6868
6869         mutex_enter(&stp->sd_lock);
6870         if (savemp) {
6871                 if (pr && (savemp->b_datap->db_type == M_DATA) &&
6872                     msgnodata(savemp)) {
6873                         /*
6874                          * Avoid queuing a zero-length tail part of
6875                          * a message. pr=1 indicates that we read some of
6876                          * the message.
6877                          */
6878                         freemsg(savemp);
6879                         more &= ~MOREDATA;
6880                         /*
6881                          * clear stream head hi pri flag based on
6882                          * first message
6883                          */
6884                         if (type >= QPCTL) {
6885                                 ASSERT(type == M_PCPROTO);
6886                                 stp->sd_flag &= ~STRPRI;
6887                         }
6888                 } else {
6889                         savemp->b_band = pri;
6890                         /*
6891                          * If the first message was HIPRI and the one we're
6892                          * putting back isn't, then clear STRPRI, otherwise
6893                          * set STRPRI again.  Note that we must set STRPRI
6894                          * again since the flush logic in strrput_nondata()
6895                          * may have cleared it while we had sd_lock dropped.
6896                          */
6897                         if (type >= QPCTL) {
6898                                 ASSERT(type == M_PCPROTO);
6899                                 if (queclass(savemp) < QPCTL)
6900                                         stp->sd_flag &= ~STRPRI;
6901                                 else
6902                                         stp->sd_flag |= STRPRI;
6903                         } else if (queclass(savemp) >= QPCTL) {
6904                                 /*
6905                                  * The first message was not a HIPRI message,
6906                                  * but the one we are about to putback is.
6907                                  * For simplicitly, we do not allow for HIPRI
6908                                  * messages to be embedded in the message
6909                                  * body, so just force it to same type as
6910                                  * first message.
6911                                  */
6912                                 ASSERT(type == M_DATA || type == M_PROTO);
6913                                 ASSERT(savemp->b_datap->db_type == M_PCPROTO);
6914                                 savemp->b_datap->db_type = type;
6915                         }
6916                         if (mark != 0) {
6917                                 savemp->b_flag |= mark & ~_LASTMARK;
6918                                 if ((mark & _LASTMARK) &&
6919                                     (stp->sd_mark == NULL)) {
6920                                         /*
6921                                          * If another marked message arrived
6922                                          * while sd_lock was not held sd_mark
6923                                          * would be non-NULL.
6924                                          */
6925                                         stp->sd_mark = savemp;
```



```
6926                                 }
6927                         }
6928                         putback(stp, q, savemp, pri);
6929                 }
6930         } else {
6931                 /*
6932                  * The complete message was consumed.
6933                  *
6934                  * If another M_PCPROTO arrived while sd_lock was not held
6935                  * it would have been discarded since STRPRI was still set.
6936                  *
6937                  * Move the MSG*MARKNEXT information
6938                  * to the stream head just in case
6939                  * the read queue becomes empty.
6940                  * clear stream head hi pri flag based on
6941                  * first message
6942                  *
6943                  * If the stream head was at the mark
6944                  * (STRATMARK) before we dropped sd_lock above
6945                  * and some data was consumed then we have
6946                  * moved past the mark thus STRATMARK is
6947                  * cleared. However, if a message arrived in
6948                  * strrput during the copyout above causing
6949                  * STRATMARK to be set we can not clear that
6950                  * flag.
6951                  */
6952                 if (type >= QPCTL) {
6953                         ASSERT(type == M_PCPROTO);
6954                         stp->sd_flag &= ~STRPRI;
6955                 }
6956                 if (mark & (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
6957                         if (mark & MSGMARKNEXT) {
6958                                 stp->sd_flag &= ~STRNOTATMARK;
6959                                 stp->sd_flag |= STRATMARK;
6960                         } else if (mark & MSGNOTMARKNEXT) {
6961                                 stp->sd_flag &= ~STRATMARK;
6962                                 stp->sd_flag |= STRNOTATMARK;
6963                         } else {
6964                                 stp->sd_flag &= ~(STRATMARK|STRNOTATMARK);
6965                         }
6966                 } else if (pr && (old_sd_flag & STRATMARK)) {
6967                         stp->sd_flag &= ~STRATMARK;
6968                 }
6969         }
6970
6971         *flagsp = flg;
6972         *prip = pri;
6973
6974         /*
6975          * Getmsg cleanup processing - if the state of the queue has changed
6976          * some signals may need to be sent and/or poll awakened.
6977          */
6978 getmout:
6979         qbackenable(q, pri);
6980
6981         /*
6982          * We dropped the stream head lock above. Send all M_SIG messages
6983          * before processing stream head for SIGPOLL messages.
6984          */
6985         ASSERT(MUTEX_HELD(&stp->sd_lock));
6986         while ((bp = q->q_first) != NULL &&
6987             (bp->b_datap->db_type == M_SIG)) {
6988                 /*
6989                  * sd_lock is held so the content of the read queue can not
6990                  * change.
6991                  */
```

```
6992                bp = getq(q);
6993                ASSERT(bp != NULL && bp->b_datap->db_type == M_SIG);

6995                strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
6996                mutex_exit(&stp->sd_lock);
6997                freemsg(bp);
6998                if (STREAM_NEEDSERVICE(stp))
6999                        stream_runservice(stp);
7000                mutex_enter(&stp->sd_lock);
7001        }

7003        /*
7004         * stream head cannot change while we make the determination
7005         * whether or not to send a signal. Drop the flag to allow strrput
7006         * to send firstmsgsigs again.
7007         */
7008        stp->sd_flag &= ~STRGETINPROG;

7010        /*
7011         * If the type of message at the front of the queue changed
7012         * due to the receive the appropriate signals and pollwakeup events
7013         * are generated. The type of changes are:
7014         *      Processed a hipri message, q_first is not hipri.
7015         *      Processed a band X message, and q_first is band Y.
7016         * The generated signals and pollwakeups are identical to what
7017         * strrput() generates should the message that is now on q_first
7018         * arrive to an empty read queue.
7019         *
7020         * Note: only strrput will send a signal for a hipri message.
7021         */
7022        if ((bp = q->q_first) != NULL && !(stp->sd_flag & STRPRI)) {
7023                strsigset_t signals = 0;
7024                strpollset_t pollwakeups = 0;

7026                if (flg & MSG_HIPRI) {
7027                        /*
7028                         * Removed a hipri message. Regular data at
7029                         * the front of  the queue.
7030                         */
7031                        if (bp->b_band == 0) {
7032                                signals = S_INPUT | S_RDNORM;
7033                                pollwakeups = POLLIN | POLLRDNORM;
7034                        } else {
7035                                signals = S_INPUT | S_RDBAND;
7036                                pollwakeups = POLLIN | POLLRDBAND;
7037                        }
7038                } else if (pri != bp->b_band) {
7039                        /*
7040                         * The band is different for the new q_first.
7041                         */
7042                        if (bp->b_band == 0) {
7043                                signals = S_RDNORM;
7044                                pollwakeups = POLLIN | POLLRDNORM;
7045                        } else {
7046                                signals = S_RDBAND;
7047                                pollwakeups = POLLIN | POLLRDBAND;
7048                        }
7049                }

7051                if (pollwakeups != 0) {
7052                        if (pollwakeups == (POLLIN | POLLRDNORM)) {
7053                                if (!(stp->sd_rput_opt & SR_POLLIN))
7054                                        goto no_pollwake;
7055                                stp->sd_rput_opt &= ~SR_POLLIN;
7056                        }
7057                        mutex_exit(&stp->sd_lock);
```



```
7058                        pollwakeup(&stp->sd_pollist, pollwakeups);
7059                        mutex_enter(&stp->sd_lock);
7060                }
7061 no_pollwake:

7063                if (stp->sd_sigflags & signals)
7064                        strsendsig(stp->sd_siglist, signals, bp->b_band, 0);
7065        }
7066        mutex_exit(&stp->sd_lock);

7068        rvp->r_val1 = more;
7069        return (error);
7070 #undef  _LASTMARK
7071 }

7073 /*
7074  * Get the next message from the read queue.  If the message is
7075  * priority, STRPRI will have been set by strrput().  This flag
7076  * should be reset only when the entire message at the front of the
7077  * queue as been consumed.
7078  *
7079  * If uiop is NULL all data is returned in mctlp.
7080  * Note that a NULL uiop implies that FNDELAY and FNONBLOCK are assumed
7081  * not enabled.
7082  * The timeout parameter is in milliseconds; -1 for infinity.
7083  * This routine handles the consolidation private flags:
7084  *      MSG_IGNERROR    Ignore any stream head error except STPLEX.
7085  *      MSG_DELAYERROR  Defer the error check until the queue is empty.
7086  *      MSG_HOLDSIG     Hold signals while waiting for data.
7087  *      MSG_IPEEK       Only peek at messages.
7088  *      MSG_DISCARDTAIL Discard the tail M_DATA part of the message
7089  *                      that doesn't fit.
7090  *      MSG_NOMARK      If the message is marked leave it on the queue.
7091  *
7092  * NOTE: strgetmsg and kstrgetmsg have much of the logic in common.
7093  */
7094 int
7095 kstrgetmsg(
7096        struct vnode *vp,
7097        mblk_t **mctlp,
7098        struct uio *uiop,
7099        unsigned char *prip,
7100        int *flagsp,
7101        clock_t timout,
7102        rval_t *rvp)
7103 {
7104        struct stdata *stp;
7105        mblk_t *bp, *nbp;
7106        mblk_t *savemp = NULL;
7107        mblk_t *savemptail = NULL;
7108        int flags;
7109        uint_t old_sd_flag;
7110        int flg;
7111        int more = 0;
7112        int error = 0;
7113        char first = 1;
7114        uint_t mark;            /* Contains MSG*MARK and _LASTMARK */
7115 #define _LASTMARK       0x8000  /* Distinct from MSG*MARK */
7116        unsigned char pri = 0;
7117        queue_t *q;
7118        int     pr = 0;                 /* Partial read successful */
7119        unsigned char type;

7121        TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_ENTER,
7122            "kstrgetmsg:%p", vp);
```

```
7124         ASSERT(vp->v_stream);
7125         stp = vp->v_stream;
7126         rvp->r_val1 = 0;

7128         mutex_enter(&stp->sd_lock);

7130         if ((error = i_straccess(stp, JCREAD)) != 0) {
7131                 mutex_exit(&stp->sd_lock);
7132                 return (error);
7133         }

7135         flags = *flagsp;
7136         if (stp->sd_flag & (STRDERR|STPLEX)) {
7137                 if ((stp->sd_flag & STPLEX) ||
7138                     (flags & (MSG_IGNERROR|MSG_DELAYERROR)) == 0) {
7139                         error = strgeterr(stp, STRDERR|STPLEX,
7140                             (flags & MSG_IPEEK));
7141                         if (error != 0) {
7142                                 mutex_exit(&stp->sd_lock);
7143                                 return (error);
7144                         }
7145                 }
7146         }
7147         mutex_exit(&stp->sd_lock);

7149         switch (flags & (MSG_HIPRI|MSG_ANY|MSG_BAND)) {
7150         case MSG_HIPRI:
7151                 if (*prip != 0)
7152                         return (EINVAL);
7153                 break;

7155         case MSG_ANY:
7156         case MSG_BAND:
7157                 break;

7159         default:
7160                 return (EINVAL);
7161         }

7163 retry:
7164         q = _RD(stp->sd_wrq);
7165         mutex_enter(&stp->sd_lock);
7166         old_sd_flag = stp->sd_flag;
7167         mark = 0;
7168         for (;;) {
7169                 int done = 0;
7170                 int waitflag;
7171                 int fmode;
7172                 mblk_t *q_first = q->q_first;

7174                 /*
7175                  * This section of the code operates just like the code
7176                  * in strgetmsg().  There is a comment there about what
7177                  * is going on here.
7178                  */
7179                 if (!(flags & (MSG_HIPRI|MSG_BAND))) {
7180                         /* Asking for normal, band0 data */
7181                         bp = strget(stp, q, uiop, first, &error);
7182                         ASSERT(MUTEX_HELD(&stp->sd_lock));
7183                         if (bp != NULL) {
7184                                 if (DB_TYPE(bp) == M_SIG) {
7185                                         strsignal_nolock(stp, *bp->b_rptr,
7186                                             bp->b_band);
7187                                         freemsg(bp);
7188                                         continue;
7189                                 } else {
```



```
7190                                         break;
7191                                 }
7192                         }
7193                         if (error != 0) {
7194                                 goto getmout;
7195                         }
7196                 /*
7197                  * We can't depend on the value of STRPRI here because
7198                  * the stream head may be in transit. Therefore, we
7199                  * must look at the type of the first message to
7200                  * determine if a high priority messages is waiting
7201                  */
7202                 } else if ((flags & MSG_HIPRI) && q_first != NULL &&
7203                     DB_TYPE(q_first) >= QPCTL &&
7204                     (bp = getq_noenab(q, 0)) != NULL) {
7205                         ASSERT(DB_TYPE(bp) >= QPCTL);
7206                         break;
7207                 } else if ((flags & MSG_BAND) && q_first != NULL &&
7208                     ((q_first->b_band >= *prip) || DB_TYPE(q_first) >= QPCTL) &&
7209                     (bp = getq_noenab(q, 0)) != NULL) {
7210                         /*
7211                          * Asked for at least band "prip" and got either at
7212                          * least that band or a hipri message.
7213                          */
7214                         ASSERT(bp->b_band >= *prip || DB_TYPE(bp) >= QPCTL);
7215                         if (DB_TYPE(bp) == M_SIG) {
7216                                 strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
7217                                 freemsg(bp);
7218                                 continue;
7219                         } else {
7220                                 break;
7221                         }
7222                 }

7224                 /* No data. Time to sleep? */
7225                 qbackenable(q, 0);

7227                 /*
7228                  * Delayed error notification?
7229                  */
7230                 if ((stp->sd_flag & (STRDERR|STPLEX)) &&
7231                     (flags & (MSG_IGNERROR|MSG_DELAYERROR)) == MSG_DELAYERROR) {
7232                         error = strgeterr(stp, STRDERR|STPLEX,
7233                             (flags & MSG_IPEEK));
7234                         if (error != 0) {
7235                                 mutex_exit(&stp->sd_lock);
7236                                 return (error);
7237                         }
7238                 }

7240                 /*
7241                  * If STRHUP or STREOF, return 0 length control and data.
7242                  * If a read(fd,buf,0) has been done, do not sleep, just
7243                  * return.
7244                  *
7245                  * If mctlp == NULL and uiop == NULL, then the code will
7246                  * do the strwaitq. This is an understood way of saying
7247                  * sleep "polling" until a message is received.
7248                  */
7249                 if ((stp->sd_flag & (STRHUP|STREOF)) ||
7250                     (uiop != NULL && uiop->uio_resid == 0)) {
7251                         if (mctlp != NULL)
7252                                 *mctlp = NULL;
7253                         *flagsp = 0;
7254                         mutex_exit(&stp->sd_lock);
7255                         return (0);
```

```
7256                }

7258                waitflag = GETWAIT;
7259                if (flags &
7260                    (MSG_HOLDSIG|MSG_IGNERROR|MSG_IPEEK|MSG_DELAYERROR)) {
7261                        if (flags & MSG_HOLDSIG)
7262                                waitflag |= STR_NOSIG;
7263                        if (flags & MSG_IGNERROR)
7264                                waitflag |= STR_NOERROR;
7265                        if (flags & MSG_IPEEK)
7266                                waitflag |= STR_PEEK;
7267                        if (flags & MSG_DELAYERROR)
7268                                waitflag |= STR_DELAYERR;
7269                }
7270                if (uiop != NULL)
7271                        fmode = uiop->uio_fmode;
7272                else
7273                        fmode = 0;

7275                TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_WAIT,
7276                    "kstrgetmsg calls strwaitq:%p, %p",
7277                    vp, uiop);
7278                if (((error = strwaitq(stp, waitflag, (ssize_t)0,
7279                    fmode, timout, &done))) != 0 || done) {
7280                        TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_DONE,
7281                            "kstrgetmsg error or done:%p, %p",
7282                            vp, uiop);
7283                        mutex_exit(&stp->sd_lock);
7284                        return (error);
7285                }
7286                TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_AWAKE,
7287                    "kstrgetmsg awakes:%p, %p", vp, uiop);
7288                if ((error = i_straccess(stp, JCREAD)) != 0) {
7289                        mutex_exit(&stp->sd_lock);
7290                        return (error);
7291                }
7292                first = 0;
7293        }
7294        ASSERT(bp != NULL);
7295        /*
7296         * Extract any mark information. If the message is not completely
7297         * consumed this information will be put in the mblk
7298         * that is putback.
7299         * If MSGMARKNEXT is set and the message is completely consumed
7300         * the STRATMARK flag will be set below. Likewise, if
7301         * MSGNOTMARKNEXT is set and the message is
7302         * completely consumed STRNOTATMARK will be set.
7303         */
7304        mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
7305        ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
7306            (MSGMARKNEXT|MSGNOTMARKNEXT));
7307        pri = bp->b_band;
7308        if (mark != 0) {
7309                /*
7310                 * If the caller doesn't want the mark return.
7311                 * Used to implement MSG_WAITALL in sockets.
7312                 */
7313                if (flags & MSG_NOMARK) {
7314                        putback(stp, q, bp, pri);
7315                        qbackenable(q, pri);
7316                        mutex_exit(&stp->sd_lock);
7317                        return (EWOULDBLOCK);
7318                }
7319                if (bp == stp->sd_mark) {
7320                        mark |= _LASTMARK;
7321                        stp->sd_mark = NULL;
```



```
7322                }
7323        }

7325        /*
7326         * keep track of the first message type
7327         */
7328        type = bp->b_datap->db_type;

7330        if (bp->b_datap->db_type == M_PASSFP) {
7331                if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7332                        stp->sd_mark = bp;
7333                bp->b_flag |= mark & ~_LASTMARK;
7334                putback(stp, q, bp, pri);
7335                qbackenable(q, pri);
7336                mutex_exit(&stp->sd_lock);
7337                return (EBADMSG);
7338        }
7339        ASSERT(type != M_SIG);

7341        if (flags & MSG_IPEEK) {
7342                /*
7343                 * Clear any struioflag - we do the uiomove over again
7344                 * when peeking since it simplifies the code.
7345                 *
7346                 * Dup the message and put the original back on the queue.
7347                 * If dupmsg() fails, try again with copymsg() to see if
7348                 * there is indeed a shortage of memory.  dupmsg() may fail
7349                 * if db_ref in any of the messages reaches its limit.
7350                 */

7352                if ((nbp = dupmsg(bp)) == NULL && (nbp = copymsg(bp)) == NULL) {
7353                        /*
7354                         * Restore the state of the stream head since we
7355                         * need to drop sd_lock (strwaitbuf is sleeping).
7356                         */
7357                        size_t size = msgdsize(bp);

7359                        if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7360                                stp->sd_mark = bp;
7361                        bp->b_flag |= mark & ~_LASTMARK;
7362                        putback(stp, q, bp, pri);
7363                        mutex_exit(&stp->sd_lock);
7364                        error = strwaitbuf(size, BPRI_HI);
7365                        if (error) {
7366                                /*
7367                                 * There is no net change to the queue thus
7368                                 * no need to qbackenable.
7369                                 */
7370                                return (error);
7371                        }
7372                        goto retry;
7373                }

7375                if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7376                        stp->sd_mark = bp;
7377                bp->b_flag |= mark & ~_LASTMARK;
7378                putback(stp, q, bp, pri);
7379                bp = nbp;
7380        }

7382        /*
7383         * Set this flag so strrput will not generate signals. Need to
7384         * make sure this flag is cleared before leaving this routine
7385         * else signals will stop being sent.
7386         */
7387        stp->sd_flag |= STRGETINPROG;
```

```
7388         mutex_exit(&stp->sd_lock);
7389
7390         if ((stp->sd_rputdatafunc != NULL) && (DB_TYPE(bp) == M_DATA)) {
7391                 mblk_t *tmp, *prevmp;
7392
7393                 /*
7394                  * Put first non-data mblk back to stream head and
7395                  * cut the mblk chain so sd_rputdatafunc only sees
7396                  * M_DATA mblks. We can skip the first mblk since it
7397                  * is M_DATA according to the condition above.
7398                  */
7399                 for (prevmp = bp, tmp = bp->b_cont; tmp != NULL;
7400                     prevmp = tmp, tmp = tmp->b_cont) {
7401                         if (DB_TYPE(tmp) != M_DATA) {
7402                                 prevmp->b_cont = NULL;
7403                                 mutex_enter(&stp->sd_lock);
7404                                 putback(stp, q, tmp, tmp->b_band);
7405                                 mutex_exit(&stp->sd_lock);
7406                                 break;
7407                         }
7408                 }
7409
7410                 bp = (stp->sd_rputdatafunc)(stp->sd_vnode, bp,
7411                     NULL, NULL, NULL, NULL);
7412
7413                 if (bp == NULL)
7414                         goto retry;
7415         }
7416
7417         if (STREAM_NEEDSERVICE(stp))
7418                 stream_runservice(stp);
7419
7420         /*
7421          * Set HIPRI flag if message is priority.
7422          */
7423         if (type >= QPCTL)
7424                 flg = MSG_HIPRI;
7425         else
7426                 flg = MSG_BAND;
7427
7428         /*
7429          * First process PROTO or PCPROTO blocks, if any.
7430          */
7431         if (mctlp != NULL && type != M_DATA) {
7432                 mblk_t *nbp;
7433
7434                 *mctlp = bp;
7435                 while (bp->b_cont && bp->b_cont->b_datap->db_type != M_DATA)
7436                         bp = bp->b_cont;
7437                 nbp = bp->b_cont;
7438                 bp->b_cont = NULL;
7439                 bp = nbp;
7440         }
7441
7442         if (bp && bp->b_datap->db_type != M_DATA) {
7443                 /*
7444                  * More PROTO blocks in msg. Will only happen if mctlp is NULL.
7445                  */
7446                 more |= MORECTL;
7447                 savemp = bp;
7448                 while (bp && bp->b_datap->db_type != M_DATA) {
7449                         savemptail = bp;
7450                         bp = bp->b_cont;
7451                 }
7452                 savemptail->b_cont = NULL;
7453         }
```



```
7454
7455         /*
7456          * Now process DATA blocks, if any.
7457          */
7458         if (uiop == NULL) {
7459                 /* Append data to tail of mctlp */
7460
7461                 if (mctlp != NULL) {
7462                         mblk_t **mpp = mctlp;
7463
7464                         while (*mpp != NULL)
7465                                 mpp = &((*mpp)->b_cont);
7466                         *mpp = bp;
7467                         bp = NULL;
7468                 }
7469         } else if (uiop->uio_resid >= 0 && bp) {
7470                 size_t oldresid = uiop->uio_resid;
7471
7472                 /*
7473                  * If a streams message is likely to consist
7474                  * of many small mblks, it is pulled up into
7475                  * one continuous chunk of memory.
7476                  * The size of the first mblk may be bogus because
7477                  * successive read() calls on the socket reduce
7478                  * the size of this mblk until it is exhausted
7479                  * and then the code walks on to the next. Thus
7480                  * the size of the mblk may not be the original size
7481                  * that was passed up, it's simply a remainder
7482                  * and hence can be very small without any
7483                  * implication that the packet is badly fragmented.
7484                  * So the size of the possible second mblk is
7485                  * used to spot a badly fragmented packet.
7486                  * see longer comment at top of page
7487                  * by mblk_pull_len declaration.
7488                  */
7489
7490                 if (bp->b_cont != NULL && MBLKL(bp->b_cont) < mblk_pull_len) {
7491                         (void) pullupmsg(bp, -1);
7492                 }
7493
7494                 bp = struiocopyout(bp, uiop, &error);
7495                 if (error != 0) {
7496                         if (mctlp != NULL) {
7497                                 freemsg(*mctlp);
7498                                 *mctlp = NULL;
7499                         } else
7500                                 freemsg(savemp);
7501                         mutex_enter(&stp->sd_lock);
7502                         /*
7503                          * clear stream head hi pri flag based on
7504                          * first message
7505                          */
7506                         if (!(flags & MSG_IPEEK) && (type >= QPCTL)) {
7507                                 ASSERT(type == M_PCPROTO);
7508                                 stp->sd_flag &= ~STRPRI;
7509                         }
7510                         more = 0;
7511                         goto getmout;
7512                 }
7513                 /*
7514                  * (pr == 1) indicates a partial read.
7515                  */
7516                 if (oldresid > uiop->uio_resid)
7517                         pr = 1;
7518         }
```

```
7520         if (bp) {                       /* more data blocks in msg */
7521                 more |= MOREDATA;
7522                 if (savemp)
7523                         savemptail->b_cont = bp;
7524                 else
7525                         savemp = bp;
7526         }

7528         mutex_enter(&stp->sd_lock);
7529         if (savemp) {
7530                 if (flags & (MSG_IPEEK|MSG_DISCARDTAIL)) {
7531                         /*
7532                          * When MSG_DISCARDTAIL is set or
7533                          * when peeking discard any tail. When peeking this
7534                          * is the tail of the dup that was copied out - the
7535                          * message has already been putback on the queue.
7536                          * Return MOREDATA to the caller even though the data
7537                          * is discarded. This is used by sockets (to
7538                          * set MSG_TRUNC).
7539                          */
7540                         freemsg(savemp);
7541                         if (!(flags & MSG_IPEEK) && (type >= QPCTL)) {
7542                                 ASSERT(type == M_PCPROTO);
7543                                 stp->sd_flag &= ~STRPRI;
7544                         }
7545                 } else if (pr && (savemp->b_datap->db_type == M_DATA) &&
7546                     msgnodata(savemp)) {
7547                         /*
7548                          * Avoid queuing a zero-length tail part of
7549                          * a message. pr=1 indicates that we read some of
7550                          * the message.
7551                          */
7552                         freemsg(savemp);
7553                         more &= ~MOREDATA;
7554                         if (type >= QPCTL) {
7555                                 ASSERT(type == M_PCPROTO);
7556                                 stp->sd_flag &= ~STRPRI;
7557                         }
7558                 } else {
7559                         savemp->b_band = pri;
7560                         /*
7561                          * If the first message was HIPRI and the one we're
7562                          * putting back isn't, then clear STRPRI, otherwise
7563                          * set STRPRI again.  Note that we must set STRPRI
7564                          * again since the flush logic in strrput_nondata()
7565                          * may have cleared it while we had sd_lock dropped.
7566                          */

7568                         if (type >= QPCTL) {
7569                                 ASSERT(type == M_PCPROTO);
7570                                 if (queclass(savemp) < QPCTL)
7571                                         stp->sd_flag &= ~STRPRI;
7572                                 else
7573                                         stp->sd_flag |= STRPRI;
7574                         } else if (queclass(savemp) >= QPCTL) {
7575                                 /*
7576                                  * The first message was not a HIPRI message,
7577                                  * but the one we are about to putback is.
7578                                  * For simplicitly, we do not allow for HIPRI
7579                                  * messages to be embedded in the message
7580                                  * body, so just force it to same type as
7581                                  * first message.
7582                                  */
7583                                 ASSERT(type == M_DATA || type == M_PROTO);
7584                                 ASSERT(savemp->b_datap->db_type == M_PCPROTO);
7585                                 savemp->b_datap->db_type = type;
```



```
7586                         }
7587                         if (mark != 0) {
7588                                 if ((mark & _LASTMARK) &&
7589                                     (stp->sd_mark == NULL)) {
7590                                         /*
7591                                          * If another marked message arrived
7592                                          * while sd_lock was not held sd_mark
7593                                          * would be non-NULL.
7594                                          */
7595                                         stp->sd_mark = savemp;
7596                                 }
7597                                 savemp->b_flag |= mark & ~_LASTMARK;
7598                         }
7599                         putback(stp, q, savemp, pri);
7600                 }
7601         } else if (!(flags & MSG_IPEEK)) {
7602                 /*
7603                  * The complete message was consumed.
7604                  *
7605                  * If another M_PCPROTO arrived while sd_lock was not held
7606                  * it would have been discarded since STRPRI was still set.
7607                  *
7608                  * Move the MSG*MARKNEXT information
7609                  * to the stream head just in case
7610                  * the read queue becomes empty.
7611                  * clear stream head hi pri flag based on
7612                  * first message
7613                  *
7614                  * If the stream head was at the mark
7615                  * (STRATMARK) before we dropped sd_lock above
7616                  * and some data was consumed then we have
7617                  * moved past the mark thus STRATMARK is
7618                  * cleared. However, if a message arrived in
7619                  * strrput during the copyout above causing
7620                  * STRATMARK to be set we can not clear that
7621                  * flag.
7622                  * XXX A "perimeter" would help by single-threading strrput,
7623                  * strread, strgetmsg and kstrgetmsg.
7624                  */
7625                 if (type >= QPCTL) {
7626                         ASSERT(type == M_PCPROTO);
7627                         stp->sd_flag &= ~STRPRI;
7628                 }
7629                 if (mark & (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
7630                         if (mark & MSGMARKNEXT) {
7631                                 stp->sd_flag &= ~STRNOTATMARK;
7632                                 stp->sd_flag |= STRATMARK;
7633                         } else if (mark & MSGNOTMARKNEXT) {
7634                                 stp->sd_flag &= ~STRATMARK;
7635                                 stp->sd_flag |= STRNOTATMARK;
7636                         } else {
7637                                 stp->sd_flag &= ~(STRATMARK|STRNOTATMARK);
7638                         }
7639                 } else if (pr && (old_sd_flag & STRATMARK)) {
7640                         stp->sd_flag &= ~STRATMARK;
7641                 }
7642         }

7644         *flagsp = flg;
7645         *prip = pri;

7647         /*
7648          * Getmsg cleanup processing - if the state of the queue has changed
7649          * some signals may need to be sent and/or poll awakened.
7650          */
7651 getmout:
```

```
7652         qbackenable(q, pri);

7654         /*
7655          * We dropped the stream head lock above. Send all M_SIG messages
7656          * before processing stream head for SIGPOLL messages.
7657          */
7658         ASSERT(MUTEX_HELD(&stp->sd_lock));
7659         while ((bp = q->q_first) != NULL &&
7660             (bp->b_datap->db_type == M_SIG)) {
7661                 /*
7662                  * sd_lock is held so the content of the read queue can not
7663                  * change.
7664                  */
7665                 bp = getq(q);
7666                 ASSERT(bp != NULL && bp->b_datap->db_type == M_SIG);

7668                 strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
7669                 mutex_exit(&stp->sd_lock);
7670                 freemsg(bp);
7671                 if (STREAM_NEEDSERVICE(stp))
7672                         stream_runservice(stp);
7673                 mutex_enter(&stp->sd_lock);
7674         }

7676         /*
7677          * stream head cannot change while we make the determination
7678          * whether or not to send a signal. Drop the flag to allow strrput
7679          * to send firstmsgsigs again.
7680          */
7681         stp->sd_flag &= ~STRGETINPROG;

7683         /*
7684          * If the type of message at the front of the queue changed
7685          * due to the receive the appropriate signals and pollwakeup events
7686          * are generated. The type of changes are:
7687          *      Processed a hipri message, q_first is not hipri.
7688          *      Processed a band X message, and q_first is band Y.
7689          * The generated signals and pollwakeups are identical to what
7690          * strrput() generates should the message that is now on q_first
7691          * arrive to an empty read queue.
7692          *
7693          * Note: only strrput will send a signal for a hipri message.
7694          */
7695         if ((bp = q->q_first) != NULL && !(stp->sd_flag & STRPRI)) {
7696                 strsigset_t signals = 0;
7697                 strpollset_t pollwakeups = 0;

7699                 if (flg & MSG_HIPRI) {
7700                         /*
7701                          * Removed a hipri message. Regular data at
7702                          * the front of  the queue.
7703                          */
7704                         if (bp->b_band == 0) {
7705                                 signals = S_INPUT | S_RDNORM;
7706                                 pollwakeups = POLLIN | POLLRDNORM;
7707                         } else {
7708                                 signals = S_INPUT | S_RDBAND;
7709                                 pollwakeups = POLLIN | POLLRDBAND;
7710                         }
7711                 } else if (pri != bp->b_band) {
7712                         /*
7713                          * The band is different for the new q_first.
7714                          */
7715                         if (bp->b_band == 0) {
7716                                 signals = S_RDNORM;
7717                                 pollwakeups = POLLIN | POLLRDNORM;
```



```
7718                         } else {
7719                                 signals = S_RDBAND;
7720                                 pollwakeups = POLLIN | POLLRDBAND;
7721                         }
7722                 }

7724                 if (pollwakeups != 0) {
7725                         if (pollwakeups == (POLLIN | POLLRDNORM)) {
7726                                 if (!(stp->sd_rput_opt & SR_POLLIN))
7727                                         goto no_pollwake;
7728                                 stp->sd_rput_opt &= ~SR_POLLIN;
7729                         }
7730                         mutex_exit(&stp->sd_lock);
7731                         pollwakeup(&stp->sd_pollist, pollwakeups);
7732                         mutex_enter(&stp->sd_lock);
7733                 }
7734 no_pollwake:

7736                 if (stp->sd_sigflags & signals)
7737                         strsendsig(stp->sd_siglist, signals, bp->b_band, 0);
7738         }
7739         mutex_exit(&stp->sd_lock);

7741         rvp->r_val1 = more;
7742         return (error);
7743 #undef  _LASTMARK
7744 }

7746 /*
7747  * Put a message downstream.
7748  *
7749  * NOTE: strputmsg and kstrputmsg have much of the logic in common.
7750  */
7751 int
7752 strputmsg(
7753         struct vnode *vp,
7754         struct strbuf *mctl,
7755         struct strbuf *mdata,
7756         unsigned char pri,
7757         int flag,
7758         int fmode)
7759 {
7760         struct stdata *stp;
7761         queue_t *wqp;
7762         mblk_t *mp;
7763         ssize_t msgsize;
7764         ssize_t rmin, rmax;
7765         int error;
7766         struct uio uios;
7767         struct uio *uiop = &uios;
7768         struct iovec iovs;
7769         int xpg4 = 0;

7771         ASSERT(vp->v_stream);
7772         stp = vp->v_stream;
7773         wqp = stp->sd_wrq;

7775         /*
7776          * If it is an XPG4 application, we need to send
7777          * SIGPIPE below
7778          */

7780         xpg4 = (flag & MSG_XPG4) ? 1 : 0;
7781         flag &= ~MSG_XPG4;

7783         if (AU_AUDITING())
```

```
7784                audit_strputmsg(vp, mctl, mdata, pri, flag, fmode);

7786        mutex_enter(&stp->sd_lock);

7788        if ((error = i_straccess(stp, JCWRITE)) != 0) {
7789                mutex_exit(&stp->sd_lock);
7790                return (error);
7791        }

7793        if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
7794                error = strwriteable(stp, B_FALSE, xpg4);
7795                if (error != 0) {
7796                        mutex_exit(&stp->sd_lock);
7797                        return (error);
7798                }
7799        }

7801        mutex_exit(&stp->sd_lock);

7803        /*
7804         * Check for legal flag value.
7805         */
7806        switch (flag) {
7807        case MSG_HIPRI:
7808                if ((mctl->len < 0) || (pri != 0))
7809                        return (EINVAL);
7810                break;
7811        case MSG_BAND:
7812                break;

7814        default:
7815                return (EINVAL);
7816        }

7818        TRACE_1(TR_FAC_STREAMS_FR, TR_STRPUTMSG_IN,
7819            "strputmsg in:stp %p", stp);

7821        /* get these values from those cached in the stream head */
7822        rmin = stp->sd_qn_minpsz;
7823        rmax = stp->sd_qn_maxpsz;

7825        /*
7826         * Make sure ctl and data sizes together fall within the
7827         * limits of the max and min receive packet sizes and do
7828         * not exceed system limit.
7829         */
7830        ASSERT((rmax >= 0) || (rmax == INFPSZ));
7831        if (rmax == 0) {
7832                return (ERANGE);
7833        }
7834        /*
7835         * Use the MAXIMUM of sd_maxblk and q_maxpsz.
7836         * Needed to prevent partial failures in the strmakedata loop.
7837         */
7838        if (stp->sd_maxblk != INFPSZ && rmax != INFPSZ && rmax < stp->sd_maxblk)
7839                rmax = stp->sd_maxblk;

7841        if ((msgsize = mdata->len) < 0) {
7842                msgsize = 0;
7843                rmin = 0;       /* no range check for NULL data part */
7844        }
7845        if ((msgsize < rmin) ||
7846            ((msgsize > rmax) && (rmax != INFPSZ)) ||
7847            (mctl->len > strctlsz)) {
7848                return (ERANGE);
7849        }
```



```
7851        /*
7852         * Setup uio and iov for data part
7853         */
7854        iovs.iov_base = mdata->buf;
7855        iovs.iov_len = msgsize;
7856        uios.uio_iov = &iovs;
7857        uios.uio_iovcnt = 1;
7858        uios.uio_loffset = 0;
7859        uios.uio_segflg = UIO_USERSPACE;
7860        uios.uio_fmode = fmode;
7861        uios.uio_extflg = UIO_COPY_DEFAULT;
7862        uios.uio_resid = msgsize;
7863        uios.uio_offset = 0;

7865        /* Ignore flow control in strput for HIPRI */
7866        if (flag & MSG_HIPRI)
7867                flag |= MSG_IGNFLOW;

7869        for (;;) {
7870                int done = 0;

7872                /*
7873                 * strput will always free the ctl mblk - even when strput
7874                 * fails.
7875                 */
7876                if ((error = strmakectl(mctl, flag, fmode, &mp)) != 0) {
7877                        TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7878                            "strputmsg out:stp %p out %d error %d",
7879                            stp, 1, error);
7880                        return (error);
7881                }
7882                /*
7883                 * Verify that the whole message can be transferred by
7884                 * strput.
7885                 */
7886                ASSERT(stp->sd_maxblk == INFPSZ ||
7887                    stp->sd_maxblk >= mdata->len);

7889                msgsize = mdata->len;
7890                error = strput(stp, mp, uiop, &msgsize, 0, pri, flag);
7891                mdata->len = msgsize;

7893                if (error == 0)
7894                        break;

7896                if (error != EWOULDBLOCK)
7897                        goto out;

7899                mutex_enter(&stp->sd_lock);
7900                /*
7901                 * Check for a missed wakeup.
7902                 * Needed since strput did not hold sd_lock across
7903                 * the canputnext.
7904                 */
7905                if (bcanputnext(wqp, pri)) {
7906                        /* Try again */
7907                        mutex_exit(&stp->sd_lock);
7908                        continue;
7909                }
7910                TRACE_2(TR_FAC_STREAMS_FR, TR_STRPUTMSG_WAIT,
7911                    "strputmsg wait:stp %p waits pri %d", stp, pri);
7912                if (((error = strwaitq(stp, WRITEWAIT, (ssize_t)0, fmode, -1,
7913                    &done)) != 0) || done) {
7914                        mutex_exit(&stp->sd_lock);
7915                        TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
```

```
7916                            "strputmsg out:q %p out %d error %d",
7917                            stp, 0, error);
7918                    return (error);
7919            }
7920            TRACE_1(TR_FAC_STREAMS_FR, TR_STRPUTMSG_WAKE,
7921                "strputmsg wake:stp %p wakes", stp);
7922            if ((error = i_straccess(stp, JCWRITE)) != 0) {
7923                    mutex_exit(&stp->sd_lock);
7924                    return (error);
7925            }
7926            mutex_exit(&stp->sd_lock);
7927    }
7928 out:
7929    /*
7930     * For historic reasons, applications expect EAGAIN
7931     * when data mblk could not be allocated. so change
7932     * ENOMEM back to EAGAIN
7933     */
7934    if (error == ENOMEM)
7935            error = EAGAIN;
7936    TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7937        "strputmsg out:stp %p out %d error %d", stp, 2, error);
7938    return (error);
7939 }

7941 /*
7942  * Put a message downstream.
7943  * Can send only an M_PROTO/M_PCPROTO by passing in a NULL uiop.
7944  * The fmode flag (NDELAY, NONBLOCK) is the or of the flags in the uio
7945  * and the fmode parameter.
7946  *
7947  * This routine handles the consolidation private flags:
7948  *      MSG_IGNERROR    Ignore any stream head error except STPLEX.
7949  *      MSG_HOLDSIG     Hold signals while waiting for data.
7950  *      MSG_IGNFLOW     Don't check streams flow control.
7951  *
7952  * NOTE: strputmsg and kstrputmsg have much of the logic in common.
7953  */
7954 int
7955 kstrputmsg(
7956        struct vnode *vp,
7957        mblk_t *mctl,
7958        struct uio *uiop,
7959        ssize_t msgsize,
7960        unsigned char pri,
7961        int flag,
7962        int fmode)
7963 {
7964        struct stdata *stp;
7965        queue_t *wqp;
7966        ssize_t rmin, rmax;
7967        int error;

7969        ASSERT(vp->v_stream);
7970        stp = vp->v_stream;
7971        wqp = stp->sd_wrq;
7972        if (AU_AUDITING())
7973                audit_strputmsg(vp, NULL, NULL, pri, flag, fmode);
7974        if (mctl == NULL)
7975                return (EINVAL);

7977        mutex_enter(&stp->sd_lock);

7979        if ((error = i_straccess(stp, JCWRITE)) != 0) {
7980                mutex_exit(&stp->sd_lock);
7981                freemsg(mctl);
```



```
7982                return (error);
7983        }

7985        if ((stp->sd_flag & STPLEX) || !(flag & MSG_IGNERROR)) {
7986                if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
7987                        error = strwriteable(stp, B_FALSE, B_TRUE);
7988                        if (error != 0) {
7989                                mutex_exit(&stp->sd_lock);
7990                                freemsg(mctl);
7991                                return (error);
7992                        }
7993                }
7994        }

7996        mutex_exit(&stp->sd_lock);

7998        /*
7999         * Check for legal flag value.
8000         */
8001        switch (flag & (MSG_HIPRI|MSG_BAND|MSG_ANY)) {
8002        case MSG_HIPRI:
8003                if (pri != 0) {
8004                        freemsg(mctl);
8005                        return (EINVAL);
8006                }
8007                break;
8008        case MSG_BAND:
8009                break;
8010        default:
8011                freemsg(mctl);
8012                return (EINVAL);
8013        }

8015        TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_IN,
8016            "kstrputmsg in:stp %p", stp);

8018        /* get these values from those cached in the stream head */
8019        rmin = stp->sd_qn_minpsz;
8020        rmax = stp->sd_qn_maxpsz;

8022        /*
8023         * Make sure ctl and data sizes together fall within the
8024         * limits of the max and min receive packet sizes and do
8025         * not exceed system limit.
8026         */
8027        ASSERT((rmax >= 0) || (rmax == INFPSZ));
8028        if (rmax == 0) {
8029                freemsg(mctl);
8030                return (ERANGE);
8031        }
8032        /*
8033         * Use the MAXIMUM of sd_maxblk and q_maxpsz.
8034         * Needed to prevent partial failures in the strmakedata loop.
8035         */
8036        if (stp->sd_maxblk != INFPSZ && rmax != INFPSZ && rmax < stp->sd_maxblk)
8037                rmax = stp->sd_maxblk;

8039        if (uiop == NULL) {
8040                msgsize = -1;
8041                rmin = -1;      /* no range check for NULL data part */
8042        } else {
8043                /* Use uio flags as well as the fmode parameter flags */
8044                fmode |= uiop->uio_fmode;

8046                if ((msgsize < rmin) ||
8047                    ((msgsize > rmax) && (rmax != INFPSZ))) {
```

```
8048                        freemsg(mctl);
8049                        return (ERANGE);
8050                }
8051        }

8053        /* Ignore flow control in strput for HIPRI */
8054        if (flag & MSG_HIPRI)
8055                flag |= MSG_IGNFLOW;

8057        for (;;) {
8058                int done = 0;
8059                int waitflag;
8060                mblk_t *mp;

8062                /*
8063                 * strput will always free the ctl mblk - even when strput
8064                 * fails. If MSG_IGNFLOW is set then any error returned
8065                 * will cause us to break the loop, so we don't need a copy
8066                 * of the message. If MSG_IGNFLOW is not set, then we can
8067                 * get hit by flow control and be forced to try again. In
8068                 * this case we need to have a copy of the message. We
8069                 * do this using copymsg since the message may get modified
8070                 * by something below us.
8071                 *
8072                 * We've observed that many TPI providers do not check db_ref
8073                 * on the control messages but blindly reuse them for the
8074                 * T_OK_ACK/T_ERROR_ACK. Thus using copymsg is more
8075                 * friendly to such providers than using dupmsg. Also, note
8076                 * that sockfs uses MSG_IGNFLOW for all TPI control messages.
8077                 * Only data messages are subject to flow control, hence
8078                 * subject to this copymsg.
8079                 */
8080                if (flag & MSG_IGNFLOW) {
8081                        mp = mctl;
8082                        mctl = NULL;
8083                } else {
8084                        do {
8085                                /*
8086                                 * If a message has a free pointer, the message
8087                                 * must be dupmsg to maintain this pointer.
8088                                 * Code using this facility must be sure
8089                                 * that modules below will not change the
8090                                 * contents of the dblk without checking db_ref
8091                                 * first. If db_ref is > 1, then the module
8092                                 * needs to do a copymsg first. Otherwise,
8093                                 * the contents of the dblk may become
8094                                 * inconsistent because the freesmg/freeb below
8095                                 * may end up calling atomic_add_32_nv.
8096                                 * The atomic_add_32_nv in freeb (accessing
8097                                 * all of db_ref, db_type, db_flags, and
8098                                 * db_struioflag) does not prevent other threads
8099                                 * from concurrently trying to modify e.g.
8100                                 * db_type.
8101                                 */
8102                                if (mctl->b_datap->db_frtnp != NULL)
8103                                        mp = dupmsg(mctl);
8104                                else
8105                                        mp = copymsg(mctl);

8107                                if (mp != NULL)
8108                                        break;

8110                                error = strwaitbuf(msgdsize(mctl), BPRI_MED);
8111                                if (error) {
8112                                        freemsg(mctl);
8113                                        return (error);
```



```
8114                                }
8115                        } while (mp == NULL);
8116                }
8117                /*
8118                 * Verify that all of msgsize can be transferred by
8119                 * strput.
8120                 */
8121                ASSERT(stp->sd_maxblk == INFPSZ || stp->sd_maxblk >= msgsize);
8122                error = strput(stp, mp, uiop, &msgsize, 0, pri, flag);
8123                if (error == 0)
8124                        break;

8126                if (error != EWOULDBLOCK)
8127                        goto out;

8129                /*
8130                 * IF MSG_IGNFLOW is set we should have broken out of loop
8131                 * above.
8132                 */
8133                ASSERT(!(flag & MSG_IGNFLOW));
8134                mutex_enter(&stp->sd_lock);
8135                /*
8136                 * Check for a missed wakeup.
8137                 * Needed since strput did not hold sd_lock across
8138                 * the canputnext.
8139                 */
8140                if (bcanputnext(wqp, pri)) {
8141                        /* Try again */
8142                        mutex_exit(&stp->sd_lock);
8143                        continue;
8144                }
8145                TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_WAIT,
8146                    "kstrputmsg wait:stp %p waits pri %d", stp, pri);

8148                waitflag = WRITEWAIT;
8149                if (flag & (MSG_HOLDSIG|MSG_IGNERROR)) {
8150                        if (flag & MSG_HOLDSIG)
8151                                waitflag |= STR_NOSIG;
8152                        if (flag & MSG_IGNERROR)
8153                                waitflag |= STR_NOERROR;
8154                }
8155                if (((error = strwaitq(stp, waitflag,
8156                    (ssize_t)0, fmode, -1, &done)) != 0) || done) {
8157                        mutex_exit(&stp->sd_lock);
8158                        TRACE_3(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_OUT,
8159                            "kstrputmsg out:stp %p out %d error %d",
8160                            stp, 0, error);
8161                        freemsg(mctl);
8162                        return (error);
8163                }
8164                TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_WAKE,
8165                    "kstrputmsg wake:stp %p wakes", stp);
8166                if ((error = i_straccess(stp, JCWRITE)) != 0) {
8167                        mutex_exit(&stp->sd_lock);
8168                        freemsg(mctl);
8169                        return (error);
8170                }
8171                mutex_exit(&stp->sd_lock);
8172        }
8173 out:
8174        freemsg(mctl);
8175        /*
8176         * For historic reasons, applications expect EAGAIN
8177         * when data mblk could not be allocated. so change
8178         * ENOMEM back to EAGAIN
8179         */
```

```
8180         if (error == ENOMEM)
8181                 error = EAGAIN;
8182         TRACE_3(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_OUT,
8183             "kstrputmsg out:stp %p out %d error %d", stp, 2, error);
8184         return (error);
8185 }

8187 /*
8188  * Determines whether the necessary conditions are set on a stream
8189  * for it to be readable, writeable, or have exceptions.
8190  *
8191  * strpoll handles the consolidation private events:
8192  *      POLLNOERR       Do not return POLLERR even if there are stream
8193  *                      head errors.
8194  *                      Used by sockfs.
8195  *      POLLRDDATA      Do not return POLLIN unless at least one message on
8196  *                      the queue contains one or more M_DATA mblks. Thus
8197  *                      when this flag is set a queue with only
8198  *                      M_PROTO/M_PCPROTO mblks does not return POLLIN.
8199  *                      Used by sockfs to ignore T_EXDATA_IND messages.
8200  *
8201  * Note: POLLRDDATA assumes that synch streams only return messages with
8202  * an M_DATA attached (i.e. not messages consisting of only
8203  * an M_PROTO/M_PCPROTO part).
8204  */
8205 int
8206 strpoll(
8207         struct stdata *stp,
8208         short events_arg,
8209         int anyyet,
8210         short *reventsp,
8211         struct pollhead **phpp)
8212 {
8213         int events = (ushort_t)events_arg;
8214         int retevents = 0;
8215         mblk_t *mp;
8216         qband_t *qbp;
8217         long sd_flags = stp->sd_flag;
8218         int headlocked = 0;

8220         /*
8221          * For performance, a single 'if' tests for most possible edge
8222          * conditions in one shot
8223          */
8224         if (sd_flags & (STPLEX | STRDERR | STWRERR)) {
8225                 if (sd_flags & STPLEX) {
8226                         *reventsp = POLLNVAL;
8227                         return (EINVAL);
8228                 }
8229                 if (((events & (POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI)) &&
8230                     (sd_flags & STRDERR)) ||
8231                     ((events & (POLLOUT | POLLWRNORM | POLLWRBAND)) &&
8232                     (sd_flags & STWRERR))) {
8233                         if (!(events & POLLNOERR)) {
8234                                 *reventsp = POLLERR;
8235                                 return (0);
8236                         }
8237                 }
8238         }
8239         if (sd_flags & STRHUP) {
8240                 retevents |= POLLHUP;
8241         } else if (events & (POLLWRNORM | POLLWRBAND)) {
8242                 queue_t *tq;
8243                 queue_t *qp = stp->sd_wrq;

8245                 claimstr(qp);
```



```
8246                 /* Find next module forward that has a service procedure */
8247                 tq = qp->q_next->q_nfsrv;
8248                 ASSERT(tq != NULL);

8250                 polllock(&stp->sd_pollist, QLOCK(tq));
8251                 if (events & POLLWRNORM) {
8252                         queue_t *sqp;

8254                         if (tq->q_flag & QFULL)
8255                                 /* ensure backq svc procedure runs */
8256                                 tq->q_flag |= QWANTW;
8257                         else if ((sqp = stp->sd_struiowrq) != NULL) {
8258                                 /* Check sync stream barrier write q */
8259                                 mutex_exit(QLOCK(tq));
8260                                 polllock(&stp->sd_pollist, QLOCK(sqp));
8261                                 if (sqp->q_flag & QFULL)
8262                                         /* ensure pollwakeup() is done */
8263                                         sqp->q_flag |= QWANTWSYNC;
8264                                 else
8265                                         retevents |= POLLOUT;
8266                                 /* More write events to process ??? */
8267                                 if (! (events & POLLWRBAND)) {
8268                                         mutex_exit(QLOCK(sqp));
8269                                         releasestr(qp);
8270                                         goto chkrd;
8271                                 }
8272                                 mutex_exit(QLOCK(sqp));
8273                                 polllock(&stp->sd_pollist, QLOCK(tq));
8274                         } else
8275                                 retevents |= POLLOUT;
8276                 }
8277                 if (events & POLLWRBAND) {
8278                         qbp = tq->q_bandp;
8279                         if (qbp) {
8280                                 while (qbp) {
8281                                         if (qbp->qb_flag & QB_FULL)
8282                                                 qbp->qb_flag |= QB_WANTW;
8283                                         else
8284                                                 retevents |= POLLWRBAND;
8285                                         qbp = qbp->qb_next;
8286                                 }
8287                         } else {
8288                                 retevents |= POLLWRBAND;
8289                         }
8290                 }
8291                 mutex_exit(QLOCK(tq));
8292                 releasestr(qp);
8293         }
8294 chkrd:
8295         if (sd_flags & STRPRI) {
8296                 retevents |= (events & POLLPRI);
8297         } else if (events & (POLLRDNORM | POLLRDBAND | POLLIN)) {
8298                 queue_t *qp = _RD(stp->sd_wrq);
8299                 int normevents = (events & (POLLIN | POLLRDNORM));

8301                 /*
8302                  * Note: Need to do polllock() here since ps_lock may be
8303                  * held. See bug 4191544.
8304                  */
8305                 polllock(&stp->sd_pollist, &stp->sd_lock);
8306                 headlocked = 1;
8307                 mp = qp->q_first;
8308                 while (mp) {
8309                         /*
8310                          * For POLLRDDATA we scan b_cont and b_next until we
8311                          * find an M_DATA.
```

```
8312                         */
8313                        if ((events & POLLRDDATA) &&
8314                            mp->b_datap->db_type != M_DATA) {
8315                                mblk_t *nmp = mp->b_cont;

8317                                while (nmp != NULL &&
8318                                    nmp->b_datap->db_type != M_DATA)
8319                                        nmp = nmp->b_cont;
8320                                if (nmp == NULL) {
8321                                        mp = mp->b_next;
8322                                        continue;
8323                                }
8324                        }
8325                        if (mp->b_band == 0)
8326                                retevents |= normevents;
8327                        else
8328                                retevents |= (events & (POLLIN | POLLRDBAND));
8329                        break;
8330                }
8331                if (! (retevents & normevents) &&
8332                    (stp->sd_wakeq & RSLEEP)) {
8333                        /*
8334                         * Sync stream barrier read queue has data.
8335                         */
8336                        retevents |= normevents;
8337                }
8338                /* Treat eof as normal data */
8339                if (sd_flags & STREOF)
8340                        retevents |= normevents;
8341        }

8343        *reventsp = (short)retevents;
8344        if (retevents) {
8345                if (headlocked)
8346                        mutex_exit(&stp->sd_lock);
8347                return (0);
8348        }

8350        /*
8351         * If poll() has not found any events yet, set up event cell
8352         * to wake up the poll if a requested event occurs on this
8353         * stream.  Check for collisions with outstanding poll requests.
8354         */
8355        if (!anyyet) {
8356                *phpp = &stp->sd_pollist;
8357                if (headlocked == 0) {
8358                        polllock(&stp->sd_pollist, &stp->sd_lock);
8359                        headlocked = 1;
8360                }
8361                stp->sd_rput_opt |= SR_POLLIN;
8362        }
8363        if (headlocked)
8364                mutex_exit(&stp->sd_lock);
8365        return (0);
8366 }

8368 /*
8369  * The purpose of putback() is to assure sleeping polls/reads
8370  * are awakened when there are no new messages arriving at the,
8371  * stream head, and a message is placed back on the read queue.
8372  *
8373  * sd_lock must be held when messages are placed back on stream
8374  * head.  (getq() holds sd_lock when it removes messages from
8375  * the queue)
8376  */
```



```
8378 static void
8379 putback(struct stdata *stp, queue_t *q, mblk_t *bp, int band)
8380 {
8381        mblk_t  *qfirst;
8382        ASSERT(MUTEX_HELD(&stp->sd_lock));

8384        /*
8385         * As a result of lock-step ordering around q_lock and sd_lock,
8386         * it's possible for function calls like putnext() and
8387         * canputnext() to get an inaccurate picture of how much
8388         * data is really being processed at the stream head.
8389         * We only consolidate with existing messages on the queue
8390         * if the length of the message we want to put back is smaller
8391         * than the queue hiwater mark.
8392         */
8393        if ((stp->sd_rput_opt & SR_CONSOL_DATA) &&
8394            (DB_TYPE(bp) == M_DATA) && ((qfirst = q->q_first) != NULL) &&
8395            (DB_TYPE(qfirst) == M_DATA) &&
8396            ((qfirst->b_flag & (MSGMARK|MSGDELIM)) == 0) &&
8397            ((bp->b_flag & (MSGMARK|MSGDELIM|MSGMARKNEXT)) == 0) &&
8398            (mp_cont_len(bp, NULL) < q->q_hiwat)) {
8399                /*
8400                 * We use the same logic as defined in strrput()
8401                 * but in reverse as we are putting back onto the
8402                 * queue and want to retain byte ordering.
8403                 * Consolidate M_DATA messages with M_DATA ONLY.
8404                 * strrput() allows the consolidation of M_DATA onto
8405                 * M_PROTO | M_PCPROTO but not the other way round.
8406                 *
8407                 * The consolidation does not take place if the message
8408                 * we are returning to the queue is marked with either
8409                 * of the marks or the delim flag or if q_first
8410                 * is marked with MSGMARK. The MSGMARK check is needed to
8411                 * handle the odd semantics of MSGMARK where essentially
8412                 * the whole message is to be treated as marked.
8413                 * Carry any MSGMARKNEXT and MSGNOTMARKNEXT from q_first
8414                 * to the front of the b_cont chain.
8415                 */
8416                rmvq_noenab(q, qfirst);

8418                /*
8419                 * The first message in the b_cont list
8420                 * tracks MSGMARKNEXT and MSGNOTMARKNEXT.
8421                 * We need to handle the case where we
8422                 * are appending:
8423                 *
8424                 * 1) a MSGMARKNEXT to a MSGNOTMARKNEXT.
8425                 * 2) a MSGMARKNEXT to a plain message.
8426                 * 3) a MSGNOTMARKNEXT to a plain message
8427                 * 4) a MSGNOTMARKNEXT to a MSGNOTMARKNEXT
8428                 *    message.
8429                 *
8430                 * Thus we never append a MSGMARKNEXT or
8431                 * MSGNOTMARKNEXT to a MSGMARKNEXT message.
8432                 */
8433                if (qfirst->b_flag & MSGMARKNEXT) {
8434                        bp->b_flag |= MSGMARKNEXT;
8435                        bp->b_flag &= ~MSGNOTMARKNEXT;
8436                        qfirst->b_flag &= ~MSGMARKNEXT;
8437                } else if (qfirst->b_flag & MSGNOTMARKNEXT) {
8438                        bp->b_flag |= MSGNOTMARKNEXT;
8439                        qfirst->b_flag &= ~MSGNOTMARKNEXT;
8440                }

8442                linkb(bp, qfirst);
8443        }
```

```
8444         (void) putbq(q, bp);

8446         /*
8447          * A message may have come in when the sd_lock was dropped in the
8448          * calling routine. If this is the case and STR*ATMARK info was
8449          * received, need to move that from the stream head to the q_last
8450          * so that SIOCATMARK can return the proper value.
8451          */
8452         if (stp->sd_flag & (STRATMARK | STRNOTATMARK)) {
8453                 unsigned short *flagp = &q->q_last->b_flag;
8454                 uint_t b_flag = (uint_t)*flagp;

8456                 if (stp->sd_flag & STRATMARK) {
8457                         b_flag &= ~MSGNOTMARKNEXT;
8458                         b_flag |= MSGMARKNEXT;
8459                         stp->sd_flag &= ~STRATMARK;
8460                 } else {
8461                         b_flag &= ~MSGMARKNEXT;
8462                         b_flag |= MSGNOTMARKNEXT;
8463                         stp->sd_flag &= ~STRNOTATMARK;
8464                 }
8465                 *flagp = (unsigned short) b_flag;
8466         }

8468 #ifdef  DEBUG
8469         /*
8470          * Make sure that the flags are not messed up.
8471          */
8472         {
8473                 mblk_t *mp;
8474                 mp = q->q_last;
8475                 while (mp != NULL) {
8476                         ASSERT((mp->b_flag & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
8477                             (MSGMARKNEXT|MSGNOTMARKNEXT));
8478                         mp = mp->b_cont;
8479                 }
8480         }
8481 #endif
8482         if (q->q_first == bp) {
8483                 short pollevents;

8485                 if (stp->sd_flag & RSLEEP) {
8486                         stp->sd_flag &= ~RSLEEP;
8487                         cv_broadcast(&q->q_wait);
8488                 }
8489                 if (stp->sd_flag & STRPRI) {
8490                         pollevents = POLLPRI;
8491                 } else {
8492                         if (band == 0) {
8493                                 if (!(stp->sd_rput_opt & SR_POLLIN))
8494                                         return;
8495                                 stp->sd_rput_opt &= ~SR_POLLIN;
8496                                 pollevents = POLLIN | POLLRDNORM;
8497                         } else {
8498                                 pollevents = POLLIN | POLLRDBAND;
8499                         }
8500                 }
8501                 mutex_exit(&stp->sd_lock);
8502                 pollwakeup(&stp->sd_pollist, pollevents);
8503                 mutex_enter(&stp->sd_lock);
8504         }
8505 }

8507 /*
8508  * Return the held vnode attached to the stream head of a
8509  * given queue
```



```
8510  * It is the responsibility of the calling routine to ensure
8511  * that the queue does not go away (e.g. pop).
8512  */
8513 vnode_t *
8514 strq2vp(queue_t *qp)
8515 {
8516         vnode_t *vp;
8517         vp = STREAM(qp)->sd_vnode;
8518         ASSERT(vp != NULL);
8519         VN_HOLD(vp);
8520         return (vp);
8521 }

8523 /*
8524  * return the stream head write queue for the given vp
8525  * It is the responsibility of the calling routine to ensure
8526  * that the stream or vnode do not close.
8527  */
8528 queue_t *
8529 strvp2wq(vnode_t *vp)
8530 {
8531         ASSERT(vp->v_stream != NULL);
8532         return (vp->v_stream->sd_wrq);
8533 }

8535 /*
8536  * pollwakeup stream head
8537  * It is the responsibility of the calling routine to ensure
8538  * that the stream or vnode do not close.
8539  */
8540 void
8541 strpollwakeup(vnode_t *vp, short event)
8542 {
8543         ASSERT(vp->v_stream);
8544         pollwakeup(&vp->v_stream->sd_pollist, event);
8545 }

8547 /*
8548  * Mate the stream heads of two vnodes together. If the two vnodes are the
8549  * same, we just make the write-side point at the read-side -- otherwise,
8550  * we do a full mate.  Only works on vnodes associated with streams that are
8551  * still being built and thus have only a stream head.
8552  */
8553 void
8554 strmate(vnode_t *vp1, vnode_t *vp2)
8555 {
8556         queue_t *wrq1 = strvp2wq(vp1);
8557         queue_t *wrq2 = strvp2wq(vp2);

8559         /*
8560          * Verify that there are no modules on the stream yet.  We also
8561          * rely on the stream head always having a service procedure to
8562          * avoid tweaking q_nfsrv.
8563          */
8564         ASSERT(wrq1->q_next == NULL && wrq2->q_next == NULL);
8565         ASSERT(wrq1->q_qinfo->qi_srvp != NULL);
8566         ASSERT(wrq2->q_qinfo->qi_srvp != NULL);

8568         /*
8569          * If the queues are the same, just twist; otherwise do a full mate.
8570          */
8571         if (wrq1 == wrq2) {
8572                 wrq1->q_next = _RD(wrq1);
8573         } else {
8574                 wrq1->q_next = _RD(wrq2);
8575                 wrq2->q_next = _RD(wrq1);
```

```
8576                 STREAM(wrq1)->sd_mate = STREAM(wrq2);
8577                 STREAM(wrq1)->sd_flag |= STRMATE;
8578                 STREAM(wrq2)->sd_mate = STREAM(wrq1);
8579                 STREAM(wrq2)->sd_flag |= STRMATE;
8580         }
8581 }

8583 /*
8584  * XXX will go away when console is correctly fixed.
8585  * Clean up the console PIDS, from previous I_SETSIG,
8586  * called only for cnopen which never calls strclean().
8587  */
8588 void
8589 str_cn_clean(struct vnode *vp)
8590 {
8591         strsig_t *ssp, *pssp, *tssp;
8592         struct stdata *stp;
8593         struct pid  *pidp;
8594         int update = 0;

8596         ASSERT(vp->v_stream);
8597         stp = vp->v_stream;
8598         pssp = NULL;
8599         mutex_enter(&stp->sd_lock);
8600         ssp = stp->sd_siglist;
8601         while (ssp) {
8602                 mutex_enter(&pidlock);
8603                 pidp = ssp->ss_pidp;
8604                 /*
8605                  * Get rid of PID if the proc is gone.
8606                  */
8607                 if (pidp->pid_prinactive) {
8608                         tssp = ssp->ss_next;
8609                         if (pssp)
8610                                 pssp->ss_next = tssp;
8611                         else
8612                                 stp->sd_siglist = tssp;
8613                         ASSERT(pidp->pid_ref <= 1);
8614                         PID_RELE(ssp->ss_pidp);
8615                         mutex_exit(&pidlock);
8616                         kmem_free(ssp, sizeof (strsig_t));
8617                         update = 1;
8618                         ssp = tssp;
8619                         continue;
8620                 } else
8621                         mutex_exit(&pidlock);
8622                 pssp = ssp;
8623                 ssp = ssp->ss_next;
8624         }
8625         if (update) {
8626                 stp->sd_sigflags = 0;
8627                 for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
8628                         stp->sd_sigflags |= ssp->ss_events;
8629         }
8630         mutex_exit(&stp->sd_lock);
8631 }

8633 /*
8634  * Return B_TRUE if there is data in the message, B_FALSE otherwise.
8635  */
8636 static boolean_t
8637 msghasdata(mblk_t *bp)
8638 {
8639         for (; bp; bp = bp->b_cont)
8640                 if (bp->b_datap->db_type == M_DATA) {
8641                         ASSERT(bp->b_wptr >= bp->b_rptr);
```



```
8642                         if (bp->b_wptr > bp->b_rptr)
8643                                 return (B_TRUE);
8644                 }
8645         return (B_FALSE);
8646 }

8648 /*
8649  * Check whether a stream is an XTI stream or not.
8650  */
8651 static boolean_t
8652 is_xti_str(const struct stdata *stp)
8653 {
8654         struct devnames *dnp;
8655         vnode_t *vn;
8656         major_t major;
8657         if ((vn = stp->sd_vnode) != NULL && vn->v_type == VCHR &&
8658             vn->v_rdev != 0) {
8659                 major = getmajor(vn->v_rdev);
8660                 dnp = (major != DDI_MAJOR_T_NONE && major >= 0 &&
8661                     major < devcnt) ? &devnamesp[major] : NULL;
8662                 if (dnp != NULL && dnp->dn_name != NULL &&
8663                     (strcmp(dnp->dn_name, "ip") == 0 ||
8664                     strcmp(dnp->dn_name, "tcp") == 0 ||
8665                     strcmp(dnp->dn_name, "udp") == 0 ||
8666                     strcmp(dnp->dn_name, "icmp") == 0 ||
8667                     strcmp(dnp->dn_name, "tl") == 0 ||
8668                     strcmp(dnp->dn_name, "ip6") == 0 ||
8669                     strcmp(dnp->dn_name, "tcp6") == 0 ||
8670                     strcmp(dnp->dn_name, "udp6") == 0 ||
8671                     strcmp(dnp->dn_name, "icmp6") == 0)) {
8672                         return (B_TRUE);
8673                 }
8674         }
8675 #endif /* ! codereview */
8676         return (B_FALSE);
8677 }
```

```
**********************************************************
  232274 Mon Aug 17 21:08:08 2015
new/usr/src/uts/common/os/strsubr.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 642 /*
 643  * Constructor/destructor routines for the stream head cache
 644  */
 645 /* ARGSUSED */
 646 static int
 647 stream_head_constructor(void *buf, void *cdrarg, int kmflags)
 648 {
 649         stdata_t *stp = buf;

 651         mutex_init(&stp->sd_lock, NULL, MUTEX_DEFAULT, NULL);
 652         mutex_init(&stp->sd_reflock, NULL, MUTEX_DEFAULT, NULL);
 653         mutex_init(&stp->sd_qlock, NULL, MUTEX_DEFAULT, NULL);
 654         mutex_init(&stp->sd_pid_list_lock, NULL, MUTEX_DEFAULT, NULL);
 655 #endif /* ! codereview */
 656         cv_init(&stp->sd_monitor, NULL, CV_DEFAULT, NULL);
 657         cv_init(&stp->sd_iocmonitor, NULL, CV_DEFAULT, NULL);
 658         cv_init(&stp->sd_refmonitor, NULL, CV_DEFAULT, NULL);
 659         cv_init(&stp->sd_qcv, NULL, CV_DEFAULT, NULL);
 660         cv_init(&stp->sd_zcopy_wait, NULL, CV_DEFAULT, NULL);
 661         list_create(&stp->sd_pid_list, sizeof (pid_node_t),
 662                 offsetof(pid_node_t, pn_ref_link));
 663 #endif /* ! codereview */
 664         stp->sd_wrq = NULL;

 666         return (0);
 667 }

 669 /* ARGSUSED */
 670 static void
 671 stream_head_destructor(void *buf, void *cdrarg)
 672 {
 673         stdata_t *stp = buf;

 675         mutex_destroy(&stp->sd_lock);
 676         mutex_destroy(&stp->sd_reflock);
 677         mutex_destroy(&stp->sd_qlock);
 678         mutex_destroy(&stp->sd_pid_list_lock);
 679 #endif /* ! codereview */
 680         cv_destroy(&stp->sd_monitor);
 681         cv_destroy(&stp->sd_iocmonitor);
 682         cv_destroy(&stp->sd_refmonitor);
 683         cv_destroy(&stp->sd_qcv);
 684         cv_destroy(&stp->sd_zcopy_wait);
 685         list_destroy(&stp->sd_pid_list);
 686 #endif /* ! codereview */
 687 }

 689 /*
 690  * Constructor/destructor routines for the queue cache
 691  */
 692 /* ARGSUSED */
 693 static int
 694 queue_constructor(void *buf, void *cdrarg, int kmflags)
 695 {
 696         queinfo_t *qip = buf;
 697         queue_t *qp = &qip->qu_rqueue;
 698         queue_t *wqp = &qip->qu_wqueue;
 699         syncq_t *sq = &qip->qu_syncq;
```



```
 701         qp->q_first = NULL;
 702         qp->q_link = NULL;
 703         qp->q_count = 0;
 704         qp->q_mblkcnt = 0;
 705         qp->q_sqhead = NULL;
 706         qp->q_sqtail = NULL;
 707         qp->q_sqnext = NULL;
 708         qp->q_sqprev = NULL;
 709         qp->q_sqflags = 0;
 710         qp->q_rwcnt = 0;
 711         qp->q_spri = 0;

 713         mutex_init(QLOCK(qp), NULL, MUTEX_DEFAULT, NULL);
 714         cv_init(&qp->q_wait, NULL, CV_DEFAULT, NULL);

 716         wqp->q_first = NULL;
 717         wqp->q_link = NULL;
 718         wqp->q_count = 0;
 719         wqp->q_mblkcnt = 0;
 720         wqp->q_sqhead = NULL;
 721         wqp->q_sqtail = NULL;
 722         wqp->q_sqnext = NULL;
 723         wqp->q_sqprev = NULL;
 724         wqp->q_sqflags = 0;
 725         wqp->q_rwcnt = 0;
 726         wqp->q_spri = 0;

 728         mutex_init(QLOCK(wqp), NULL, MUTEX_DEFAULT, NULL);
 729         cv_init(&wqp->q_wait, NULL, CV_DEFAULT, NULL);

 731         sq->sq_head = NULL;
 732         sq->sq_tail = NULL;
 733         sq->sq_evhead = NULL;
 734         sq->sq_evtail = NULL;
 735         sq->sq_callbpend = NULL;
 736         sq->sq_outer = NULL;
 737         sq->sq_onext = NULL;
 738         sq->sq_oprev = NULL;
 739         sq->sq_next = NULL;
 740         sq->sq_svcflags = 0;
 741         sq->sq_servcount = 0;
 742         sq->sq_needexcl = 0;
 743         sq->sq_nqueues = 0;
 744         sq->sq_pri = 0;

 746         mutex_init(&sq->sq_lock, NULL, MUTEX_DEFAULT, NULL);
 747         cv_init(&sq->sq_wait, NULL, CV_DEFAULT, NULL);
 748         cv_init(&sq->sq_exitwait, NULL, CV_DEFAULT, NULL);

 750         return (0);
 751 }

 753 /* ARGSUSED */
 754 static void
 755 queue_destructor(void *buf, void *cdrarg)
 756 {
 757         queinfo_t *qip = buf;
 758         queue_t *qp = &qip->qu_rqueue;
 759         queue_t *wqp = &qip->qu_wqueue;
 760         syncq_t *sq = &qip->qu_syncq;

 762         ASSERT(qp->q_sqhead == NULL);
 763         ASSERT(wqp->q_sqhead == NULL);
 764         ASSERT(qp->q_sqnext == NULL);
 765         ASSERT(wqp->q_sqnext == NULL);
 766         ASSERT(qp->q_rwcnt == 0);
```

```
767         ASSERT(wqp->q_rwcnt == 0);

769         mutex_destroy(&qp->q_lock);
770         cv_destroy(&qp->q_wait);

772         mutex_destroy(&wqp->q_lock);
773         cv_destroy(&wqp->q_wait);

775         mutex_destroy(&sq->sq_lock);
776         cv_destroy(&sq->sq_wait);
777         cv_destroy(&sq->sq_exitwait);
778 }

780 /*
781  * Constructor/destructor routines for the syncq cache
782  */
783 /* ARGSUSED */
784 static int
785 syncq_constructor(void *buf, void *cdrarg, int kmflags)
786 {
787         syncq_t *sq = buf;

789         bzero(buf, sizeof (syncq_t));

791         mutex_init(&sq->sq_lock, NULL, MUTEX_DEFAULT, NULL);
792         cv_init(&sq->sq_wait, NULL, CV_DEFAULT, NULL);
793         cv_init(&sq->sq_exitwait, NULL, CV_DEFAULT, NULL);

795         return (0);
796 }

798 /* ARGSUSED */
799 static void
800 syncq_destructor(void *buf, void *cdrarg)
801 {
802         syncq_t *sq = buf;

804         ASSERT(sq->sq_head == NULL);
805         ASSERT(sq->sq_tail == NULL);
806         ASSERT(sq->sq_evhead == NULL);
807         ASSERT(sq->sq_evtail == NULL);
808         ASSERT(sq->sq_callbpend == NULL);
809         ASSERT(sq->sq_callbflags == 0);
810         ASSERT(sq->sq_outer == NULL);
811         ASSERT(sq->sq_onext == NULL);
812         ASSERT(sq->sq_oprev == NULL);
813         ASSERT(sq->sq_next == NULL);
814         ASSERT(sq->sq_needexcl == 0);
815         ASSERT(sq->sq_svcflags == 0);
816         ASSERT(sq->sq_servcount == 0);
817         ASSERT(sq->sq_nqueues == 0);
818         ASSERT(sq->sq_pri == 0);
819         ASSERT(sq->sq_count == 0);
820         ASSERT(sq->sq_rmqcount == 0);
821         ASSERT(sq->sq_cancelid == 0);
822         ASSERT(sq->sq_ciputctrl == NULL);
823         ASSERT(sq->sq_nciputctrl == 0);
824         ASSERT(sq->sq_type == 0);
825         ASSERT(sq->sq_flags == 0);

827         mutex_destroy(&sq->sq_lock);
828         cv_destroy(&sq->sq_wait);
829         cv_destroy(&sq->sq_exitwait);
830 }

832 /* ARGSUSED */
```



```
833 static int
834 ciputctrl_constructor(void *buf, void *cdrarg, int kmflags)
835 {
836         ciputctrl_t *cip = buf;
837         int i;

839         for (i = 0; i < n_ciputctrl; i++) {
840                 cip[i].ciputctrl_count = SQ_FASTPUT;
841                 mutex_init(&cip[i].ciputctrl_lock, NULL, MUTEX_DEFAULT, NULL);
842         }

844         return (0);
845 }

847 /* ARGSUSED */
848 static void
849 ciputctrl_destructor(void *buf, void *cdrarg)
850 {
851         ciputctrl_t *cip = buf;
852         int i;

854         for (i = 0; i < n_ciputctrl; i++) {
855                 ASSERT(cip[i].ciputctrl_count & SQ_FASTPUT);
856                 mutex_destroy(&cip[i].ciputctrl_lock);
857         }
858 }

860 /*
861  * Init routine run from main at boot time.
862  */
863 void
864 strinit(void)
865 {
866         int ncpus = ((boot_max_ncpus == -1) ? max_ncpus : boot_max_ncpus);

868         stream_head_cache = kmem_cache_create("stream_head_cache",
869             sizeof (stdata_t), 0,
870             stream_head_constructor, stream_head_destructor, NULL,
871             NULL, NULL, 0);

873         queue_cache = kmem_cache_create("queue_cache", sizeof (queinfo_t), 0,
874             queue_constructor, queue_destructor, NULL, NULL, NULL, 0);

876         syncq_cache = kmem_cache_create("syncq_cache", sizeof (syncq_t), 0,
877             syncq_constructor, syncq_destructor, NULL, NULL, NULL, 0);

879         qband_cache = kmem_cache_create("qband_cache",
880             sizeof (qband_t), 0, NULL, NULL, NULL, NULL, NULL, 0);

882         linkinfo_cache = kmem_cache_create("linkinfo_cache",
883             sizeof (linkinfo_t), 0, NULL, NULL, NULL, NULL, NULL, 0);

885         n_ciputctrl = ncpus;
886         n_ciputctrl = 1 << highbit(n_ciputctrl - 1);
887         ASSERT(n_ciputctrl >= 1);
888         n_ciputctrl = MIN(n_ciputctrl, max_n_ciputctrl);
889         if (n_ciputctrl >= min_n_ciputctrl) {
890                 ciputctrl_cache = kmem_cache_create("ciputctrl_cache",
891                     sizeof (ciputctrl_t) * n_ciputctrl,
892                     sizeof (ciputctrl_t), ciputctrl_constructor,
893                     ciputctrl_destructor, NULL, NULL, NULL, 0);
894         }

896         streams_taskq = system_taskq;

898         if (streams_taskq == NULL)
```

```
899                 panic("strinit: no memory for streams taskq!");

901         bc_bkgrnd_thread = thread_create(NULL, 0,
902             streams_bufcall_service, NULL, 0, &p0, TS_RUN, streams_lopri);

904         streams_qbkgrnd_thread = thread_create(NULL, 0,
905             streams_qbkgrnd_service, NULL, 0, &p0, TS_RUN, streams_lopri);

907         streams_sqbkgrnd_thread = thread_create(NULL, 0,
908             streams_sqbkgrnd_service, NULL, 0, &p0, TS_RUN, streams_lopri);

910         /*
911          * Create STREAMS kstats.
912          */
913         str_kstat = kstat_create("streams", 0, "strstat",
914             "net", KSTAT_TYPE_NAMED,
915             sizeof (str_statistics) / sizeof (kstat_named_t),
916             KSTAT_FLAG_VIRTUAL);

918         if (str_kstat != NULL) {
919                 str_kstat->ks_data = &str_statistics;
920                 kstat_install(str_kstat);
921         }

923         /*
924          * TPI support routine initialisation.
925          */
926         tpi_init();

928         /*
929          * Handle to have autopush and persistent link information per
930          * zone.
931          * Note: uses shutdown hook instead of destroy hook so that the
932          * persistent links can be torn down before the destroy hooks
933          * in the TCP/IP stack are called.
934          */
935         netstack_register(NS_STR, str_stack_init, str_stack_shutdown,
936             str_stack_fini);
937 }

939 void
940 str_sendsig(vnode_t *vp, int event, uchar_t band, int error)
941 {
942         struct stdata *stp;

944         ASSERT(vp->v_stream);
945         stp = vp->v_stream;
946         /* Have to hold sd_lock to prevent siglist from changing */
947         mutex_enter(&stp->sd_lock);
948         if (stp->sd_sigflags & event)
949                 strsendsig(stp->sd_siglist, event, band, error);
950         mutex_exit(&stp->sd_lock);
951 }

953 /*
954  * Send the "sevent" set of signals to a process.
955  * This might send more than one signal if the process is registered
956  * for multiple events. The caller should pass in an sevent that only
957  * includes the events for which the process has registered.
958  */
959 static void
960 dosendsig(proc_t *proc, int events, int sevent, k_siginfo_t *info,
961         uchar_t band, int error)
962 {
963         ASSERT(MUTEX_HELD(&proc->p_lock));
```



```
965         info->si_band = 0;
966         info->si_errno = 0;

968         if (sevent & S_ERROR) {
969                 sevent &= ~S_ERROR;
970                 info->si_code = POLL_ERR;
971                 info->si_errno = error;
972                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRSENDSIG,
973                     "strsendsig:proc %p info %p", proc, info);
974                 sigaddq(proc, NULL, info, KM_NOSLEEP);
975                 info->si_errno = 0;
976         }
977         if (sevent & S_HANGUP) {
978                 sevent &= ~S_HANGUP;
979                 info->si_code = POLL_HUP;
980                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRSENDSIG,
981                     "strsendsig:proc %p info %p", proc, info);
982                 sigaddq(proc, NULL, info, KM_NOSLEEP);
983         }
984         if (sevent & S_HIPRI) {
985                 sevent &= ~S_HIPRI;
986                 info->si_code = POLL_PRI;
987                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRSENDSIG,
988                     "strsendsig:proc %p info %p", proc, info);
989                 sigaddq(proc, NULL, info, KM_NOSLEEP);
990         }
991         if (sevent & S_RDBAND) {
992                 sevent &= ~S_RDBAND;
993                 if (events & S_BANDURG)
994                         sigtoproc(proc, NULL, SIGURG);
995                 else
996                         sigtoproc(proc, NULL, SIGPOLL);
997         }
998         if (sevent & S_WRBAND) {
999                 sevent &= ~S_WRBAND;
1000                sigtoproc(proc, NULL, SIGPOLL);
1001        }
1002        if (sevent & S_INPUT) {
1003                sevent &= ~S_INPUT;
1004                info->si_code = POLL_IN;
1005                info->si_band = band;
1006                TRACE_2(TR_FAC_STREAMS_FR, TR_STRSENDSIG,
1007                    "strsendsig:proc %p info %p", proc, info);
1008                sigaddq(proc, NULL, info, KM_NOSLEEP);
1009                info->si_band = 0;
1010        }
1011        if (sevent & S_OUTPUT) {
1012                sevent &= ~S_OUTPUT;
1013                info->si_code = POLL_OUT;
1014                info->si_band = band;
1015                TRACE_2(TR_FAC_STREAMS_FR, TR_STRSENDSIG,
1016                    "strsendsig:proc %p info %p", proc, info);
1017                sigaddq(proc, NULL, info, KM_NOSLEEP);
1018                info->si_band = 0;
1019        }
1020        if (sevent & S_MSG) {
1021                sevent &= ~S_MSG;
1022                info->si_code = POLL_MSG;
1023                info->si_band = band;
1024                TRACE_2(TR_FAC_STREAMS_FR, TR_STRSENDSIG,
1025                    "strsendsig:proc %p info %p", proc, info);
1026                sigaddq(proc, NULL, info, KM_NOSLEEP);
1027                info->si_band = 0;
1028        }
1029        if (sevent & S_RDNORM) {
1030                sevent &= ~S_RDNORM;
```

```
1031                 sigtoproc(proc, NULL, SIGPOLL);
1032         }
1033         if (sevent != 0) {
1034                 panic("strsendsig: unknown event(s) %x", sevent);
1035         }
1036 }

1038 /*
1039  * Send SIGPOLL/SIGURG signal to all processes and process groups
1040  * registered on the given signal list that want a signal for at
1041  * least one of the specified events.
1042  *
1043  * Must be called with exclusive access to siglist (caller holding sd_lock).
1044  *
1045  * strioctl(I_SETSIG/I_ESETSIG) will only change siglist when holding
1046  * sd_lock and the ioctl code maintains a PID_HOLD on the pid structure
1047  * while it is in the siglist.
1048  *
1049  * For performance reasons (MP scalability) the code drops pidlock
1050  * when sending signals to a single process.
1051  * When sending to a process group the code holds
1052  * pidlock to prevent the membership in the process group from changing
1053  * while walking the p_pglink list.
1054  */
1055 void
1056 strsendsig(strsig_t *siglist, int event, uchar_t band, int error)
1057 {
1058         strsig_t *ssp;
1059         k_siginfo_t info;
1060         struct pid *pidp;
1061         proc_t  *proc;

1063         info.si_signo = SIGPOLL;
1064         info.si_errno = 0;
1065         for (ssp = siglist; ssp; ssp = ssp->ss_next) {
1066                 int sevent;

1068                 sevent = ssp->ss_events & event;
1069                 if (sevent == 0)
1070                         continue;

1072                 if ((pidp = ssp->ss_pidp) == NULL) {
1073                         /* pid was released but still on event list */
1074                         continue;
1075                 }


1078                 if (ssp->ss_pid > 0) {
1079                         /*
1080                          * XXX This unfortunately still generates
1081                          * a signal when a fd is closed but
1082                          * the proc is active.
1083                          */
1084                         ASSERT(ssp->ss_pid == pidp->pid_id);

1086                         mutex_enter(&pidlock);
1087                         proc = prfind_zone(pidp->pid_id, ALL_ZONES);
1088                         if (proc == NULL) {
1089                                 mutex_exit(&pidlock);
1090                                 continue;
1091                         }
1092                         mutex_enter(&proc->p_lock);
1093                         mutex_exit(&pidlock);
1094                         dosendsig(proc, ssp->ss_events, sevent, &info,
1095                             band, error);
1096                         mutex_exit(&proc->p_lock);
```



```
1097                 } else {
1098                         /*
1099                          * Send to process group. Hold pidlock across
1100                          * calls to dosendsig().
1101                          */
1102                         pid_t pgrp = -ssp->ss_pid;

1104                         mutex_enter(&pidlock);
1105                         proc = pgfind_zone(pgrp, ALL_ZONES);
1106                         while (proc != NULL) {
1107                                 mutex_enter(&proc->p_lock);
1108                                 dosendsig(proc, ssp->ss_events, sevent,
1109                                     &info, band, error);
1110                                 mutex_exit(&proc->p_lock);
1111                                 proc = proc->p_pglink;
1112                         }
1113                         mutex_exit(&pidlock);
1114                 }
1115         }
1116 }

1118 /*
1119  * Attach a stream device or module.
1120  * qp is a read queue; the new queue goes in so its next
1121  * read ptr is the argument, and the write queue corresponding
1122  * to the argument points to this queue. Return 0 on success,
1123  * or a non-zero errno on failure.
1124  */
1125 int
1126 qattach(queue_t *qp, dev_t *devp, int oflag, cred_t *crp, fmodsw_impl_t *fp,
1127     boolean_t is_insert)
1128 {
1129         major_t                 major;
1130         cdevsw_impl_t           *dp;
1131         struct streamtab        *str;
1132         queue_t                 *rq;
1133         queue_t                 *wrq;
1134         uint32_t                qflag;
1135         uint32_t                sqtype;
1136         perdm_t                 *dmp;
1137         int                     error;
1138         int                     sflag;

1140         rq = allocq();
1141         wrq = _WR(rq);
1142         STREAM(rq) = STREAM(wrq) = STREAM(qp);

1144         if (fp != NULL) {
1145                 str = fp->f_str;
1146                 qflag = fp->f_qflag;
1147                 sqtype = fp->f_sqtype;
1148                 dmp = fp->f_dmp;
1149                 IMPLY((qflag & (QPERMOD | QMTOUTPERIM)), dmp != NULL);
1150                 sflag = MODOPEN;

1152                 /*
1153                  * stash away a pointer to the module structure so we can
1154                  * unref it in qdetach.
1155                  */
1156                 rq->q_fp = fp;
1157         } else {
1158                 ASSERT(!is_insert);

1160                 major = getmajor(*devp);
1161                 dp = &devimpl[major];
```

```
1163                 str = dp->d_str;
1164                 ASSERT(str == STREAMSTAB(major));

1166                 qflag = dp->d_qflag;
1167                 ASSERT(qflag & QISDRV);
1168                 sqtype = dp->d_sqtype;

1170                 /* create perdm_t if needed */
1171                 if (NEED_DM(dp->d_dmp, qflag))
1172                         dp->d_dmp = hold_dm(str, qflag, sqtype);

1174                 dmp = dp->d_dmp;
1175                 sflag = 0;
1176         }

1178         TRACE_2(TR_FAC_STREAMS_FR, TR_QATTACH_FLAGS,
1179             "qattach:qflag == %X(%X)", qflag, *devp);

1181         /* setq might sleep in allocator - avoid holding locks. */
1182         setq(rq, str->st_rdinit, str->st_wrinit, dmp, qflag, sqtype, B_FALSE);

1184         /*
1185          * Before calling the module's open routine, set up the q_next
1186          * pointer for inserting a module in the middle of a stream.
1187          *
1188          * Note that we can always set _QINSERTING and set up q_next
1189          * pointer for both inserting and pushing a module.  Then there
1190          * is no need for the is_insert parameter.  In insertq(), called
1191          * by qprocson(), assume that q_next of the new module always points
1192          * to the correct queue and use it for insertion.  Everything should
1193          * work out fine.  But in the first release of _I_INSERT, we
1194          * distinguish between inserting and pushing to make sure that
1195          * pushing a module follows the same code path as before.
1196          */
1197         if (is_insert) {
1198                 rq->q_flag |= _QINSERTING;
1199                 rq->q_next = qp;
1200         }

1202         /*
1203          * If there is an outer perimeter get exclusive access during
1204          * the open procedure.  Bump up the reference count on the queue.
1205          */
1206         entersq(rq->q_syncq, SQ_OPENCLOSE);
1207         error = (*rq->q_qinfo->qi_qopen)(rq, devp, oflag, sflag, crp);
1208         if (error != 0)
1209                 goto failed;
1210         leavesq(rq->q_syncq, SQ_OPENCLOSE);
1211         ASSERT(qprocsareon(rq));
1212         return (0);

1214 failed:
1215         rq->q_flag &= ~_QINSERTING;
1216         if (backq(wrq) != NULL && backq(wrq)->q_next == wrq)
1217                 qprocsoff(rq);
1218         leavesq(rq->q_syncq, SQ_OPENCLOSE);
1219         rq->q_next = wrq->q_next = NULL;
1220         qdetach(rq, 0, 0, crp, B_FALSE);
1221         return (error);
1222 }

1224 /*
1225  * Handle second open of stream. For modules, set the
1226  * last argument to MODOPEN and do not pass any open flags.
1227  * Ignore dummydev since this is not the first open.
1228  */
```



```
1229 int
1230 qreopen(queue_t *qp, dev_t *devp, int flag, cred_t *crp)
1231 {
1232         int     error;
1233         dev_t dummydev;
1234         queue_t *wqp = _WR(qp);

1236         ASSERT(qp->q_flag & QREADR);
1237         entersq(qp->q_syncq, SQ_OPENCLOSE);

1239         dummydev = *devp;
1240         if (error = ((*qp->q_qinfo->qi_qopen)(qp, &dummydev,
1241             (wqp->q_next ? 0 : flag), (wqp->q_next ? MODOPEN : 0), crp))) {
1242                 leavesq(qp->q_syncq, SQ_OPENCLOSE);
1243                 mutex_enter(&STREAM(qp)->sd_lock);
1244                 qp->q_stream->sd_flag |= STREOPENFAIL;
1245                 mutex_exit(&STREAM(qp)->sd_lock);
1246                 return (error);
1247         }
1248         leavesq(qp->q_syncq, SQ_OPENCLOSE);

1250         /*
1251          * successful open should have done qprocson()
1252          */
1253         ASSERT(qprocsareon(_RD(qp)));
1254         return (0);
1255 }

1257 /*
1258  * Detach a stream module or device.
1259  * If clmode == 1 then the module or driver was opened and its
1260  * close routine must be called. If clmode == 0, the module
1261  * or driver was never opened or the open failed, and so its close
1262  * should not be called.
1263  */
1264 void
1265 qdetach(queue_t *qp, int clmode, int flag, cred_t *crp, boolean_t is_remove)
1266 {
1267         queue_t *wqp = _WR(qp);
1268         ASSERT(STREAM(qp)->sd_flag & (STRCLOSE|STWOPEN|STRPLUMB));

1270         if (STREAM_NEEDSERVICE(STREAM(qp)))
1271                 stream_runservice(STREAM(qp));

1273         if (clmode) {
1274                 /*
1275                  * Make sure that all the messages on the write side syncq are
1276                  * processed and nothing is left. Since we are closing, no new
1277                  * messages may appear there.
1278                  */
1279                 wait_q_syncq(wqp);

1281                 entersq(qp->q_syncq, SQ_OPENCLOSE);
1282                 if (is_remove) {
1283                         mutex_enter(QLOCK(qp));
1284                         qp->q_flag |= _QREMOVING;
1285                         mutex_exit(QLOCK(qp));
1286                 }
1287                 (*qp->q_qinfo->qi_qclose)(qp, flag, crp);
1288                 /*
1289                  * Check that qprocsoff() was actually called.
1290                  */
1291                 ASSERT((qp->q_flag & QWCLOSE) && (wqp->q_flag & QWCLOSE));

1293                 leavesq(qp->q_syncq, SQ_OPENCLOSE);
1294         } else {
```

```
1295                    disable_svc(qp);
1296            }
1297
1298            /*
1299             * Allow any threads blocked in entersq to proceed and discover
1300             * the QWCLOSE is set.
1301             * Note: This assumes that all users of entersq check QWCLOSE.
1302             * Currently runservice is the only entersq that can happen
1303             * after removeq has finished.
1304             * Removeq will have discarded all messages destined to the closing
1305             * pair of queues from the syncq.
1306             * NOTE: Calling a function inside an assert is unconventional.
1307             * However, it does not cause any problem since flush_syncq() does
1308             * not change any state except when it returns non-zero i.e.
1309             * when the assert will trigger.
1310             */
1311            ASSERT(flush_syncq(qp->q_syncq, qp) == 0);
1312            ASSERT(flush_syncq(wqp->q_syncq, wqp) == 0);
1313            ASSERT((qp->q_flag & QPERMOD) ||
1314                ((qp->q_syncq->sq_head == NULL) &&
1315                (wqp->q_syncq->sq_head == NULL)));
1316
1317            /* release any fmodsw_impl_t structure held on behalf of the queue */
1318            ASSERT(qp->q_fp != NULL || qp->q_flag & QISDRV);
1319            if (qp->q_fp != NULL)
1320                    fmodsw_rele(qp->q_fp);
1321
1322            /* freeq removes us from the outer perimeter if any */
1323            freeq(qp);
1324    }
1325
1326    /* Prevent service procedures from being called */
1327    void
1328    disable_svc(queue_t *qp)
1329    {
1330            queue_t *wqp = _WR(qp);
1331
1332            ASSERT(qp->q_flag & QREADR);
1333            mutex_enter(QLOCK(qp));
1334            qp->q_flag |= QWCLOSE;
1335            mutex_exit(QLOCK(qp));
1336            mutex_enter(QLOCK(wqp));
1337            wqp->q_flag |= QWCLOSE;
1338            mutex_exit(QLOCK(wqp));
1339    }
1340
1341    /* Allow service procedures to be called again */
1342    void
1343    enable_svc(queue_t *qp)
1344    {
1345            queue_t *wqp = _WR(qp);
1346
1347            ASSERT(qp->q_flag & QREADR);
1348            mutex_enter(QLOCK(qp));
1349            qp->q_flag &= ~QWCLOSE;
1350            mutex_exit(QLOCK(qp));
1351            mutex_enter(QLOCK(wqp));
1352            wqp->q_flag &= ~QWCLOSE;
1353            mutex_exit(QLOCK(wqp));
1354    }
1355
1356    /*
1357     * Remove queue from qhead/qtail if it is enabled.
1358     * Only reset QENAB if the queue was removed from the runlist.
1359     * A queue goes through 3 stages:
1360     *      It is on the service list and QENAB is set.
```



```
1361     *      It is removed from the service list but QENAB is still set.
1362     *      QENAB gets changed to QINSERVICE.
1363     *      QINSERVICE is reset (when the service procedure is done)
1364     * Thus we can not reset QENAB unless we actually removed it from the service
1365     * queue.
1366     */
1367    void
1368    remove_runlist(queue_t *qp)
1369    {
1370            if (qp->q_flag & QENAB && qhead != NULL) {
1371                    queue_t *q_chase;
1372                    queue_t *q_curr;
1373                    int removed;
1374
1375                    mutex_enter(&service_queue);
1376                    RMQ(qp, qhead, qtail, q_link, q_chase, q_curr, removed);
1377                    mutex_exit(&service_queue);
1378                    if (removed) {
1379                            STRSTAT(qremoved);
1380                            qp->q_flag &= ~QENAB;
1381                    }
1382            }
1383    }
1384
1385
1386    /*
1387     * Wait for any pending service processing to complete.
1388     * The removal of queues from the runlist is not atomic with the
1389     * clearing of the QENABLED flag and setting the INSERVICE flag.
1390     * consequently it is possible for remove_runlist in strclose
1391     * to not find the queue on the runlist but for it to be QENABLED
1392     * and not yet INSERVICE -> hence wait_svc needs to check QENABLED
1393     * as well as INSERVICE.
1394     */
1395    void
1396    wait_svc(queue_t *qp)
1397    {
1398            queue_t *wqp = _WR(qp);
1399
1400            ASSERT(qp->q_flag & QREADR);
1401
1402            /*
1403             * Try to remove queues from qhead/qtail list.
1404             */
1405            if (qhead != NULL) {
1406                    remove_runlist(qp);
1407                    remove_runlist(wqp);
1408            }
1409            /*
1410             * Wait till the syncqs associated with the queue disappear from the
1411             * background processing list.
1412             * This only needs to be done for non-PERMOD perimeters since
1413             * for PERMOD perimeters the syncq may be shared and will only be freed
1414             * when the last module/driver is unloaded.
1415             * If for PERMOD perimeters queue was on the syncq list, removeq()
1416             * should call propagate_syncq() or drain_syncq() for it. Both of these
1417             * functions remove the queue from its syncq list, so sqthread will not
1418             * try to access the queue.
1419             */
1420            if (!(qp->q_flag & QPERMOD)) {
1421                    syncq_t *rsq = qp->q_syncq;
1422                    syncq_t *wsq = wqp->q_syncq;
1423
1424                    /*
1425                     * Disable rsq and wsq and wait for any background processing of
1426                     * syncq to complete.
```

```
1427                 */
1428                wait_sq_svc(rsq);
1429                if (wsq != rsq)
1430                        wait_sq_svc(wsq);
1431        }

1433        mutex_enter(QLOCK(qp));
1434        while (qp->q_flag & (QINSERVICE|QENAB))
1435                cv_wait(&qp->q_wait, QLOCK(qp));
1436        mutex_exit(QLOCK(qp));
1437        mutex_enter(QLOCK(wqp));
1438        while (wqp->q_flag & (QINSERVICE|QENAB))
1439                cv_wait(&wqp->q_wait, QLOCK(wqp));
1440        mutex_exit(QLOCK(wqp));
1441 }

1443 /*
1444  * Put ioctl data from userland buffer ‘arg’ into the mblk chain ‘bp’.
1445  * ‘flag’ must always contain either K_TO_K or U_TO_K; STR_NOSIG may
1446  * also be set, and is passed through to allocb_cred_wait().
1447  *
1448  * Returns errno on failure, zero on success.
1449  */
1450 int
1451 putiocd(mblk_t *bp, char *arg, int flag, cred_t *cr)
1452 {
1453        mblk_t *tmp;
1454        ssize_t  count;
1455        int error = 0;

1457        ASSERT((flag & (U_TO_K | K_TO_K)) == U_TO_K ||
1458            (flag & (U_TO_K | K_TO_K)) == K_TO_K);

1460        if (bp->b_datap->db_type == M_IOCTL) {
1461                count = ((struct iocblk *)bp->b_rptr)->ioc_count;
1462        } else {
1463                ASSERT(bp->b_datap->db_type == M_COPYIN);
1464                count = ((struct copyreq *)bp->b_rptr)->cq_size;
1465        }
1466        /*
1467         * strdoioctl validates ioc_count, so if this assert fails it
1468         * cannot be due to user error.
1469         */
1470        ASSERT(count >= 0);

1472        if ((tmp = allocb_cred_wait(count, (flag & STR_NOSIG), &error, cr,
1473            curproc->p_pid)) == NULL) {
1474                return (error);
1475        }
1476        error = strcopyin(arg, tmp->b_wptr, count, flag & (U_TO_K|K_TO_K));
1477        if (error != 0) {
1478                freeb(tmp);
1479                return (error);
1480        }
1481        DB_CPID(tmp) = curproc->p_pid;
1482        tmp->b_wptr += count;
1483        bp->b_cont = tmp;

1485        return (0);
1486 }

1488 /*
1489  * Copy ioctl data to user-land. Return non-zero errno on failure,
1490  * 0 for success.
1491  */
1492 int
```



```
1493 getiocd(mblk_t *bp, char *arg, int copymode)
1494 {
1495        ssize_t count;
1496        size_t  n;
1497        int     error;

1499        if (bp->b_datap->db_type == M_IOCACK)
1500                count = ((struct iocblk *)bp->b_rptr)->ioc_count;
1501        else {
1502                ASSERT(bp->b_datap->db_type == M_COPYOUT);
1503                count = ((struct copyreq *)bp->b_rptr)->cq_size;
1504        }
1505        ASSERT(count >= 0);

1507        for (bp = bp->b_cont; bp && count;
1508            count -= n, bp = bp->b_cont, arg += n) {
1509                n = MIN(count, bp->b_wptr - bp->b_rptr);
1510                error = strcopyout(bp->b_rptr, arg, n, copymode);
1511                if (error)
1512                        return (error);
1513        }
1514        ASSERT(count == 0);
1515        return (0);
1516 }

1518 /*
1519  * Allocate a linkinfo entry given the write queue of the
1520  * bottom module of the top stream and the write queue of the
1521  * stream head of the bottom stream.
1522  */
1523 linkinfo_t *
1524 alloclink(queue_t *qup, queue_t *qdown, file_t *fpdown)
1525 {
1526        linkinfo_t *linkp;

1528        linkp = kmem_cache_alloc(linkinfo_cache, KM_SLEEP);

1530        linkp->li_lblk.l_qtop = qup;
1531        linkp->li_lblk.l_qbot = qdown;
1532        linkp->li_fpdown = fpdown;

1534        mutex_enter(&strresources);
1535        linkp->li_next = linkinfo_list;
1536        linkp->li_prev = NULL;
1537        if (linkp->li_next)
1538                linkp->li_next->li_prev = linkp;
1539        linkinfo_list = linkp;
1540        linkp->li_lblk.l_index = ++lnk_id;
1541        ASSERT(lnk_id != 0);    /* this should never wrap in practice */
1542        mutex_exit(&strresources);

1544        return (linkp);
1545 }

1547 /*
1548  * Free a linkinfo entry.
1549  */
1550 void
1551 lbfree(linkinfo_t *linkp)
1552 {
1553        mutex_enter(&strresources);
1554        if (linkp->li_next)
1555                linkp->li_next->li_prev = linkp->li_prev;
1556        if (linkp->li_prev)
1557                linkp->li_prev->li_next = linkp->li_next;
1558        else
```

```
1559                 linkinfo_list = linkp->li_next;
1560         mutex_exit(&strresources);
1561 
1562         kmem_cache_free(linkinfo_cache, linkp);
1563 }
1564 
1565 /*
1566  * Check for a potential linking cycle.
1567  * Return 1 if a link will result in a cycle,
1568  * and 0 otherwise.
1569  */
1570 int
1571 linkcycle(stdata_t *upstp, stdata_t *lostp, str_stack_t *ss)
1572 {
1573         struct mux_node *np;
1574         struct mux_edge *ep;
1575         int i;
1576         major_t lomaj;
1577         major_t upmaj;
1578         /*
1579          * if the lower stream is a pipe/FIFO, return, since link
1580          * cycles can not happen on pipes/FIFOs
1581          */
1582         if (lostp->sd_vnode->v_type == VFIFO)
1583                 return (0);
1584 
1585         for (i = 0; i < ss->ss_devcnt; i++) {
1586                 np = &ss->ss_mux_nodes[i];
1587                 MUX_CLEAR(np);
1588         }
1589         lomaj = getmajor(lostp->sd_vnode->v_rdev);
1590         upmaj = getmajor(upstp->sd_vnode->v_rdev);
1591         np = &ss->ss_mux_nodes[lomaj];
1592         for (;;) {
1593                 if (!MUX_DIDVISIT(np)) {
1594                         if (np->mn_imaj == upmaj)
1595                                 return (1);
1596                         if (np->mn_outp == NULL) {
1597                                 MUX_VISIT(np);
1598                                 if (np->mn_originp == NULL)
1599                                         return (0);
1600                                 np = np->mn_originp;
1601                                 continue;
1602                         }
1603                         MUX_VISIT(np);
1604                         np->mn_startp = np->mn_outp;
1605                 } else {
1606                         if (np->mn_startp == NULL) {
1607                                 if (np->mn_originp == NULL)
1608                                         return (0);
1609                                 else {
1610                                         np = np->mn_originp;
1611                                         continue;
1612                                 }
1613                         }
1614                         /*
1615                          * If ep->me_nodep is a FIFO (me_nodep == NULL),
1616                          * ignore the edge and move on. ep->me_nodep gets
1617                          * set to NULL in mux_addedge() if it is a FIFO.
1618                          *
1619                          */
1620                         ep = np->mn_startp;
1621                         np->mn_startp = ep->me_nextp;
1622                         if (ep->me_nodep == NULL)
1623                                 continue;
1624                         ep->me_nodep->mn_originp = np;
```



```
1625                         np = ep->me_nodep;
1626                 }
1627         }
1628 }
1629 
1630 /*
1631  * Find linkinfo entry corresponding to the parameters.
1632  */
1633 linkinfo_t *
1634 findlinks(stdata_t *stp, int index, int type, str_stack_t *ss)
1635 {
1636         linkinfo_t *linkp;
1637         struct mux_edge *mep;
1638         struct mux_node *mnp;
1639         queue_t *qup;
1640 
1641         mutex_enter(&strresources);
1642         if ((type & LINKTYPEMASK) == LINKNORMAL) {
1643                 qup = getendq(stp->sd_wrq);
1644                 for (linkp = linkinfo_list; linkp; linkp = linkp->li_next) {
1645                         if ((qup == linkp->li_lblk.l_qtop) &&
1646                             (!index || (index == linkp->li_lblk.l_index))) {
1647                                 mutex_exit(&strresources);
1648                                 return (linkp);
1649                         }
1650                 }
1651         } else {
1652                 ASSERT((type & LINKTYPEMASK) == LINKPERSIST);
1653                 mnp = &ss->ss_mux_nodes[getmajor(stp->sd_vnode->v_rdev)];
1654                 mep = mnp->mn_outp;
1655                 while (mep) {
1656                         if ((index == 0) || (index == mep->me_muxid))
1657                                 break;
1658                         mep = mep->me_nextp;
1659                 }
1660                 if (!mep) {
1661                         mutex_exit(&strresources);
1662                         return (NULL);
1663                 }
1664                 for (linkp = linkinfo_list; linkp; linkp = linkp->li_next) {
1665                         if ((!linkp->li_lblk.l_qtop) &&
1666                             (mep->me_muxid == linkp->li_lblk.l_index)) {
1667                                 mutex_exit(&strresources);
1668                                 return (linkp);
1669                         }
1670                 }
1671         }
1672         mutex_exit(&strresources);
1673         return (NULL);
1674 }
1675 
1676 /*
1677  * Given a queue ptr, follow the chain of q_next pointers until you reach the
1678  * last queue on the chain and return it.
1679  */
1680 queue_t *
1681 getendq(queue_t *q)
1682 {
1683         ASSERT(q != NULL);
1684         while (_SAMESTR(q))
1685                 q = q->q_next;
1686         return (q);
1687 }
1688 
1689 /*
1690  * Wait for the syncq count to drop to zero.
```

```
 * sq could be either outer or inner.
 */

static void
wait_syncq(syncq_t *sq)
{
        uint16_t count;

        mutex_enter(SQLOCK(sq));
        count = sq->sq_count;
        SQ_PUTLOCKS_ENTER(sq);
        SUM_SQ_PUTCOUNTS(sq, count);
        while (count != 0) {
                sq->sq_flags |= SQ_WANTWAKEUP;
                SQ_PUTLOCKS_EXIT(sq);
                cv_wait(&sq->sq_wait, SQLOCK(sq));
                count = sq->sq_count;
                SQ_PUTLOCKS_ENTER(sq);
                SUM_SQ_PUTCOUNTS(sq, count);
        }
        SQ_PUTLOCKS_EXIT(sq);
        mutex_exit(SQLOCK(sq));
}

/*
 * Wait while there are any messages for the queue in its syncq.
 */
static void
wait_q_syncq(queue_t *q)
{
        if ((q->q_sqflags & Q_SQQUEUED) || (q->q_syncqmsgs > 0)) {
                syncq_t *sq = q->q_syncq;

                mutex_enter(SQLOCK(sq));
                while ((q->q_sqflags & Q_SQQUEUED) || (q->q_syncqmsgs > 0)) {
                        sq->sq_flags |= SQ_WANTWAKEUP;
                        cv_wait(&sq->sq_wait, SQLOCK(sq));
                }
                mutex_exit(SQLOCK(sq));
        }
}


int
mlink_file(vnode_t *vp, int cmd, struct file *fpdown, cred_t *crp, int *rvalp,
    int lhlink)
{
        struct stdata *stp;
        struct strioctl strioc;
        struct linkinfo *linkp;
        struct stdata *stpdown;
        struct streamtab *str;
        queue_t *passq;
        syncq_t *passyncq;
        queue_t *rq;
        cdevsw_impl_t *dp;
        uint32_t qflag;
        uint32_t sqtype;
        perdm_t *dmp;
        int error = 0;
        netstack_t *ns;
        str_stack_t *ss;

        stp = vp->v_stream;
        TRACE_1(TR_FAC_STREAMS_FR,
            TR_I_LINK, "I_LINK/I_PLINK:stp %p", stp);
```



```
        /*
         * Test for invalid upper stream
         */
        if (stp->sd_flag & STRHUP) {
                return (ENXIO);
        }
        if (vp->v_type == VFIFO) {
                return (EINVAL);
        }
        if (stp->sd_strtab == NULL) {
                return (EINVAL);
        }
        if (!stp->sd_strtab->st_muxwinit) {
                return (EINVAL);
        }
        if (fpdown == NULL) {
                return (EBADF);
        }
        ns = netstack_find_by_cred(crp);
        ASSERT(ns != NULL);
        ss = ns->netstack_str;
        ASSERT(ss != NULL);

        if (getmajor(stp->sd_vnode->v_rdev) >= ss->ss_devcnt) {
                netstack_rele(ss->ss_netstack);
                return (EINVAL);
        }
        mutex_enter(&muxifier);
        if (stp->sd_flag & STPLEX) {
                mutex_exit(&muxifier);
                netstack_rele(ss->ss_netstack);
                return (ENXIO);
        }

        /*
         * Test for invalid lower stream.
         * The check for the v_type != VFIFO and having a major
         * number not >= devcnt is done to avoid problems with
         * adding mux_node entry past the end of mux_nodes[].
         * For FIFO's we don't add an entry so this isn't a
         * problem.
         */
        if (((stpdown = fpdown->f_vnode->v_stream) == NULL) ||
            (stpdown == stp) || (stpdown->sd_flag &
            (STPLEX|STRHUP|STRDERR|STWRERR|IOCWAIT|STRPLUMB)) ||
            ((stpdown->sd_vnode->v_type != VFIFO) &&
            (getmajor(stpdown->sd_vnode->v_rdev) >= ss->ss_devcnt)) ||
            linkcycle(stp, stpdown, ss)) {
                mutex_exit(&muxifier);
                netstack_rele(ss->ss_netstack);
                return (EINVAL);
        }
        TRACE_1(TR_FAC_STREAMS_FR,
            TR_STPDOWN, "stpdown:%p", stpdown);
        rq = getendq(stp->sd_wrq);
        if (cmd == I_PLINK)
                rq = NULL;

        linkp = alloclink(rq, stpdown->sd_wrq, fpdown);

        strioc.ic_cmd = cmd;
        strioc.ic_timout = INFTIM;
        strioc.ic_len = sizeof (struct linkblk);
        strioc.ic_dp = (char *)&linkp->li_lblk;

        /*
```

```
1823          * STRPLUMB protects plumbing changes and should be set before
1824          * link_addpassthru()/link_rempassthru() are called, so it is set here
1825          * and cleared in the end of mlink when passthru queue is removed.
1826          * Setting of STRPLUMB prevents reopens of the stream while passthru
1827          * queue is in-place (it is not a proper module and doesn't have open
1828          * entry point).
1829          *
1830          * STPLEX prevents any threads from entering the stream from above. It
1831          * can't be set before the call to link_addpassthru() because putnext
1832          * from below may cause stream head I/O routines to be called and these
1833          * routines assert that STPLEX is not set. After link_addpassthru()
1834          * nothing may come from below since the pass queue syncq is blocked.
1835          * Note also that STPLEX should be cleared before the call to
1836          * link_rempassthru() since when messages start flowing to the stream
1837          * head (e.g. because of message propagation from the pass queue) stream
1838          * head I/O routines may be called with STPLEX flag set.
1839          *
1840          * When STPLEX is set, nothing may come into the stream from above and
1841          * it is safe to do a setq which will change stream head. So, the
1842          * correct sequence of actions is:
1843          *
1844          * 1) Set STRPLUMB
1845          * 2) Call link_addpassthru()
1846          * 3) Set STPLEX
1847          * 4) Call setq and update the stream state
1848          * 5) Clear STPLEX
1849          * 6) Call link_rempassthru()
1850          * 7) Clear STRPLUMB
1851          *
1852          * The same sequence applies to munlink() code.
1853          */
1854         mutex_enter(&stpdown->sd_lock);
1855         stpdown->sd_flag |= STRPLUMB;
1856         mutex_exit(&stpdown->sd_lock);
1857         /*
1858          * Add passthru queue below lower mux. This will block
1859          * syncqs of lower muxs read queue during I_LINK/I_UNLINK.
1860          */
1861         passq = link_addpassthru(stpdown);

1863         mutex_enter(&stpdown->sd_lock);
1864         stpdown->sd_flag |= STPLEX;
1865         mutex_exit(&stpdown->sd_lock);

1867         rq = _RD(stpdown->sd_wrq);
1868         /*
1869          * There may be messages in the streamhead's syncq due to messages
1870          * that arrived before link_addpassthru() was done. To avoid
1871          * background processing of the syncq happening simultaneous with
1872          * setq processing, we disable the streamhead syncq and wait until
1873          * existing background thread finishes working on it.
1874          */
1875         wait_sq_svc(rq->q_syncq);
1876         passyncq = passq->q_syncq;
1877         if (!(passyncq->sq_flags & SQ_BLOCKED))
1878                 blocksq(passyncq, SQ_BLOCKED, 0);

1880         ASSERT((rq->q_flag & QMT_TYPEMASK) == QMTSAFE);
1881         ASSERT(rq->q_syncq == SQ(rq) && _WR(rq)->q_syncq == SQ(rq));
1882         rq->q_ptr = _WR(rq)->q_ptr = NULL;

1884         /* setq might sleep in allocator - avoid holding locks. */
1885         /* Note: we are holding muxifier here. */

1887         str = stp->sd_strtab;
1888         dp = &devimpl[getmajor(vp->v_rdev)];
```



```
1889         ASSERT(dp->d_str == str);

1891         qflag = dp->d_qflag;
1892         sqtype = dp->d_sqtype;

1894         /* create perdm_t if needed */
1895         if (NEED_DM(dp->d_dmp, qflag))
1896                 dp->d_dmp = hold_dm(str, qflag, sqtype);

1898         dmp = dp->d_dmp;

1900         setq(rq, str->st_muxrinit, str->st_muxwinit, dmp, qflag, sqtype,
1901             B_TRUE);

1903         /*
1904          * XXX Remove any "odd" messages from the queue.
1905          * Keep only M_DATA, M_PROTO, M_PCPROTO.
1906          */
1907         error = strdoioctl(stp, &strioc, FNATIVE,
1908             K_TO_K | STR_NOERROR | STR_NOSIG, crp, rvalp);
1909         if (error != 0) {
1910                 lbfree(linkp);

1912                 if (!(passyncq->sq_flags & SQ_BLOCKED))
1913                         blocksq(passyncq, SQ_BLOCKED, 0);
1914                 /*
1915                  * Restore the stream head queue and then remove
1916                  * the passq. Turn off STPLEX before we turn on
1917                  * the stream by removing the passq.
1918                  */
1919                 rq->q_ptr = _WR(rq)->q_ptr = stpdown;
1920                 setq(rq, &strdata, &stwdata, NULL, QMTSAFE, SQ_CI|SQ_CO,
1921                     B_TRUE);

1923                 mutex_enter(&stpdown->sd_lock);
1924                 stpdown->sd_flag &= ~STPLEX;
1925                 mutex_exit(&stpdown->sd_lock);

1927                 link_rempassthru(passq);

1929                 mutex_enter(&stpdown->sd_lock);
1930                 stpdown->sd_flag &= ~STRPLUMB;
1931                 /* Wakeup anyone waiting for STRPLUMB to clear. */
1932                 cv_broadcast(&stpdown->sd_monitor);
1933                 mutex_exit(&stpdown->sd_lock);

1935                 mutex_exit(&muxifier);
1936                 netstack_rele(ss->ss_netstack);
1937                 return (error);
1938         }
1939         mutex_enter(&fpdown->f_tlock);
1940         fpdown->f_count++;
1941         mutex_exit(&fpdown->f_tlock);

1943         /*
1944          * if we've made it here the linkage is all set up so we should also
1945          * set up the layered driver linkages
1946          */

1948         ASSERT((cmd == I_LINK) || (cmd == I_PLINK));
1949         if (cmd == I_LINK) {
1950                 ldi_mlink_fp(stp, fpdown, lhlink, LINKNORMAL);
1951         } else {
1952                 ldi_mlink_fp(stp, fpdown, lhlink, LINKPERSIST);
1953         }
```

```
1955         link_rempassthru(passq);

1957         mux_addedge(stp, stpdown, linkp->li_lblk.l_index, ss);

1959         /*
1960          * Mark the upper stream as having dependent links
1961          * so that strclose can clean it up.
1962          */
1963         if (cmd == I_LINK) {
1964                 mutex_enter(&stp->sd_lock);
1965                 stp->sd_flag |= STRHASLINKS;
1966                 mutex_exit(&stp->sd_lock);
1967         }
1968         /*
1969          * Wake up any other processes that may have been
1970          * waiting on the lower stream. These will all
1971          * error out.
1972          */
1973         mutex_enter(&stpdown->sd_lock);
1974         /* The passthru module is removed so we may release STRPLUMB */
1975         stpdown->sd_flag &= ~STRPLUMB;
1976         cv_broadcast(&rq->q_wait);
1977         cv_broadcast(&_WR(rq)->q_wait);
1978         cv_broadcast(&stpdown->sd_monitor);
1979         mutex_exit(&stpdown->sd_lock);
1980         mutex_exit(&muxifier);
1981         *rvalp = linkp->li_lblk.l_index;
1982         netstack_rele(ss->ss_netstack);
1983         return (0);
1984 }

1986 int
1987 mlink(vnode_t *vp, int cmd, int arg, cred_t *crp, int *rvalp, int lhlink)
1988 {
1989         int             ret;
1990         struct file     *fpdown;

1992         fpdown = getf(arg);
1993         ret = mlink_file(vp, cmd, fpdown, crp, rvalp, lhlink);
1994         if (fpdown != NULL)
1995                 releasef(arg);
1996         return (ret);
1997 }

1999 /*
2000  * Unlink a multiplexor link. Stp is the controlling stream for the
2001  * link, and linkp points to the link's entry in the linkinfo list.
2002  * The muxifier lock must be held on entry and is dropped on exit.
2003  *
2004  * NOTE : Currently it is assumed that mux would process all the messages
2005  * sitting on it's queue before ACKing the UNLINK. It is the responsibility
2006  * of the mux to handle all the messages that arrive before UNLINK.
2007  * If the mux has to send down messages on its lower stream before
2008  * ACKing I_UNLINK, then it *should* know to handle messages even
2009  * after the UNLINK is acked (actually it should be able to handle till we
2010  * re-block the read side of the pass queue here). If the mux does not
2011  * open up the lower stream, any messages that arrive during UNLINK
2012  * will be put in the stream head. In the case of lower stream opening
2013  * up, some messages might land in the stream head depending on when
2014  * the message arrived and when the read side of the pass queue was
2015  * re-blocked.
2016  */
2017 int
2018 munlink(stdata_t *stp, linkinfo_t *linkp, int flag, cred_t *crp, int *rvalp,
2019     str_stack_t *ss)
2020 {
```



```
2021         struct strioctl strioc;
2022         struct stdata *stpdown;
2023         queue_t *rq, *wrq;
2024         queue_t *passq;
2025         syncq_t *passyncq;
2026         int error = 0;
2027         file_t *fpdown;

2029         ASSERT(MUTEX_HELD(&muxifier));

2031         stpdown = linkp->li_fpdown->f_vnode->v_stream;

2033         /*
2034          * See the comment in mlink() concerning STRPLUMB/STPLEX flags.
2035          */
2036         mutex_enter(&stpdown->sd_lock);
2037         stpdown->sd_flag |= STRPLUMB;
2038         mutex_exit(&stpdown->sd_lock);

2040         /*
2041          * Add passthru queue below lower mux. This will block
2042          * syncqs of lower muxs read queue during I_LINK/I_UNLINK.
2043          */
2044         passq = link_addpassthru(stpdown);

2046         if ((flag & LINKTYPEMASK) == LINKNORMAL)
2047                 strioc.ic_cmd = I_UNLINK;
2048         else
2049                 strioc.ic_cmd = I_PUNLINK;
2050         strioc.ic_timout = INFTIM;
2051         strioc.ic_len = sizeof (struct linkblk);
2052         strioc.ic_dp = (char *)&linkp->li_lblk;

2054         error = strdoioctl(stp, &strioc, FNATIVE,
2055             K_TO_K | STR_NOERROR | STR_NOSIG, crp, rvalp);

2057         /*
2058          * If there was an error and this is not called via strclose,
2059          * return to the user. Otherwise, pretend there was no error
2060          * and close the link.
2061          */
2062         if (error) {
2063                 if (flag & LINKCLOSE) {
2064                         cmn_err(CE_WARN, "KERNEL: munlink: could not perform "
2065                             "unlink ioctl, closing anyway (%d)\n", error);
2066                 } else {
2067                         link_rempassthru(passq);
2068                         mutex_enter(&stpdown->sd_lock);
2069                         stpdown->sd_flag &= ~STRPLUMB;
2070                         cv_broadcast(&stpdown->sd_monitor);
2071                         mutex_exit(&stpdown->sd_lock);
2072                         mutex_exit(&muxifier);
2073                         return (error);
2074                 }
2075         }

2077         mux_rmvedge(stp, linkp->li_lblk.l_index, ss);
2078         fpdown = linkp->li_fpdown;
2079         lbfree(linkp);

2081         /*
2082          * We go ahead and drop muxifier here--it's a nasty global lock that
2083          * can slow others down. It's okay to since attempts to mlink() this
2084          * stream will be stopped because STPLEX is still set in the stdata
2085          * structure, and munlink() is stopped because mux_rmvedge() and
2086          * lbfree() have removed it from mux_nodes[] and linkinfo_list,
```

```
2087          * respectively.  Note that we defer the closef() of fpdown until
2088          * after we drop muxifier since strclose() can call munlinkall().
2089          */
2090         mutex_exit(&muxifier);

2092         wrq = stpdown->sd_wrq;
2093         rq = _RD(wrq);

2095         /*
2096          * Get rid of outstanding service procedure runs, before we make
2097          * it a stream head, since a stream head doesn't have any service
2098          * procedure.
2099          */
2100         disable_svc(rq);
2101         wait_svc(rq);

2103         /*
2104          * Since we don't disable the syncq for QPERMOD, we wait for whatever
2105          * is queued up to be finished. mux should take care that nothing is
2106          * send down to this queue. We should do it now as we're going to block
2107          * passyncq if it was unblocked.
2108          */
2109         if (wrq->q_flag & QPERMOD) {
2110                 syncq_t *sq = wrq->q_syncq;

2112                 mutex_enter(SQLOCK(sq));
2113                 while (wrq->q_sqflags & Q_SQQUEUED) {
2114                         sq->sq_flags |= SQ_WANTWAKEUP;
2115                         cv_wait(&sq->sq_wait, SQLOCK(sq));
2116                 }
2117                 mutex_exit(SQLOCK(sq));
2118         }
2119         passyncq = passq->q_syncq;
2120         if (!(passyncq->sq_flags & SQ_BLOCKED)) {

2122                 syncq_t *sq, *outer;

2124                 /*
2125                  * Messages could be flowing from underneath. We will
2126                  * block the read side of the passq. This would be
2127                  * sufficient for QPAIR and QPERQ muxes to ensure
2128                  * that no data is flowing up into this queue
2129                  * and hence no thread active in this instance of
2130                  * lower mux. But for QPERMOD and QMTOUTPERIM there
2131                  * could be messages on the inner and outer/inner
2132                  * syncqs respectively. We will wait for them to drain.
2133                  * Because passq is blocked messages end up in the syncq
2134                  * And qfill_syncq could possibly end up setting QFULL
2135                  * which will access the rq->q_flag. Hence, we have to
2136                  * acquire the QLOCK in setq.
2137                  *
2138                  * XXX Messages can also flow from top into this
2139                  * queue though the unlink is over (Ex. some instance
2140                  * in putnext() called from top that has still not
2141                  * accessed this queue. And also putq(lowerq) ?).
2142                  * Solution : How about blocking the l_qtop queue ?
2143                  * Do we really care about such pure D_MP muxes ?
2144                  */

2146                 blocksq(passyncq, SQ_BLOCKED, 0);

2148                 sq = rq->q_syncq;
2149                 if ((outer = sq->sq_outer) != NULL) {

2151                         /*
2152                          * We have to just wait for the outer sq_count
```



```
2153                          * drop to zero. As this does not prevent new
2154                          * messages to enter the outer perimeter, this
2155                          * is subject to starvation.
2156                          *
2157                          * NOTE :Because of blocksq above, messages could
2158                          * be in the inner syncq only because of some
2159                          * thread holding the outer perimeter exclusively.
2160                          * Hence it would be sufficient to wait for the
2161                          * exclusive holder of the outer perimeter to drain
2162                          * the inner and outer syncqs. But we will not depend
2163                          * on this feature and hence check the inner syncqs
2164                          * separately.
2165                          */
2166                         wait_syncq(outer);
2167                 }


2170                 /*
2171                  * There could be messages destined for
2172                  * this queue. Let the exclusive holder
2173                  * drain it.
2174                  */

2176                 wait_syncq(sq);
2177                 ASSERT((rq->q_flag & QPERMOD) ||
2178                     ((rq->q_syncq->sq_head == NULL) &&
2179                     (_WR(rq)->q_syncq->sq_head == NULL)));
2180         }

2182         /*
2183          * We haven't taken care of QPERMOD case yet. QPERMOD is a special
2184          * case as we don't disable its syncq or remove it off the syncq
2185          * service list.
2186          */
2187         if (rq->q_flag & QPERMOD) {
2188                 syncq_t *sq = rq->q_syncq;

2190                 mutex_enter(SQLOCK(sq));
2191                 while (rq->q_sqflags & Q_SQQUEUED) {
2192                         sq->sq_flags |= SQ_WANTWAKEUP;
2193                         cv_wait(&sq->sq_wait, SQLOCK(sq));
2194                 }
2195                 mutex_exit(SQLOCK(sq));
2196         }

2198         /*
2199          * flush_syncq changes states only when there are some messages to
2200          * free, i.e. when it returns non-zero value to return.
2201          */
2202         ASSERT(flush_syncq(rq->q_syncq, rq) == 0);
2203         ASSERT(flush_syncq(wrq->q_syncq, wrq) == 0);

2205         /*
2206          * Nobody else should know about this queue now.
2207          * If the mux did not process the messages before
2208          * acking the I_UNLINK, free them now.
2209          */

2211         flushq(rq, FLUSHALL);
2212         flushq(_WR(rq), FLUSHALL);

2214         /*
2215          * Convert the mux lower queue into a stream head queue.
2216          * Turn off STPLEX before we turn on the stream by removing the passq.
2217          */
2218         rq->q_ptr = wrq->q_ptr = stpdown;
```

```
2219         setq(rq, &strdata, &stwdata, NULL, QMTSAFE, SQ_CI|SQ_CO, B_TRUE);

2221         ASSERT((rq->q_flag & QMT_TYPEMASK) == QMTSAFE);
2222         ASSERT(rq->q_syncq == SQ(rq) && _WR(rq)->q_syncq == SQ(rq));

2224         enable_svc(rq);

2226         /*
2227          * Now it is a proper stream, so STPLEX is cleared. But STRPLUMB still
2228          * needs to be set to prevent reopen() of the stream - such reopen may
2229          * try to call non-existent pass queue open routine and panic.
2230          */
2231         mutex_enter(&stpdown->sd_lock);
2232         stpdown->sd_flag &= ~STPLEX;
2233         mutex_exit(&stpdown->sd_lock);

2235         ASSERT(((flag & LINKTYPEMASK) == LINKNORMAL) ||
2236             ((flag & LINKTYPEMASK) == LINKPERSIST));

2238         /* clean up the layered driver linkages */
2239         if ((flag & LINKTYPEMASK) == LINKNORMAL) {
2240                 ldi_munlink_fp(stp, fpdown, LINKNORMAL);
2241         } else {
2242                 ldi_munlink_fp(stp, fpdown, LINKPERSIST);
2243         }

2245         link_rempassthru(passq);

2247         /*
2248          * Now all plumbing changes are finished and STRPLUMB is no
2249          * longer needed.
2250          */
2251         mutex_enter(&stpdown->sd_lock);
2252         stpdown->sd_flag &= ~STRPLUMB;
2253         cv_broadcast(&stpdown->sd_monitor);
2254         mutex_exit(&stpdown->sd_lock);

2256         (void) closef(fpdown);
2257         return (0);
2258 }

2260 /*
2261  * Unlink all multiplexor links for which stp is the controlling stream.
2262  * Return 0, or a non-zero errno on failure.
2263  */
2264 int
2265 munlinkall(stdata_t *stp, int flag, cred_t *crp, int *rvalp, str_stack_t *ss)
2266 {
2267         linkinfo_t *linkp;
2268         int error = 0;

2270         mutex_enter(&muxifier);
2271         while (linkp = findlinks(stp, 0, flag, ss)) {
2272                 /*
2273                  * munlink() releases the muxifier lock.
2274                  */
2275                 if (error = munlink(stp, linkp, flag, crp, rvalp, ss))
2276                         return (error);
2277                 mutex_enter(&muxifier);
2278         }
2279         mutex_exit(&muxifier);
2280         return (0);
2281 }

2283 /*
2284  * A multiplexor link has been made. Add an
```



```
2285  * edge to the directed graph.
2286  */
2287 void
2288 mux_addedge(stdata_t *upstp, stdata_t *lostp, int muxid, str_stack_t *ss)
2289 {
2290         struct mux_node *np;
2291         struct mux_edge *ep;
2292         major_t upmaj;
2293         major_t lomaj;

2295         upmaj = getmajor(upstp->sd_vnode->v_rdev);
2296         lomaj = getmajor(lostp->sd_vnode->v_rdev);
2297         np = &ss->ss_mux_nodes[upmaj];
2298         if (np->mn_outp) {
2299                 ep = np->mn_outp;
2300                 while (ep->me_nextp)
2301                         ep = ep->me_nextp;
2302                 ep->me_nextp = kmem_alloc(sizeof (struct mux_edge), KM_SLEEP);
2303                 ep = ep->me_nextp;
2304         } else {
2305                 np->mn_outp = kmem_alloc(sizeof (struct mux_edge), KM_SLEEP);
2306                 ep = np->mn_outp;
2307         }
2308         ep->me_nextp = NULL;
2309         ep->me_muxid = muxid;
2310         /*
2311          * Save the dev_t for the purposes of str_stack_shutdown.
2312          * str_stack_shutdown assumes that the device allows reopen, since
2313          * this dev_t is the one after any cloning by xx_open().
2314          * Would prefer finding the dev_t from before any cloning,
2315          * but specfs doesn't retain that.
2316          */
2317         ep->me_dev = upstp->sd_vnode->v_rdev;
2318         if (lostp->sd_vnode->v_type == VFIFO)
2319                 ep->me_nodep = NULL;
2320         else
2321                 ep->me_nodep = &ss->ss_mux_nodes[lomaj];
2322 }

2324 /*
2325  * A multiplexor link has been removed. Remove the
2326  * edge in the directed graph.
2327  */
2328 void
2329 mux_rmvedge(stdata_t *upstp, int muxid, str_stack_t *ss)
2330 {
2331         struct mux_node *np;
2332         struct mux_edge *ep;
2333         struct mux_edge *pep = NULL;
2334         major_t upmaj;

2336         upmaj = getmajor(upstp->sd_vnode->v_rdev);
2337         np = &ss->ss_mux_nodes[upmaj];
2338         ASSERT(np->mn_outp != NULL);
2339         ep = np->mn_outp;
2340         while (ep) {
2341                 if (ep->me_muxid == muxid) {
2342                         if (pep)
2343                                 pep->me_nextp = ep->me_nextp;
2344                         else
2345                                 np->mn_outp = ep->me_nextp;
2346                         kmem_free(ep, sizeof (struct mux_edge));
2347                         return;
2348                 }
2349                 pep = ep;
2350                 ep = ep->me_nextp;
```

```
2351         }
2352         ASSERT(0);      /* should not reach here */
2353 }

2355 /*
2356  * Translate the device flags (from conf.h) to the corresponding
2357  * qflag and sq_flag (type) values.
2358  */
2359 int
2360 devflg_to_qflag(struct streamtab *stp, uint32_t devflag, uint32_t *qflagp,
2361         uint32_t *sqtypep)
2362 {
2363         uint32_t qflag = 0;
2364         uint32_t sqtype = 0;

2366         if (devflag & _D_OLD)
2367                 goto bad;

2369         /* Inner perimeter presence and scope */
2370         switch (devflag & D_MTINNER_MASK) {
2371         case D_MP:
2372                 qflag |= QMTSAFE;
2373                 sqtype |= SQ_CI;
2374                 break;
2375         case D_MTPERQ|D_MP:
2376                 qflag |= QPERQ;
2377                 break;
2378         case D_MTQPAIR|D_MP:
2379                 qflag |= QPAIR;
2380                 break;
2381         case D_MTPERMOD|D_MP:
2382                 qflag |= QPERMOD;
2383                 break;
2384         default:
2385                 goto bad;
2386         }

2388         /* Outer perimeter */
2389         if (devflag & D_MTOUTPERIM) {
2390                 switch (devflag & D_MTINNER_MASK) {
2391                 case D_MP:
2392                 case D_MTPERQ|D_MP:
2393                 case D_MTQPAIR|D_MP:
2394                         break;
2395                 default:
2396                         goto bad;
2397                 }
2398                 qflag |= QMTOUTPERIM;
2399         }

2401         /* Inner perimeter modifiers */
2402         if (devflag & D_MTINNER_MOD) {
2403                 switch (devflag & D_MTINNER_MASK) {
2404                 case D_MP:
2405                         goto bad;
2406                 default:
2407                         break;
2408                 }
2409                 if (devflag & D_MTPUTSHARED)
2410                         sqtype |= SQ_CIPUT;
2411                 if (devflag & _D_MTOCSHARED) {
2412                         /*
2413                          * The code in putnext assumes that it has the
2414                          * highest concurrency by not checking sq_count.
2415                          * Thus _D_MTOCSHARED can only be supported when
2416                          * D_MTPUTSHARED is set.
```



```
2417                          */
2418                         if (!(devflag & D_MTPUTSHARED))
2419                                 goto bad;
2420                         sqtype |= SQ_CIOC;
2421                 }
2422                 if (devflag & _D_MTCBSHARED) {
2423                         /*
2424                          * The code in putnext assumes that it has the
2425                          * highest concurrency by not checking sq_count.
2426                          * Thus _D_MTCBSHARED can only be supported when
2427                          * D_MTPUTSHARED is set.
2428                          */
2429                         if (!(devflag & D_MTPUTSHARED))
2430                                 goto bad;
2431                         sqtype |= SQ_CICB;
2432                 }
2433                 if (devflag & _D_MTSVCSHARED) {
2434                         /*
2435                          * The code in putnext assumes that it has the
2436                          * highest concurrency by not checking sq_count.
2437                          * Thus _D_MTSVCSHARED can only be supported when
2438                          * D_MTPUTSHARED is set. Also _D_MTSVCSHARED is
2439                          * supported only for QPERMOD.
2440                          */
2441                         if (!(devflag & D_MTPUTSHARED) || !(qflag & QPERMOD))
2442                                 goto bad;
2443                         sqtype |= SQ_CISVC;
2444                 }
2445         }

2447         /* Default outer perimeter concurrency */
2448         sqtype |= SQ_CO;

2450         /* Outer perimeter modifiers */
2451         if (devflag & D_MTOCEXCL) {
2452                 if (!(devflag & D_MTOUTPERIM)) {
2453                         /* No outer perimeter */
2454                         goto bad;
2455                 }
2456                 sqtype &= ~SQ_COOC;
2457         }

2459         /* Synchronous Streams extended qinit structure */
2460         if (devflag & D_SYNCSTR)
2461                 qflag |= QSYNCSTR;

2463         /*
2464          * Private flag used by a transport module to indicate
2465          * to sockfs that it supports direct-access mode without
2466          * having to go through STREAMS.
2467          */
2468         if (devflag & _D_DIRECT) {
2469                 /* Reject unless the module is fully-MT (no perimeter) */
2470                 if ((qflag & QMT_TYPEMASK) != QMTSAFE)
2471                         goto bad;
2472                 qflag |= _QDIRECT;
2473         }

2475         *qflagp = qflag;
2476         *sqtypep = sqtype;
2477         return (0);

2479 bad:
2480         cmn_err(CE_WARN,
2481             "stropen: bad MT flags (0x%x) in driver '%s'",
2482             (int)(qflag & D_MTSAFETY_MASK),
```

```
2483             stp->st_rdinit->qi_minfo->mi_idname);

2485         return (EINVAL);
2486 }

2488 /*
2489  * Set the interface values for a pair of queues (qinit structure,
2490  * packet sizes, water marks).
2491  * setq assumes that the caller does not have a claim (entersq or claimq)
2492  * on the queue.
2493  */
2494 void
2495 setq(queue_t *rq, struct qinit *rinit, struct qinit *winit,
2496     perdm_t *dmp, uint32_t qflag, uint32_t sqtype, boolean_t lock_needed)
2497 {
2498         queue_t *wq;
2499         syncq_t *sq, *outer;

2501         ASSERT(rq->q_flag & QREADR);
2502         ASSERT((qflag & QMT_TYPEMASK) != 0);
2503         IMPLY((qflag & (QPERMOD | QMTOUTPERIM)), dmp != NULL);

2505         wq = _WR(rq);
2506         rq->q_qinfo = rinit;
2507         rq->q_hiwat = rinit->qi_minfo->mi_hiwat;
2508         rq->q_lowat = rinit->qi_minfo->mi_lowat;
2509         rq->q_minpsz = rinit->qi_minfo->mi_minpsz;
2510         rq->q_maxpsz = rinit->qi_minfo->mi_maxpsz;
2511         wq->q_qinfo = winit;
2512         wq->q_hiwat = winit->qi_minfo->mi_hiwat;
2513         wq->q_lowat = winit->qi_minfo->mi_lowat;
2514         wq->q_minpsz = winit->qi_minfo->mi_minpsz;
2515         wq->q_maxpsz = winit->qi_minfo->mi_maxpsz;

2517         /* Remove old syncqs */
2518         sq = rq->q_syncq;
2519         outer = sq->sq_outer;
2520         if (outer != NULL) {
2521                 ASSERT(wq->q_syncq->sq_outer == outer);
2522                 outer_remove(outer, rq->q_syncq);
2523                 if (wq->q_syncq != rq->q_syncq)
2524                         outer_remove(outer, wq->q_syncq);
2525         }
2526         ASSERT(sq->sq_outer == NULL);
2527         ASSERT(sq->sq_onext == NULL && sq->sq_oprev == NULL);

2529         if (sq != SQ(rq)) {
2530                 if (!(rq->q_flag & QPERMOD))
2531                         free_syncq(sq);
2532                 if (wq->q_syncq == rq->q_syncq)
2533                         wq->q_syncq = NULL;
2534                 rq->q_syncq = NULL;
2535         }
2536         if (wq->q_syncq != NULL && wq->q_syncq != sq &&
2537             wq->q_syncq != SQ(rq)) {
2538                 free_syncq(wq->q_syncq);
2539                 wq->q_syncq = NULL;
2540         }
2541         ASSERT(rq->q_syncq == NULL || (rq->q_syncq->sq_head == NULL &&
2542             rq->q_syncq->sq_tail == NULL));
2543         ASSERT(wq->q_syncq == NULL || (wq->q_syncq->sq_head == NULL &&
2544             wq->q_syncq->sq_tail == NULL));

2546         if (!(rq->q_flag & QPERMOD) &&
2547             rq->q_syncq != NULL && rq->q_syncq->sq_ciputctrl != NULL) {
2548                 ASSERT(rq->q_syncq->sq_nciputctrl == n_ciputctrl - 1);
```



```
2549                 SUMCHECK_CIPUTCTRL_COUNTS(rq->q_syncq->sq_ciputctrl,
2550                     rq->q_syncq->sq_nciputctrl, 0);
2551                 ASSERT(ciputctrl_cache != NULL);
2552                 kmem_cache_free(ciputctrl_cache, rq->q_syncq->sq_ciputctrl);
2553                 rq->q_syncq->sq_ciputctrl = NULL;
2554                 rq->q_syncq->sq_nciputctrl = 0;
2555         }

2557         if (!(wq->q_flag & QPERMOD) &&
2558             wq->q_syncq != NULL && wq->q_syncq->sq_ciputctrl != NULL) {
2559                 ASSERT(wq->q_syncq->sq_nciputctrl == n_ciputctrl - 1);
2560                 SUMCHECK_CIPUTCTRL_COUNTS(wq->q_syncq->sq_ciputctrl,
2561                     wq->q_syncq->sq_nciputctrl, 0);
2562                 ASSERT(ciputctrl_cache != NULL);
2563                 kmem_cache_free(ciputctrl_cache, wq->q_syncq->sq_ciputctrl);
2564                 wq->q_syncq->sq_ciputctrl = NULL;
2565                 wq->q_syncq->sq_nciputctrl = 0;
2566         }

2568         sq = SQ(rq);
2569         ASSERT(sq->sq_head == NULL && sq->sq_tail == NULL);
2570         ASSERT(sq->sq_outer == NULL);
2571         ASSERT(sq->sq_onext == NULL && sq->sq_oprev == NULL);

2573         /*
2574          * Create syncqs based on qflag and sqtype. Set the SQ_TYPES_IN_FLAGS
2575          * bits in sq_flag based on the sqtype.
2576          */
2577         ASSERT((sq->sq_flags & ~SQ_TYPES_IN_FLAGS) == 0);

2579         rq->q_syncq = wq->q_syncq = sq;
2580         sq->sq_type = sqtype;
2581         sq->sq_flags = (sqtype & SQ_TYPES_IN_FLAGS);

2583         /*
2584          *  We are making sq_svcflags zero,
2585          *  resetting SQ_DISABLED in case it was set by
2586          *  wait_svc() in the munlink path.
2587          *
2588          */
2589         ASSERT((sq->sq_svcflags & SQ_SERVICE) == 0);
2590         sq->sq_svcflags = 0;

2592         /*
2593          * We need to acquire the lock here for the mlink and munlink case,
2594          * where canputnext, backenable, etc can access the q_flag.
2595          */
2596         if (lock_needed) {
2597                 mutex_enter(QLOCK(rq));
2598                 rq->q_flag = (rq->q_flag & ~QMT_TYPEMASK) | QWANTR | qflag;
2599                 mutex_exit(QLOCK(rq));
2600                 mutex_enter(QLOCK(wq));
2601                 wq->q_flag = (wq->q_flag & ~QMT_TYPEMASK) | QWANTR | qflag;
2602                 mutex_exit(QLOCK(wq));
2603         } else {
2604                 rq->q_flag = (rq->q_flag & ~QMT_TYPEMASK) | QWANTR | qflag;
2605                 wq->q_flag = (wq->q_flag & ~QMT_TYPEMASK) | QWANTR | qflag;
2606         }

2608         if (qflag & QPERQ) {
2609                 /* Allocate a separate syncq for the write side */
2610                 sq = new_syncq();
2611                 sq->sq_type = rq->q_syncq->sq_type;
2612                 sq->sq_flags = rq->q_syncq->sq_flags;
2613                 ASSERT(sq->sq_outer == NULL && sq->sq_onext == NULL &&
2614                     sq->sq_oprev == NULL);
```

```
2615                    wq->q_syncq = sq;
2616            }
2617            if (qflag & QPERMOD) {
2618                    sq = dmp->dm_sq;

2620                    /*
2621                     * Assert that we do have an inner perimeter syncq and that it
2622                     * does not have an outer perimeter associated with it.
2623                     */
2624                    ASSERT(sq->sq_outer == NULL && sq->sq_onext == NULL &&
2625                        sq->sq_oprev == NULL);
2626                    rq->q_syncq = wq->q_syncq = sq;
2627            }
2628            if (qflag & QMTOUTPERIM) {
2629                    outer = dmp->dm_sq;

2631                    ASSERT(outer->sq_outer == NULL);
2632                    outer_insert(outer, rq->q_syncq);
2633                    if (wq->q_syncq != rq->q_syncq)
2634                            outer_insert(outer, wq->q_syncq);
2635            }
2636            ASSERT((rq->q_syncq->sq_flags & SQ_TYPES_IN_FLAGS) ==
2637                (rq->q_syncq->sq_type & SQ_TYPES_IN_FLAGS));
2638            ASSERT((wq->q_syncq->sq_flags & SQ_TYPES_IN_FLAGS) ==
2639                (wq->q_syncq->sq_type & SQ_TYPES_IN_FLAGS));
2640            ASSERT((rq->q_flag & QMT_TYPEMASK) == (qflag & QMT_TYPEMASK));

2642            /*
2643             * Initialize struio() types.
2644             */
2645            rq->q_struiot =
2646                (rq->q_flag & QSYNCSTR) ? rinit->qi_struiot : STRUIOT_NONE;
2647            wq->q_struiot =
2648                (wq->q_flag & QSYNCSTR) ? winit->qi_struiot : STRUIOT_NONE;
2649    }

2651    perdm_t *
2652    hold_dm(struct streamtab *str, uint32_t qflag, uint32_t sqtype)
2653    {
2654            syncq_t *sq;
2655            perdm_t **pp;
2656            perdm_t *p;
2657            perdm_t *dmp;

2659            ASSERT(str != NULL);
2660            ASSERT(qflag & (QPERMOD | QMTOUTPERIM));

2662            rw_enter(&perdm_rwlock, RW_READER);
2663            for (p = perdm_list; p != NULL; p = p->dm_next) {
2664                    if (p->dm_str == str) { /* found one */
2665                            atomic_inc_32(&(p->dm_ref));
2666                            rw_exit(&perdm_rwlock);
2667                            return (p);
2668                    }
2669            }
2670            rw_exit(&perdm_rwlock);

2672            sq = new_syncq();
2673            if (qflag & QPERMOD) {
2674                    sq->sq_type = sqtype | SQ_PERMOD;
2675                    sq->sq_flags = sqtype & SQ_TYPES_IN_FLAGS;
2676            } else {
2677                    ASSERT(qflag & QMTOUTPERIM);
2678                    sq->sq_onext = sq->sq_oprev = sq;
2679            }
```



```
2681            dmp = kmem_alloc(sizeof (perdm_t), KM_SLEEP);
2682            dmp->dm_sq = sq;
2683            dmp->dm_str = str;
2684            dmp->dm_ref = 1;
2685            dmp->dm_next = NULL;

2687            rw_enter(&perdm_rwlock, RW_WRITER);
2688            for (pp = &perdm_list; (p = *pp) != NULL; pp = &(p->dm_next)) {
2689                    if (p->dm_str == str) { /* already present */
2690                            p->dm_ref++;
2691                            rw_exit(&perdm_rwlock);
2692                            free_syncq(sq);
2693                            kmem_free(dmp, sizeof (perdm_t));
2694                            return (p);
2695                    }
2696            }

2698            *pp = dmp;
2699            rw_exit(&perdm_rwlock);
2700            return (dmp);
2701    }

2703    void
2704    rele_dm(perdm_t *dmp)
2705    {
2706            perdm_t **pp;
2707            perdm_t *p;

2709            rw_enter(&perdm_rwlock, RW_WRITER);
2710            ASSERT(dmp->dm_ref > 0);

2712            if (--dmp->dm_ref > 0) {
2713                    rw_exit(&perdm_rwlock);
2714                    return;
2715            }

2717            for (pp = &perdm_list; (p = *pp) != NULL; pp = &(p->dm_next))
2718                    if (p == dmp)
2719                            break;
2720            ASSERT(p == dmp);
2721            *pp = p->dm_next;
2722            rw_exit(&perdm_rwlock);

2724            /*
2725             * Wait for any background processing that relies on the
2726             * syncq to complete before it is freed.
2727             */
2728            wait_sq_svc(p->dm_sq);
2729            free_syncq(p->dm_sq);
2730            kmem_free(p, sizeof (perdm_t));
2731    }

2733    /*
2734     * Make a protocol message given control and data buffers.
2735     * n.b., this can block; be careful of what locks you hold when calling it.
2736     *
2737     * If sd_maxblk is less than *iosize this routine can fail part way through
2738     * (due to an allocation failure). In this case on return *iosize will contain
2739     * the amount that was consumed. Otherwise *iosize will not be modified
2740     * i.e. it will contain the amount that was consumed.
2741     */
2742    int
2743    strmakemsg(
2744            struct strbuf *mctl,
2745            ssize_t *iosize,
2746            struct uio *uiop,
```

```
2747         stdata_t *stp,
2748         int32_t flag,
2749         mblk_t **mpp)
2750 {
2751         mblk_t *mpctl = NULL;
2752         mblk_t *mpdata = NULL;
2753         int error;

2755         ASSERT(uiop != NULL);

2757         *mpp = NULL;
2758         /* Create control part, if any */
2759         if ((mctl != NULL) && (mctl->len >= 0)) {
2760                 error = strmakectl(mctl, flag, uiop->uio_fmode, &mpctl);
2761                 if (error)
2762                         return (error);
2763         }
2764         /* Create data part, if any */
2765         if (*iosize >= 0) {
2766                 error = strmakedata(iosize, uiop, stp, flag, &mpdata);
2767                 if (error) {
2768                         freemsg(mpctl);
2769                         return (error);
2770                 }
2771         }
2772         if (mpctl != NULL) {
2773                 if (mpdata != NULL)
2774                         linkb(mpctl, mpdata);
2775                 *mpp = mpctl;
2776         } else {
2777                 *mpp = mpdata;
2778         }
2779         return (0);
2780 }

2782 /*
2783  * Make the control part of a protocol message given a control buffer.
2784  * n.b., this can block; be careful of what locks you hold when calling it.
2785  */
2786 int
2787 strmakectl(
2788         struct strbuf *mctl,
2789         int32_t flag,
2790         int32_t fflag,
2791         mblk_t **mpp)
2792 {
2793         mblk_t *bp = NULL;
2794         unsigned char msgtype;
2795         int error = 0;
2796         cred_t *cr = CRED();

2798         /* We do not support interrupt threads using the stream head to send */
2799         ASSERT(cr != NULL);

2801         *mpp = NULL;
2802         /*
2803          * Create control part of message, if any.
2804          */
2805         if ((mctl != NULL) && (mctl->len >= 0)) {
2806                 caddr_t base;
2807                 int ctlcount;
2808                 int allocsz;

2810                 if (flag & RS_HIPRI)
2811                         msgtype = M_PCPROTO;
2812                 else
```



```
2813                         msgtype = M_PROTO;

2815                 ctlcount = mctl->len;
2816                 base = mctl->buf;

2818                 /*
2819                  * Give modules a better chance to reuse M_PROTO/M_PCPROTO
2820                  * blocks by increasing the size to something more usable.
2821                  */
2822                 allocsz = MAX(ctlcount, 64);

2824                 /*
2825                  * Range checking has already been done; simply try
2826                  * to allocate a message block for the ctl part.
2827                  */
2828                 while ((bp = allocb_cred(allocsz, cr,
2829                     curproc->p_pid)) == NULL) {
2830                         if (fflag & (FNDELAY|FNONBLOCK))
2831                                 return (EAGAIN);
2832                         if (error = strwaitbuf(allocsz, BPRI_MED))
2833                                 return (error);
2834                 }

2836                 bp->b_datap->db_type = msgtype;
2837                 if (copyin(base, bp->b_wptr, ctlcount)) {
2838                         freeb(bp);
2839                         return (EFAULT);
2840                 }
2841                 bp->b_wptr += ctlcount;
2842         }
2843         *mpp = bp;
2844         return (0);
2845 }

2847 /*
2848  * Make a protocol message given data buffers.
2849  * n.b., this can block; be careful of what locks you hold when calling it.
2850  *
2851  * If sd_maxblk is less than *iosize this routine can fail part way through
2852  * (due to an allocation failure). In this case on return *iosize will contain
2853  * the amount that was consumed. Otherwise *iosize will not be modified
2854  * i.e. it will contain the amount that was consumed.
2855  */
2856 int
2857 strmakedata(
2858         ssize_t   *iosize,
2859         struct uio *uiop,
2860         stdata_t *stp,
2861         int32_t flag,
2862         mblk_t **mpp)
2863 {
2864         mblk_t *mp = NULL;
2865         mblk_t *bp;
2866         int wroff = (int)stp->sd_wroff;
2867         int tail_len = (int)stp->sd_tail;
2868         int extra = wroff + tail_len;
2869         int error = 0;
2870         ssize_t maxblk;
2871         ssize_t count = *iosize;
2872         cred_t *cr;

2874         *mpp = NULL;
2875         if (count < 0)
2876                 return (0);

2878         /* We do not support interrupt threads using the stream head to send */
```

```
2879         cr = CRED();
2880         ASSERT(cr != NULL);
2881 
2882         maxblk = stp->sd_maxblk;
2883         if (maxblk == INFPSZ)
2884                 maxblk = count;
2885 
2886         /*
2887          * Create data part of message, if any.
2888          */
2889         do {
2890                 ssize_t size;
2891                 dblk_t  *dp;
2892 
2893                 ASSERT(uiop);
2894 
2895                 size = MIN(count, maxblk);
2896 
2897                 while ((bp = allocb_cred(size + extra, cr,
2898                     curproc->p_pid)) == NULL) {
2899                         error = EAGAIN;
2900                         if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
2901                             (error = strwaitbuf(size + extra, BPRI_MED)) != 0) {
2902                                 if (count == *iosize) {
2903                                         freemsg(mp);
2904                                         return (error);
2905                                 } else {
2906                                         *iosize -= count;
2907                                         *mpp = mp;
2908                                         return (0);
2909                                 }
2910                         }
2911                 }
2912                 dp = bp->b_datap;
2913                 dp->db_cpid = curproc->p_pid;
2914                 ASSERT(wroff <= dp->db_lim - bp->b_wptr);
2915                 bp->b_wptr = bp->b_rptr = bp->b_rptr + wroff;
2916 
2917                 if (flag & STRUIO_POSTPONE) {
2918                         /*
2919                          * Setup the stream uio portion of the
2920                          * dblk for subsequent use by struioget().
2921                          */
2922                         dp->db_struioflag = STRUIO_SPEC;
2923                         dp->db_cksumstart = 0;
2924                         dp->db_cksumstuff = 0;
2925                         dp->db_cksumend = size;
2926                         *(long long *)dp->db_struioun.data = 0ll;
2927                         bp->b_wptr += size;
2928                 } else {
2929                         if (stp->sd_copyflag & STRCOPYCACHED)
2930                                 uiop->uio_extflg |= UIO_COPY_CACHED;
2931 
2932                         if (size != 0) {
2933                                 error = uiomove(bp->b_wptr, size, UIO_WRITE,
2934                                     uiop);
2935                                 if (error != 0) {
2936                                         freeb(bp);
2937                                         freemsg(mp);
2938                                         return (error);
2939                                 }
2940                         }
2941                         bp->b_wptr += size;
2942 
2943                         if (stp->sd_wputdatafunc != NULL) {
2944                                 mblk_t *newbp;
```



```
2945 
2946                                 newbp = (stp->sd_wputdatafunc)(stp->sd_vnode,
2947                                     bp, NULL, NULL, NULL, NULL);
2948                                 if (newbp == NULL) {
2949                                         freeb(bp);
2950                                         freemsg(mp);
2951                                         return (ECOMM);
2952                                 }
2953                                 bp = newbp;
2954                         }
2955                 }
2956 
2957                 count -= size;
2958 
2959                 if (mp == NULL)
2960                         mp = bp;
2961                 else
2962                         linkb(mp, bp);
2963         } while (count > 0);
2964 
2965         *mpp = mp;
2966         return (0);
2967 }
2968 
2969 /*
2970  * Wait for a buffer to become available. Return non-zero errno
2971  * if not able to wait, 0 if buffer is probably there.
2972  */
2973 int
2974 strwaitbuf(size_t size, int pri)
2975 {
2976         bufcall_id_t id;
2977 
2978         mutex_enter(&bcall_monitor);
2979         if ((id = bufcall(size, pri, (void (*)(void *))cv_broadcast,
2980             &ttoproc(curthread)->p_flag_cv)) == 0) {
2981                 mutex_exit(&bcall_monitor);
2982                 return (ENOSR);
2983         }
2984         if (!cv_wait_sig(&(ttoproc(curthread)->p_flag_cv), &bcall_monitor)) {
2985                 unbufcall(id);
2986                 mutex_exit(&bcall_monitor);
2987                 return (EINTR);
2988         }
2989         unbufcall(id);
2990         mutex_exit(&bcall_monitor);
2991         return (0);
2992 }
2993 
2994 /*
2995  * This function waits for a read or write event to happen on a stream.
2996  * fmode can specify FNDELAY and/or FNONBLOCK.
2997  * The timeout is in ms with -1 meaning infinite.
2998  * The flag values work as follows:
2999  *      READWAIT        Check for read side errors, send M_READ
3000  *      GETWAIT         Check for read side errors, no M_READ
3001  *      WRITEWAIT       Check for write side errors.
3002  *      NOINTR          Do not return error if nonblocking or timeout.
3003  *      STR_NOERROR     Ignore all errors except STPLEX.
3004  *      STR_NOSIG       Ignore/hold signals during the duration of the call.
3005  *      STR_PEEK        Pass through the strgeterr().
3006  */
3007 int
3008 strwaitq(stdata_t *stp, int flag, ssize_t count, int fmode, clock_t timout,
3009     int *done)
3010 {
```

```
3011         int slpflg, errs;
3012         int error;
3013         kcondvar_t *sleepon;
3014         mblk_t *mp;
3015         ssize_t *rd_count;
3016         clock_t rval;

3018         ASSERT(MUTEX_HELD(&stp->sd_lock));
3019         if ((flag & READWAIT) || (flag & GETWAIT)) {
3020                 slpflg = RSLEEP;
3021                 sleepon = &_RD(stp->sd_wrq)->q_wait;
3022                 errs = STRDERR|STPLEX;
3023         } else {
3024                 slpflg = WSLEEP;
3025                 sleepon = &stp->sd_wrq->q_wait;
3026                 errs = STWRERR|STRHUP|STPLEX;
3027         }
3028         if (flag & STR_NOERROR)
3029                 errs = STPLEX;

3031         if (stp->sd_wakeq & slpflg) {
3032                 /*
3033                  * A strwakeq() is pending, no need to sleep.
3034                  */
3035                 stp->sd_wakeq &= ~slpflg;
3036                 *done = 0;
3037                 return (0);
3038         }

3040         if (stp->sd_flag & errs) {
3041                 /*
3042                  * Check for errors before going to sleep since the
3043                  * caller might not have checked this while holding
3044                  * sd_lock.
3045                  */
3046                 error = strgeterr(stp, errs, (flag & STR_PEEK));
3047                 if (error != 0) {
3048                         *done = 1;
3049                         return (error);
3050                 }
3051         }

3053         /*
3054          * If any module downstream has requested read notification
3055          * by setting SNDMREAD flag using M_SETOPTS, send a message
3056          * down stream.
3057          */
3058         if ((flag & READWAIT) && (stp->sd_flag & SNDMREAD)) {
3059                 mutex_exit(&stp->sd_lock);
3060                 if (!(mp = allocb_wait(sizeof (ssize_t), BPRI_MED,
3061                     (flag & STR_NOSIG), &error))) {
3062                         mutex_enter(&stp->sd_lock);
3063                         *done = 1;
3064                         return (error);
3065                 }
3066                 mp->b_datap->db_type = M_READ;
3067                 rd_count = (ssize_t *)mp->b_wptr;
3068                 *rd_count = count;
3069                 mp->b_wptr += sizeof (ssize_t);
3070                 /*
3071                  * Send the number of bytes requested by the
3072                  * read as the argument to M_READ.
3073                  */
3074                 stream_willservice(stp);
3075                 putnext(stp->sd_wrq, mp);
3076                 stream_runservice(stp);
```



```
3077                 mutex_enter(&stp->sd_lock);

3079                 /*
3080                  * If any data arrived due to inline processing
3081                  * of putnext(), don't sleep.
3082                  */
3083                 if (_RD(stp->sd_wrq)->q_first != NULL) {
3084                         *done = 0;
3085                         return (0);
3086                 }
3087         }

3089         if (fmode & (FNDELAY|FNONBLOCK)) {
3090                 if (!(flag & NOINTR))
3091                         error = EAGAIN;
3092                 else
3093                         error = 0;
3094                 *done = 1;
3095                 return (error);
3096         }

3098         stp->sd_flag |= slpflg;
3099         TRACE_5(TR_FAC_STREAMS_FR, TR_STRWAITQ_WAIT2,
3100             "strwaitq sleeps (2):%p, %X, %lX, %X, %p",
3101             stp, flag, count, fmode, done);

3103         rval = str_cv_wait(sleepon, &stp->sd_lock, timout, flag & STR_NOSIG);
3104         if (rval > 0) {
3105                 /* EMPTY */
3106                 TRACE_5(TR_FAC_STREAMS_FR, TR_STRWAITQ_WAKE2,
3107                     "strwaitq awakes(2):%X, %X, %X, %X, %X",
3108                     stp, flag, count, fmode, done);
3109         } else if (rval == 0) {
3110                 TRACE_5(TR_FAC_STREAMS_FR, TR_STRWAITQ_INTR2,
3111                     "strwaitq interrupt #2:%p, %X, %lX, %X, %p",
3112                     stp, flag, count, fmode, done);
3113                 stp->sd_flag &= ~slpflg;
3114                 cv_broadcast(sleepon);
3115                 if (!(flag & NOINTR))
3116                         error = EINTR;
3117                 else
3118                         error = 0;
3119                 *done = 1;
3120                 return (error);
3121         } else {
3122                 /* timeout */
3123                 TRACE_5(TR_FAC_STREAMS_FR, TR_STRWAITQ_TIME,
3124                     "strwaitq timeout:%p, %X, %lX, %X, %p",
3125                     stp, flag, count, fmode, done);
3126                 *done = 1;
3127                 if (!(flag & NOINTR))
3128                         return (ETIME);
3129                 else
3130                         return (0);
3131         }
3132         /*
3133          * If the caller implements delayed errors (i.e. queued after data)
3134          * we can not check for errors here since data as well as an
3135          * error might have arrived at the stream head. We return to
3136          * have the caller check the read queue before checking for errors.
3137          */
3138         if ((stp->sd_flag & errs) && !(flag & STR_DELAYERR)) {
3139                 error = strgeterr(stp, errs, (flag & STR_PEEK));
3140                 if (error != 0) {
3141                         *done = 1;
3142                         return (error);
```

```
3143                }
3144        }
3145        *done = 0;
3146        return (0);
3147 }

3149 /*
3150  * Perform job control discipline access checks.
3151  * Return 0 for success and the errno for failure.
3152  */

3154 #define cantsend(p, t, sig) \
3155         (sigismember(&(p)->p_ignore, sig) || signal_is_blocked((t), sig))

3157 int
3158 straccess(struct stdata *stp, enum jcaccess mode)
3159 {
3160         extern kcondvar_t lbolt_cv;     /* XXX: should be in a header file */
3161         kthread_t *t = curthread;
3162         proc_t *p = ttoproc(t);
3163         sess_t *sp;

3165         ASSERT(mutex_owned(&stp->sd_lock));

3167         if (stp->sd_sidp == NULL || stp->sd_vnode->v_type == VFIFO)
3168                 return (0);

3170         mutex_enter(&p->p_lock);                /* protects p_pgidp */

3172         for (;;) {
3173                 mutex_enter(&p->p_splock);      /* protects p->p_sessp */
3174                 sp = p->p_sessp;
3175                 mutex_enter(&sp->s_lock);       /* protects sp->* */

3177                 /*
3178                  * If this is not the calling process's controlling terminal
3179                  * or if the calling process is already in the foreground
3180                  * then allow access.
3181                  */
3182                 if (sp->s_dev != stp->sd_vnode->v_rdev ||
3183                     p->p_pgidp == stp->sd_pgidp) {
3184                         mutex_exit(&sp->s_lock);
3185                         mutex_exit(&p->p_splock);
3186                         mutex_exit(&p->p_lock);
3187                         return (0);
3188                 }

3190                 /*
3191                  * Check to see if controlling terminal has been deallocated.
3192                  */
3193                 if (sp->s_vp == NULL) {
3194                         if (!cantsend(p, t, SIGHUP))
3195                                 sigtoproc(p, t, SIGHUP);
3196                         mutex_exit(&sp->s_lock);
3197                         mutex_exit(&p->p_splock);
3198                         mutex_exit(&p->p_lock);
3199                         return (EIO);
3200                 }

3202                 mutex_exit(&sp->s_lock);
3203                 mutex_exit(&p->p_splock);

3205                 if (mode == JCGETP) {
3206                         mutex_exit(&p->p_lock);
3207                         return (0);
3208                 }
```



```
3210                 if (mode == JCREAD) {
3211                         if (p->p_detached || cantsend(p, t, SIGTTIN)) {
3212                                 mutex_exit(&p->p_lock);
3213                                 return (EIO);
3214                         }
3215                         mutex_exit(&p->p_lock);
3216                         mutex_exit(&stp->sd_lock);
3217                         pgsignal(p->p_pgidp, SIGTTIN);
3218                         mutex_enter(&stp->sd_lock);
3219                         mutex_enter(&p->p_lock);
3220                 } else {  /* mode == JCWRITE or JCSETP */
3221                         if ((mode == JCWRITE && !(stp->sd_flag & STRTOSTOP)) ||
3222                             cantsend(p, t, SIGTTOU)) {
3223                                 mutex_exit(&p->p_lock);
3224                                 return (0);
3225                         }
3226                         if (p->p_detached) {
3227                                 mutex_exit(&p->p_lock);
3228                                 return (EIO);
3229                         }
3230                         mutex_exit(&p->p_lock);
3231                         mutex_exit(&stp->sd_lock);
3232                         pgsignal(p->p_pgidp, SIGTTOU);
3233                         mutex_enter(&stp->sd_lock);
3234                         mutex_enter(&p->p_lock);
3235                 }

3237                 /*
3238                  * We call cv_wait_sig_swap() to cause the appropriate
3239                  * action for the jobcontrol signal to take place.
3240                  * If the signal is being caught, we will take the
3241                  * EINTR error return.  Otherwise, the default action
3242                  * of causing the process to stop will take place.
3243                  * In this case, we rely on the periodic cv_broadcast() on
3244                  * &lbolt_cv to wake us up to loop around and test again.
3245                  * We can't get here if the signal is ignored or
3246                  * if the current thread is blocking the signal.
3247                  */
3248                 mutex_exit(&stp->sd_lock);
3249                 if (!cv_wait_sig_swap(&lbolt_cv, &p->p_lock)) {
3250                         mutex_exit(&p->p_lock);
3251                         mutex_enter(&stp->sd_lock);
3252                         return (EINTR);
3253                 }
3254                 mutex_exit(&p->p_lock);
3255                 mutex_enter(&stp->sd_lock);
3256                 mutex_enter(&p->p_lock);
3257         }
3258 }

3260 /*
3261  * Return size of message of block type (bp->b_datap->db_type)
3262  */
3263 size_t
3264 xmsgsize(mblk_t *bp)
3265 {
3266         unsigned char type;
3267         size_t count = 0;

3269         type = bp->b_datap->db_type;

3271         for (; bp; bp = bp->b_cont) {
3272                 if (type != bp->b_datap->db_type)
3273                         break;
3274                 ASSERT(bp->b_wptr >= bp->b_rptr);
```

```
3275                 count += bp->b_wptr - bp->b_rptr;
3276         }
3277         return (count);
3278 }

3280 /*
3281  * Allocate a stream head.
3282  */
3283 struct stdata *
3284 shalloc(queue_t *qp)
3285 {
3286         stdata_t *stp;

3288         stp = kmem_cache_alloc(stream_head_cache, KM_SLEEP);

3290         stp->sd_wrq = _WR(qp);
3291         stp->sd_strtab = NULL;
3292         stp->sd_iocid = 0;
3293         stp->sd_mate = NULL;
3294         stp->sd_freezer = NULL;
3295         stp->sd_refcnt = 0;
3296         stp->sd_wakeq = 0;
3297         stp->sd_anchor = 0;
3298         stp->sd_struiowrq = NULL;
3299         stp->sd_struiordq = NULL;
3300         stp->sd_struiodnak = 0;
3301         stp->sd_struionak = NULL;
3302         stp->sd_t_audit_data = NULL;
3303         stp->sd_rput_opt = 0;
3304         stp->sd_wput_opt = 0;
3305         stp->sd_read_opt = 0;
3306         stp->sd_rprotofunc = strrput_proto;
3307         stp->sd_rmiscfunc = strrput_misc;
3308         stp->sd_rderrfunc = stp->sd_wrerrfunc = NULL;
3309         stp->sd_rputdatafunc = stp->sd_wputdatafunc = NULL;
3310         stp->sd_ciputctrl = NULL;
3311         stp->sd_nciputctrl = 0;
3312         stp->sd_qhead = NULL;
3313         stp->sd_qtail = NULL;
3314         stp->sd_servid = NULL;
3315         stp->sd_nqueues = 0;
3316         stp->sd_svcflags = 0;
3317         stp->sd_copyflag = 0;

3319         return (stp);
3320 }

3322 /*
3323  * Free a stream head.
3324  */
3325 void
3326 shfree(stdata_t *stp)
3327 {
3328         pid_node_t *pn;

3330 #endif /* ! codereview */
3331         ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));

3333         stp->sd_wrq = NULL;

3335         mutex_enter(&stp->sd_qlock);
3336         while (stp->sd_svcflags & STRS_SCHEDULED) {
3337                 STRSTAT(strwaits);
3338                 cv_wait(&stp->sd_qcv, &stp->sd_qlock);
3339         }
3340         mutex_exit(&stp->sd_qlock);
```



```
3342         if (stp->sd_ciputctrl != NULL) {
3343                 ASSERT(stp->sd_nciputctrl == n_ciputctrl - 1);
3344                 SUMCHECK_CIPUTCTRL_COUNTS(stp->sd_ciputctrl,
3345                     stp->sd_nciputctrl, 0);
3346                 ASSERT(ciputctrl_cache != NULL);
3347                 kmem_cache_free(ciputctrl_cache, stp->sd_ciputctrl);
3348                 stp->sd_ciputctrl = NULL;
3349                 stp->sd_nciputctrl = 0;
3350         }
3351         ASSERT(stp->sd_qhead == NULL);
3352         ASSERT(stp->sd_qtail == NULL);
3353         ASSERT(stp->sd_nqueues == 0);

3355         mutex_enter(&stp->sd_pid_list_lock);
3356         while ((pn = list_head(&stp->sd_pid_list)) != NULL) {
3357                 list_remove(&stp->sd_pid_list, pn);
3358                 kmem_free(pn, sizeof (*pn));
3359         }
3360         mutex_exit(&stp->sd_pid_list_lock);

3362 #endif /* ! codereview */
3363         kmem_cache_free(stream_head_cache, stp);
3364 }

3366 void
3367 sh_insert_pid(struct stdata *stp, pid_t pid)
3368 {
3369         pid_node_t *pn;

3371         mutex_enter(&stp->sd_pid_list_lock);
3372         for (pn = list_head(&stp->sd_pid_list);
3373             pn != NULL && pn->pn_pid != pid;
3374             pn = list_next(&stp->sd_pid_list, pn))
3375                 ;

3377         if (pn != NULL) {
3378                 pn->pn_count++;
3379         } else {
3380                 pn = kmem_zalloc(sizeof (*pn), KM_SLEEP);
3381                 list_link_init(&pn->pn_ref_link);
3382                 pn->pn_pid = pid;
3383                 pn->pn_count = 1;
3384                 list_insert_tail(&stp->sd_pid_list, pn);
3385         }
3386         mutex_exit(&stp->sd_pid_list_lock);
3387 }

3389 void
3390 sh_remove_pid(struct stdata *stp, pid_t pid)
3391 {
3392         pid_node_t *pn;

3394         mutex_enter(&stp->sd_pid_list_lock);
3395         for (pn = list_head(&stp->sd_pid_list);
3396             pn != NULL && pn->pn_pid != pid;
3397             pn = list_next(&stp->sd_pid_list, pn))
3398                 ;

3400         if (pn != NULL) {
3401                 if (pn->pn_count > 1) {
3402                         pn->pn_count--;
3403                 } else {
3404                         list_remove(&stp->sd_pid_list, pn);
3405                         kmem_free(pn, sizeof (*pn));
3406                 }
```

```
3407         }
3408         mutex_exit(&stp->sd_pid_list_lock);
3409 }

3411 mblk_t *
3412 sh_get_pid_mblk(struct stdata *stp)
3413 {
3414         mblk_t *mblk;
3415         int sz, n = 0;
3416         pid_t *pids;
3417         pid_node_t *pn;
3418         conn_pid_info_t *cpi;

3420         mutex_enter(&stp->sd_pid_list_lock);

3422         n = list_numnodes(&stp->sd_pid_list);
3423         sz = sizeof (conn_pid_info_t);
3424         sz += (n > 1) ? ((n - 1) * sizeof (pid_t)) : 0;
3425         if ((mblk = allocb(sz, BPRI_HI)) == NULL) {
3426                 mutex_exit(&stp->sd_pid_list_lock);
3427                 return (NULL);
3428         }
3429         mblk->b_wptr += sz;
3430         cpi = (conn_pid_info_t *)mblk->b_datap->db_base;
3431         cpi->cpi_magic = CONN_PID_INFO_MGC;
3432         cpi->cpi_contents = CONN_PID_INFO_XTI;
3433         cpi->cpi_pids_cnt = n;
3434         cpi->cpi_tot_size = sz;
3435         cpi->cpi_pids[0] = 0;

3437         if (cpi->cpi_pids_cnt > 0) {
3438                 pids = cpi->cpi_pids;
3439                 for (pn = list_head(&stp->sd_pid_list); pn != NULL;
3440                     pids++, pn = list_next(&stp->sd_pid_list, pn))
3441                         *pids = pn->pn_pid;
3442         }
3443         mutex_exit(&stp->sd_pid_list_lock);
3444         return (mblk);
3445 #endif /* ! codereview */
3446 }

3448 /*
3449  * Allocate a pair of queues and a syncq for the pair
3450  */
3451 queue_t *
3452 allocq(void)
3453 {
3454         queinfo_t *qip;
3455         queue_t *qp, *wqp;
3456         syncq_t *sq;

3458         qip = kmem_cache_alloc(queue_cache, KM_SLEEP);

3460         qp = &qip->qu_rqueue;
3461         wqp = &qip->qu_wqueue;
3462         sq = &qip->qu_syncq;

3464         qp->q_last      = NULL;
3465         qp->q_next      = NULL;
3466         qp->q_ptr       = NULL;
3467         qp->q_flag      = QUSE | QREADR;
3468         qp->q_bandp     = NULL;
3469         qp->q_stream    = NULL;
3470         qp->q_syncq     = sq;
3471         qp->q_nband     = 0;
3472         qp->q_nfsrv     = NULL;
```



```
3473         qp->q_draining  = 0;
3474         qp->q_syncqmsgs = 0;
3475         qp->q_spri      = 0;
3476         qp->q_qtstamp   = 0;
3477         qp->q_sqtstamp  = 0;
3478         qp->q_fp        = NULL;

3480         wqp->q_last     = NULL;
3481         wqp->q_next     = NULL;
3482         wqp->q_ptr      = NULL;
3483         wqp->q_flag     = QUSE;
3484         wqp->q_bandp    = NULL;
3485         wqp->q_stream   = NULL;
3486         wqp->q_syncq    = sq;
3487         wqp->q_nband    = 0;
3488         wqp->q_nfsrv    = NULL;
3489         wqp->q_draining = 0;
3490         wqp->q_syncqmsgs = 0;
3491         wqp->q_qtstamp  = 0;
3492         wqp->q_sqtstamp = 0;
3493         wqp->q_spri     = 0;

3495         sq->sq_count    = 0;
3496         sq->sq_rmqcount = 0;
3497         sq->sq_flags    = 0;
3498         sq->sq_type     = 0;
3499         sq->sq_callbflags = 0;
3500         sq->sq_cancelid = 0;
3501         sq->sq_ciputctrl = NULL;
3502         sq->sq_nciputctrl = 0;
3503         sq->sq_needexcl = 0;
3504         sq->sq_svcflags = 0;

3506         return (qp);
3507 }

3509 /*
3510  * Free a pair of queues and the "attached" syncq.
3511  * Discard any messages left on the syncq(s), remove the syncq(s) from the
3512  * outer perimeter, and free the syncq(s) if they are not the "attached" syncq.
3513  */
3514 void
3515 freeq(queue_t *qp)
3516 {
3517         qband_t *qbp, *nqbp;
3518         syncq_t *sq, *outer;
3519         queue_t *wqp = _WR(qp);

3521         ASSERT(qp->q_flag & QREADR);

3523         /*
3524          * If a previously dispatched taskq job is scheduled to run
3525          * sync_service() or a service routine is scheduled for the
3526          * queues about to be freed, wait here until all service is
3527          * done on the queue and all associated queues and syncqs.
3528          */
3529         wait_svc(qp);

3531         (void) flush_syncq(qp->q_syncq, qp);
3532         (void) flush_syncq(wqp->q_syncq, wqp);
3533         ASSERT(qp->q_syncqmsgs == 0 && wqp->q_syncqmsgs == 0);

3535         /*
3536          * Flush the queues before q_next is set to NULL This is needed
3537          * in order to backenable any downstream queue before we go away.
3538          * Note: we are already removed from the stream so that the
```

```
3539          * backenabling will not cause any messages to be delivered to our
3540          * put procedures.
3541          */
3542         flushq(qp, FLUSHALL);
3543         flushq(wqp, FLUSHALL);

3545         /* Tidy up - removeq only does a half-remove from stream */
3546         qp->q_next = wqp->q_next = NULL;
3547         ASSERT(!(qp->q_flag & QENAB));
3548         ASSERT(!(wqp->q_flag & QENAB));

3550         outer = qp->q_syncq->sq_outer;
3551         if (outer != NULL) {
3552                 outer_remove(outer, qp->q_syncq);
3553                 if (wqp->q_syncq != qp->q_syncq)
3554                         outer_remove(outer, wqp->q_syncq);
3555         }
3556         /*
3557          * Free any syncqs that are outside what allocq returned.
3558          */
3559         if (qp->q_syncq != SQ(qp) && !(qp->q_flag & QPERMOD))
3560                 free_syncq(qp->q_syncq);
3561         if (qp->q_syncq != wqp->q_syncq && wqp->q_syncq != SQ(qp))
3562                 free_syncq(wqp->q_syncq);

3564         ASSERT((qp->q_sqflags & (Q_SQQUEUED | Q_SQDRAINING)) == 0);
3565         ASSERT((wqp->q_sqflags & (Q_SQQUEUED | Q_SQDRAINING)) == 0);
3566         ASSERT(MUTEX_NOT_HELD(QLOCK(qp)));
3567         ASSERT(MUTEX_NOT_HELD(QLOCK(wqp)));
3568         sq = SQ(qp);
3569         ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
3570         ASSERT(sq->sq_head == NULL && sq->sq_tail == NULL);
3571         ASSERT(sq->sq_outer == NULL);
3572         ASSERT(sq->sq_onext == NULL && sq->sq_oprev == NULL);
3573         ASSERT(sq->sq_callbpend == NULL);
3574         ASSERT(sq->sq_needexcl == 0);

3576         if (sq->sq_ciputctrl != NULL) {
3577                 ASSERT(sq->sq_nciputctrl == n_ciputctrl - 1);
3578                 SUMCHECK_CIPUTCTRL_COUNTS(sq->sq_ciputctrl,
3579                     sq->sq_nciputctrl, 0);
3580                 ASSERT(ciputctrl_cache != NULL);
3581                 kmem_cache_free(ciputctrl_cache, sq->sq_ciputctrl);
3582                 sq->sq_ciputctrl = NULL;
3583                 sq->sq_nciputctrl = 0;
3584         }

3586         ASSERT(qp->q_first == NULL && wqp->q_first == NULL);
3587         ASSERT(qp->q_count == 0 && wqp->q_count == 0);
3588         ASSERT(qp->q_mblkcnt == 0 && wqp->q_mblkcnt == 0);

3590         qp->q_flag &= ~QUSE;
3591         wqp->q_flag &= ~QUSE;

3593         /* NOTE: Uncomment the assert below once bugid 1159635 is fixed. */
3594         /* ASSERT((qp->q_flag & QWANTW) == 0 && (wqp->q_flag & QWANTW) == 0); */

3596         qbp = qp->q_bandp;
3597         while (qbp) {
3598                 nqbp = qbp->qb_next;
3599                 freeband(qbp);
3600                 qbp = nqbp;
3601         }
3602         qbp = wqp->q_bandp;
3603         while (qbp) {
3604                 nqbp = qbp->qb_next;
```



```
3605                 freeband(qbp);
3606                 qbp = nqbp;
3607         }
3608         kmem_cache_free(queue_cache, qp);
3609 }

3611 /*
3612  * Allocate a qband structure.
3613  */
3614 qband_t *
3615 allocband(void)
3616 {
3617         qband_t *qbp;

3619         qbp = kmem_cache_alloc(qband_cache, KM_NOSLEEP);
3620         if (qbp == NULL)
3621                 return (NULL);

3623         qbp->qb_next    = NULL;
3624         qbp->qb_count   = 0;
3625         qbp->qb_mblkcnt = 0;
3626         qbp->qb_first   = NULL;
3627         qbp->qb_last    = NULL;
3628         qbp->qb_flag    = 0;

3630         return (qbp);
3631 }

3633 /*
3634  * Free a qband structure.
3635  */
3636 void
3637 freeband(qband_t *qbp)
3638 {
3639         kmem_cache_free(qband_cache, qbp);
3640 }

3642 /*
3643  * Just like putnextctl(9F), except that allocb_wait() is used.
3644  *
3645  * Consolidation Private, and of course only callable from the stream head or
3646  * routines that may block.
3647  */
3648 int
3649 putnextctl_wait(queue_t *q, int type)
3650 {
3651         mblk_t *bp;
3652         int error;

3654         if ((datamsg(type) && (type != M_DELAY)) ||
3655             (bp = allocb_wait(0, BPRI_HI, 0, &error)) == NULL)
3656                 return (0);

3658         bp->b_datap->db_type = (unsigned char)type;
3659         putnext(q, bp);
3660         return (1);
3661 }

3663 /*
3664  * Run any possible bufcalls.
3665  */
3666 void
3667 runbufcalls(void)
3668 {
3669         strbufcall_t *bcp;
```

```
3671         mutex_enter(&bcall_monitor);
3672         mutex_enter(&strbcall_lock);

3674         if (strbcalls.bc_head) {
3675                 size_t count;
3676                 int nevent;

3678                 /*
3679                  * count how many events are on the list
3680                  * now so we can check to avoid looping
3681                  * in low memory situations
3682                  */
3683                 nevent = 0;
3684                 for (bcp = strbcalls.bc_head; bcp; bcp = bcp->bc_next)
3685                         nevent++;

3687                 /*
3688                  * get estimate of available memory from kmem_avail().
3689                  * awake all bufcall functions waiting for
3690                  * memory whose request could be satisfied
3691                  * by 'count' memory and let 'em fight for it.
3692                  */
3693                 count = kmem_avail();
3694                 while ((bcp = strbcalls.bc_head) != NULL && nevent) {
3695                         STRSTAT(bufcalls);
3696                         --nevent;
3697                         if (bcp->bc_size <= count) {
3698                                 bcp->bc_executor = curthread;
3699                                 mutex_exit(&strbcall_lock);
3700                                 (*bcp->bc_func)(bcp->bc_arg);
3701                                 mutex_enter(&strbcall_lock);
3702                                 bcp->bc_executor = NULL;
3703                                 cv_broadcast(&bcall_cv);
3704                                 strbcalls.bc_head = bcp->bc_next;
3705                                 kmem_free(bcp, sizeof (strbufcall_t));
3706                         } else {
3707                                 /*
3708                                  * too big, try again later - note
3709                                  * that nevent was decremented above
3710                                  * so we won't retry this one on this
3711                                  * iteration of the loop
3712                                  */
3713                                 if (bcp->bc_next != NULL) {
3714                                         strbcalls.bc_head = bcp->bc_next;
3715                                         bcp->bc_next = NULL;
3716                                         strbcalls.bc_tail->bc_next = bcp;
3717                                         strbcalls.bc_tail = bcp;
3718                                 }
3719                         }
3720                 }
3721                 if (strbcalls.bc_head == NULL)
3722                         strbcalls.bc_tail = NULL;
3723         }

3725         mutex_exit(&strbcall_lock);
3726         mutex_exit(&bcall_monitor);
3727 }



3730 /*
3731  * Actually run queue's service routine.
3732  */
3733 static void
3734 runservice(queue_t *q)
3735 {
3736         qband_t *qbp;
```



```
3738         ASSERT(q->q_qinfo->qi_srvp);
3739 again:
3740         entersq(q->q_syncq, SQ_SVC);
3741         TRACE_1(TR_FAC_STREAMS_FR, TR_QRUNSERVICE_START,
3742             "runservice starts:%p", q);

3744         if (!(q->q_flag & QWCLOSE))
3745                 (*q->q_qinfo->qi_srvp)(q);

3747         TRACE_1(TR_FAC_STREAMS_FR, TR_QRUNSERVICE_END,
3748             "runservice ends:(%p)", q);

3750         leavesq(q->q_syncq, SQ_SVC);

3752         mutex_enter(QLOCK(q));
3753         if (q->q_flag & QENAB) {
3754                 q->q_flag &= ~QENAB;
3755                 mutex_exit(QLOCK(q));
3756                 goto again;
3757         }
3758         q->q_flag &= ~QINSERVICE;
3759         q->q_flag &= ~QBACK;
3760         for (qbp = q->q_bandp; qbp; qbp = qbp->qb_next)
3761                 qbp->qb_flag &= ~QB_BACK;
3762         /*
3763          * Wakeup thread waiting for the service procedure
3764          * to be run (strclose and qdetach).
3765          */
3766         cv_broadcast(&q->q_wait);

3768         mutex_exit(QLOCK(q));
3769 }

3771 /*
3772  * Background processing of bufcalls.
3773  */
3774 void
3775 streams_bufcall_service(void)
3776 {
3777         callb_cpr_t     cprinfo;

3779         CALLB_CPR_INIT(&cprinfo, &strbcall_lock, callb_generic_cpr,
3780             "streams_bufcall_service");

3782         mutex_enter(&strbcall_lock);

3784         for (;;) {
3785                 if (strbcalls.bc_head != NULL && kmem_avail() > 0) {
3786                         mutex_exit(&strbcall_lock);
3787                         runbufcalls();
3788                         mutex_enter(&strbcall_lock);
3789                 }
3790                 if (strbcalls.bc_head != NULL) {
3791                         STRSTAT(bcwaits);
3792                         /* Wait for memory to become available */
3793                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
3794                         (void) cv_reltimedwait(&memavail_cv, &strbcall_lock,
3795                             SEC_TO_TICK(60), TR_CLOCK_TICK);
3796                         CALLB_CPR_SAFE_END(&cprinfo, &strbcall_lock);
3797                 }

3799                 /* Wait for new work to arrive */
3800                 if (strbcalls.bc_head == NULL) {
3801                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
3802                         cv_wait(&strbcall_cv, &strbcall_lock);
```

```
3803                        CALLB_CPR_SAFE_END(&cprinfo, &strbcall_lock);
3804                }
3805        }
3806 }

3808 /*
3809  * Background processing of streams background tasks which failed
3810  * taskq_dispatch.
3811  */
3812 static void
3813 streams_qbkgrnd_service(void)
3814 {
3815        callb_cpr_t cprinfo;
3816        queue_t *q;

3818        CALLB_CPR_INIT(&cprinfo, &service_queue, callb_generic_cpr,
3819            "streams_bkgrnd_service");

3821        mutex_enter(&service_queue);

3823        for (;;) {
3824                /*
3825                 * Wait for work to arrive.
3826                 */
3827                while ((freebs_list == NULL) && (qhead == NULL)) {
3828                        CALLB_CPR_SAFE_BEGIN(&cprinfo);
3829                        cv_wait(&services_to_run, &service_queue);
3830                        CALLB_CPR_SAFE_END(&cprinfo, &service_queue);
3831                }
3832                /*
3833                 * Handle all pending freebs requests to free memory.
3834                 */
3835                while (freebs_list != NULL) {
3836                        mblk_t *mp = freebs_list;
3837                        freebs_list = mp->b_next;
3838                        mutex_exit(&service_queue);
3839                        mblk_free(mp);
3840                        mutex_enter(&service_queue);
3841                }
3842                /*
3843                 * Run pending queues.
3844                 */
3845                while (qhead != NULL) {
3846                        DQ(q, qhead, qtail, q_link);
3847                        ASSERT(q != NULL);
3848                        mutex_exit(&service_queue);
3849                        queue_service(q);
3850                        mutex_enter(&service_queue);
3851                }
3852                ASSERT(qhead == NULL && qtail == NULL);
3853        }
3854 }

3856 /*
3857  * Background processing of streams background tasks which failed
3858  * taskq_dispatch.
3859  */
3860 static void
3861 streams_sqbkgrnd_service(void)
3862 {
3863        callb_cpr_t cprinfo;
3864        syncq_t *sq;

3866        CALLB_CPR_INIT(&cprinfo, &service_queue, callb_generic_cpr,
3867            "streams_sqbkgrnd_service");
```



```
3869        mutex_enter(&service_queue);

3871        for (;;) {
3872                /*
3873                 * Wait for work to arrive.
3874                 */
3875                while (sqhead == NULL) {
3876                        CALLB_CPR_SAFE_BEGIN(&cprinfo);
3877                        cv_wait(&syncqs_to_run, &service_queue);
3878                        CALLB_CPR_SAFE_END(&cprinfo, &service_queue);
3879                }

3881                /*
3882                 * Run pending syncqs.
3883                 */
3884                while (sqhead != NULL) {
3885                        DQ(sq, sqhead, sqtail, sq_next);
3886                        ASSERT(sq != NULL);
3887                        ASSERT(sq->sq_svcflags & SQ_BGTHREAD);
3888                        mutex_exit(&service_queue);
3889                        syncq_service(sq);
3890                        mutex_enter(&service_queue);
3891                }
3892        }
3893 }

3895 /*
3896  * Disable the syncq and wait for background syncq processing to complete.
3897  * If the syncq is placed on the sqhead/sqtail queue, try to remove it from the
3898  * list.
3899  */
3900 void
3901 wait_sq_svc(syncq_t *sq)
3902 {
3903        mutex_enter(SQLOCK(sq));
3904        sq->sq_svcflags |= SQ_DISABLED;
3905        if (sq->sq_svcflags & SQ_BGTHREAD) {
3906                syncq_t *sq_chase;
3907                syncq_t *sq_curr;
3908                int removed;

3910                ASSERT(sq->sq_servcount == 1);
3911                mutex_enter(&service_queue);
3912                RMQ(sq, sqhead, sqtail, sq_next, sq_chase, sq_curr, removed);
3913                mutex_exit(&service_queue);
3914                if (removed) {
3915                        sq->sq_svcflags &= ~SQ_BGTHREAD;
3916                        sq->sq_servcount = 0;
3917                        STRSTAT(sqremoved);
3918                        goto done;
3919                }
3920        }
3921        while (sq->sq_servcount != 0) {
3922                sq->sq_flags |= SQ_WANTWAKEUP;
3923                cv_wait(&sq->sq_wait, SQLOCK(sq));
3924        }
3925 done:
3926        mutex_exit(SQLOCK(sq));
3927 }

3929 /*
3930  * Put a syncq on the list of syncq's to be serviced by the sqthread.
3931  * Add the argument to the end of the sqhead list and set the flag
3932  * indicating this syncq has been enabled.  If it has already been
3933  * enabled, don't do anything.
3934  * This routine assumes that SQLOCK is held.
```

```
3935  * NOTE that the lock order is to have the SQLOCK first,
3936  * so if the service_syncq lock is held, we need to release it
3937  * before acquiring the SQLOCK (mostly relevant for the background
3938  * thread, and this seems to be common among the STREAMS global locks).
3939  * Note that the sq_svcflags are protected by the SQLOCK.
3940  */
3941 void
3942 sqenable(syncq_t *sq)
3943 {
3944         /*
3945          * This is probably not important except for where I believe it
3946          * is being called.  At that point, it should be held (and it
3947          * is a pain to release it just for this routine, so don't do
3948          * it).
3949          */
3950         ASSERT(MUTEX_HELD(SQLOCK(sq)));

3952         IMPLY(sq->sq_servcount == 0, sq->sq_next == NULL);
3953         IMPLY(sq->sq_next != NULL, sq->sq_svcflags & SQ_BGTHREAD);

3955         /*
3956          * Do not put on list if background thread is scheduled or
3957          * syncq is disabled.
3958          */
3959         if (sq->sq_svcflags & (SQ_DISABLED | SQ_BGTHREAD))
3960                 return;

3962         /*
3963          * Check whether we should enable sq at all.
3964          * Non PERMOD syncqs may be drained by at most one thread.
3965          * PERMOD syncqs may be drained by several threads but we limit the
3966          * total amount to the lesser of
3967          *      Number of queues on the squeue and
3968          *      Number of CPUs.
3969          */
3970         if (sq->sq_servcount != 0) {
3971                 if (((sq->sq_type & SQ_PERMOD) == 0) ||
3972                     (sq->sq_servcount >= MIN(sq->sq_nqueues, ncpus_online))) {
3973                         STRSTAT(sqtoomany);
3974                         return;
3975                 }
3976         }

3978         sq->sq_tstamp = ddi_get_lbolt();
3979         STRSTAT(sqenables);

3981         /* Attempt a taskq dispatch */
3982         sq->sq_servid = (void *)taskq_dispatch(streams_taskq,
3983             (task_func_t *)syncq_service, sq, TQ_NOSLEEP | TQ_NOQUEUE);
3984         if (sq->sq_servid != NULL) {
3985                 sq->sq_servcount++;
3986                 return;
3987         }

3989         /*
3990          * This taskq dispatch failed, but a previous one may have succeeded.
3991          * Don't try to schedule on the background thread whilst there is
3992          * outstanding taskq processing.
3993          */
3994         if (sq->sq_servcount != 0)
3995                 return;

3997         /*
3998          * System is low on resources and can't perform a non-sleeping
3999          * dispatch. Schedule the syncq for a background thread and mark the
4000          * syncq to avoid any further taskq dispatch attempts.
```



```
4001          */
4002         mutex_enter(&service_queue);
4003         STRSTAT(taskqfails);
4004         ENQUEUE(sq, sqhead, sqtail, sq_next);
4005         sq->sq_svcflags |= SQ_BGTHREAD;
4006         sq->sq_servcount = 1;
4007         cv_signal(&syncqs_to_run);
4008         mutex_exit(&service_queue);
4009 }

4011 /*
4012  * Note: fifo_close() depends on the mblk_t on the queue being freed
4013  * asynchronously. The asynchronous freeing of messages breaks the
4014  * recursive call chain of fifo_close() while there are I_SENDFD type of
4015  * messages referring to other file pointers on the queue. Then when
4016  * closing pipes it can avoid stack overflow in case of daisy-chained
4017  * pipes, and also avoid deadlock in case of fifonode_t pairs (which
4018  * share the same fifolock_t).
4019  *
4020  * No need to kpreempt_disable to access cpu_seqid.  If we migrate and
4021  * the esb queue does not match the new CPU, that is OK.
4022  */
4023 void
4024 freebs_enqueue(mblk_t *mp, dblk_t *dbp)
4025 {
4026         int qindex = CPU->cpu_seqid >> esbq_log2_cpus_per_q;
4027         esb_queue_t *eqp;

4029         ASSERT(dbp->db_mblk == mp);
4030         ASSERT(qindex < esbq_nelem);

4032         eqp = system_esbq_array;
4033         if (eqp != NULL) {
4034                 eqp += qindex;
4035         } else {
4036                 mutex_enter(&esbq_lock);
4037                 if (kmem_ready && system_esbq_array == NULL)
4038                         system_esbq_array = (esb_queue_t *)kmem_zalloc(
4039                             esbq_nelem * sizeof (esb_queue_t), KM_NOSLEEP);
4040                 mutex_exit(&esbq_lock);
4041                 eqp = system_esbq_array;
4042                 if (eqp != NULL)
4043                         eqp += qindex;
4044                 else
4045                         eqp = &system_esbq;
4046         }

4048         /*
4049          * Check data sanity. The dblock should have non-empty free function.
4050          * It is better to panic here then later when the dblock is freed
4051          * asynchronously when the context is lost.
4052          */
4053         if (dbp->db_frtnp->free_func == NULL) {
4054                 panic("freebs_enqueue: dblock %p has a NULL free callback",
4055                     (void *)dbp);
4056         }

4058         mutex_enter(&eqp->eq_lock);
4059         /* queue the new mblk on the esballoc queue */
4060         if (eqp->eq_head == NULL) {
4061                 eqp->eq_head = eqp->eq_tail = mp;
4062         } else {
4063                 eqp->eq_tail->b_next = mp;
4064                 eqp->eq_tail = mp;
4065         }
4066         eqp->eq_len++;
```

```
4068         /* If we're the first thread to reach the threshold, process */
4069         if (eqp->eq_len >= esbq_max_qlen &&
4070             !(eqp->eq_flags & ESBQ_PROCESSING))
4071                 esballoc_process_queue(eqp);
4073         esballoc_set_timer(eqp, esbq_timeout);
4074         mutex_exit(&eqp->eq_lock);
4075 }
4077 static void
4078 esballoc_process_queue(esb_queue_t *eqp)
4079 {
4080         mblk_t  *mp;
4082         ASSERT(MUTEX_HELD(&eqp->eq_lock));
4084         eqp->eq_flags |= ESBQ_PROCESSING;
4086         do {
4087                 /*
4088                  * Detach the message chain for processing.
4089                  */
4090                 mp = eqp->eq_head;
4091                 eqp->eq_tail->b_next = NULL;
4092                 eqp->eq_head = eqp->eq_tail = NULL;
4093                 eqp->eq_len = 0;
4094                 mutex_exit(&eqp->eq_lock);
4096                 /*
4097                  * Process the message chain.
4098                  */
4099                 esballoc_enqueue_mblk(mp);
4100                 mutex_enter(&eqp->eq_lock);
4101         } while ((eqp->eq_len >= esbq_max_qlen) && (eqp->eq_len > 0));
4103         eqp->eq_flags &= ~ESBQ_PROCESSING;
4104 }
4106 /*
4107  * taskq callback routine to free esballoced mblk's
4108  */
4109 static void
4110 esballoc_mblk_free(mblk_t *mp)
4111 {
4112         mblk_t  *nextmp;
4114         for (; mp != NULL; mp = nextmp) {
4115                 nextmp = mp->b_next;
4116                 mp->b_next = NULL;
4117                 mblk_free(mp);
4118         }
4119 }
4121 static void
4122 esballoc_enqueue_mblk(mblk_t *mp)
4123 {
4125         if (taskq_dispatch(system_taskq, (task_func_t *)esballoc_mblk_free, mp,
4126             TQ_NOSLEEP) == NULL) {
4127                 mblk_t *first_mp = mp;
4128                 /*
4129                  * System is low on resources and can't perform a non-sleeping
4130                  * dispatch. Schedule for a background thread.
4131                  */
4132                 mutex_enter(&service_queue);
```



```
4133                 STRSTAT(taskqfails);
4135                 while (mp->b_next != NULL)
4136                         mp = mp->b_next;
4138                 mp->b_next = freebs_list;
4139                 freebs_list = first_mp;
4140                 cv_signal(&services_to_run);
4141                 mutex_exit(&service_queue);
4142         }
4143 }
4145 static void
4146 esballoc_timer(void *arg)
4147 {
4148         esb_queue_t *eqp = arg;
4150         mutex_enter(&eqp->eq_lock);
4151         eqp->eq_flags &= ~ESBQ_TIMER;
4153         if (!(eqp->eq_flags & ESBQ_PROCESSING) &&
4154             eqp->eq_len > 0)
4155                 esballoc_process_queue(eqp);
4157         esballoc_set_timer(eqp, esbq_timeout);
4158         mutex_exit(&eqp->eq_lock);
4159 }
4161 static void
4162 esballoc_set_timer(esb_queue_t *eqp, clock_t eq_timeout)
4163 {
4164         ASSERT(MUTEX_HELD(&eqp->eq_lock));
4166         if (eqp->eq_len > 0 && !(eqp->eq_flags & ESBQ_TIMER)) {
4167                 (void) timeout(esballoc_timer, eqp, eq_timeout);
4168                 eqp->eq_flags |= ESBQ_TIMER;
4169         }
4170 }
4172 /*
4173  * Setup esbq array length based upon NCPU scaled by CPUs per
4174  * queue. Use static system_esbq until kmem_ready and we can
4175  * create an array in freebs_enqueue().
4176  */
4177 void
4178 esballoc_queue_init(void)
4179 {
4180         esbq_log2_cpus_per_q = highbit(esbq_cpus_per_q - 1);
4181         esbq_cpus_per_q = 1 << esbq_log2_cpus_per_q;
4182         esbq_nelem = howmany(NCPU, esbq_cpus_per_q);
4183         system_esbq.eq_len = 0;
4184         system_esbq.eq_head = system_esbq.eq_tail = NULL;
4185         system_esbq.eq_flags = 0;
4186 }
4188 /*
4189  * Set the QBACK or QB_BACK flag in the given queue for
4190  * the given priority band.
4191  */
4192 void
4193 setqback(queue_t *q, unsigned char pri)
4194 {
4195         int i;
4196         qband_t *qbp;
4197         qband_t **qbpp;
```

```
4199         ASSERT(MUTEX_HELD(QLOCK(q)));
4200         if (pri != 0) {
4201                 if (pri > q->q_nband) {
4202                         qbpp = &q->q_bandp;
4203                         while (*qbpp)
4204                                 qbpp = &(*qbpp)->qb_next;
4205                         while (pri > q->q_nband) {
4206                                 if ((*qbpp = allocband()) == NULL) {
4207                                         cmn_err(CE_WARN,
4208                                             "setqback: can't allocate qband\n");
4209                                         return;
4210                                 }
4211                                 (*qbpp)->qb_hiwat = q->q_hiwat;
4212                                 (*qbpp)->qb_lowat = q->q_lowat;
4213                                 q->q_nband++;
4214                                 qbpp = &(*qbpp)->qb_next;
4215                         }
4216                 }
4217                 qbp = q->q_bandp;
4218                 i = pri;
4219                 while (--i)
4220                         qbp = qbp->qb_next;
4221                 qbp->qb_flag |= QB_BACK;
4222         } else {
4223                 q->q_flag |= QBACK;
4224         }
4225 }

4227 int
4228 strcopyin(void *from, void *to, size_t len, int copyflag)
4229 {
4230         if (copyflag & U_TO_K) {
4231                 ASSERT((copyflag & K_TO_K) == 0);
4232                 if (copyin(from, to, len))
4233                         return (EFAULT);
4234         } else {
4235                 ASSERT(copyflag & K_TO_K);
4236                 bcopy(from, to, len);
4237         }
4238         return (0);
4239 }

4241 int
4242 strcopyout(void *from, void *to, size_t len, int copyflag)
4243 {
4244         if (copyflag & U_TO_K) {
4245                 if (copyout(from, to, len))
4246                         return (EFAULT);
4247         } else {
4248                 ASSERT(copyflag & K_TO_K);
4249                 bcopy(from, to, len);
4250         }
4251         return (0);
4252 }

4254 /*
4255  * strsignal_nolock() posts a signal to the process(es) at the stream head.
4256  * It assumes that the stream head lock is already held, whereas strsignal()
4257  * acquires the lock first.  This routine was created because a few callers
4258  * release the stream head lock before calling only to re-acquire it after
4259  * it returns.
4260  */
4261 void
4262 strsignal_nolock(stdata_t *stp, int sig, uchar_t band)
4263 {
4264         ASSERT(MUTEX_HELD(&stp->sd_lock));
```



```
4265         switch (sig) {
4266         case SIGPOLL:
4267                 if (stp->sd_sigflags & S_MSG)
4268                         strsendsig(stp->sd_siglist, S_MSG, band, 0);
4269                 break;
4270         default:
4271                 if (stp->sd_pgidp)
4272                         pgsignal(stp->sd_pgidp, sig);
4273                 break;
4274         }
4275 }

4277 void
4278 strsignal(stdata_t *stp, int sig, int32_t band)
4279 {
4280         TRACE_3(TR_FAC_STREAMS_FR, TR_SENDSIG,
4281             "strsignal:%p, %X, %X", stp, sig, band);

4283         mutex_enter(&stp->sd_lock);
4284         switch (sig) {
4285         case SIGPOLL:
4286                 if (stp->sd_sigflags & S_MSG)
4287                         strsendsig(stp->sd_siglist, S_MSG, (uchar_t)band, 0);
4288                 break;

4290         default:
4291                 if (stp->sd_pgidp) {
4292                         pgsignal(stp->sd_pgidp, sig);
4293                 }
4294                 break;
4295         }
4296         mutex_exit(&stp->sd_lock);
4297 }

4299 void
4300 strhup(stdata_t *stp)
4301 {
4302         ASSERT(mutex_owned(&stp->sd_lock));
4303         pollwakeup(&stp->sd_pollist, POLLHUP);
4304         if (stp->sd_sigflags & S_HANGUP)
4305                 strsendsig(stp->sd_siglist, S_HANGUP, 0, 0);
4306 }

4308 /*
4309  * Backenable the first queue upstream from 'q' with a service procedure.
4310  */
4311 void
4312 backenable(queue_t *q, uchar_t pri)
4313 {
4314         queue_t *nq;

4316         /*
4317          * Our presence might not prevent other modules in our own
4318          * stream from popping/pushing since the caller of getq might not
4319          * have a claim on the queue (some drivers do a getq on somebody
4320          * else's queue - they know that the queue itself is not going away
4321          * but the framework has to guarantee q_next in that stream).
4322          */
4323         claimstr(q);

4325         /* Find nearest back queue with service proc */
4326         for (nq = backq(q); nq && !nq->q_qinfo->qi_srvp; nq = backq(nq)) {
4327                 ASSERT(STRMATED(q->q_stream) || STREAM(q) == STREAM(nq));
4328         }

4330         if (nq) {
```

```
4331                kthread_t *freezer;
4332                /*
4333                 * backenable can be called either with no locks held
4334                 * or with the stream frozen (the latter occurs when a module
4335                 * calls rmvq with the stream frozen). If the stream is frozen
4336                 * by the caller the caller will hold all qlocks in the stream.
4337                 * Note that a frozen stream doesn't freeze a mated stream,
4338                 * so we explicitly check for that.
4339                 */
4340                freezer = STREAM(q)->sd_freezer;
4341                if (freezer != curthread || STREAM(q) != STREAM(nq)) {
4342                        mutex_enter(QLOCK(nq));
4343                }
4344 #ifdef DEBUG
4345                else {
4346                        ASSERT(frozenstr(q));
4347                        ASSERT(MUTEX_HELD(QLOCK(q)));
4348                        ASSERT(MUTEX_HELD(QLOCK(nq)));
4349                }
4350 #endif
4351                setqback(nq, pri);
4352                qenable_locked(nq);
4353                if (freezer != curthread || STREAM(q) != STREAM(nq))
4354                        mutex_exit(QLOCK(nq));
4355        }
4356        releasestr(q);
4357 }

4359 /*
4360  * Return the appropriate errno when one of flags_to_check is set
4361  * in sd_flags. Uses the exported error routines if they are set.
4362  * Will return 0 if non error is set (or if the exported error routines
4363  * do not return an error).
4364  *
4365  * If there is both a read and write error to check, we prefer the read error.
4366  * Also, give preference to recorded errno's over the error functions.
4367  * The flags that are handled are:
4368  *      STPLEX          return EINVAL
4369  *      STRDERR         return sd_rerror (and clear if STRDERRNONPERSIST)
4370  *      STWRERR         return sd_werror (and clear if STWRERRNONPERSIST)
4371  *      STRHUP          return sd_werror
4372  *
4373  * If the caller indicates that the operation is a peek, a nonpersistent error
4374  * is not cleared.
4375  */
4376 int
4377 strgeterr(stdata_t *stp, int32_t flags_to_check, int ispeek)
4378 {
4379        int32_t sd_flag = stp->sd_flag & flags_to_check;
4380        int error = 0;

4382        ASSERT(MUTEX_HELD(&stp->sd_lock));
4383        ASSERT((flags_to_check & ~(STRDERR|STWRERR|STRHUP|STPLEX)) == 0);
4384        if (sd_flag & STPLEX)
4385                error = EINVAL;
4386        else if (sd_flag & STRDERR) {
4387                error = stp->sd_rerror;
4388                if ((stp->sd_flag & STRDERRNONPERSIST) && !ispeek) {
4389                        /*
4390                         * Read errors are non-persistent i.e. discarded once
4391                         * returned to a non-peeking caller,
4392                         */
4393                        stp->sd_rerror = 0;
4394                        stp->sd_flag &= ~STRDERR;
4395                }
4396                if (error == 0 && stp->sd_rderrfunc != NULL) {
```



```
4397                        int clearerr = 0;

4399                        error = (*stp->sd_rderrfunc)(stp->sd_vnode, ispeek,
4400                            &clearerr);
4401                        if (clearerr) {
4402                                stp->sd_flag &= ~STRDERR;
4403                                stp->sd_rderrfunc = NULL;
4404                        }
4405                }
4406        } else if (sd_flag & STWRERR) {
4407                error = stp->sd_werror;
4408                if ((stp->sd_flag & STWRERRNONPERSIST) && !ispeek) {
4409                        /*
4410                         * Write errors are non-persistent i.e. discarded once
4411                         * returned to a non-peeking caller,
4412                         */
4413                        stp->sd_werror = 0;
4414                        stp->sd_flag &= ~STWRERR;
4415                }
4416                if (error == 0 && stp->sd_wrerrfunc != NULL) {
4417                        int clearerr = 0;

4419                        error = (*stp->sd_wrerrfunc)(stp->sd_vnode, ispeek,
4420                            &clearerr);
4421                        if (clearerr) {
4422                                stp->sd_flag &= ~STWRERR;
4423                                stp->sd_wrerrfunc = NULL;
4424                        }
4425                }
4426        } else if (sd_flag & STRHUP) {
4427                /* sd_werror set when STRHUP */
4428                error = stp->sd_werror;
4429        }
4430        return (error);
4431 }


4434 /*
4435  * Single-thread open/close/push/pop
4436  * for twisted streams also
4437  */
4438 int
4439 strstartplumb(stdata_t *stp, int flag, int cmd)
4440 {
4441        int waited = 1;
4442        int error = 0;

4444        if (STRMATED(stp)) {
4445                struct stdata *stmatep = stp->sd_mate;

4447                STRLOCKMATES(stp);
4448                while (waited) {
4449                        waited = 0;
4450                        while (stmatep->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4451                                if ((cmd == I_POP) &&
4452                                    (flag & (FNDELAY|FNONBLOCK))) {
4453                                        STRUNLOCKMATES(stp);
4454                                        return (EAGAIN);
4455                                }
4456                                waited = 1;
4457                                mutex_exit(&stp->sd_lock);
4458                                if (!cv_wait_sig(&stmatep->sd_monitor,
4459                                    &stmatep->sd_lock)) {
4460                                        mutex_exit(&stmatep->sd_lock);
4461                                        return (EINTR);
4462                                }
```

```
4463                                mutex_exit(&stmatep->sd_lock);
4464                                STRLOCKMATES(stp);
4465                        }
4466                        while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4467                                if ((cmd == I_POP) &&
4468                                    (flag & (FNDELAY|FNONBLOCK))) {
4469                                        STRUNLOCKMATES(stp);
4470                                        return (EAGAIN);
4471                                }
4472                                waited = 1;
4473                                mutex_exit(&stmatep->sd_lock);
4474                                if (!cv_wait_sig(&stp->sd_monitor,
4475                                    &stp->sd_lock)) {
4476                                        mutex_exit(&stp->sd_lock);
4477                                        return (EINTR);
4478                                }
4479                                mutex_exit(&stp->sd_lock);
4480                                STRLOCKMATES(stp);
4481                        }
4482                        if (stp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
4483                                error = strgeterr(stp,
4484                                    STRDERR|STWRERR|STRHUP|STPLEX, 0);
4485                                if (error != 0) {
4486                                        STRUNLOCKMATES(stp);
4487                                        return (error);
4488                                }
4489                        }
4490                }
4491                stp->sd_flag |= STRPLUMB;
4492                STRUNLOCKMATES(stp);
4493        } else {
4494                mutex_enter(&stp->sd_lock);
4495                while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4496                        if (((cmd == I_POP) || (cmd == _I_REMOVE)) &&
4497                            (flag & (FNDELAY|FNONBLOCK))) {
4498                                mutex_exit(&stp->sd_lock);
4499                                return (EAGAIN);
4500                        }
4501                        if (!cv_wait_sig(&stp->sd_monitor, &stp->sd_lock)) {
4502                                mutex_exit(&stp->sd_lock);
4503                                return (EINTR);
4504                        }
4505                        if (stp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
4506                                error = strgeterr(stp,
4507                                    STRDERR|STWRERR|STRHUP|STPLEX, 0);
4508                                if (error != 0) {
4509                                        mutex_exit(&stp->sd_lock);
4510                                        return (error);
4511                                }
4512                        }
4513                }
4514                stp->sd_flag |= STRPLUMB;
4515                mutex_exit(&stp->sd_lock);
4516        }
4517        return (0);
4518 }

4520 /*
4521  * Complete the plumbing operation associated with stream ‘stp’.
4522  */
4523 void
4524 strendplumb(stdata_t *stp)
4525 {
4526        ASSERT(MUTEX_HELD(&stp->sd_lock));
4527        ASSERT(stp->sd_flag & STRPLUMB);
4528        stp->sd_flag &= ~STRPLUMB;
```



```
4529        cv_broadcast(&stp->sd_monitor);
4530 }

4532 /*
4533  * This describes how the STREAMS framework handles synchronization
4534  * during open/push and close/pop.
4535  * The key interfaces for open and close are qprocson and qprocsoff,
4536  * respectively. While the close case in general is harder both open
4537  * have close have significant similarities.
4538  *
4539  * During close the STREAMS framework has to both ensure that there
4540  * are no stale references to the queue pair (and syncq) that
4541  * are being closed and also provide the guarantees that are documented
4542  * in qprocsoff(9F).
4543  * If there are stale references to the queue that is closing it can
4544  * result in kernel memory corruption or kernel panics.
4545  *
4546  * Note that is it up to the module/driver to ensure that it itself
4547  * does not have any stale references to the closing queues once its close
4548  * routine returns. This includes:
4549  *  - Cancelling any timeout/bufcall/qtimeout/qbufcall callback routines
4550  *    associated with the queues. For timeout and bufcall callbacks the
4551  *    module/driver also has to ensure (or wait for) any callbacks that
4552  *    are in progress.
4553  *  - If the module/driver is using esballoc it has to ensure that any
4554  *    esballoc free functions do not refer to a queue that has closed.
4555  *    (Note that in general the close routine can not wait for the esballoc’ed
4556  *    messages to be freed since that can cause a deadlock.)
4557  *  - Cancelling any interrupts that refer to the closing queues and
4558  *    also ensuring that there are no interrupts in progress that will
4559  *    refer to the closing queues once the close routine returns.
4560  *  - For multiplexors removing any driver global state that refers to
4561  *    the closing queue and also ensuring that there are no threads in
4562  *    the multiplexor that has picked up a queue pointer but not yet
4563  *    finished using it.
4564  *
4565  * In addition, a driver/module can only reference the q_next pointer
4566  * in its open, close, put, or service procedures or in a
4567  * qtimeout/qbufcall callback procedure executing "on" the correct
4568  * stream. Thus it can not reference the q_next pointer in an interrupt
4569  * routine or a timeout, bufcall or esballoc callback routine. Likewise
4570  * it can not reference q_next of a different queue e.g. in a mux that
4571  * passes messages from one queues put/service procedure to another queue.
4572  * In all the cases when the driver/module can not access the q_next
4573  * field it must use the *next* versions e.g. canputnext instead of
4574  * canput(q->q_next) and putnextctl instead of putctl(q->q_next, ...).
4575  *
4576  *
4577  * Assuming that the driver/module conforms to the above constraints
4578  * the STREAMS framework has to avoid stale references to q_next for all
4579  * the framework internal cases which include (but are not limited to):
4580  *  - Threads in canput/canputnext/backenable and elsewhere that are
4581  *    walking q_next.
4582  *  - Messages on a syncq that have a reference to the queue through b_queue.
4583  *  - Messages on an outer perimeter (syncq) that have a reference to the
4584  *    queue through b_queue.
4585  *  - Threads that use q_nfsrv (e.g. canput) to find a queue.
4586  *    Note that only canput and bcanput use q_nfsrv without any locking.
4587  *
4588  * The STREAMS framework providing the qprocsoff(9F) guarantees means that
4589  * after qprocsoff returns, the framework has to ensure that no threads can
4590  * enter the put or service routines for the closing read or write-side queue.
4591  * In addition to preventing "direct" entry into the put procedures
4592  * the framework also has to prevent messages being drained from
4593  * the syncq or the outer perimeter.
4594  * XXX Note that currently qdetach does relies on D_MTOCEXCL as the only
```

```
 * mechanism to prevent qwriter(PERIM_OUTER) from running after
 * qprocsoff has returned.
 * Note that if a module/driver uses put(9F) on one of its own queues
 * it is up to the module/driver to ensure that the put() doesn't
 * get called when the queue is closing.
 *
 *
 * The framework aspects of the above "contract" is implemented by
 * qprocsoff, removeq, and strlock:
 *  - qprocsoff (disable_svc) sets QWCLOSE to prevent runservice from
 *    entering the service procedures.
 *  - strlock acquires the sd_lock and sd_reflock to prevent putnext,
 *    canputnext, backenable etc from dereferencing the q_next that will
 *    soon change.
 *  - strlock waits for sd_refcnt to be zero to wait for e.g. any canputnext
 *    or other q_next walker that uses claimstr/releasestr to finish.
 *  - optionally for every syncq in the stream strlock acquires all the
 *    sq_lock's and waits for all sq_counts to drop to a value that indicates
 *    that no thread executes in the put or service procedures and that no
 *    thread is draining into the module/driver. This ensures that no
 *    open, close, put, service, or qtimeout/qbufcall callback procedure is
 *    currently executing hence no such thread can end up with the old stale
 *    q_next value and no canput/backenable can have the old stale
 *    q_nfsrv/q_next.
 *  - qdetach (wait_svc) makes sure that any scheduled or running threads
 *    have either finished or observed the QWCLOSE flag and gone away.
 */


/*
 * Get all the locks necessary to change q_next.
 *
 * Wait for sd_refcnt to reach 0 and, if sqlist is present, wait for the
 * sq_count of each syncq in the list to drop to sq_rmqcount, indicating that
 * the only threads inside the syncq are threads currently calling removeq().
 * Since threads calling removeq() are in the process of removing their queues
 * from the stream, we do not need to worry about them accessing a stale q_next
 * pointer and thus we do not need to wait for them to exit (in fact, waiting
 * for them can cause deadlock).
 *
 * This routine is subject to starvation since it does not set any flag to
 * prevent threads from entering a module in the stream (i.e. sq_count can
 * increase on some syncq while it is waiting on some other syncq).
 *
 * Assumes that only one thread attempts to call strlock for a given
 * stream. If this is not the case the two threads would deadlock.
 * This assumption is guaranteed since strlock is only called by insertq
 * and removeq and streams plumbing changes are single-threaded for
 * a given stream using the STWOPEN, STRCLOSE, and STRPLUMB flags.
 *
 * For pipes, it is not difficult to atomically designate a pair of streams
 * to be mated. Once mated atomically by the framework the twisted pair remain
 * configured that way until dismantled atomically by the framework.
 * When plumbing takes place on a twisted stream it is necessary to ensure that
 * this operation is done exclusively on the twisted stream since two such
 * operations, each initiated on different ends of the pipe will deadlock
 * waiting for each other to complete.
 *
 * On entry, no locks should be held.
 * The locks acquired and held by strlock depends on a few factors.
 * - If sqlist is non-NULL all the syncq locks in the sqlist will be acquired
 *   and held on exit and all sq_count are at an acceptable level.
 * - In all cases, sd_lock and sd_reflock are acquired and held on exit with
 *   sd_refcnt being zero.
 */
```



```
static void
strlock(struct stdata *stp, sqlist_t *sqlist)
{
        syncql_t *sql, *sql2;
retry:
        /*
         * Wait for any claimstr to go away.
         */
        if (STRMATED(stp)) {
                struct stdata *stp1, *stp2;

                STRLOCKMATES(stp);
                /*
                 * Note that the selection of locking order is not
                 * important, just that they are always acquired in
                 * the same order.  To assure this, we choose this
                 * order based on the value of the pointer, and since
                 * the pointer will not change for the life of this
                 * pair, we will always grab the locks in the same
                 * order (and hence, prevent deadlocks).
                 */
                if (&(stp->sd_lock) > &((stp->sd_mate)->sd_lock)) {
                        stp1 = stp;
                        stp2 = stp->sd_mate;
                } else {
                        stp2 = stp;
                        stp1 = stp->sd_mate;
                }
                mutex_enter(&stp1->sd_reflock);
                if (stp1->sd_refcnt > 0) {
                        STRUNLOCKMATES(stp);
                        cv_wait(&stp1->sd_refmonitor, &stp1->sd_reflock);
                        mutex_exit(&stp1->sd_reflock);
                        goto retry;
                }
                mutex_enter(&stp2->sd_reflock);
                if (stp2->sd_refcnt > 0) {
                        STRUNLOCKMATES(stp);
                        mutex_exit(&stp1->sd_reflock);
                        cv_wait(&stp2->sd_refmonitor, &stp2->sd_reflock);
                        mutex_exit(&stp2->sd_reflock);
                        goto retry;
                }
                STREAM_PUTLOCKS_ENTER(stp1);
                STREAM_PUTLOCKS_ENTER(stp2);
        } else {
                mutex_enter(&stp->sd_lock);
                mutex_enter(&stp->sd_reflock);
                while (stp->sd_refcnt > 0) {
                        mutex_exit(&stp->sd_lock);
                        cv_wait(&stp->sd_refmonitor, &stp->sd_reflock);
                        if (mutex_tryenter(&stp->sd_lock) == 0) {
                                mutex_exit(&stp->sd_reflock);
                                mutex_enter(&stp->sd_lock);
                                mutex_enter(&stp->sd_reflock);
                        }
                }
                STREAM_PUTLOCKS_ENTER(stp);
        }

        if (sqlist == NULL)
                return;

        for (sql = sqlist->sqlist_head; sql; sql = sql->sql_next) {
                syncq_t *sq = sql->sql_sq;
                uint16_t count;
```

```
4728                 mutex_enter(SQLOCK(sq));
4729                 count = sq->sq_count;
4730                 ASSERT(sq->sq_rmqcount <= count);
4731                 SQ_PUTLOCKS_ENTER(sq);
4732                 SUM_SQ_PUTCOUNTS(sq, count);
4733                 if (count == sq->sq_rmqcount)
4734                         continue;

4736                 /* Failed - drop all locks that we have acquired so far */
4737                 if (STRMATED(stp)) {
4738                         STREAM_PUTLOCKS_EXIT(stp);
4739                         STREAM_PUTLOCKS_EXIT(stp->sd_mate);
4740                         STRUNLOCKMATES(stp);
4741                         mutex_exit(&stp->sd_reflock);
4742                         mutex_exit(&stp->sd_mate->sd_reflock);
4743                 } else {
4744                         STREAM_PUTLOCKS_EXIT(stp);
4745                         mutex_exit(&stp->sd_lock);
4746                         mutex_exit(&stp->sd_reflock);
4747                 }
4748                 for (sql2 = sqlist->sqlist_head; sql2 != sql;
4749                     sql2 = sql2->sql_next) {
4750                         SQ_PUTLOCKS_EXIT(sql2->sql_sq);
4751                         mutex_exit(SQLOCK(sql2->sql_sq));
4752                 }

4754                 /*
4755                  * The wait loop below may starve when there are many threads
4756                  * claiming the syncq. This is especially a problem with permod
4757                  * syncqs (IP). To lessen the impact of the problem we increment
4758                  * sq_needexcl and clear fastbits so that putnexts will slow
4759                  * down and call sqenable instead of draining right away.
4760                  */
4761                 sq->sq_needexcl++;
4762                 SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
4763                 while (count > sq->sq_rmqcount) {
4764                         sq->sq_flags |= SQ_WANTWAKEUP;
4765                         SQ_PUTLOCKS_EXIT(sq);
4766                         cv_wait(&sq->sq_wait, SQLOCK(sq));
4767                         count = sq->sq_count;
4768                         SQ_PUTLOCKS_ENTER(sq);
4769                         SUM_SQ_PUTCOUNTS(sq, count);
4770                 }
4771                 sq->sq_needexcl--;
4772                 if (sq->sq_needexcl == 0)
4773                         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
4774                 SQ_PUTLOCKS_EXIT(sq);
4775                 ASSERT(count == sq->sq_rmqcount);
4776                 mutex_exit(SQLOCK(sq));
4777                 goto retry;
4778         }
4779 }

4781 /*
4782  * Drop all the locks that strlock acquired.
4783  */
4784 static void
4785 strunlock(struct stdata *stp, sqlist_t *sqlist)
4786 {
4787         syncql_t *sql;

4789         if (STRMATED(stp)) {
4790                 STREAM_PUTLOCKS_EXIT(stp);
4791                 STREAM_PUTLOCKS_EXIT(stp->sd_mate);
4792                 STRUNLOCKMATES(stp);
```



```
4793                 mutex_exit(&stp->sd_reflock);
4794                 mutex_exit(&stp->sd_mate->sd_reflock);
4795         } else {
4796                 STREAM_PUTLOCKS_EXIT(stp);
4797                 mutex_exit(&stp->sd_lock);
4798                 mutex_exit(&stp->sd_reflock);
4799         }

4801         if (sqlist == NULL)
4802                 return;

4804         for (sql = sqlist->sqlist_head; sql; sql = sql->sql_next) {
4805                 SQ_PUTLOCKS_EXIT(sql->sql_sq);
4806                 mutex_exit(SQLOCK(sql->sql_sq));
4807         }
4808 }

4810 /*
4811  * When the module has service procedure, we need check if the next
4812  * module which has service procedure is in flow control to trigger
4813  * the backenable.
4814  */
4815 static void
4816 backenable_insertedq(queue_t *q)
4817 {
4818         qband_t *qbp;

4820         claimstr(q);
4821         if (q->q_qinfo->qi_srvp != NULL && q->q_next != NULL) {
4822                 if (q->q_next->q_nfsrv->q_flag & QWANTW)
4823                         backenable(q, 0);

4825                 qbp = q->q_next->q_nfsrv->q_bandp;
4826                 for (; qbp != NULL; qbp = qbp->qb_next)
4827                         if ((qbp->qb_flag & QB_WANTW) && qbp->qb_first != NULL)
4828                                 backenable(q, qbp->qb_first->b_band);
4829         }
4830         releasestr(q);
4831 }

4833 /*
4834  * Given two read queues, insert a new single one after another.
4835  *
4836  * This routine acquires all the necessary locks in order to change
4837  * q_next and related pointer using strlock().
4838  * It depends on the stream head ensuring that there are no concurrent
4839  * insertq or removeq on the same stream. The stream head ensures this
4840  * using the flags STWOPEN, STRCLOSE, and STRPLUMB.
4841  *
4842  * Note that no syncq locks are held during the q_next change. This is
4843  * applied to all streams since, unlike removeq, there is no problem of stale
4844  * pointers when adding a module to the stream. Thus drivers/modules that do a
4845  * canput(rq->q_next) would never get a closed/freed queue pointer even if we
4846  * applied this optimization to all streams.
4847  */
4848 void
4849 insertq(struct stdata *stp, queue_t *new)
4850 {
4851         queue_t *after;
4852         queue_t *wafter;
4853         queue_t *wnew = _WR(new);
4854         boolean_t have_fifo = B_FALSE;

4856         if (new->q_flag & _QINSERTING) {
4857                 ASSERT(stp->sd_vnode->v_type != VFIFO);
4858                 after = new->q_next;
```

```
4859                wafter = _WR(new->q_next);
4860        } else {
4861                after = _RD(stp->sd_wrq);
4862                wafter = stp->sd_wrq;
4863        }

4865        TRACE_2(TR_FAC_STREAMS_FR, TR_INSERTQ,
4866            "insertq:%p, %p", after, new);
4867        ASSERT(after->q_flag & QREADR);
4868        ASSERT(new->q_flag & QREADR);

4870        strlock(stp, NULL);

4872        /* Do we have a FIFO? */
4873        if (wafter->q_next == after) {
4874                have_fifo = B_TRUE;
4875                wnew->q_next = new;
4876        } else {
4877                wnew->q_next = wafter->q_next;
4878        }
4879        new->q_next = after;

4881        set_nfsrv_ptr(new, wnew, after, wafter);
4882        /*
4883         * set_nfsrv_ptr() needs to know if this is an insertion or not,
4884         * so only reset this flag after calling it.
4885         */
4886        new->q_flag &= ~_QINSERTING;

4888        if (have_fifo) {
4889                wafter->q_next = wnew;
4890        } else {
4891                if (wafter->q_next)
4892                        _OTHERQ(wafter->q_next)->q_next = new;
4893                wafter->q_next = wnew;
4894        }

4896        set_qend(new);
4897        /* The QEND flag might have to be updated for the upstream guy */
4898        set_qend(after);

4900        ASSERT(_SAMESTR(new) == O_SAMESTR(new));
4901        ASSERT(_SAMESTR(wnew) == O_SAMESTR(wnew));
4902        ASSERT(_SAMESTR(after) == O_SAMESTR(after));
4903        ASSERT(_SAMESTR(wafter) == O_SAMESTR(wafter));
4904        strsetuio(stp);

4906        /*
4907         * If this was a module insertion, bump the push count.
4908         */
4909        if (!(new->q_flag & QISDRV))
4910                stp->sd_pushcnt++;

4912        strunlock(stp, NULL);

4914        /* check if the write Q needs backenable */
4915        backenable_insertedq(wnew);

4917        /* check if the read Q needs backenable */
4918        backenable_insertedq(new);
4919 }

4921 /*
4922  * Given a read queue, unlink it from any neighbors.
4923  *
4924  * This routine acquires all the necessary locks in order to
```



```
4925  * change q_next and related pointers and also guard against
4926  * stale references (e.g. through q_next) to the queue that
4927  * is being removed. It also plays part of the role in ensuring
4928  * that the module's/driver's put procedure doesn't get called
4929  * after qprocsoff returns.
4930  *
4931  * Removeq depends on the stream head ensuring that there are
4932  * no concurrent insertq or removeq on the same stream. The
4933  * stream head ensures this using the flags STWOPEN, STRCLOSE and
4934  * STRPLUMB.
4935  *
4936  * The set of locks needed to remove the queue is different in
4937  * different cases:
4938  *
4939  * Acquire sd_lock, sd_reflock, and all the syncq locks in the stream after
4940  * waiting for the syncq reference count to drop to 0 indicating that no
4941  * non-close threads are present anywhere in the stream. This ensures that any
4942  * module/driver can reference q_next in its open, close, put, or service
4943  * procedures.
4944  *
4945  * The sq_rmqcount counter tracks the number of threads inside removeq().
4946  * strlock() ensures that there is either no threads executing inside perimeter
4947  * or there is only a thread calling qprocsoff().
4948  *
4949  * strlock() compares the value of sq_count with the number of threads inside
4950  * removeq() and waits until sq_count is equal to sq_rmqcount. We need to wakeup
4951  * any threads waiting in strlock() when the sq_rmqcount increases.
4952  */

4954 void
4955 removeq(queue_t *qp)
4956 {
4957         queue_t *wqp = _WR(qp);
4958         struct stdata *stp = STREAM(qp);
4959         sqlist_t *sqlist = NULL;
4960         boolean_t isdriver;
4961         int moved;
4962         syncq_t *sq = qp->q_syncq;
4963         syncq_t *wsq = wqp->q_syncq;

4965         ASSERT(stp);

4967         TRACE_2(TR_FAC_STREAMS_FR, TR_REMOVEQ,
4968             "removeq:%p %p", qp, wqp);
4969         ASSERT(qp->q_flag&QREADR);

4971         /*
4972          * For queues using Synchronous streams, we must wait for all threads in
4973          * rwnext() to drain out before proceeding.
4974          */
4975         if (qp->q_flag & QSYNCSTR) {
4976                 /* First, we need wakeup any threads blocked in rwnext() */
4977                 mutex_enter(SQLOCK(sq));
4978                 if (sq->sq_flags & SQ_WANTWAKEUP) {
4979                         sq->sq_flags &= ~SQ_WANTWAKEUP;
4980                         cv_broadcast(&sq->sq_wait);
4981                 }
4982                 mutex_exit(SQLOCK(sq));

4984                 if (wsq != sq) {
4985                         mutex_enter(SQLOCK(wsq));
4986                         if (wsq->sq_flags & SQ_WANTWAKEUP) {
4987                                 wsq->sq_flags &= ~SQ_WANTWAKEUP;
4988                                 cv_broadcast(&wsq->sq_wait);
4989                         }
4990                         mutex_exit(SQLOCK(wsq));
```

```
4991                }

4993                mutex_enter(QLOCK(qp));
4994                while (qp->q_rwcnt > 0) {
4995                        qp->q_flag |= QWANTRMQSYNC;
4996                        cv_wait(&qp->q_wait, QLOCK(qp));
4997                }
4998                mutex_exit(QLOCK(qp));

5000                mutex_enter(QLOCK(wqp));
5001                while (wqp->q_rwcnt > 0) {
5002                        wqp->q_flag |= QWANTRMQSYNC;
5003                        cv_wait(&wqp->q_wait, QLOCK(wqp));
5004                }
5005                mutex_exit(QLOCK(wqp));
5006        }

5008        mutex_enter(SQLOCK(sq));
5009        sq->sq_rmqcount++;
5010        if (sq->sq_flags & SQ_WANTWAKEUP) {
5011                sq->sq_flags &= ~SQ_WANTWAKEUP;
5012                cv_broadcast(&sq->sq_wait);
5013        }
5014        mutex_exit(SQLOCK(sq));

5016        isdriver = (qp->q_flag & QISDRV);

5018        sqlist = sqlist_build(qp, stp, STRMATED(stp));
5019        strlock(stp, sqlist);

5021        reset_nfsrv_ptr(qp, wqp);

5023        ASSERT(wqp->q_next == NULL || backq(qp)->q_next == qp);
5024        ASSERT(qp->q_next == NULL || backq(wqp)->q_next == wqp);
5025        /* Do we have a FIFO? */
5026        if (wqp->q_next == qp) {
5027                stp->sd_wrq->q_next = _RD(stp->sd_wrq);
5028        } else {
5029                if (wqp->q_next)
5030                        backq(qp)->q_next = qp->q_next;
5031                if (qp->q_next)
5032                        backq(wqp)->q_next = wqp->q_next;
5033        }

5035        /* The QEND flag might have to be updated for the upstream guy */
5036        if (qp->q_next)
5037                set_qend(qp->q_next);

5039        ASSERT(_SAMESTR(stp->sd_wrq) == O_SAMESTR(stp->sd_wrq));
5040        ASSERT(_SAMESTR(_RD(stp->sd_wrq)) == O_SAMESTR(_RD(stp->sd_wrq)));

5042        /*
5043         * Move any messages destined for the put procedures to the next
5044         * syncq in line. Otherwise free them.
5045         */
5046        moved = 0;
5047        /*
5048         * Quick check to see whether there are any messages or events.
5049         */
5050        if (qp->q_syncqmsgs != 0 || (qp->q_syncq->sq_flags & SQ_EVENTS))
5051                moved += propagate_syncq(qp);
5052        if (wqp->q_syncqmsgs != 0 ||
5053            (wqp->q_syncq->sq_flags & SQ_EVENTS))
5054                moved += propagate_syncq(wqp);

5056        strsetuio(stp);
```



```
5058        /*
5059         * If this was a module removal, decrement the push count.
5060         */
5061        if (!isdriver)
5062                stp->sd_pushcnt--;

5064        strunlock(stp, sqlist);
5065        sqlist_free(sqlist);

5067        /*
5068         * Make sure any messages that were propagated are drained.
5069         * Also clear any QFULL bit caused by messages that were propagated.
5070         */

5072        if (qp->q_next != NULL) {
5073                clr_qfull(qp);
5074                /*
5075                 * For the driver calling qprocsoff, propagate_syncq
5076                 * frees all the messages instead of putting it in
5077                 * the stream head
5078                 */
5079                if (!isdriver && (moved > 0))
5080                        emptysq(qp->q_next->q_syncq);
5081        }
5082        if (wqp->q_next != NULL) {
5083                clr_qfull(wqp);
5084                /*
5085                 * We come here for any pop of a module except for the
5086                 * case of driver being removed. We don't call emptysq
5087                 * if we did not move any messages. This will avoid holding
5088                 * PERMOD syncq locks in emptysq
5089                 */
5090                if (moved > 0)
5091                        emptysq(wqp->q_next->q_syncq);
5092        }

5094        mutex_enter(SQLOCK(sq));
5095        sq->sq_rmqcount--;
5096        mutex_exit(SQLOCK(sq));
5097 }

5099 /*
5100  * Prevent further entry by setting a flag (like SQ_FROZEN, SQ_BLOCKED or
5101  * SQ_WRITER) on a syncq.
5102  * If maxcnt is not -1 it assumes that caller has "maxcnt" claim(s) on the
5103  * sync queue and waits until sq_count reaches maxcnt.
5104  *
5105  * If maxcnt is -1 there's no need to grab sq_putlocks since the caller
5106  * does not care about putnext threads that are in the middle of calling put
5107  * entry points.
5108  *
5109  * This routine is used for both inner and outer syncqs.
5110  */
5111 static void
5112 blocksq(syncq_t *sq, ushort_t flag, int maxcnt)
5113 {
5114         uint16_t count = 0;

5116         mutex_enter(SQLOCK(sq));
5117         /*
5118          * Wait for SQ_FROZEN/SQ_BLOCKED to be reset.
5119          * SQ_FROZEN will be set if there is a frozen stream that has a
5120          * queue which also refers to this "shared" syncq.
5121          * SQ_BLOCKED will be set if there is "off" queue which also
5122          * refers to this "shared" syncq.
```

```
5123          */
5124         if (maxcnt != -1) {
5125                 count = sq->sq_count;
5126                 SQ_PUTLOCKS_ENTER(sq);
5127                 SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
5128                 SUM_SQ_PUTCOUNTS(sq, count);
5129         }
5130         sq->sq_needexcl++;
5131         ASSERT(sq->sq_needexcl != 0);   /* wraparound */

5133         while ((sq->sq_flags & flag) ||
5134             (maxcnt != -1 && count > (unsigned)maxcnt)) {
5135                 sq->sq_flags |= SQ_WANTWAKEUP;
5136                 if (maxcnt != -1) {
5137                         SQ_PUTLOCKS_EXIT(sq);
5138                 }
5139                 cv_wait(&sq->sq_wait, SQLOCK(sq));
5140                 if (maxcnt != -1) {
5141                         count = sq->sq_count;
5142                         SQ_PUTLOCKS_ENTER(sq);
5143                         SUM_SQ_PUTCOUNTS(sq, count);
5144                 }
5145         }
5146         sq->sq_needexcl--;
5147         sq->sq_flags |= flag;
5148         ASSERT(maxcnt == -1 || count == maxcnt);
5149         if (maxcnt != -1) {
5150                 if (sq->sq_needexcl == 0) {
5151                         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
5152                 }
5153                 SQ_PUTLOCKS_EXIT(sq);
5154         } else if (sq->sq_needexcl == 0) {
5155                 SQ_PUTCOUNT_SETFAST(sq);
5156         }

5158         mutex_exit(SQLOCK(sq));
5159 }

5161 /*
5162  * Reset a flag that was set with blocksq.
5163  *
5164  * Can not use this routine to reset SQ_WRITER.
5165  *
5166  * If "isouter" is set then the syncq is assumed to be an outer perimeter
5167  * and drain_syncq is not called. Instead we rely on the qwriter_outer thread
5168  * to handle the queued qwriter operations.
5169  *
5170  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5171  * sq_putlocks are used.
5172  */
5173 static void
5174 unblocksq(syncq_t *sq, uint16_t resetflag, int isouter)
5175 {
5176         uint16_t flags;

5178         mutex_enter(SQLOCK(sq));
5179         ASSERT(resetflag != SQ_WRITER);
5180         ASSERT(sq->sq_flags & resetflag);
5181         flags = sq->sq_flags & ~resetflag;
5182         sq->sq_flags = flags;
5183         if (flags & (SQ_QUEUED | SQ_WANTWAKEUP)) {
5184                 if (flags & SQ_WANTWAKEUP) {
5185                         flags &= ~SQ_WANTWAKEUP;
5186                         cv_broadcast(&sq->sq_wait);
5187                 }
5188                 sq->sq_flags = flags;
```



```
5189                 if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5190                         if (!isouter) {
5191                                 /* drain_syncq drops SQLOCK */
5192                                 drain_syncq(sq);
5193                                 return;
5194                         }
5195                 }
5196         }
5197         mutex_exit(SQLOCK(sq));
5198 }

5200 /*
5201  * Reset a flag that was set with blocksq.
5202  * Does not drain the syncq. Use emptysq() for that.
5203  * Returns 1 if SQ_QUEUED is set. Otherwise 0.
5204  *
5205  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5206  * sq_putlocks are used.
5207  */
5208 static int
5209 dropsq(syncq_t *sq, uint16_t resetflag)
5210 {
5211         uint16_t flags;

5213         mutex_enter(SQLOCK(sq));
5214         ASSERT(sq->sq_flags & resetflag);
5215         flags = sq->sq_flags & ~resetflag;
5216         if (flags & SQ_WANTWAKEUP) {
5217                 flags &= ~SQ_WANTWAKEUP;
5218                 cv_broadcast(&sq->sq_wait);
5219         }
5220         sq->sq_flags = flags;
5221         mutex_exit(SQLOCK(sq));
5222         if (flags & SQ_QUEUED)
5223                 return (1);
5224         return (0);
5225 }

5227 /*
5228  * Empty all the messages on a syncq.
5229  *
5230  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5231  * sq_putlocks are used.
5232  */
5233 static void
5234 emptysq(syncq_t *sq)
5235 {
5236         uint16_t flags;

5238         mutex_enter(SQLOCK(sq));
5239         flags = sq->sq_flags;
5240         if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5241                 /*
5242                  * To prevent potential recursive invocation of drain_syncq we
5243                  * do not call drain_syncq if count is non-zero.
5244                  */
5245                 if (sq->sq_count == 0) {
5246                         /* drain_syncq() drops SQLOCK */
5247                         drain_syncq(sq);
5248                         return;
5249                 } else
5250                         sqenable(sq);
5251         }
5252         mutex_exit(SQLOCK(sq));
5253 }
```

```
5255 /*
5256  * Ordered insert while removing duplicates.
5257  */
5258 static void
5259 sqlist_insert(sqlist_t *sqlist, syncq_t *sqp)
5260 {
5261         syncql_t *sqlp, **prev_sqlpp, *new_sqlp;

5263         prev_sqlpp = &sqlist->sqlist_head;
5264         while ((sqlp = *prev_sqlpp) != NULL) {
5265                 if (sqlp->sql_sq >= sqp) {
5266                         if (sqlp->sql_sq == sqp)        /* duplicate */
5267                                 return;
5268                         break;
5269                 }
5270                 prev_sqlpp = &sqlp->sql_next;
5271         }
5272         new_sqlp = &sqlist->sqlist_array[sqlist->sqlist_index++];
5273         ASSERT((char *)new_sqlp < (char *)sqlist + sqlist->sqlist_size);
5274         new_sqlp->sql_next = sqlp;
5275         new_sqlp->sql_sq = sqp;
5276         *prev_sqlpp = new_sqlp;
5277 }

5279 /*
5280  * Walk the write side queues until we hit either the driver
5281  * or a twist in the stream (_SAMESTR will return false in both
5282  * these cases) then turn around and walk the read side queues
5283  * back up to the stream head.
5284  */
5285 static void
5286 sqlist_insertall(sqlist_t *sqlist, queue_t *q)
5287 {
5288         while (q != NULL) {
5289                 sqlist_insert(sqlist, q->q_syncq);

5291                 if (_SAMESTR(q))
5292                         q = q->q_next;
5293                 else if (!(q->q_flag & QREADR))
5294                         q = _RD(q);
5295                 else
5296                         q = NULL;
5297         }
5298 }

5300 /*
5301  * Allocate and build a list of all syncqs in a stream and the syncq(s)
5302  * associated with the "q" parameter. The resulting list is sorted in a
5303  * canonical order and is free of duplicates.
5304  * Assumes the passed queue is a _RD(q).
5305  */
5306 static sqlist_t *
5307 sqlist_build(queue_t *q, struct stdata *stp, boolean_t do_twist)
5308 {
5309         sqlist_t *sqlist = sqlist_alloc(stp, KM_SLEEP);

5311         /*
5312          * start with the current queue/qpair
5313          */
5314         ASSERT(q->q_flag & QREADR);

5316         sqlist_insert(sqlist, q->q_syncq);
5317         sqlist_insert(sqlist, _WR(q)->q_syncq);

5319         sqlist_insertall(sqlist, stp->sd_wrq);
5320         if (do_twist)
```



```
5321                 sqlist_insertall(sqlist, stp->sd_mate->sd_wrq);

5323         return (sqlist);
5324 }

5326 static sqlist_t *
5327 sqlist_alloc(struct stdata *stp, int kmflag)
5328 {
5329         size_t sqlist_size;
5330         sqlist_t *sqlist;

5332         /*
5333          * Allocate 2 syncql_t's for each pushed module. Note that
5334          * the sqlist_t structure already has 4 syncql_t's built in:
5335          * 2 for the stream head, and 2 for the driver/other stream head.
5336          */
5337         sqlist_size = 2 * sizeof (syncql_t) * stp->sd_pushcnt +
5338             sizeof (sqlist_t);
5339         if (STRMATED(stp))
5340                 sqlist_size += 2 * sizeof (syncql_t) * stp->sd_mate->sd_pushcnt;
5341         sqlist = kmem_alloc(sqlist_size, kmflag);

5343         sqlist->sqlist_head = NULL;
5344         sqlist->sqlist_size = sqlist_size;
5345         sqlist->sqlist_index = 0;

5347         return (sqlist);
5348 }

5350 /*
5351  * Free the list created by sqlist_alloc()
5352  */
5353 static void
5354 sqlist_free(sqlist_t *sqlist)
5355 {
5356         kmem_free(sqlist, sqlist->sqlist_size);
5357 }

5359 /*
5360  * Prevent any new entries into any syncq in this stream.
5361  * Used by freezestr.
5362  */
5363 void
5364 strblock(queue_t *q)
5365 {
5366         struct stdata   *stp;
5367         syncql_t        *sql;
5368         sqlist_t        *sqlist;

5370         q = _RD(q);

5372         stp = STREAM(q);
5373         ASSERT(stp != NULL);

5375         /*
5376          * Get a sorted list with all the duplicates removed containing
5377          * all the syncqs referenced by this stream.
5378          */
5379         sqlist = sqlist_build(q, stp, B_FALSE);
5380         for (sql = sqlist->sqlist_head; sql != NULL; sql = sql->sql_next)
5381                 blocksq(sql->sql_sq, SQ_FROZEN, -1);
5382         sqlist_free(sqlist);
5383 }

5385 /*
5386  * Release the block on new entries into this stream
```

```
 */
void
strunblock(queue_t *q)
{
        struct stdata   *stp;
        syncql_t        *sql;
        sqlist_t        *sqlist;
        int             drain_needed;

        q = _RD(q);

        /*
         * Get a sorted list with all the duplicates removed containing
         * all the syncqs referenced by this stream.
         * Have to drop the SQ_FROZEN flag on all the syncqs before
         * starting to drain them; otherwise the draining might
         * cause a freezestr in some module on the stream (which
         * would deadlock).
         */
        stp = STREAM(q);
        ASSERT(stp != NULL);
        sqlist = sqlist_build(q, stp, B_FALSE);
        drain_needed = 0;
        for (sql = sqlist->sqlist_head; sql != NULL; sql = sql->sql_next)
                drain_needed += dropsq(sql->sql_sq, SQ_FROZEN);
        if (drain_needed) {
                for (sql = sqlist->sqlist_head; sql != NULL;
                    sql = sql->sql_next)
                        emptysq(sql->sql_sq);
        }
        sqlist_free(sqlist);
}

#ifdef DEBUG
static int
qprocsareon(queue_t *rq)
{
        if (rq->q_next == NULL)
                return (0);
        return (_WR(rq->q_next)->q_next == _WR(rq));
}

int
qclaimed(queue_t *q)
{
        uint_t count;

        count = q->q_syncq->sq_count;
        SUM_SQ_PUTCOUNTS(q->q_syncq, count);
        return (count != 0);
}

/*
 * Check if anyone has frozen this stream with freezestr
 */
int
frozenstr(queue_t *q)
{
        return ((q->q_syncq->sq_flags & SQ_FROZEN) != 0);
}
#endif /* DEBUG */

/*
 * Enter a queue.
 * Obsoleted interface. Should not be used.
 */
```



```
void
enterq(queue_t *q)
{
        entersq(q->q_syncq, SQ_CALLBACK);
}

void
leaveq(queue_t *q)
{
        leavesq(q->q_syncq, SQ_CALLBACK);
}

/*
 * Enter a perimeter. c_inner and c_outer specifies which concurrency bits
 * to check.
 * Wait if SQ_QUEUED is set to preserve ordering between messages and qwriter
 * calls and the running of open, close and service procedures.
 *
 * If c_inner bit is set no need to grab sq_putlocks since we don't care
 * if other threads have entered or are entering put entry point.
 *
 * If c_inner bit is set it might have been possible to use
 * sq_putlocks/sq_putcounts instead of SQLOCK/sq_count (e.g. to optimize
 * open/close path for IP) but since the count may need to be decremented in
 * qwait() we wouldn't know which counter to decrement. Currently counter is
 * selected by current cpu_seqid and current CPU can change at any moment. XXX
 * in the future we might use curthread id bits to select the counter and this
 * would stay constant across routine calls.
 */
void
entersq(syncq_t *sq, int entrypoint)
{
        uint16_t        count = 0;
        uint16_t        flags;
        uint16_t        waitflags = SQ_STAYAWAY | SQ_EVENTS | SQ_EXCL;
        uint16_t        type;
        uint_t          c_inner = entrypoint & SQ_CI;
        uint_t          c_outer = entrypoint & SQ_CO;

        /*
         * Increment ref count to keep closes out of this queue.
         */
        ASSERT(sq);
        ASSERT(c_inner && c_outer);
        mutex_enter(SQLOCK(sq));
        flags = sq->sq_flags;
        type = sq->sq_type;
        if (!(type & c_inner)) {
                /* Make sure all putcounts now use slowlock. */
                count = sq->sq_count;
                SQ_PUTLOCKS_ENTER(sq);
                SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
                SUM_SQ_PUTCOUNTS(sq, count);
                sq->sq_needexcl++;
                ASSERT(sq->sq_needexcl != 0);   /* wraparound */
                waitflags |= SQ_MESSAGES;
        }
        /*
         * Wait until we can enter the inner perimeter.
         * If we want exclusive access we wait until sq_count is 0.
         * We have to do this before entering the outer perimeter in order
         * to preserve put/close message ordering.
         */
        while ((flags & waitflags) || (!(type & c_inner) && count != 0)) {
                sq->sq_flags = flags | SQ_WANTWAKEUP;
                if (!(type & c_inner)) {
```

```
5519                        SQ_PUTLOCKS_EXIT(sq);
5520                }
5521                cv_wait(&sq->sq_wait, SQLOCK(sq));
5522                if (!(type & c_inner)) {
5523                        count = sq->sq_count;
5524                        SQ_PUTLOCKS_ENTER(sq);
5525                        SUM_SQ_PUTCOUNTS(sq, count);
5526                }
5527                flags = sq->sq_flags;
5528        }

5530        if (!(type & c_inner)) {
5531                ASSERT(sq->sq_needexcl > 0);
5532                sq->sq_needexcl--;
5533                if (sq->sq_needexcl == 0) {
5534                        SQ_PUTCOUNT_SETFAST_LOCKED(sq);
5535                }
5536        }

5538        /* Check if we need to enter the outer perimeter */
5539        if (!(type & c_outer)) {
5540                /*
5541                 * We have to enter the outer perimeter exclusively before
5542                 * we can increment sq_count to avoid deadlock. This implies
5543                 * that we have to re-check sq_flags and sq_count.
5544                 *
5545                 * is it possible to have c_inner set when c_outer is not set?
5546                 */
5547                if (!(type & c_inner)) {
5548                        SQ_PUTLOCKS_EXIT(sq);
5549                }
5550                mutex_exit(SQLOCK(sq));
5551                outer_enter(sq->sq_outer, SQ_GOAWAY);
5552                mutex_enter(SQLOCK(sq));
5553                flags = sq->sq_flags;
5554                /*
5555                 * there should be no need to recheck sq_putcounts
5556                 * because outer_enter() has already waited for them to clear
5557                 * after setting SQ_WRITER.
5558                 */
5559                count = sq->sq_count;
5560 #ifdef DEBUG
5561                /*
5562                 * SUMCHECK_SQ_PUTCOUNTS should return the sum instead
5563                 * of doing an ASSERT internally. Others should do
5564                 * something like
5565                 *       ASSERT(SUMCHECK_SQ_PUTCOUNTS(sq) == 0);
5566                 * without the need to #ifdef DEBUG it.
5567                 */
5568                SUMCHECK_SQ_PUTCOUNTS(sq, 0);
5569 #endif
5570                while ((flags & (SQ_EXCL|SQ_BLOCKED|SQ_FROZEN)) ||
5571                    (!(type & c_inner) && count != 0)) {
5572                        sq->sq_flags = flags | SQ_WANTWAKEUP;
5573                        cv_wait(&sq->sq_wait, SQLOCK(sq));
5574                        count = sq->sq_count;
5575                        flags = sq->sq_flags;
5576                }
5577        }

5579        sq->sq_count++;
5580        ASSERT(sq->sq_count != 0);      /* Wraparound */
5581        if (!(type & c_inner)) {
5582                /* Exclusive entry */
5583                ASSERT(sq->sq_count == 1);
5584                sq->sq_flags |= SQ_EXCL;
```



```
5585                if (type & c_outer) {
5586                        SQ_PUTLOCKS_EXIT(sq);
5587                }
5588        }
5589        mutex_exit(SQLOCK(sq));
5590 }

5592 /*
5593  * Leave a syncq. Announce to framework that closes may proceed.
5594  * c_inner and c_outer specify which concurrency bits to check.
5595  *
5596  * Must never be called from driver or module put entry point.
5597  *
5598  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5599  * sq_putlocks are used.
5600  */
5601 void
5602 leavesq(syncq_t *sq, int entrypoint)
5603 {
5604        uint16_t        flags;
5605        uint16_t        type;
5606        uint_t          c_outer = entrypoint & SQ_CO;
5607 #ifdef DEBUG
5608        uint_t          c_inner = entrypoint & SQ_CI;
5609 #endif

5611        /*
5612         * Decrement ref count, drain the syncq if possible, and wake up
5613         * any waiting close.
5614         */
5615        ASSERT(sq);
5616        ASSERT(c_inner && c_outer);
5617        mutex_enter(SQLOCK(sq));
5618        flags = sq->sq_flags;
5619        type = sq->sq_type;
5620        if (flags & (SQ_QUEUED|SQ_WANTWAKEUP|SQ_WANTEXWAKEUP)) {

5622                if (flags & SQ_WANTWAKEUP) {
5623                        flags &= ~SQ_WANTWAKEUP;
5624                        cv_broadcast(&sq->sq_wait);
5625                }
5626                if (flags & SQ_WANTEXWAKEUP) {
5627                        flags &= ~SQ_WANTEXWAKEUP;
5628                        cv_broadcast(&sq->sq_exitwait);
5629                }

5631                if ((flags & SQ_QUEUED) && !(flags & SQ_STAYAWAY)) {
5632                        /*
5633                         * The syncq needs to be drained. "Exit" the syncq
5634                         * before calling drain_syncq.
5635                         */
5636                        ASSERT(sq->sq_count != 0);
5637                        sq->sq_count--;
5638                        ASSERT((flags & SQ_EXCL) || (type & c_inner));
5639                        sq->sq_flags = flags & ~SQ_EXCL;
5640                        drain_syncq(sq);
5641                        ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
5642                        /* Check if we need to exit the outer perimeter */
5643                        /* XXX will this ever be true? */
5644                        if (!(type & c_outer))
5645                                outer_exit(sq->sq_outer);
5646                        return;
5647                }
5648        }
5649        ASSERT(sq->sq_count != 0);
5650        sq->sq_count--;
```

```
5651         ASSERT((flags & SQ_EXCL) || (type & c_inner));
5652         sq->sq_flags = flags & ~SQ_EXCL;
5653         mutex_exit(SQLOCK(sq));

5655         /* Check if we need to exit the outer perimeter */
5656         if (!(sq->sq_type & c_outer))
5657                 outer_exit(sq->sq_outer);
5658 }

5660 /*
5661  * Prevent q_next from changing in this stream by incrementing sq_count.
5662  *
5663  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5664  * sq_putlocks are used.
5665  */
5666 void
5667 claimq(queue_t *qp)
5668 {
5669         syncq_t *sq = qp->q_syncq;

5671         mutex_enter(SQLOCK(sq));
5672         sq->sq_count++;
5673         ASSERT(sq->sq_count != 0);      /* Wraparound */
5674         mutex_exit(SQLOCK(sq));
5675 }

5677 /*
5678  * Undo claimq.
5679  *
5680  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5681  * sq_putlocks are used.
5682  */
5683 void
5684 releaseq(queue_t *qp)
5685 {
5686         syncq_t *sq = qp->q_syncq;
5687         uint16_t flags;

5689         mutex_enter(SQLOCK(sq));
5690         ASSERT(sq->sq_count > 0);
5691         sq->sq_count--;

5693         flags = sq->sq_flags;
5694         if (flags & (SQ_WANTWAKEUP|SQ_QUEUED)) {
5695                 if (flags & SQ_WANTWAKEUP) {
5696                         flags &= ~SQ_WANTWAKEUP;
5697                         cv_broadcast(&sq->sq_wait);
5698                 }
5699                 sq->sq_flags = flags;
5700                 if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5701                         /*
5702                          * To prevent potential recursive invocation of
5703                          * drain_syncq we do not call drain_syncq if count is
5704                          * non-zero.
5705                          */
5706                         if (sq->sq_count == 0) {
5707                                 drain_syncq(sq);
5708                                 return;
5709                         } else
5710                                 sqenable(sq);
5711                 }
5712         }
5713         mutex_exit(SQLOCK(sq));
5714 }

5716 /*
```



```
5717  * Prevent q_next from changing in this stream by incrementing sd_refcnt.
5718  */
5719 void
5720 claimstr(queue_t *qp)
5721 {
5722         struct stdata *stp = STREAM(qp);

5724         mutex_enter(&stp->sd_reflock);
5725         stp->sd_refcnt++;
5726         ASSERT(stp->sd_refcnt != 0);    /* Wraparound */
5727         mutex_exit(&stp->sd_reflock);
5728 }

5730 /*
5731  * Undo claimstr.
5732  */
5733 void
5734 releasestr(queue_t *qp)
5735 {
5736         struct stdata *stp = STREAM(qp);

5738         mutex_enter(&stp->sd_reflock);
5739         ASSERT(stp->sd_refcnt != 0);
5740         if (--stp->sd_refcnt == 0)
5741                 cv_broadcast(&stp->sd_refmonitor);
5742         mutex_exit(&stp->sd_reflock);
5743 }

5745 static syncq_t *
5746 new_syncq(void)
5747 {
5748         return (kmem_cache_alloc(syncq_cache, KM_SLEEP));
5749 }

5751 static void
5752 free_syncq(syncq_t *sq)
5753 {
5754         ASSERT(sq->sq_head == NULL);
5755         ASSERT(sq->sq_outer == NULL);
5756         ASSERT(sq->sq_callbpend == NULL);
5757         ASSERT((sq->sq_onext == NULL && sq->sq_oprev == NULL) ||
5758             (sq->sq_onext == sq && sq->sq_oprev == sq));

5760         if (sq->sq_ciputctrl != NULL) {
5761                 ASSERT(sq->sq_nciputctrl == n_ciputctrl - 1);
5762                 SUMCHECK_CIPUTCTRL_COUNTS(sq->sq_ciputctrl,
5763                     sq->sq_nciputctrl, 0);
5764                 ASSERT(ciputctrl_cache != NULL);
5765                 kmem_cache_free(ciputctrl_cache, sq->sq_ciputctrl);
5766         }

5768         sq->sq_tail = NULL;
5769         sq->sq_evhead = NULL;
5770         sq->sq_evtail = NULL;
5771         sq->sq_ciputctrl = NULL;
5772         sq->sq_nciputctrl = 0;
5773         sq->sq_count = 0;
5774         sq->sq_rmqcount = 0;
5775         sq->sq_callbflags = 0;
5776         sq->sq_cancelid = 0;
5777         sq->sq_next = NULL;
5778         sq->sq_needexcl = 0;
5779         sq->sq_svcflags = 0;
5780         sq->sq_nqueues = 0;
5781         sq->sq_pri = 0;
5782         sq->sq_onext = NULL;
```

```
5783         sq->sq_oprev = NULL;
5784         sq->sq_flags = 0;
5785         sq->sq_type = 0;
5786         sq->sq_servcount = 0;

5788         kmem_cache_free(syncq_cache, sq);
5789 }

5791 /* Outer perimeter code */

5793 /*
5794  * The outer syncq uses the fields and flags in the syncq slightly
5795  * differently from the inner syncqs.
5796  *      sq_count        Incremented when there are pending or running
5797  *                      writers at the outer perimeter to prevent the set of
5798  *                      inner syncqs that belong to the outer perimeter from
5799  *                      changing.
5800  *      sq_head/tail    List of deferred qwriter(OUTER) operations.
5801  *
5802  *      SQ_BLOCKED      Set to prevent traversing of sq_next,sq_prev while
5803  *                      inner syncqs are added to or removed from the
5804  *                      outer perimeter.
5805  *      SQ_QUEUED       sq_head/tail has messages or events queued.
5806  *
5807  *      SQ_WRITER       A thread is currently traversing all the inner syncqs
5808  *                      setting the SQ_WRITER flag.
5809  */

5811 /*
5812  * Get write access at the outer perimeter.
5813  * Note that read access is done by entersq, putnext, and put by simply
5814  * incrementing sq_count in the inner syncq.
5815  *
5816  * Waits until "flags" is no longer set in the outer to prevent multiple
5817  * threads from having write access at the same time. SQ_WRITER has to be part
5818  * of "flags".
5819  *
5820  * Increases sq_count on the outer syncq to keep away outer_insert/remove
5821  * until the outer_exit is finished.
5822  *
5823  * outer_enter is vulnerable to starvation since it does not prevent new
5824  * threads from entering the inner syncqs while it is waiting for sq_count to
5825  * go to zero.
5826  */
5827 void
5828 outer_enter(syncq_t *outer, uint16_t flags)
5829 {
5830         syncq_t *sq;
5831         int     wait_needed;
5832         uint16_t        count;

5834         ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
5835             outer->sq_oprev != NULL);
5836         ASSERT(flags & SQ_WRITER);

5838 retry:
5839         mutex_enter(SQLOCK(outer));
5840         while (outer->sq_flags & flags) {
5841                 outer->sq_flags |= SQ_WANTWAKEUP;
5842                 cv_wait(&outer->sq_wait, SQLOCK(outer));
5843         }

5845         ASSERT(!(outer->sq_flags & SQ_WRITER));
5846         outer->sq_flags |= SQ_WRITER;
5847         outer->sq_count++;
5848         ASSERT(outer->sq_count != 0);   /* wraparound */
```



```
5849         wait_needed = 0;
5850         /*
5851          * Set SQ_WRITER on all the inner syncqs while holding
5852          * the SQLOCK on the outer syncq. This ensures that the changing
5853          * of SQ_WRITER is atomic under the outer SQLOCK.
5854          */
5855         for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
5856                 mutex_enter(SQLOCK(sq));
5857                 count = sq->sq_count;
5858                 SQ_PUTLOCKS_ENTER(sq);
5859                 sq->sq_flags |= SQ_WRITER;
5860                 SUM_SQ_PUTCOUNTS(sq, count);
5861                 if (count != 0)
5862                         wait_needed = 1;
5863                 SQ_PUTLOCKS_EXIT(sq);
5864                 mutex_exit(SQLOCK(sq));
5865         }
5866         mutex_exit(SQLOCK(outer));

5868         /*
5869          * Get everybody out of the syncqs sequentially.
5870          * Note that we don't actually need to acquire the PUTLOCKS, since
5871          * we have already cleared the fastbit, and set QWRITER.  By
5872          * definition, the count can not increase since putnext will
5873          * take the slowlock path (and the purpose of acquiring the
5874          * putlocks was to make sure it didn't increase while we were
5875          * waiting).
5876          *
5877          * Note that we still acquire the PUTLOCKS to be safe.
5878          */
5879         if (wait_needed) {
5880                 for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
5881                         mutex_enter(SQLOCK(sq));
5882                         count = sq->sq_count;
5883                         SQ_PUTLOCKS_ENTER(sq);
5884                         SUM_SQ_PUTCOUNTS(sq, count);
5885                         while (count != 0) {
5886                                 sq->sq_flags |= SQ_WANTWAKEUP;
5887                                 SQ_PUTLOCKS_EXIT(sq);
5888                                 cv_wait(&sq->sq_wait, SQLOCK(sq));
5889                                 count = sq->sq_count;
5890                                 SQ_PUTLOCKS_ENTER(sq);
5891                                 SUM_SQ_PUTCOUNTS(sq, count);
5892                         }
5893                         SQ_PUTLOCKS_EXIT(sq);
5894                         mutex_exit(SQLOCK(sq));
5895                 }
5896                 /*
5897                  * Verify that none of the flags got set while we
5898                  * were waiting for the sq_counts to drop.
5899                  * If this happens we exit and retry entering the
5900                  * outer perimeter.
5901                  */
5902                 mutex_enter(SQLOCK(outer));
5903                 if (outer->sq_flags & (flags & ~SQ_WRITER)) {
5904                         mutex_exit(SQLOCK(outer));
5905                         outer_exit(outer);
5906                         goto retry;
5907                 }
5908                 mutex_exit(SQLOCK(outer));
5909         }
5910 }

5912 /*
5913  * Drop the write access at the outer perimeter.
5914  * Read access is dropped implicitly (by putnext, put, and leavesq) by
```

```
 * decrementing sq_count.
 */
void
outer_exit(syncq_t *outer)
{
        syncq_t *sq;
        int      drain_needed;
        uint16_t flags;

        ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
            outer->sq_oprev != NULL);
        ASSERT(MUTEX_NOT_HELD(SQLOCK(outer)));

        /*
         * Atomically (from the perspective of threads calling become_writer)
         * drop the write access at the outer perimeter by holding
         * SQLOCK(outer) across all the dropsq calls and the resetting of
         * SQ_WRITER.
         * This defines a locking order between the outer perimeter
         * SQLOCK and the inner perimeter SQLOCKs.
         */
        mutex_enter(SQLOCK(outer));
        flags = outer->sq_flags;
        ASSERT(outer->sq_flags & SQ_WRITER);
        if (flags & SQ_QUEUED) {
                write_now(outer);
                flags = outer->sq_flags;
        }

        /*
         * sq_onext is stable since sq_count has not yet been decreased.
         * Reset the SQ_WRITER flags in all syncqs.
         * After dropping SQ_WRITER on the outer syncq we empty all the
         * inner syncqs.
         */
        drain_needed = 0;
        for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext)
                drain_needed += dropsq(sq, SQ_WRITER);
        ASSERT(!(outer->sq_flags & SQ_QUEUED));
        flags &= ~SQ_WRITER;
        if (drain_needed) {
                outer->sq_flags = flags;
                mutex_exit(SQLOCK(outer));
                for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext)
                        emptysq(sq);
                mutex_enter(SQLOCK(outer));
                flags = outer->sq_flags;
        }
        if (flags & SQ_WANTWAKEUP) {
                flags &= ~SQ_WANTWAKEUP;
                cv_broadcast(&outer->sq_wait);
        }
        outer->sq_flags = flags;
        ASSERT(outer->sq_count > 0);
        outer->sq_count--;
        mutex_exit(SQLOCK(outer));
}

/*
 * Add another syncq to an outer perimeter.
 * Block out all other access to the outer perimeter while it is being
 * changed using blocksq.
 * Assumes that the caller has *not* done an outer_enter.
 *
 * Vulnerable to starvation in blocksq.
 */
```



```
static void
outer_insert(syncq_t *outer, syncq_t *sq)
{
        ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
            outer->sq_oprev != NULL);
        ASSERT(sq->sq_outer == NULL && sq->sq_onext == NULL &&
            sq->sq_oprev == NULL);      /* Can't be in an outer perimeter */

        /* Get exclusive access to the outer perimeter list */
        blocksq(outer, SQ_BLOCKED, 0);
        ASSERT(outer->sq_flags & SQ_BLOCKED);
        ASSERT(!(outer->sq_flags & SQ_WRITER));

        mutex_enter(SQLOCK(sq));
        sq->sq_outer = outer;
        outer->sq_onext->sq_oprev = sq;
        sq->sq_onext = outer->sq_onext;
        outer->sq_onext = sq;
        sq->sq_oprev = outer;
        mutex_exit(SQLOCK(sq));
        unblocksq(outer, SQ_BLOCKED, 1);
}

/*
 * Remove a syncq from an outer perimeter.
 * Block out all other access to the outer perimeter while it is being
 * changed using blocksq.
 * Assumes that the caller has *not* done an outer_enter.
 *
 * Vulnerable to starvation in blocksq.
 */
static void
outer_remove(syncq_t *outer, syncq_t *sq)
{
        ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
            outer->sq_oprev != NULL);
        ASSERT(sq->sq_outer == outer);

        /* Get exclusive access to the outer perimeter list */
        blocksq(outer, SQ_BLOCKED, 0);
        ASSERT(outer->sq_flags & SQ_BLOCKED);
        ASSERT(!(outer->sq_flags & SQ_WRITER));

        mutex_enter(SQLOCK(sq));
        sq->sq_outer = NULL;
        sq->sq_onext->sq_oprev = sq->sq_oprev;
        sq->sq_oprev->sq_onext = sq->sq_onext;
        sq->sq_oprev = sq->sq_onext = NULL;
        mutex_exit(SQLOCK(sq));
        unblocksq(outer, SQ_BLOCKED, 1);
}

/*
 * Queue a deferred qwriter(OUTER) callback for this outer perimeter.
 * If this is the first callback for this outer perimeter then add
 * this outer perimeter to the list of outer perimeters that
 * the qwriter_outer_thread will process.
 *
 * Increments sq_count in the outer syncq to prevent the membership
 * of the outer perimeter (in terms of inner syncqs) to change while
 * the callback is pending.
 */
static void
queue_writer(syncq_t *outer, void (*func)(), queue_t *q, mblk_t *mp)
{
        ASSERT(MUTEX_HELD(SQLOCK(outer)));
```

```
6048         mp->b_prev = (mblk_t *)func;
6049         mp->b_queue = q;
6050         mp->b_next = NULL;
6051         outer->sq_count++;      /* Decremented when dequeued */
6052         ASSERT(outer->sq_count != 0);   /* Wraparound */
6053         if (outer->sq_evhead == NULL) {
6054                 /* First message. */
6055                 outer->sq_evhead = outer->sq_evtail = mp;
6056                 outer->sq_flags |= SQ_EVENTS;
6057                 mutex_exit(SQLOCK(outer));
6058                 STRSTAT(qwr_outer);
6059                 (void) taskq_dispatch(streams_taskq,
6060                     (task_func_t *)qwriter_outer_service, outer, TQ_SLEEP);
6061         } else {
6062                 ASSERT(outer->sq_flags & SQ_EVENTS);
6063                 outer->sq_evtail->b_next = mp;
6064                 outer->sq_evtail = mp;
6065                 mutex_exit(SQLOCK(outer));
6066         }
6067 }

6069 /*
6070  * Try and upgrade to write access at the outer perimeter. If this can
6071  * not be done without blocking then queue the callback to be done
6072  * by the qwriter_outer_thread.
6073  *
6074  * This routine can only be called from put or service procedures plus
6075  * asynchronous callback routines that have properly entered the queue (with
6076  * entersq). Thus qwriter(OUTER) assumes the caller has one claim on the syncq
6077  * associated with q.
6078  */
6079 void
6080 qwriter_outer(queue_t *q, mblk_t *mp, void (*func)())
6081 {
6082         syncq_t *osq, *sq, *outer;
6083         int     failed;
6084         uint16_t flags;

6086         osq = q->q_syncq;
6087         outer = osq->sq_outer;
6088         if (outer == NULL)
6089                 panic("qwriter(PERIM_OUTER): no outer perimeter");
6090         ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
6091             outer->sq_oprev != NULL);

6093         mutex_enter(SQLOCK(outer));
6094         flags = outer->sq_flags;
6095         /*
6096          * If some thread is traversing sq_next, or if we are blocked by
6097          * outer_insert or outer_remove, or if the we already have queued
6098          * callbacks, then queue this callback for later processing.
6099          *
6100          * Also queue the qwriter for an interrupt thread in order
6101          * to reduce the time spent running at high IPL.
6102          * to identify there are events.
6103          */
6104         if ((flags & SQ_GOAWAY) || (curthread->t_pri >= kpreemptpri)) {
6105                 /*
6106                  * Queue the become_writer request.
6107                  * The queueing is atomic under SQLOCK(outer) in order
6108                  * to synchronize with outer_exit.
6109                  * queue_writer will drop the outer SQLOCK
6110                  */
6111                 if (flags & SQ_BLOCKED) {
6112                         /* Must set SQ_WRITER on inner perimeter */
```



```
6113                         mutex_enter(SQLOCK(osq));
6114                         osq->sq_flags |= SQ_WRITER;
6115                         mutex_exit(SQLOCK(osq));
6116                 } else {
6117                         if (!(flags & SQ_WRITER)) {
6118                                 /*
6119                                  * The outer could have been SQ_BLOCKED thus
6120                                  * SQ_WRITER might not be set on the inner.
6121                                  */
6122                                 mutex_enter(SQLOCK(osq));
6123                                 osq->sq_flags |= SQ_WRITER;
6124                                 mutex_exit(SQLOCK(osq));
6125                         }
6126                         ASSERT(osq->sq_flags & SQ_WRITER);
6127                 }
6128                 queue_writer(outer, func, q, mp);
6129                 return;
6130         }
6131         /*
6132          * We are half-way to exclusive access to the outer perimeter.
6133          * Prevent any outer_enter, qwriter(OUTER), or outer_insert/remove
6134          * while the inner syncqs are traversed.
6135          */
6136         outer->sq_count++;
6137         ASSERT(outer->sq_count != 0);   /* wraparound */
6138         flags |= SQ_WRITER;
6139         /*
6140          * Check if we can run the function immediately. Mark all
6141          * syncqs with the writer flag to prevent new entries into
6142          * put and service procedures.
6143          *
6144          * Set SQ_WRITER on all the inner syncqs while holding
6145          * the SQLOCK on the outer syncq. This ensures that the changing
6146          * of SQ_WRITER is atomic under the outer SQLOCK.
6147          */
6148         failed = 0;
6149         for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
6150                 uint16_t count;
6151                 uint_t  maxcnt = (sq == osq) ? 1 : 0;

6153                 mutex_enter(SQLOCK(sq));
6154                 count = sq->sq_count;
6155                 SQ_PUTLOCKS_ENTER(sq);
6156                 SUM_SQ_PUTCOUNTS(sq, count);
6157                 if (sq->sq_count > maxcnt)
6158                         failed = 1;
6159                 sq->sq_flags |= SQ_WRITER;
6160                 SQ_PUTLOCKS_EXIT(sq);
6161                 mutex_exit(SQLOCK(sq));
6162         }
6163         if (failed) {
6164                 /*
6165                  * Some other thread has a read claim on the outer perimeter.
6166                  * Queue the callback for deferred processing.
6167                  *
6168                  * queue_writer will set SQ_QUEUED before we drop SQ_WRITER
6169                  * so that other qwriter(OUTER) calls will queue their
6170                  * callbacks as well. queue_writer increments sq_count so we
6171                  * decrement to compensate for the our increment.
6172                  *
6173                  * Dropping SQ_WRITER enables the writer thread to work
6174                  * on this outer perimeter.
6175                  */
6176                 outer->sq_flags = flags;
6177                 queue_writer(outer, func, q, mp);
6178                 /* queue_writer dropper the lock */
```

```
6179                mutex_enter(SQLOCK(outer));
6180                ASSERT(outer->sq_count > 0);
6181                outer->sq_count--;
6182                ASSERT(outer->sq_flags & SQ_WRITER);
6183                flags = outer->sq_flags;
6184                flags &= ~SQ_WRITER;
6185                if (flags & SQ_WANTWAKEUP) {
6186                        flags &= ~SQ_WANTWAKEUP;
6187                        cv_broadcast(&outer->sq_wait);
6188                }
6189                outer->sq_flags = flags;
6190                mutex_exit(SQLOCK(outer));
6191                return;
6192        } else {
6193                outer->sq_flags = flags;
6194                mutex_exit(SQLOCK(outer));
6195        }

6197        /* Can run it immediately */
6198        (*func)(q, mp);

6200        outer_exit(outer);
6201 }

6203 /*
6204  * Dequeue all writer callbacks from the outer perimeter and run them.
6205  */
6206 static void
6207 write_now(syncq_t *outer)
6208 {
6209        mblk_t          *mp;
6210        queue_t         *q;
6211        void    (*func)();

6213        ASSERT(MUTEX_HELD(SQLOCK(outer)));
6214        ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
6215            outer->sq_oprev != NULL);
6216        while ((mp = outer->sq_evhead) != NULL) {
6217                /*
6218                 * queues cannot be placed on the queuelist on the outer
6219                 * perimeter.
6220                 */
6221                ASSERT(!(outer->sq_flags & SQ_MESSAGES));
6222                ASSERT((outer->sq_flags & SQ_EVENTS));

6224                outer->sq_evhead = mp->b_next;
6225                if (outer->sq_evhead == NULL) {
6226                        outer->sq_evtail = NULL;
6227                        outer->sq_flags &= ~SQ_EVENTS;
6228                }
6229                ASSERT(outer->sq_count != 0);
6230                outer->sq_count--;      /* Incremented when enqueued. */
6231                mutex_exit(SQLOCK(outer));
6232                /*
6233                 * Drop the message if the queue is closing.
6234                 * Make sure that the queue is "claimed" when the callback
6235                 * is run in order to satisfy various ASSERTs.
6236                 */
6237                q = mp->b_queue;
6238                func = (void (*)())mp->b_prev;
6239                ASSERT(func != NULL);
6240                mp->b_next = mp->b_prev = NULL;
6241                if (q->q_flag & QWCLOSE) {
6242                        freemsg(mp);
6243                } else {
6244                        claimq(q);
```



```
6245                        (*func)(q, mp);
6246                        releaseq(q);
6247                }
6248                mutex_enter(SQLOCK(outer));
6249        }
6250        ASSERT(MUTEX_HELD(SQLOCK(outer)));
6251 }

6253 /*
6254  * The list of messages on the inner syncq is effectively hashed
6255  * by destination queue.  These destination queues are doubly
6256  * linked lists (hopefully) in priority order.  Messages are then
6257  * put on the queue referenced by the q_sqhead/q_sqtail elements.
6258  * Additional messages are linked together by the b_next/b_prev
6259  * elements in the mblk, with (similar to putq()) the first message
6260  * having a NULL b_prev and the last message having a NULL b_next.
6261  *
6262  * Events, such as qwriter callbacks, are put onto a list in FIFO
6263  * order referenced by sq_evhead, and sq_evtail.  This is a singly
6264  * linked list, and messages here MUST be processed in the order queued.
6265  */

6267 /*
6268  * Run the events on the syncq event list (sq_evhead).
6269  * Assumes there is only one claim on the syncq, it is
6270  * already exclusive (SQ_EXCL set), and the SQLOCK held.
6271  * Messages here are processed in order, with the SQ_EXCL bit
6272  * held all the way through till the last message is processed.
6273  */
6274 void
6275 sq_run_events(syncq_t *sq)
6276 {
6277        mblk_t          *bp;
6278        queue_t         *qp;
6279        uint16_t        flags = sq->sq_flags;
6280        void            (*func)();

6282        ASSERT(MUTEX_HELD(SQLOCK(sq)));
6283        ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6284            sq->sq_oprev == NULL) ||
6285            (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6286            sq->sq_oprev != NULL));

6288        ASSERT(flags & SQ_EXCL);
6289        ASSERT(sq->sq_count == 1);

6291        /*
6292         * We need to process all of the events on this list.  It
6293         * is possible that new events will be added while we are
6294         * away processing a callback, so on every loop, we start
6295         * back at the beginning of the list.
6296         */
6297        /*
6298         * We have to reaccess sq_evhead since there is a
6299         * possibility of a new entry while we were running
6300         * the callback.
6301         */
6302        for (bp = sq->sq_evhead; bp != NULL; bp = sq->sq_evhead) {
6303                ASSERT(bp->b_queue->q_syncq == sq);
6304                ASSERT(sq->sq_flags & SQ_EVENTS);

6306                qp = bp->b_queue;
6307                func = (void (*)())bp->b_prev;
6308                ASSERT(func != NULL);

6310                /*
```

```
6311                 * Messages from the event queue must be taken off in
6312                 * FIFO order.
6313                 */
6314                ASSERT(sq->sq_evhead == bp);
6315                sq->sq_evhead = bp->b_next;

6317                if (bp->b_next == NULL) {
6318                        /* Deleting last */
6319                        ASSERT(sq->sq_evtail == bp);
6320                        sq->sq_evtail = NULL;
6321                        sq->sq_flags &= ~SQ_EVENTS;
6322                }
6323                bp->b_prev = bp->b_next = NULL;
6324                ASSERT(bp->b_datap->db_ref != 0);

6326                mutex_exit(SQLOCK(sq));

6328                (*func)(qp, bp);

6330                mutex_enter(SQLOCK(sq));
6331                /*
6332                 * re-read the flags, since they could have changed.
6333                 */
6334                flags = sq->sq_flags;
6335                ASSERT(flags & SQ_EXCL);
6336        }
6337        ASSERT(sq->sq_evhead == NULL && sq->sq_evtail == NULL);
6338        ASSERT(!(sq->sq_flags & SQ_EVENTS));

6340        if (flags & SQ_WANTWAKEUP) {
6341                flags &= ~SQ_WANTWAKEUP;
6342                cv_broadcast(&sq->sq_wait);
6343        }
6344        if (flags & SQ_WANTEXWAKEUP) {
6345                flags &= ~SQ_WANTEXWAKEUP;
6346                cv_broadcast(&sq->sq_exitwait);
6347        }
6348        sq->sq_flags = flags;
6349 }

6351 /*
6352  * Put messages on the event list.
6353  * If we can go exclusive now, do so and process the event list, otherwise
6354  * let the last claim service this list (or wake the sqthread).
6355  * This procedure assumes SQLOCK is held.  To run the event list, it
6356  * must be called with no claims.
6357  */
6358 static void
6359 sqfill_events(syncq_t *sq, queue_t *q, mblk_t *mp, void (*func)())
6360 {
6361        uint16_t count;

6363        ASSERT(MUTEX_HELD(SQLOCK(sq)));
6364        ASSERT(func != NULL);

6366        /*
6367         * This is a callback.  Add it to the list of callbacks
6368         * and see about upgrading.
6369         */
6370        mp->b_prev = (mblk_t *)func;
6371        mp->b_queue = q;
6372        mp->b_next = NULL;
6373        if (sq->sq_evhead == NULL) {
6374                sq->sq_evhead = sq->sq_evtail = mp;
6375                sq->sq_flags |= SQ_EVENTS;
6376        } else {
```



```
6377                ASSERT(sq->sq_evtail != NULL);
6378                ASSERT(sq->sq_evtail->b_next == NULL);
6379                ASSERT(sq->sq_flags & SQ_EVENTS);
6380                sq->sq_evtail->b_next = mp;
6381                sq->sq_evtail = mp;
6382        }
6383        /*
6384         * We have set SQ_EVENTS, so threads will have to
6385         * unwind out of the perimeter, and new entries will
6386         * not grab a putlock.  But we still need to know
6387         * how many threads have already made a claim to the
6388         * syncq, so grab the putlocks, and sum the counts.
6389         * If there are no claims on the syncq, we can upgrade
6390         * to exclusive, and run the event list.
6391         * NOTE: We hold the SQLOCK, so we can just grab the
6392         * putlocks.
6393         */
6394        count = sq->sq_count;
6395        SQ_PUTLOCKS_ENTER(sq);
6396        SUM_SQ_PUTCOUNTS(sq, count);
6397        /*
6398         * We have no claim, so we need to check if there
6399         * are no others, then we can upgrade.
6400         */
6401        /*
6402         * There are currently no claims on
6403         * the syncq by this thread (at least on this entry). The thread who has
6404         * the claim should drain syncq.
6405         */
6406        if (count > 0) {
6407                /*
6408                 * Can't upgrade - other threads inside.
6409                 */
6410                SQ_PUTLOCKS_EXIT(sq);
6411                mutex_exit(SQLOCK(sq));
6412                return;
6413        }
6414        /*
6415         * Need to set SQ_EXCL and make a claim on the syncq.
6416         */
6417        ASSERT((sq->sq_flags & SQ_EXCL) == 0);
6418        sq->sq_flags |= SQ_EXCL;
6419        ASSERT(sq->sq_count == 0);
6420        sq->sq_count++;
6421        SQ_PUTLOCKS_EXIT(sq);

6423        /* Process the events list */
6424        sq_run_events(sq);

6426        /*
6427         * Release our claim...
6428         */
6429        sq->sq_count--;

6431        /*
6432         * And release SQ_EXCL.
6433         * We don't need to acquire the putlocks to release
6434         * SQ_EXCL, since we are exclusive, and hold the SQLOCK.
6435         */
6436        sq->sq_flags &= ~SQ_EXCL;

6438        /*
6439         * sq_run_events should have released SQ_EXCL
6440         */
6441        ASSERT(!(sq->sq_flags & SQ_EXCL));
```

```
6443         /*
6444          * If anything happened while we were running the
6445          * events (or was there before), we need to process
6446          * them now.  We shouldn't be exclusive sine we
6447          * released the perimeter above (plus, we asserted
6448          * for it).
6449          */
6450         if (!(sq->sq_flags & SQ_STAYAWAY) && (sq->sq_flags & SQ_QUEUED))
6451                 drain_syncq(sq);
6452         else
6453                 mutex_exit(SQLOCK(sq));
6454 }

6456 /*
6457  * Perform delayed processing. The caller has to make sure that it is safe
6458  * to enter the syncq (e.g. by checking that none of the SQ_STAYAWAY bits are
6459  * set).
6460  *
6461  * Assume that the caller has NO claims on the syncq.  However, a claim
6462  * on the syncq does not indicate that a thread is draining the syncq.
6463  * There may be more claims on the syncq than there are threads draining
6464  * (i.e.  #_threads_draining <= sq_count)
6465  *
6466  * drain_syncq has to terminate when one of the SQ_STAYAWAY bits gets set
6467  * in order to preserve qwriter(OUTER) ordering constraints.
6468  *
6469  * sq_putcount only needs to be checked when dispatching the queued
6470  * writer call for CIPUT sync queue, but this is handled in sq_run_events.
6471  */
6472 void
6473 drain_syncq(syncq_t *sq)
6474 {
6475         queue_t         *qp;
6476         uint16_t        count;
6477         uint16_t        type = sq->sq_type;
6478         uint16_t        flags = sq->sq_flags;
6479         boolean_t       bg_service = sq->sq_svcflags & SQ_SERVICE;

6481         TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_START,
6482             "drain_syncq start:%p", sq);
6483         ASSERT(MUTEX_HELD(SQLOCK(sq)));
6484         ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6485             sq->sq_oprev == NULL) ||
6486             (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6487             sq->sq_oprev != NULL));

6489         /*
6490          * Drop SQ_SERVICE flag.
6491          */
6492         if (bg_service)
6493                 sq->sq_svcflags &= ~SQ_SERVICE;

6495         /*
6496          * If SQ_EXCL is set, someone else is processing this syncq - let him
6497          * finish the job.
6498          */
6499         if (flags & SQ_EXCL) {
6500                 if (bg_service) {
6501                         ASSERT(sq->sq_servcount != 0);
6502                         sq->sq_servcount--;
6503                 }
6504                 mutex_exit(SQLOCK(sq));
6505                 return;
6506         }

6508         /*
```



```
6509          * This routine can be called by a background thread if
6510          * it was scheduled by a hi-priority thread.  SO, if there are
6511          * NOT messages queued, return (remember, we have the SQLOCK,
6512          * and it cannot change until we release it). Wakeup any waiters also.
6513          */
6514         if (!(flags & SQ_QUEUED)) {
6515                 if (flags & SQ_WANTWAKEUP) {
6516                         flags &= ~SQ_WANTWAKEUP;
6517                         cv_broadcast(&sq->sq_wait);
6518                 }
6519                 if (flags & SQ_WANTEXWAKEUP) {
6520                         flags &= ~SQ_WANTEXWAKEUP;
6521                         cv_broadcast(&sq->sq_exitwait);
6522                 }
6523                 sq->sq_flags = flags;
6524                 if (bg_service) {
6525                         ASSERT(sq->sq_servcount != 0);
6526                         sq->sq_servcount--;
6527                 }
6528                 mutex_exit(SQLOCK(sq));
6529                 return;
6530         }

6532         /*
6533          * If this is not a concurrent put perimeter, we need to
6534          * become exclusive to drain.  Also, if not CIPUT, we would
6535          * not have acquired a putlock, so we don't need to check
6536          * the putcounts.  If not entering with a claim, we test
6537          * for sq_count == 0.
6538          */
6539         type = sq->sq_type;
6540         if (!(type & SQ_CIPUT)) {
6541                 if (sq->sq_count > 1) {
6542                         if (bg_service) {
6543                                 ASSERT(sq->sq_servcount != 0);
6544                                 sq->sq_servcount--;
6545                         }
6546                         mutex_exit(SQLOCK(sq));
6547                         return;
6548                 }
6549                 sq->sq_flags |= SQ_EXCL;
6550         }

6552         /*
6553          * This is where we make a claim to the syncq.
6554          * This can either be done by incrementing a putlock, or
6555          * the sq_count.  But since we already have the SQLOCK
6556          * here, we just bump the sq_count.
6557          *
6558          * Note that after we make a claim, we need to let the code
6559          * fall through to the end of this routine to clean itself
6560          * up.  A return in the while loop will put the syncq in a
6561          * very bad state.
6562          */
6563         sq->sq_count++;
6564         ASSERT(sq->sq_count != 0);      /* wraparound */

6566         while ((flags = sq->sq_flags) & SQ_QUEUED) {
6567                 /*
6568                  * If we are told to stayaway or went exclusive,
6569                  * we are done.
6570                  */
6571                 if (flags & (SQ_STAYAWAY)) {
6572                         break;
6573                 }
```

```
6575                /*
6576                 * If there are events to run, do so.
6577                 * We have one claim to the syncq, so if there are
6578                 * more than one, other threads are running.
6579                 */
6580                if (sq->sq_evhead != NULL) {
6581                        ASSERT(sq->sq_flags & SQ_EVENTS);

6583                        count = sq->sq_count;
6584                        SQ_PUTLOCKS_ENTER(sq);
6585                        SUM_SQ_PUTCOUNTS(sq, count);
6586                        if (count > 1) {
6587                                SQ_PUTLOCKS_EXIT(sq);
6588                                /* Can't upgrade - other threads inside */
6589                                break;
6590                        }
6591                        ASSERT((flags & SQ_EXCL) == 0);
6592                        sq->sq_flags = flags | SQ_EXCL;
6593                        SQ_PUTLOCKS_EXIT(sq);
6594                        /*
6595                         * we have the only claim, run the events,
6596                         * sq_run_events will clear the SQ_EXCL flag.
6597                         */
6598                        sq_run_events(sq);

6600                        /*
6601                         * If this is a CIPUT perimeter, we need
6602                         * to drop the SQ_EXCL flag so we can properly
6603                         * continue draining the syncq.
6604                         */
6605                        if (type & SQ_CIPUT) {
6606                                ASSERT(sq->sq_flags & SQ_EXCL);
6607                                sq->sq_flags &= ~SQ_EXCL;
6608                        }

6610                        /*
6611                         * And go back to the beginning just in case
6612                         * anything changed while we were away.
6613                         */
6614                        ASSERT((sq->sq_flags & SQ_EXCL) || (type & SQ_CIPUT));
6615                        continue;
6616                }

6618                ASSERT(sq->sq_evhead == NULL);
6619                ASSERT(!(sq->sq_flags & SQ_EVENTS));

6621                /*
6622                 * Find the queue that is not draining.
6623                 *
6624                 * q_draining is protected by QLOCK which we do not hold.
6625                 * But if it was set, then a thread was draining, and if it gets
6626                 * cleared, then it was because the thread has successfully
6627                 * drained the syncq, or a GOAWAY state occurred. For the GOAWAY
6628                 * state to happen, a thread needs the SQLOCK which we hold, and
6629                 * if there was such a flag, we would have already seen it.
6630                 */

6632                for (qp = sq->sq_head;
6633                    qp != NULL && (qp->q_draining ||
6634                    (qp->q_sqflags & Q_SQDRAINING));
6635                    qp = qp->q_sqnext)
6636                        ;

6638                if (qp == NULL)
6639                        break;
```



```
6641                /*
6642                 * We have a queue to work on, and we hold the
6643                 * SQLOCK and one claim, call qdrain_syncq.
6644                 * This means we need to release the SQLOCK and
6645                 * acquire the QLOCK (OK since we have a claim).
6646                 * Note that qdrain_syncq will actually dequeue
6647                 * this queue from the sq_head list when it is
6648                 * convinced all the work is done and release
6649                 * the QLOCK before returning.
6650                 */
6651                qp->q_sqflags |= Q_SQDRAINING;
6652                mutex_exit(SQLOCK(sq));
6653                mutex_enter(QLOCK(qp));
6654                qdrain_syncq(sq, qp);
6655                mutex_enter(SQLOCK(sq));

6657                /* The queue is drained */
6658                ASSERT(qp->q_sqflags & Q_SQDRAINING);
6659                qp->q_sqflags &= ~Q_SQDRAINING;
6660                /*
6661                 * NOTE: After this point qp should not be used since it may be
6662                 * closed.
6663                 */
6664        }

6666        ASSERT(MUTEX_HELD(SQLOCK(sq)));
6667        flags = sq->sq_flags;

6669        /*
6670         * sq->sq_head cannot change because we hold the
6671         * sqlock. However, a thread CAN decide that it is no longer
6672         * going to drain that queue.  However, this should be due to
6673         * a GOAWAY state, and we should see that here.
6674         *
6675         * This loop is not very efficient. One solution may be adding a second
6676         * pointer to the "draining" queue, but it is difficult to do when
6677         * queues are inserted in the middle due to priority ordering. Another
6678         * possibility is to yank the queue out of the sq list and put it onto
6679         * the "draining list" and then put it back if it can't be drained.
6680         */

6682        ASSERT((sq->sq_head == NULL) || (flags & SQ_GOAWAY) ||
6683            (type & SQ_CI) || sq->sq_head->q_draining);

6685        /* Drop SQ_EXCL for non-CIPUT perimeters */
6686        if (!(type & SQ_CIPUT))
6687                flags &= ~SQ_EXCL;
6688        ASSERT((flags & SQ_EXCL) == 0);

6690        /* Wake up any waiters. */
6691        if (flags & SQ_WANTWAKEUP) {
6692                flags &= ~SQ_WANTWAKEUP;
6693                cv_broadcast(&sq->sq_wait);
6694        }
6695        if (flags & SQ_WANTEXWAKEUP) {
6696                flags &= ~SQ_WANTEXWAKEUP;
6697                cv_broadcast(&sq->sq_exitwait);
6698        }
6699        sq->sq_flags = flags;

6701        ASSERT(sq->sq_count != 0);
6702        /* Release our claim. */
6703        sq->sq_count--;

6705        if (bg_service) {
6706                ASSERT(sq->sq_servcount != 0);
```

```
6707                 sq->sq_servcount--;
6708         }

6710         mutex_exit(SQLOCK(sq));

6712         TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_END,
6713             "drain_syncq end:%p", sq);
6714 }



6717 /*
6718  *
6719  * qdrain_syncq can be called (currently) from only one of two places:
6720  *      drain_syncq
6721  *      putnext  (or some variation of it).
6722  * and eventually
6723  *      qwait(_sig)
6724  *
6725  * If called from drain_syncq, we found it in the list of queues needing
6726  * service, so there is work to be done (or it wouldn't be in the list).
6727  *
6728  * If called from some putnext variation, it was because the
6729  * perimeter is open, but messages are blocking a putnext and
6730  * there is not a thread working on it.  Now a thread could start
6731  * working on it while we are getting ready to do so ourself, but
6732  * the thread would set the q_draining flag, and we can spin out.
6733  *
6734  * As for qwait(_sig), I think I shall let it continue to call
6735  * drain_syncq directly (after all, it will get here eventually).
6736  *
6737  * qdrain_syncq has to terminate when:
6738  * - one of the SQ_STAYAWAY bits gets set to preserve qwriter(OUTER) ordering
6739  * - SQ_EVENTS gets set to preserve qwriter(INNER) ordering
6740  *
6741  * ASSUMES:
6742  *      One claim
6743  *      QLOCK held
6744  *      SQLOCK not held
6745  *      Will release QLOCK before returning
6746  */
6747 void
6748 qdrain_syncq(syncq_t *sq, queue_t *q)
6749 {
6750         mblk_t          *bp;
6751 #ifdef DEBUG
6752         uint16_t        count;
6753 #endif

6755         TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_START,
6756             "drain_syncq start:%p", sq);
6757         ASSERT(q->q_syncq == sq);
6758         ASSERT(MUTEX_HELD(QLOCK(q)));
6759         ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
6760         /*
6761          * For non-CIPUT perimeters, we should be called with the exclusive bit
6762          * set already. For CIPUT perimeters, we will be doing a concurrent
6763          * drain, so it better not be set.
6764          */
6765         ASSERT((sq->sq_flags & (SQ_EXCL|SQ_CIPUT)));
6766         ASSERT(!((sq->sq_type & SQ_CIPUT) && (sq->sq_flags & SQ_EXCL)));
6767         ASSERT((sq->sq_type & SQ_CIPUT) || (sq->sq_flags & SQ_EXCL));
6768         /*
6769          * All outer pointers are set, or none of them are
6770          */
6771         ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6772             sq->sq_oprev == NULL) ||
```



```
6773             (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6774             sq->sq_oprev != NULL));
6775 #ifdef DEBUG
6776         count = sq->sq_count;
6777         /*
6778          * This is OK without the putlocks, because we have one
6779          * claim either from the sq_count, or a putcount.  We could
6780          * get an erroneous value from other counts, but ours won't
6781          * change, so one way or another, we will have at least a
6782          * value of one.
6783          */
6784         SUM_SQ_PUTCOUNTS(sq, count);
6785         ASSERT(count >= 1);
6786 #endif /* DEBUG */

6788         /*
6789          * The first thing to do is find out if a thread is already draining
6790          * this queue. If so, we are done, just return.
6791          */
6792         if (q->q_draining) {
6793                 mutex_exit(QLOCK(q));
6794                 return;
6795         }

6797         /*
6798          * If the perimeter is exclusive, there is nothing we can do right now,
6799          * go away. Note that there is nothing to prevent this case from
6800          * changing right after this check, but the spin-out will catch it.
6801          */

6803         /* Tell other threads that we are draining this queue */
6804         q->q_draining = 1;      /* Protected by QLOCK */

6806         /*
6807          * If there is nothing to do, clear QFULL as necessary. This caters for
6808          * the case where an empty queue was enqueued onto the syncq.
6809          */
6810         if (q->q_sqhead == NULL) {
6811                 ASSERT(q->q_syncqmsgs == 0);
6812                 mutex_exit(QLOCK(q));
6813                 clr_qfull(q);
6814                 mutex_enter(QLOCK(q));
6815         }

6817         /*
6818          * Note that q_sqhead must be re-checked here in case another message
6819          * was enqueued whilst QLOCK was dropped during the call to clr_qfull.
6820          */
6821         for (bp = q->q_sqhead; bp != NULL; bp = q->q_sqhead) {
6822                 /*
6823                  * Because we can enter this routine just because a putnext is
6824                  * blocked, we need to spin out if the perimeter wants to go
6825                  * exclusive as well as just blocked. We need to spin out also
6826                  * if events are queued on the syncq.
6827                  * Don't check for SQ_EXCL, because non-CIPUT perimeters would
6828                  * set it, and it can't become exclusive while we hold a claim.
6829                  */
6830                 if (sq->sq_flags & (SQ_STAYAWAY | SQ_EVENTS)) {
6831                         break;
6832                 }

6834 #ifdef DEBUG
6835                 /*
6836                  * Since we are in qdrain_syncq, we already know the queue,
6837                  * but for sanity, we want to check this against the qp that
6838                  * was passed in by bp->b_queue.
```

```
6839                 */

6841                ASSERT(bp->b_queue == q);
6842                ASSERT(bp->b_queue->q_syncq == sq);
6843                bp->b_queue = NULL;

6845                /*
6846                 * We would have the following check in the DEBUG code:
6847                 *
6848                 * if (bp->b_prev != NULL)  {
6849                 *      ASSERT(bp->b_prev == (void (*)())q->q_qinfo->qi_putp);
6850                 * }
6851                 *
6852                 * This can't be done, however, since IP modifies qinfo
6853                 * structure at run-time (switching between IPv4 qinfo and IPv6
6854                 * qinfo), invalidating the check.
6855                 * So the assignment to func is left here, but the ASSERT itself
6856                 * is removed until the whole issue is resolved.
6857                 */
6858 #endif
6859                ASSERT(q->q_sqhead == bp);
6860                q->q_sqhead = bp->b_next;
6861                bp->b_prev = bp->b_next = NULL;
6862                ASSERT(q->q_syncqmsgs > 0);
6863                mutex_exit(QLOCK(q));

6865                ASSERT(bp->b_datap->db_ref != 0);

6867                (void) (*q->q_qinfo->qi_putp)(q, bp);

6869                mutex_enter(QLOCK(q));

6871                /*
6872                 * q_syncqmsgs should only be decremented after executing the
6873                 * put procedure to avoid message re-ordering. This is due to an
6874                 * optimisation in putnext() which can call the put procedure
6875                 * directly if it sees q_syncqmsgs == 0 (despite Q_SQQUEUED
6876                 * being set).
6877                 *
6878                 * We also need to clear QFULL in the next service procedure
6879                 * queue if this is the last message destined for that queue.
6880                 *
6881                 * It would make better sense to have some sort of tunable for
6882                 * the low water mark, but these semantics are not yet defined.
6883                 * So, alas, we use a constant.
6884                 */
6885                if (--q->q_syncqmsgs == 0) {
6886                        mutex_exit(QLOCK(q));
6887                        clr_qfull(q);
6888                        mutex_enter(QLOCK(q));
6889                }

6891                /*
6892                 * Always clear SQ_EXCL when CIPUT in order to handle
6893                 * qwriter(INNER). The putp() can call qwriter and get exclusive
6894                 * access IFF this is the only claim. So, we need to test for
6895                 * this possibility, acquire the mutex and clear the bit.
6896                 */
6897                if ((sq->sq_type & SQ_CIPUT) && (sq->sq_flags & SQ_EXCL)) {
6898                        mutex_enter(SQLOCK(sq));
6899                        sq->sq_flags &= ~SQ_EXCL;
6900                        mutex_exit(SQLOCK(sq));
6901                }
6902        }

6904        /*
```



```
6905         * We should either have no messages on this queue, or we were told to
6906         * goaway by a waiter (which we will wake up at the end of this
6907         * function).
6908         */
6909        ASSERT((q->q_sqhead == NULL) ||
6910            (sq->sq_flags & (SQ_STAYAWAY | SQ_EVENTS)));

6912        ASSERT(MUTEX_HELD(QLOCK(q)));
6913        ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));

6915        /* Remove the q from the syncq list if all the messages are drained. */
6916        if (q->q_sqhead == NULL) {
6917                ASSERT(q->q_syncqmsgs == 0);
6918                mutex_enter(SQLOCK(sq));
6919                if (q->q_sqflags & Q_SQQUEUED)
6920                        SQRM_Q(sq, q);
6921                mutex_exit(SQLOCK(sq));
6922                /*
6923                 * Since the queue is removed from the list, reset its priority.
6924                 */
6925                q->q_spri = 0;
6926        }

6928        /*
6929         * Remember, the q_draining flag is used to let another thread know
6930         * that there is a thread currently draining the messages for a queue.
6931         * Since we are now done with this queue (even if there may be messages
6932         * still there), we need to clear this flag so some thread will work on
6933         * it if needed.
6934         */
6935        ASSERT(q->q_draining);
6936        q->q_draining = 0;

6938        /* Called with a claim, so OK to drop all locks. */
6939        mutex_exit(QLOCK(q));

6941        TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_END,
6942            "drain_syncq end:%p", sq);
6943 }
6944 /* END OF QDRAIN_SYNCQ  */


6947 /*
6948  * This is the mate to qdrain_syncq, except that it is putting the message onto
6949  * the queue instead of draining. Since the message is destined for the queue
6950  * that is selected, there is no need to identify the function because the
6951  * message is intended for the put routine for the queue. For debug kernels,
6952  * this routine will do it anyway just in case.
6953  *
6954  * After the message is enqueued on the syncq, it calls putnext_tail()
6955  * which will schedule a background thread to actually process the message.
6956  *
6957  * Assumes that there is a claim on the syncq (sq->sq_count > 0) and
6958  * SQLOCK(sq) and QLOCK(q) are not held.
6959  */
6960 void
6961 qfill_syncq(syncq_t *sq, queue_t *q, mblk_t *mp)
6962 {
6963        ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
6964        ASSERT(MUTEX_NOT_HELD(QLOCK(q)));
6965        ASSERT(sq->sq_count > 0);
6966        ASSERT(q->q_syncq == sq);
6967        ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6968            sq->sq_oprev == NULL) ||
6969            (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6970            sq->sq_oprev != NULL));
```

```
6972         mutex_enter(QLOCK(q));

6974 #ifdef DEBUG
6975         /*
6976          * This is used for debug in the qfill_syncq/qdrain_syncq case
6977          * to trace the queue that the message is intended for.  Note
6978          * that the original use was to identify the queue and function
6979          * to call on the drain.  In the new syncq, we have the context
6980          * of the queue that we are draining, so call it's putproc and
6981          * don't rely on the saved values.  But for debug this is still
6982          * useful information.
6983          */
6984         mp->b_prev = (mblk_t *)q->q_qinfo->qi_putp;
6985         mp->b_queue = q;
6986         mp->b_next = NULL;
6987 #endif
6988         ASSERT(q->q_syncq == sq);
6989         /*
6990          * Enqueue the message on the list.
6991          * SQPUT_MP() accesses q_syncqmsgs.  We are already holding QLOCK to
6992          * protect it.  So it's ok to acquire SQLOCK after SQPUT_MP().
6993          */
6994         SQPUT_MP(q, mp);
6995         mutex_enter(SQLOCK(sq));

6997         /*
6998          * And queue on syncq for scheduling, if not already queued.
6999          * Note that we need the SQLOCK for this, and for testing flags
7000          * at the end to see if we will drain.  So grab it now, and
7001          * release it before we call qdrain_syncq or return.
7002          */
7003         if (!(q->q_sqflags & Q_SQQUEUED)) {
7004                 q->q_spri = curthread->t_pri;
7005                 SQPUT_Q(sq, q);
7006         }
7007 #ifdef DEBUG
7008         else {
7009                 /*
7010                  * All of these conditions MUST be true!
7011                  */
7012                 ASSERT(sq->sq_tail != NULL);
7013                 if (sq->sq_tail == sq->sq_head) {
7014                         ASSERT((q->q_sqprev == NULL) &&
7015                             (q->q_sqnext == NULL));
7016                 } else {
7017                         ASSERT((q->q_sqprev != NULL) ||
7018                             (q->q_sqnext != NULL));
7019                 }
7020                 ASSERT(sq->sq_flags & SQ_QUEUED);
7021                 ASSERT(q->q_syncqmsgs != 0);
7022                 ASSERT(q->q_sqflags & Q_SQQUEUED);
7023         }
7024 #endif
7025         mutex_exit(QLOCK(q));
7026         /*
7027          * SQLOCK is still held, so sq_count can be safely decremented.
7028          */
7029         sq->sq_count--;

7031         putnext_tail(sq, q, 0);
7032         /* Should not reference sq or q after this point. */
7033 }

7035 /*  End of qfill_syncq  */
```



```
7037 /*
7038  * Remove all messages from a syncq (if qp is NULL) or remove all messages
7039  * that would be put into qp by drain_syncq.
7040  * Used when deleting the syncq (qp == NULL) or when detaching
7041  * a queue (qp != NULL).
7042  * Return non-zero if one or more messages were freed.
7043  *
7044  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
7045  * sq_putlocks are used.
7046  *
7047  * NOTE: This function assumes that it is called from the close() context and
7048  * that all the queues in the syncq are going away. For this reason it doesn't
7049  * acquire QLOCK for modifying q_sqhead/q_sqtail fields. This assumption is
7050  * currently valid, but it is useful to rethink this function to behave properly
7051  * in other cases.
7052  */
7053 int
7054 flush_syncq(syncq_t *sq, queue_t *qp)
7055 {
7056         mblk_t          *bp, *mp_head, *mp_next, *mp_prev;
7057         queue_t         *q;
7058         int             ret = 0;

7060         mutex_enter(SQLOCK(sq));

7062         /*
7063          * Before we leave, we need to make sure there are no
7064          * events listed for this queue.  All events for this queue
7065          * will just be freed.
7066          */
7067         if (qp != NULL && sq->sq_evhead != NULL) {
7068                 ASSERT(sq->sq_flags & SQ_EVENTS);

7070                 mp_prev = NULL;
7071                 for (bp = sq->sq_evhead; bp != NULL; bp = mp_next) {
7072                         mp_next = bp->b_next;
7073                         if (bp->b_queue == qp) {
7074                                 /* Delete this message */
7075                                 if (mp_prev != NULL) {
7076                                         mp_prev->b_next = mp_next;
7077                                         /*
7078                                          * Update sq_evtail if the last element
7079                                          * is removed.
7080                                          */
7081                                         if (bp == sq->sq_evtail) {
7082                                                 ASSERT(mp_next == NULL);
7083                                                 sq->sq_evtail = mp_prev;
7084                                         }
7085                                 } else
7086                                         sq->sq_evhead = mp_next;
7087                                 if (sq->sq_evhead == NULL)
7088                                         sq->sq_flags &= ~SQ_EVENTS;
7089                                 bp->b_prev = bp->b_next = NULL;
7090                                 freemsg(bp);
7091                                 ret++;
7092                         } else {
7093                                 mp_prev = bp;
7094                         }
7095                 }
7096         }

7098         /*
7099          * Walk sq_head and:
7100          *      - match qp if qp is set, remove it's messages
7101          *      - all if qp is not set
7102          */
```

```
7103         q = sq->sq_head;
7104         while (q != NULL) {
7105                 ASSERT(q->q_syncq == sq);
7106                 if ((qp == NULL) || (qp == q)) {
7107                         /*
7108                          * Yank the messages as a list off the queue
7109                          */
7110                         mp_head = q->q_sqhead;
7111                         /*
7112                          * We do not have QLOCK(q) here (which is safe due to
7113                          * assumptions mentioned above). To obtain the lock we
7114                          * need to release SQLOCK which may allow lots of things
7115                          * to change upon us. This place requires more analysis.
7116                          */
7117                         q->q_sqhead = q->q_sqtail = NULL;
7118                         ASSERT(mp_head->b_queue &&
7119                             mp_head->b_queue->q_syncq == sq);
7120
7121                         /*
7122                          * Free each of the messages.
7123                          */
7124                         for (bp = mp_head; bp != NULL; bp = mp_next) {
7125                                 mp_next = bp->b_next;
7126                                 bp->b_prev = bp->b_next = NULL;
7127                                 freemsg(bp);
7128                                 ret++;
7129                         }
7130                         /*
7131                          * Now remove the queue from the syncq.
7132                          */
7133                         ASSERT(q->q_sqflags & Q_SQQUEUED);
7134                         SQRM_Q(sq, q);
7135                         q->q_spri = 0;
7136                         q->q_syncqmsgs = 0;
7137
7138                         /*
7139                          * If qp was specified, we are done with it and are
7140                          * going to drop SQLOCK(sq) and return. We wakeup syncq
7141                          * waiters while we still have the SQLOCK.
7142                          */
7143                         if ((qp != NULL) && (sq->sq_flags & SQ_WANTWAKEUP)) {
7144                                 sq->sq_flags &= ~SQ_WANTWAKEUP;
7145                                 cv_broadcast(&sq->sq_wait);
7146                         }
7147                         /* Drop SQLOCK across clr_qfull */
7148                         mutex_exit(SQLOCK(sq));
7149
7150                         /*
7151                          * We avoid doing the test that drain_syncq does and
7152                          * unconditionally clear qfull for every flushed
7153                          * message. Since flush_syncq is only called during
7154                          * close this should not be a problem.
7155                          */
7156                         clr_qfull(q);
7157                         if (qp != NULL) {
7158                                 return (ret);
7159                         } else {
7160                                 mutex_enter(SQLOCK(sq));
7161                                 /*
7162                                  * The head was removed by SQRM_Q above.
7163                                  * reread the new head and flush it.
7164                                  */
7165                                 q = sq->sq_head;
7166                         }
7167                 } else {
7168                         q = q->q_sqnext;
```



```
7169                 }
7170                 ASSERT(MUTEX_HELD(SQLOCK(sq)));
7171         }
7172
7173         if (sq->sq_flags & SQ_WANTWAKEUP) {
7174                 sq->sq_flags &= ~SQ_WANTWAKEUP;
7175                 cv_broadcast(&sq->sq_wait);
7176         }
7177
7178         mutex_exit(SQLOCK(sq));
7179         return (ret);
7180 }
7181
7182 /*
7183  * Propagate all messages from a syncq to the next syncq that are associated
7184  * with the specified queue. If the queue is attached to a driver or if the
7185  * messages have been added due to a qwriter(PERIM_INNER), free the messages.
7186  *
7187  * Assumes that the stream is strlock()'ed. We don't come here if there
7188  * are no messages to propagate.
7189  *
7190  * NOTE : If the queue is attached to a driver, all the messages are freed
7191  * as there is no point in propagating the messages from the driver syncq
7192  * to the closing stream head which will in turn get freed later.
7193  */
7194 static int
7195 propagate_syncq(queue_t *qp)
7196 {
7197         mblk_t          *bp, *head, *tail, *prev, *next;
7198         syncq_t         *sq;
7199         queue_t         *nqp;
7200         syncq_t         *nsq;
7201         boolean_t       isdriver;
7202         int             moved = 0;
7203         uint16_t        flags;
7204         pri_t           priority = curthread->t_pri;
7205 #ifdef DEBUG
7206         void            (*func)();
7207 #endif
7208
7209         sq = qp->q_syncq;
7210         ASSERT(MUTEX_HELD(SQLOCK(sq)));
7211         /* debug macro */
7212         SQ_PUTLOCKS_HELD(sq);
7213         /*
7214          * As entersq() does not increment the sq_count for
7215          * the write side, check sq_count for non-QPERQ
7216          * perimeters alone.
7217          */
7218         ASSERT((qp->q_flag & QPERQ) || (sq->sq_count >= 1));
7219
7220         /*
7221          * propagate_syncq() can be called because of either messages on the
7222          * queue syncq or because on events on the queue syncq. Do actual
7223          * message propagations if there are any messages.
7224          */
7225         if (qp->q_syncqmsgs) {
7226                 isdriver = (qp->q_flag & QISDRV);
7227
7228                 if (!isdriver) {
7229                         nqp = qp->q_next;
7230                         nsq = nqp->q_syncq;
7231                         ASSERT(MUTEX_HELD(SQLOCK(nsq)));
7232                         /* debug macro */
7233                         SQ_PUTLOCKS_HELD(nsq);
7234 #ifdef DEBUG
```

```
7235                        func = (void (*)())nqp->q_qinfo->qi_putp;
7236 #endif
7237                }

7239                SQRM_Q(sq, qp);
7240                priority = MAX(qp->q_spri, priority);
7241                qp->q_spri = 0;
7242                head = qp->q_sqhead;
7243                tail = qp->q_sqtail;
7244                qp->q_sqhead = qp->q_sqtail = NULL;
7245                qp->q_syncqmsgs = 0;

7247                /*
7248                 * Walk the list of messages, and free them if this is a driver,
7249                 * otherwise reset the b_prev and b_queue value to the new putp.
7250                 * Afterward, we will just add the head to the end of the next
7251                 * syncq, and point the tail to the end of this one.
7252                 */

7254                for (bp = head; bp != NULL; bp = next) {
7255                        next = bp->b_next;
7256                        if (isdriver) {
7257                                bp->b_prev = bp->b_next = NULL;
7258                                freemsg(bp);
7259                                continue;
7260                        }
7261                        /* Change the q values for this message */
7262                        bp->b_queue = nqp;
7263 #ifdef DEBUG
7264                        bp->b_prev = (mblk_t *)func;
7265 #endif
7266                        moved++;
7267                }
7268                /*
7269                 * Attach list of messages to the end of the new queue (if there
7270                 * is a list of messages).
7271                 */

7273                if (!isdriver && head != NULL) {
7274                        ASSERT(tail != NULL);
7275                        if (nqp->q_sqhead == NULL) {
7276                                nqp->q_sqhead = head;
7277                        } else {
7278                                ASSERT(nqp->q_sqtail != NULL);
7279                                nqp->q_sqtail->b_next = head;
7280                        }
7281                        nqp->q_sqtail = tail;
7282                        /*
7283                         * When messages are moved from high priority queue to
7284                         * another queue, the destination queue priority is
7285                         * upgraded.
7286                         */

7288                        if (priority > nqp->q_spri)
7289                                nqp->q_spri = priority;

7291                        SQPUT_Q(nsq, nqp);

7293                        nqp->q_syncqmsgs += moved;
7294                        ASSERT(nqp->q_syncqmsgs != 0);
7295                }
7296        }

7298        /*
7299         * Before we leave, we need to make sure there are no
7300         * events listed for this queue.  All events for this queue
```



```
7301         * will just be freed.
7302         */
7303        if (sq->sq_evhead != NULL) {
7304                ASSERT(sq->sq_flags & SQ_EVENTS);
7305                prev = NULL;
7306                for (bp = sq->sq_evhead; bp != NULL; bp = next) {
7307                        next = bp->b_next;
7308                        if (bp->b_queue == qp) {
7309                                /* Delete this message */
7310                                if (prev != NULL) {
7311                                        prev->b_next = next;
7312                                        /*
7313                                         * Update sq_evtail if the last element
7314                                         * is removed.
7315                                         */
7316                                        if (bp == sq->sq_evtail) {
7317                                                ASSERT(next == NULL);
7318                                                sq->sq_evtail = prev;
7319                                        }
7320                                } else
7321                                        sq->sq_evhead = next;
7322                                if (sq->sq_evhead == NULL)
7323                                        sq->sq_flags &= ~SQ_EVENTS;
7324                                bp->b_prev = bp->b_next = NULL;
7325                                freemsg(bp);
7326                        } else {
7327                                prev = bp;
7328                        }
7329                }
7330        }

7332        flags = sq->sq_flags;

7334        /* Wake up any waiter before leaving. */
7335        if (flags & SQ_WANTWAKEUP) {
7336                flags &= ~SQ_WANTWAKEUP;
7337                cv_broadcast(&sq->sq_wait);
7338        }
7339        sq->sq_flags = flags;

7341        return (moved);
7342 }

7344 /*
7345  * Try and upgrade to exclusive access at the inner perimeter. If this can
7346  * not be done without blocking then request will be queued on the syncq
7347  * and drain_syncq will run it later.
7348  *
7349  * This routine can only be called from put or service procedures plus
7350  * asynchronous callback routines that have properly entered the queue (with
7351  * entersq). Thus qwriter_inner assumes the caller has one claim on the syncq
7352  * associated with q.
7353  */
7354 void
7355 qwriter_inner(queue_t *q, mblk_t *mp, void (*func)())
7356 {
7357        syncq_t *sq = q->q_syncq;
7358        uint16_t count;

7360        mutex_enter(SQLOCK(sq));
7361        count = sq->sq_count;
7362        SQ_PUTLOCKS_ENTER(sq);
7363        SUM_SQ_PUTCOUNTS(sq, count);
7364        ASSERT(count >= 1);
7365        ASSERT(sq->sq_type & (SQ_CIPUT|SQ_CISVC));
```

```
7367         if (count == 1) {
7368                 /*
7369                  * Can upgrade. This case also handles nested qwriter calls
7370                  * (when the qwriter callback function calls qwriter). In that
7371                  * case SQ_EXCL is already set.
7372                  */
7373                 sq->sq_flags |= SQ_EXCL;
7374                 SQ_PUTLOCKS_EXIT(sq);
7375                 mutex_exit(SQLOCK(sq));
7376                 (*func)(q, mp);
7377                 /*
7378                  * Assumes that leavesq, putnext, and drain_syncq will reset
7379                  * SQ_EXCL for SQ_CIPUT/SQ_CISVC queues. We leave SQ_EXCL on
7380                  * until putnext, leavesq, or drain_syncq drops it.
7381                  * That way we handle nested qwriter(INNER) without dropping
7382                  * SQ_EXCL until the outermost qwriter callback routine is
7383                  * done.
7384                  */
7385                 return;
7386         }
7387         SQ_PUTLOCKS_EXIT(sq);
7388         sqfill_events(sq, q, mp, func);
7389 }

7391 /*
7392  * Synchronous callback support functions
7393  */

7395 /*
7396  * Allocate a callback parameter structure.
7397  * Assumes that caller initializes the flags and the id.
7398  * Acquires SQLOCK(sq) if non-NULL is returned.
7399  */
7400 callbparams_t *
7401 callbparams_alloc(syncq_t *sq, void (*func)(void *), void *arg, int kmflags)
7402 {
7403         callbparams_t *cbp;
7404         size_t size = sizeof (callbparams_t);

7406         cbp = kmem_alloc(size, kmflags & ~KM_PANIC);

7408         /*
7409          * Only try tryhard allocation if the caller is ready to panic.
7410          * Otherwise just fail.
7411          */
7412         if (cbp == NULL) {
7413                 if (kmflags & KM_PANIC)
7414                         cbp = kmem_alloc_tryhard(sizeof (callbparams_t),
7415                             &size, kmflags);
7416                 else
7417                         return (NULL);
7418         }

7420         ASSERT(size >= sizeof (callbparams_t));
7421         cbp->cbp_size = size;
7422         cbp->cbp_sq = sq;
7423         cbp->cbp_func = func;
7424         cbp->cbp_arg = arg;
7425         mutex_enter(SQLOCK(sq));
7426         cbp->cbp_next = sq->sq_callbpend;
7427         sq->sq_callbpend = cbp;
7428         return (cbp);
7429 }

7431 void
7432 callbparams_free(syncq_t *sq, callbparams_t *cbp)
```



```
7433 {
7434         callbparams_t **pp, *p;

7436         ASSERT(MUTEX_HELD(SQLOCK(sq)));

7438         for (pp = &sq->sq_callbpend; (p = *pp) != NULL; pp = &p->cbp_next) {
7439                 if (p == cbp) {
7440                         *pp = p->cbp_next;
7441                         kmem_free(p, p->cbp_size);
7442                         return;
7443                 }
7444         }
7445         (void) (STRLOG(0, 0, 0, SL_CONSOLE,
7446             "callbparams_free: not found\n"));
7447 }

7449 void
7450 callbparams_free_id(syncq_t *sq, callbparams_id_t id, int32_t flag)
7451 {
7452         callbparams_t **pp, *p;

7454         ASSERT(MUTEX_HELD(SQLOCK(sq)));

7456         for (pp = &sq->sq_callbpend; (p = *pp) != NULL; pp = &p->cbp_next) {
7457                 if (p->cbp_id == id && p->cbp_flags == flag) {
7458                         *pp = p->cbp_next;
7459                         kmem_free(p, p->cbp_size);
7460                         return;
7461                 }
7462         }
7463         (void) (STRLOG(0, 0, 0, SL_CONSOLE,
7464             "callbparams_free_id: not found\n"));
7465 }

7467 /*
7468  * Callback wrapper function used by once-only callbacks that can be
7469  * cancelled (qtimeout and qbufcall)
7470  * Contains inline version of entersq(sq, SQ_CALLBACK) that can be
7471  * cancelled by the qun* functions.
7472  */
7473 void
7474 qcallbwrapper(void *arg)
7475 {
7476         callbparams_t *cbp = arg;
7477         syncq_t *sq;
7478         uint16_t count = 0;
7479         uint16_t waitflags = SQ_STAYAWAY | SQ_EVENTS | SQ_EXCL;
7480         uint16_t type;

7482         sq = cbp->cbp_sq;
7483         mutex_enter(SQLOCK(sq));
7484         type = sq->sq_type;
7485         if (!(type & SQ_CICB)) {
7486                 count = sq->sq_count;
7487                 SQ_PUTLOCKS_ENTER(sq);
7488                 SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
7489                 SUM_SQ_PUTCOUNTS(sq, count);
7490                 sq->sq_needexcl++;
7491                 ASSERT(sq->sq_needexcl != 0);   /* wraparound */
7492                 waitflags |= SQ_MESSAGES;
7493         }
7494         /* Can not handle exclusive entry at outer perimeter */
7495         ASSERT(type & SQ_COCB);

7497         while ((sq->sq_flags & waitflags) || (!(type & SQ_CICB) &&count != 0)) {
7498                 if ((sq->sq_callbflags & cbp->cbp_flags) &&
```

```
7499                    (sq->sq_cancelid == cbp->cbp_id)) {
7500                        /* timeout has been cancelled */
7501                        sq->sq_callbflags |= SQ_CALLB_BYPASSED;
7502                        callbparams_free(sq, cbp);
7503                        if (!(type & SQ_CICB)) {
7504                                ASSERT(sq->sq_needexcl > 0);
7505                                sq->sq_needexcl--;
7506                                if (sq->sq_needexcl == 0) {
7507                                        SQ_PUTCOUNT_SETFAST_LOCKED(sq);
7508                                }
7509                                SQ_PUTLOCKS_EXIT(sq);
7510                        }
7511                        mutex_exit(SQLOCK(sq));
7512                        return;
7513                }
7514                sq->sq_flags |= SQ_WANTWAKEUP;
7515                if (!(type & SQ_CICB)) {
7516                        SQ_PUTLOCKS_EXIT(sq);
7517                }
7518                cv_wait(&sq->sq_wait, SQLOCK(sq));
7519                if (!(type & SQ_CICB)) {
7520                        count = sq->sq_count;
7521                        SQ_PUTLOCKS_ENTER(sq);
7522                        SUM_SQ_PUTCOUNTS(sq, count);
7523                }
7524        }

7526        sq->sq_count++;
7527        ASSERT(sq->sq_count != 0);      /* Wraparound */
7528        if (!(type & SQ_CICB)) {
7529                ASSERT(count == 0);
7530                sq->sq_flags |= SQ_EXCL;
7531                ASSERT(sq->sq_needexcl > 0);
7532                sq->sq_needexcl--;
7533                if (sq->sq_needexcl == 0) {
7534                        SQ_PUTCOUNT_SETFAST_LOCKED(sq);
7535                }
7536                SQ_PUTLOCKS_EXIT(sq);
7537        }

7539        mutex_exit(SQLOCK(sq));

7541        cbp->cbp_func(cbp->cbp_arg);

7543        /*
7544         * We drop the lock only for leavesq to re-acquire it.
7545         * Possible optimization is inline of leavesq.
7546         */
7547        mutex_enter(SQLOCK(sq));
7548        callbparams_free(sq, cbp);
7549        mutex_exit(SQLOCK(sq));
7550        leavesq(sq, SQ_CALLBACK);
7551 }

7553 /*
7554  * No need to grab sq_putlocks here. See comment in strsubr.h that
7555  * explains when sq_putlocks are used.
7556  *
7557  * sq_count (or one of the sq_putcounts) has already been
7558  * decremented by the caller, and if SQ_QUEUED, we need to call
7559  * drain_syncq (the global syncq drain).
7560  * If putnext_tail is called with the SQ_EXCL bit set, we are in
7561  * one of two states, non-CIPUT perimeter, and we need to clear
7562  * it, or we went exclusive in the put procedure.  In any case,
7563  * we want to clear the bit now, and it is probably easier to do
7564  * this at the beginning of this function (remember, we hold
```



```
7565  * the SQLOCK).  Lastly, if there are other messages queued
7566  * on the syncq (and not for our destination), enable the syncq
7567  * for background work.
7568  */

7570 /* ARGSUSED */
7571 void
7572 putnext_tail(syncq_t *sq, queue_t *qp, uint32_t passflags)
7573 {
7574        uint16_t        flags = sq->sq_flags;

7576        ASSERT(MUTEX_HELD(SQLOCK(sq)));
7577        ASSERT(MUTEX_NOT_HELD(QLOCK(qp)));

7579        /* Clear SQ_EXCL if set in passflags */
7580        if (passflags & SQ_EXCL) {
7581                flags &= ~SQ_EXCL;
7582        }
7583        if (flags & SQ_WANTWAKEUP) {
7584                flags &= ~SQ_WANTWAKEUP;
7585                cv_broadcast(&sq->sq_wait);
7586        }
7587        if (flags & SQ_WANTEXWAKEUP) {
7588                flags &= ~SQ_WANTEXWAKEUP;
7589                cv_broadcast(&sq->sq_exitwait);
7590        }
7591        sq->sq_flags = flags;

7593        /*
7594         * We have cleared SQ_EXCL if we were asked to, and started
7595         * the wakeup process for waiters.  If there are no writers
7596         * then we need to drain the syncq if we were told to, or
7597         * enable the background thread to do it.
7598         */
7599        if (!(flags & (SQ_STAYAWAY|SQ_EXCL))) {
7600                if ((passflags & SQ_QUEUED) ||
7601                    (sq->sq_svcflags & SQ_DISABLED)) {
7602                        /* drain_syncq will take care of events in the list */
7603                        drain_syncq(sq);
7604                        return;
7605                } else if (flags & SQ_QUEUED) {
7606                        sqenable(sq);
7607                }
7608        }
7609        /* Drop the SQLOCK on exit */
7610        mutex_exit(SQLOCK(sq));
7611        TRACE_3(TR_FAC_STREAMS_FR, TR_PUTNEXT_END,
7612            "putnext_end:(%p, %p, %p) done", NULL, qp, sq);
7613 }

7615 void
7616 set_qend(queue_t *q)
7617 {
7618        mutex_enter(QLOCK(q));
7619        if (!O_SAMESTR(q))
7620                q->q_flag |= QEND;
7621        else
7622                q->q_flag &= ~QEND;
7623        mutex_exit(QLOCK(q));
7624        q = _OTHERQ(q);
7625        mutex_enter(QLOCK(q));
7626        if (!O_SAMESTR(q))
7627                q->q_flag |= QEND;
7628        else
7629                q->q_flag &= ~QEND;
7630        mutex_exit(QLOCK(q));
```

```
}

/*
 * Set QFULL in next service procedure queue (that cares) if not already
 * set and if there are already more messages on the syncq than
 * sq_max_size.  If sq_max_size is 0, no flow control will be asserted on
 * any syncq.
 *
 * The fq here is the next queue with a service procedure.  This is where
 * we would fail canputnext, so this is where we need to set QFULL.
 * In the case when fq != q we need to take QLOCK(fq) to set QFULL flag.
 *
 * We already have QLOCK at this point. To avoid cross-locks with
 * freezestr() which grabs all QLOCKs and with strlock() which grabs both
 * SQLOCK and sd_reflock, we need to drop respective locks first.
 */
void
set_qfull(queue_t *q)
{
        queue_t         *fq = NULL;

        ASSERT(MUTEX_HELD(QLOCK(q)));
        if ((sq_max_size != 0) && (!(q->q_nfsrv->q_flag & QFULL)) &&
            (q->q_syncqmsgs > sq_max_size)) {
                if ((fq = q->q_nfsrv) == q) {
                        fq->q_flag |= QFULL;
                } else {
                        mutex_exit(QLOCK(q));
                        mutex_enter(QLOCK(fq));
                        fq->q_flag |= QFULL;
                        mutex_exit(QLOCK(fq));
                        mutex_enter(QLOCK(q));
                }
        }
}

void
clr_qfull(queue_t *q)
{
        queue_t *oq = q;

        q = q->q_nfsrv;
        /* Fast check if there is any work to do before getting the lock. */
        if ((q->q_flag & (QFULL|QWANTW)) == 0) {
                return;
        }

        /*
         * Do not reset QFULL (and backenable) if the q_count is the reason
         * for QFULL being set.
         */
        mutex_enter(QLOCK(q));
        /*
         * If queue is empty i.e q_mblkcnt is zero, queue can not be full.
         * Hence clear the QFULL.
         * If both q_count and q_mblkcnt are less than the hiwat mark,
         * clear the QFULL.
         */
        if (q->q_mblkcnt == 0 || ((q->q_count < q->q_hiwat) &&
            (q->q_mblkcnt < q->q_hiwat))) {
                q->q_flag &= ~QFULL;
                /*
                 * A little more confusing, how about this way:
                 * if someone wants to write,
                 * AND
                 *    both counts are less than the lowat mark
```



```
                 *    OR
                 *    the lowat mark is zero
                 * THEN
                 * backenable
                 */
                if ((q->q_flag & QWANTW) &&
                    (((q->q_count < q->q_lowat) &&
                    (q->q_mblkcnt < q->q_lowat)) || q->q_lowat == 0)) {
                        q->q_flag &= ~QWANTW;
                        mutex_exit(QLOCK(q));
                        backenable(oq, 0);
                } else
                        mutex_exit(QLOCK(q));
        } else
                mutex_exit(QLOCK(q));
}

/*
 * Set the forward service procedure pointer.
 *
 * Called at insert-time to cache a queue's next forward service procedure in
 * q_nfsrv; used by canput() and canputnext().  If the queue to be inserted
 * has a service procedure then q_nfsrv points to itself.  If the queue to be
 * inserted does not have a service procedure, then q_nfsrv points to the next
 * queue forward that has a service procedure.  If the queue is at the logical
 * end of the stream (driver for write side, stream head for the read side)
 * and does not have a service procedure, then q_nfsrv also points to itself.
 */
void
set_nfsrv_ptr(
        queue_t  *rnew,         /* read queue pointer to new module */
        queue_t  *wnew,         /* write queue pointer to new module */
        queue_t  *prev_rq,      /* read queue pointer to the module above */
        queue_t  *prev_wq)      /* write queue pointer to the module above */
{
        queue_t *qp;

        if (prev_wq->q_next == NULL) {
                /*
                 * Insert the driver, initialize the driver and stream head.
                 * In this case, prev_rq/prev_wq should be the stream head.
                 * _I_INSERT does not allow inserting a driver.  Make sure
                 * that it is not an insertion.
                 */
                ASSERT(!(rnew->q_flag & _QINSERTING));
                wnew->q_nfsrv = wnew;
                if (rnew->q_qinfo->qi_srvp)
                        rnew->q_nfsrv = rnew;
                else
                        rnew->q_nfsrv = prev_rq;
                prev_rq->q_nfsrv = prev_rq;
                prev_wq->q_nfsrv = prev_wq;
        } else {
                /*
                 * set up read side q_nfsrv pointer.  This MUST be done
                 * before setting the write side, because the setting of
                 * the write side for a fifo may depend on it.
                 *
                 * Suppose we have a fifo that only has pipemod pushed.
                 * pipemod has no read or write service procedures, so
                 * nfsrv for both pipemod queues points to prev_rq (the
                 * stream read head).  Now push bufmod (which has only a
                 * read service procedure).  Doing the write side first,
                 * wnew->q_nfsrv is set to pipemod's writeq nfsrv, which
                 * is WRONG; the next queue forward from wnew with a
                 * service procedure will be rnew, not the stream read head.
```

```
7763                  * Since the downstream queue (which in the case of a fifo
7764                  * is the read queue rnew) can affect upstream queues, it
7765                  * needs to be done first.  Setting up the read side first
7766                  * sets nfsrv for both pipemod queues to rnew and then
7767                  * when the write side is set up, wnew-q_nfsrv will also
7768                  * point to rnew.
7769                  */
7770                 if (rnew->q_qinfo->qi_srvp) {
7771                         /*
7772                          * use _OTHERQ() because, if this is a pipe, next
7773                          * module may have been pushed from other end and
7774                          * q_next could be a read queue.
7775                          */
7776                         qp = _OTHERQ(prev_wq->q_next);
7777                         while (qp && qp->q_nfsrv != qp) {
7778                                 qp->q_nfsrv = rnew;
7779                                 qp = backq(qp);
7780                         }
7781                         rnew->q_nfsrv = rnew;
7782                 } else
7783                         rnew->q_nfsrv = prev_rq->q_nfsrv;

7785                 /* set up write side q_nfsrv pointer */
7786                 if (wnew->q_qinfo->qi_srvp) {
7787                         wnew->q_nfsrv = wnew;

7789                         /*
7790                          * For insertion, need to update nfsrv of the modules
7791                          * above which do not have a service routine.
7792                          */
7793                         if (rnew->q_flag & _QINSERTING) {
7794                                 for (qp = prev_wq;
7795                                     qp != NULL && qp->q_nfsrv != qp;
7796                                     qp = backq(qp)) {
7797                                         qp->q_nfsrv = wnew->q_nfsrv;
7798                                 }
7799                         }
7800                 } else {
7801                         if (prev_wq->q_next == prev_rq)
7802                                 /*
7803                                  * Since prev_wq/prev_rq are the middle of a
7804                                  * fifo, wnew/rnew will also be the middle of
7805                                  * a fifo and wnew's nfsrv is same as rnew's.
7806                                  */
7807                                 wnew->q_nfsrv = rnew->q_nfsrv;
7808                         else
7809                                 wnew->q_nfsrv = prev_wq->q_next->q_nfsrv;
7810                 }
7811         }
7812 }

7814 /*
7815  * Reset the forward service procedure pointer; called at remove-time.
7816  */
7817 void
7818 reset_nfsrv_ptr(queue_t *rqp, queue_t *wqp)
7819 {
7820         queue_t *tmp_qp;

7822         /* Reset the write side q_nfsrv pointer for _I_REMOVE */
7823         if ((rqp->q_flag & _QREMOVING) && (wqp->q_qinfo->qi_srvp != NULL)) {
7824                 for (tmp_qp = backq(wqp);
7825                     tmp_qp != NULL && tmp_qp->q_nfsrv == wqp;
7826                     tmp_qp = backq(tmp_qp)) {
7827                         tmp_qp->q_nfsrv = wqp->q_nfsrv;
7828                 }
```



```
7829         }

7831         /* reset the read side q_nfsrv pointer */
7832         if (rqp->q_qinfo->qi_srvp) {
7833                 if (wqp->q_next) {      /* non-driver case */
7834                         tmp_qp = _OTHERQ(wqp->q_next);
7835                         while (tmp_qp && tmp_qp->q_nfsrv == rqp) {
7836                                 /* Note that rqp->q_next cannot be NULL */
7837                                 ASSERT(rqp->q_next != NULL);
7838                                 tmp_qp->q_nfsrv = rqp->q_next->q_nfsrv;
7839                                 tmp_qp = backq(tmp_qp);
7840                         }
7841                 }
7842         }
7843 }

7845 /*
7846  * This routine should be called after all stream geometry changes to update
7847  * the stream head cached struio() rd/wr queue pointers. Note must be called
7848  * with the streamlock()ed.
7849  *
7850  * Note: only enables Synchronous STREAMS for a side of a Stream which has
7851  *       an explicit synchronous barrier module queue. That is, a queue that
7852  *       has specified a struio() type.
7853  */
7854 static void
7855 strsetuio(stdata_t *stp)
7856 {
7857         queue_t *wrq;

7859         if (stp->sd_flag & STPLEX) {
7860                 /*
7861                  * Not streamhead, but a mux, so no Synchronous STREAMS.
7862                  */
7863                 stp->sd_struiowrq = NULL;
7864                 stp->sd_struiordq = NULL;
7865                 return;
7866         }
7867         /*
7868          * Scan the write queue(s) while synchronous
7869          * until we find a qinfo uio type specified.
7870          */
7871         wrq = stp->sd_wrq->q_next;
7872         while (wrq) {
7873                 if (wrq->q_struiot == STRUIOT_NONE) {
7874                         wrq = 0;
7875                         break;
7876                 }
7877                 if (wrq->q_struiot != STRUIOT_DONTCARE)
7878                         break;
7879                 if (! _SAMESTR(wrq)) {
7880                         wrq = 0;
7881                         break;
7882                 }
7883                 wrq = wrq->q_next;
7884         }
7885         stp->sd_struiowrq = wrq;
7886         /*
7887          * Scan the read queue(s) while synchronous
7888          * until we find a qinfo uio type specified.
7889          */
7890         wrq = stp->sd_wrq->q_next;
7891         while (wrq) {
7892                 if (_RD(wrq)->q_struiot == STRUIOT_NONE) {
7893                         wrq = 0;
7894                         break;
```

```
7895                }
7896                if (_RD(wrq)->q_struiot != STRUIOT_DONTCARE)
7897                        break;
7898                if (! _SAMESTR(wrq)) {
7899                        wrq = 0;
7900                        break;
7901                }
7902                wrq = wrq->q_next;
7903        }
7904        stp->sd_struiordq = wrq ? _RD(wrq) : 0;
7905 }

7907 /*
7908  * pass_wput, unblocks the passthru queues, so that
7909  * messages can arrive at muxs lower read queue, before
7910  * I_LINK/I_UNLINK is acked/nacked.
7911  */
7912 static void
7913 pass_wput(queue_t *q, mblk_t *mp)
7914 {
7915         syncq_t *sq;

7917         sq = _RD(q)->q_syncq;
7918         if (sq->sq_flags & SQ_BLOCKED)
7919                 unblocksq(sq, SQ_BLOCKED, 0);
7920         putnext(q, mp);
7921 }

7923 /*
7924  * Set up queues for the link/unlink.
7925  * Create a new queue and block it and then insert it
7926  * below the stream head on the lower stream.
7927  * This prevents any messages from arriving during the setq
7928  * as well as while the mux is processing the LINK/I_UNLINK.
7929  * The blocked passq is unblocked once the LINK/I_UNLINK has
7930  * been acked or nacked or if a message is generated and sent
7931  * down muxs write put procedure.
7932  * See pass_wput().
7933  *
7934  * After the new queue is inserted, all messages coming from below are
7935  * blocked. The call to strlock will ensure that all activity in the stream head
7936  * read queue syncq is stopped (sq_count drops to zero).
7937  */
7938 static queue_t *
7939 link_addpassthru(stdata_t *stpdown)
7940 {
7941         queue_t *passq;
7942         sqlist_t sqlist;

7944         passq = allocq();
7945         STREAM(passq) = STREAM(_WR(passq)) = stpdown;
7946         /* setq might sleep in allocator - avoid holding locks. */
7947         setq(passq, &passthru_rinit, &passthru_winit, NULL, QPERQ,
7948             SQ_CI|SQ_CO, B_FALSE);
7949         claimq(passq);
7950         blocksq(passq->q_syncq, SQ_BLOCKED, 1);
7951         insertq(STREAM(passq), passq);

7953         /*
7954          * Use strlock() to wait for the stream head sq_count to drop to zero
7955          * since we are going to change q_ptr in the stream head.  Note that
7956          * insertq() doesn't wait for any syncq counts to drop to zero.
7957          */
7958         sqlist.sqlist_head = NULL;
7959         sqlist.sqlist_index = 0;
7960         sqlist.sqlist_size = sizeof (sqlist_t);
```



```
7961         sqlist_insert(&sqlist, _RD(stpdown->sd_wrq)->q_syncq);
7962         strlock(stpdown, &sqlist);
7963         strunlock(stpdown, &sqlist);

7965         releaseq(passq);
7966         return (passq);
7967 }

7969 /*
7970  * Let messages flow up into the mux by removing
7971  * the passq.
7972  */
7973 static void
7974 link_rempassthru(queue_t *passq)
7975 {
7976         claimq(passq);
7977         removeq(passq);
7978         releaseq(passq);
7979         freeq(passq);
7980 }

7982 /*
7983  * Wait for the condition variable pointed to by 'cvp' to be signaled,
7984  * or for 'tim' milliseconds to elapse, whichever comes first.  If 'tim'
7985  * is negative, then there is no time limit.  If 'nosigs' is non-zero,
7986  * then the wait will be non-interruptible.
7987  *
7988  * Returns >0 if signaled, 0 if interrupted, or -1 upon timeout.
7989  */
7990 clock_t
7991 str_cv_wait(kcondvar_t *cvp, kmutex_t *mp, clock_t tim, int nosigs)
7992 {
7993         clock_t ret;

7995         if (tim < 0) {
7996                 if (nosigs) {
7997                         cv_wait(cvp, mp);
7998                         ret = 1;
7999                 } else {
8000                         ret = cv_wait_sig(cvp, mp);
8001                 }
8002         } else if (tim > 0) {
8003                 /*
8004                  * convert milliseconds to clock ticks
8005                  */
8006                 if (nosigs) {
8007                         ret = cv_reltimedwait(cvp, mp,
8008                             MSEC_TO_TICK_ROUNDUP(tim), TR_CLOCK_TICK);
8009                 } else {
8010                         ret = cv_reltimedwait_sig(cvp, mp,
8011                             MSEC_TO_TICK_ROUNDUP(tim), TR_CLOCK_TICK);
8012                 }
8013         } else {
8014                 ret = -1;
8015         }
8016         return (ret);
8017 }

8019 /*
8020  * Wait until the stream head can determine if it is at the mark but
8021  * don't wait forever to prevent a race condition between the "mark" state
8022  * in the stream head and any mark state in the caller/user of this routine.
8023  *
8024  * This is used by sockets and for a socket it would be incorrect
8025  * to return a failure for SIOCATMARK when there is no data in the receive
8026  * queue and the marked urgent data is traveling up the stream.
```

```
8027  *
8028  * This routine waits until the mark is known by waiting for one of these
8029  * three events:
8030  *      The stream head read queue becoming non-empty (including an EOF).
8031  *      The STRATMARK flag being set (due to a MSGMARKNEXT message).
8032  *      The STRNOTATMARK flag being set (which indicates that the transport
8033  *      has sent a MSGNOTMARKNEXT message to indicate that it is not at
8034  *      the mark).
8035  *
8036  * The routine returns 1 if the stream is at the mark; 0 if it can
8037  * be determined that the stream is not at the mark.
8038  * If the wait times out and it can't determine
8039  * whether or not the stream might be at the mark the routine will return -1.
8040  *
8041  * Note: This routine should only be used when a mark is pending i.e.,
8042  * in the socket case the SIGURG has been posted.
8043  * Note2: This can not wakeup just because synchronous streams indicate
8044  * that data is available since it is not possible to use the synchronous
8045  * streams interfaces to determine the b_flag value for the data queued below
8046  * the stream head.
8047  */
8048 int
8049 strwaitmark(vnode_t *vp)
8050 {
8051         struct stdata *stp = vp->v_stream;
8052         queue_t *rq = _RD(stp->sd_wrq);
8053         int mark;

8055         mutex_enter(&stp->sd_lock);
8056         while (rq->q_first == NULL &&
8057             !(stp->sd_flag & (STRATMARK|STRNOTATMARK|STREOF))) {
8058                 stp->sd_flag |= RSLEEP;

8060                 /* Wait for 100 milliseconds for any state change. */
8061                 if (str_cv_wait(&rq->q_wait, &stp->sd_lock, 100, 1) == -1) {
8062                         mutex_exit(&stp->sd_lock);
8063                         return (-1);
8064                 }
8065         }
8066         if (stp->sd_flag & STRATMARK)
8067                 mark = 1;
8068         else if (rq->q_first != NULL && (rq->q_first->b_flag & MSGMARK))
8069                 mark = 1;
8070         else
8071                 mark = 0;

8073         mutex_exit(&stp->sd_lock);
8074         return (mark);
8075 }

8077 /*
8078  * Set a read side error. If persist is set change the socket error
8079  * to persistent. If errfunc is set install the function as the exported
8080  * error handler.
8081  */
8082 void
8083 strsetrerror(vnode_t *vp, int error, int persist, errfunc_t errfunc)
8084 {
8085         struct stdata *stp = vp->v_stream;

8087         mutex_enter(&stp->sd_lock);
8088         stp->sd_rerror = error;
8089         if (error == 0 && errfunc == NULL)
8090                 stp->sd_flag &= ~STRDERR;
8091         else
8092                 stp->sd_flag |= STRDERR;
```



```
8093         if (persist) {
8094                 stp->sd_flag &= ~STRDERRNONPERSIST;
8095         } else {
8096                 stp->sd_flag |= STRDERRNONPERSIST;
8097         }
8098         stp->sd_rderrfunc = errfunc;
8099         if (error != 0 || errfunc != NULL) {
8100                 cv_broadcast(&_RD(stp->sd_wrq)->q_wait);        /* readers */
8101                 cv_broadcast(&stp->sd_wrq->q_wait);             /* writers */
8102                 cv_broadcast(&stp->sd_monitor);                 /* ioctllers */

8104                 mutex_exit(&stp->sd_lock);
8105                 pollwakeup(&stp->sd_pollist, POLLERR);
8106                 mutex_enter(&stp->sd_lock);

8108                 if (stp->sd_sigflags & S_ERROR)
8109                         strsendsig(stp->sd_siglist, S_ERROR, 0, error);
8110         }
8111         mutex_exit(&stp->sd_lock);
8112 }

8114 /*
8115  * Set a write side error. If persist is set change the socket error
8116  * to persistent.
8117  */
8118 void
8119 strsetwerror(vnode_t *vp, int error, int persist, errfunc_t errfunc)
8120 {
8121         struct stdata *stp = vp->v_stream;

8123         mutex_enter(&stp->sd_lock);
8124         stp->sd_werror = error;
8125         if (error == 0 && errfunc == NULL)
8126                 stp->sd_flag &= ~STWRERR;
8127         else
8128                 stp->sd_flag |= STWRERR;
8129         if (persist) {
8130                 stp->sd_flag &= ~STWRERRNONPERSIST;
8131         } else {
8132                 stp->sd_flag |= STWRERRNONPERSIST;
8133         }
8134         stp->sd_wrerrfunc = errfunc;
8135         if (error != 0 || errfunc != NULL) {
8136                 cv_broadcast(&_RD(stp->sd_wrq)->q_wait);        /* readers */
8137                 cv_broadcast(&stp->sd_wrq->q_wait);             /* writers */
8138                 cv_broadcast(&stp->sd_monitor);                 /* ioctllers */

8140                 mutex_exit(&stp->sd_lock);
8141                 pollwakeup(&stp->sd_pollist, POLLERR);
8142                 mutex_enter(&stp->sd_lock);

8144                 if (stp->sd_sigflags & S_ERROR)
8145                         strsendsig(stp->sd_siglist, S_ERROR, 0, error);
8146         }
8147         mutex_exit(&stp->sd_lock);
8148 }

8150 /*
8151  * Make the stream return 0 (EOF) when all data has been read.
8152  * No effect on write side.
8153  */
8154 void
8155 strseteof(vnode_t *vp, int eof)
8156 {
8157         struct stdata *stp = vp->v_stream;
```

```
8159         mutex_enter(&stp->sd_lock);
8160         if (!eof) {
8161                 stp->sd_flag &= ~STREOF;
8162                 mutex_exit(&stp->sd_lock);
8163                 return;
8164         }
8165         stp->sd_flag |= STREOF;
8166         if (stp->sd_flag & RSLEEP) {
8167                 stp->sd_flag &= ~RSLEEP;
8168                 cv_broadcast(&_RD(stp->sd_wrq)->q_wait);
8169         }

8171         mutex_exit(&stp->sd_lock);
8172         pollwakeup(&stp->sd_pollist, POLLIN|POLLRDNORM);
8173         mutex_enter(&stp->sd_lock);

8175         if (stp->sd_sigflags & (S_INPUT|S_RDNORM))
8176                 strsendsig(stp->sd_siglist, S_INPUT|S_RDNORM, 0, 0);
8177         mutex_exit(&stp->sd_lock);
8178 }

8180 void
8181 strflushrq(vnode_t *vp, int flag)
8182 {
8183         struct stdata *stp = vp->v_stream;

8185         mutex_enter(&stp->sd_lock);
8186         flushq(_RD(stp->sd_wrq), flag);
8187         mutex_exit(&stp->sd_lock);
8188 }

8190 void
8191 strsetrputhooks(vnode_t *vp, uint_t flags,
8192                 msgfunc_t protofunc, msgfunc_t miscfunc)
8193 {
8194         struct stdata *stp = vp->v_stream;

8196         mutex_enter(&stp->sd_lock);

8198         if (protofunc == NULL)
8199                 stp->sd_rprotofunc = strrput_proto;
8200         else
8201                 stp->sd_rprotofunc = protofunc;

8203         if (miscfunc == NULL)
8204                 stp->sd_rmiscfunc = strrput_misc;
8205         else
8206                 stp->sd_rmiscfunc = miscfunc;

8208         if (flags & SH_CONSOL_DATA)
8209                 stp->sd_rput_opt |= SR_CONSOL_DATA;
8210         else
8211                 stp->sd_rput_opt &= ~SR_CONSOL_DATA;

8213         if (flags & SH_SIGALLDATA)
8214                 stp->sd_rput_opt |= SR_SIGALLDATA;
8215         else
8216                 stp->sd_rput_opt &= ~SR_SIGALLDATA;

8218         if (flags & SH_IGN_ZEROLEN)
8219                 stp->sd_rput_opt |= SR_IGN_ZEROLEN;
8220         else
8221                 stp->sd_rput_opt &= ~SR_IGN_ZEROLEN;

8223         mutex_exit(&stp->sd_lock);
8224 }
```



```
8226 void
8227 strsetwputhooks(vnode_t *vp, uint_t flags, clock_t closetime)
8228 {
8229         struct stdata *stp = vp->v_stream;

8231         mutex_enter(&stp->sd_lock);
8232         stp->sd_closetime = closetime;

8234         if (flags & SH_SIGPIPE)
8235                 stp->sd_wput_opt |= SW_SIGPIPE;
8236         else
8237                 stp->sd_wput_opt &= ~SW_SIGPIPE;
8238         if (flags & SH_RECHECK_ERR)
8239                 stp->sd_wput_opt |= SW_RECHECK_ERR;
8240         else
8241                 stp->sd_wput_opt &= ~SW_RECHECK_ERR;

8243         mutex_exit(&stp->sd_lock);
8244 }

8246 void
8247 strsetrwputdatahooks(vnode_t *vp, msgfunc_t rdatafunc, msgfunc_t wdatafunc)
8248 {
8249         struct stdata *stp = vp->v_stream;

8251         mutex_enter(&stp->sd_lock);

8253         stp->sd_rputdatafunc = rdatafunc;
8254         stp->sd_wputdatafunc = wdatafunc;

8256         mutex_exit(&stp->sd_lock);
8257 }

8259 /* Used within framework when the queue is already locked */
8260 void
8261 qenable_locked(queue_t *q)
8262 {
8263         stdata_t *stp = STREAM(q);

8265         ASSERT(MUTEX_HELD(QLOCK(q)));

8267         if (!q->q_qinfo->qi_srvp)
8268                 return;

8270         /*
8271          * Do not place on run queue if already enabled or closing.
8272          */
8273         if (q->q_flag & (QWCLOSE|QENAB))
8274                 return;

8276         /*
8277          * mark queue enabled and place on run list if it is not already being
8278          * serviced. If it is serviced, the runservice() function will detect
8279          * that QENAB is set and call service procedure before clearing
8280          * QINSERVICE flag.
8281          */
8282         q->q_flag |= QENAB;
8283         if (q->q_flag & QINSERVICE)
8284                 return;

8286         /* Record the time of qenable */
8287         q->q_qtstamp = ddi_get_lbolt();

8289         /*
8290          * Put the queue in the stp list and schedule it for background
```

```
8291          * processing if it is not already scheduled or if stream head does not
8292          * intent to process it in the foreground later by setting
8293          * STRS_WILLSERVICE flag.
8294          */
8295         mutex_enter(&stp->sd_qlock);
8296         /*
8297          * If there are already something on the list, stp flags should show
8298          * intention to drain it.
8299          */
8300         IMPLY(STREAM_NEEDSERVICE(stp),
8301             (stp->sd_svcflags & (STRS_WILLSERVICE | STRS_SCHEDULED)));

8303         ENQUEUE(q, stp->sd_qhead, stp->sd_qtail, q_link);
8304         stp->sd_nqueues++;

8306         /*
8307          * If no one will drain this stream we are the first producer and
8308          * need to schedule it for background thread.
8309          */
8310         if (!(stp->sd_svcflags & (STRS_WILLSERVICE | STRS_SCHEDULED))) {
8311                 /*
8312                  * No one will service this stream later, so we have to
8313                  * schedule it now.
8314                  */
8315                 STRSTAT(stenables);
8316                 stp->sd_svcflags |= STRS_SCHEDULED;
8317                 stp->sd_servid = (void *)taskq_dispatch(streams_taskq,
8318                     (task_func_t *)stream_service, stp, TQ_NOSLEEP|TQ_NOQUEUE);

8320                 if (stp->sd_servid == NULL) {
8321                         /*
8322                          * Task queue failed so fail over to the backup
8323                          * servicing thread.
8324                          */
8325                         STRSTAT(taskqfails);
8326                         /*
8327                          * It is safe to clear STRS_SCHEDULED flag because it
8328                          * was set by this thread above.
8329                          */
8330                         stp->sd_svcflags &= ~STRS_SCHEDULED;

8332                         /*
8333                          * Failover scheduling is protected by service_queue
8334                          * lock.
8335                          */
8336                         mutex_enter(&service_queue);
8337                         ASSERT((stp->sd_qhead == q) && (stp->sd_qtail == q));
8338                         ASSERT(q->q_link == NULL);
8339                         /*
8340                          * Append the queue to qhead/qtail list.
8341                          */
8342                         if (qhead == NULL)
8343                                 qhead = q;
8344                         else
8345                                 qtail->q_link = q;
8346                         qtail = q;
8347                         /*
8348                          * Clear stp queue list.
8349                          */
8350                         stp->sd_qhead = stp->sd_qtail = NULL;
8351                         stp->sd_nqueues = 0;
8352                         /*
8353                          * Wakeup background queue processing thread.
8354                          */
8355                         cv_signal(&services_to_run);
8356                         mutex_exit(&service_queue);
```



```
8357                 }
8358         }
8359         mutex_exit(&stp->sd_qlock);
8360 }

8362 static void
8363 queue_service(queue_t *q)
8364 {
8365         /*
8366          * The queue in the list should have
8367          * QENAB flag set and should not have
8368          * QINSERVICE flag set. QINSERVICE is
8369          * set when the queue is dequeued and
8370          * qenable_locked doesn't enqueue a
8371          * queue with QINSERVICE set.
8372          */

8374         ASSERT(!(q->q_flag & QINSERVICE));
8375         ASSERT((q->q_flag & QENAB));
8376         mutex_enter(QLOCK(q));
8377         q->q_flag &= ~QENAB;
8378         q->q_flag |= QINSERVICE;
8379         mutex_exit(QLOCK(q));
8380         runservice(q);
8381 }

8383 static void
8384 syncq_service(syncq_t *sq)
8385 {
8386         STRSTAT(syncqservice);
8387         mutex_enter(SQLOCK(sq));
8388         ASSERT(!(sq->sq_svcflags & SQ_SERVICE));
8389         ASSERT(sq->sq_servcount != 0);
8390         ASSERT(sq->sq_next == NULL);

8392         /* if we came here from the background thread, clear the flag */
8393         if (sq->sq_svcflags & SQ_BGTHREAD)
8394                 sq->sq_svcflags &= ~SQ_BGTHREAD;

8396         /* let drain_syncq know that it's being called in the background */
8397         sq->sq_svcflags |= SQ_SERVICE;
8398         drain_syncq(sq);
8399 }

8401 static void
8402 qwriter_outer_service(syncq_t *outer)
8403 {
8404         /*
8405          * Note that SQ_WRITER is used on the outer perimeter
8406          * to signal that a qwriter(OUTER) is either investigating
8407          * running or that it is actually running a function.
8408          */
8409         outer_enter(outer, SQ_BLOCKED|SQ_WRITER);

8411         /*
8412          * All inner syncq are empty and have SQ_WRITER set
8413          * to block entering the outer perimeter.
8414          *
8415          * We do not need to explicitly call write_now since
8416          * outer_exit does it for us.
8417          */
8418         outer_exit(outer);
8419 }

8421 static void
8422 mblk_free(mblk_t *mp)
```

```
8423 {
8424         dblk_t *dbp = mp->b_datap;
8425         frtn_t *frp = dbp->db_frtnp;

8427         mp->b_next = NULL;
8428         if (dbp->db_fthdr != NULL)
8429                 str_ftfree(dbp);

8431         ASSERT(dbp->db_fthdr == NULL);
8432         frp->free_func(frp->free_arg);
8433         ASSERT(dbp->db_mblk == mp);

8435         if (dbp->db_credp != NULL) {
8436                 crfree(dbp->db_credp);
8437                 dbp->db_credp = NULL;
8438         }
8439         dbp->db_cpid = -1;
8440         dbp->db_struioflag = 0;
8441         dbp->db_struioun.cksum.flags = 0;

8443         kmem_cache_free(dbp->db_cache, dbp);
8444 }

8446 /*
8447  * Background processing of the stream queue list.
8448  */
8449 static void
8450 stream_service(stdata_t *stp)
8451 {
8452         queue_t *q;

8454         mutex_enter(&stp->sd_qlock);

8456         STR_SERVICE(stp, q);

8458         stp->sd_svcflags &= ~STRS_SCHEDULED;
8459         stp->sd_servid = NULL;
8460         cv_signal(&stp->sd_qcv);
8461         mutex_exit(&stp->sd_qlock);
8462 }

8464 /*
8465  * Foreground processing of the stream queue list.
8466  */
8467 void
8468 stream_runservice(stdata_t *stp)
8469 {
8470         queue_t *q;

8472         mutex_enter(&stp->sd_qlock);
8473         STRSTAT(rservice);
8474         /*
8475          * We are going to drain this stream queue list, so qenable_locked will
8476          * not schedule it until we finish.
8477          */
8478         stp->sd_svcflags |= STRS_WILLSERVICE;

8480         STR_SERVICE(stp, q);

8482         stp->sd_svcflags &= ~STRS_WILLSERVICE;
8483         mutex_exit(&stp->sd_qlock);
8484         /*
8485          * Help backup background thread to drain the qhead/qtail list.
8486          */
8487         while (qhead != NULL) {
8488                 STRSTAT(qhelps);
```



```
8489                 mutex_enter(&service_queue);
8490                 DQ(q, qhead, qtail, q_link);
8491                 mutex_exit(&service_queue);
8492                 if (q != NULL)
8493                         queue_service(q);
8494         }
8495 }

8497 void
8498 stream_willservice(stdata_t *stp)
8499 {
8500         mutex_enter(&stp->sd_qlock);
8501         stp->sd_svcflags |= STRS_WILLSERVICE;
8502         mutex_exit(&stp->sd_qlock);
8503 }

8505 /*
8506  * Replace the cred currently in the mblk with a different one.
8507  * Also update db_cpid.
8508  */
8509 void
8510 mblk_setcred(mblk_t *mp, cred_t *cr, pid_t cpid)
8511 {
8512         dblk_t *dbp = mp->b_datap;
8513         cred_t *ocr = dbp->db_credp;

8515         ASSERT(cr != NULL);

8517         if (cr != ocr) {
8518                 crhold(dbp->db_credp = cr);
8519                 if (ocr != NULL)
8520                         crfree(ocr);
8521         }
8522         /* Don't overwrite with NOPID */
8523         if (cpid != NOPID)
8524                 dbp->db_cpid = cpid;
8525 }

8527 /*
8528  * If the src message has a cred, then replace the cred currently in the mblk
8529  * with it.
8530  * Also update db_cpid.
8531  */
8532 void
8533 mblk_copycred(mblk_t *mp, const mblk_t *src)
8534 {
8535         dblk_t *dbp = mp->b_datap;
8536         cred_t *cr, *ocr;
8537         pid_t cpid;

8539         cr = msg_getcred(src, &cpid);
8540         if (cr == NULL)
8541                 return;

8543         ocr = dbp->db_credp;
8544         if (cr != ocr) {
8545                 crhold(dbp->db_credp = cr);
8546                 if (ocr != NULL)
8547                         crfree(ocr);
8548         }
8549         /* Don't overwrite with NOPID */
8550         if (cpid != NOPID)
8551                 dbp->db_cpid = cpid;
8552 }

8554 int
```

```
8555 hcksum_assoc(mblk_t *mp,  multidata_t *mmd, pdesc_t *pd,
8556     uint32_t start, uint32_t stuff, uint32_t end, uint32_t value,
8557     uint32_t flags, int km_flags)
8558 {
8559         int rc = 0;

8561         ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_MULTIDATA);
8562         if (mp->b_datap->db_type == M_DATA) {
8563                 /* Associate values for M_DATA type */
8564                 DB_CKSUMSTART(mp) = (intptr_t)start;
8565                 DB_CKSUMSTUFF(mp) = (intptr_t)stuff;
8566                 DB_CKSUMEND(mp) = (intptr_t)end;
8567                 DB_CKSUMFLAGS(mp) = flags;
8568                 DB_CKSUM16(mp) = (uint16_t)value;

8570         } else {
8571                 pattrinfo_t pa_info;

8573                 ASSERT(mmd != NULL);

8575                 pa_info.type = PATTR_HCKSUM;
8576                 pa_info.len = sizeof (pattr_hcksum_t);

8578                 if (mmd_addpattr(mmd, pd, &pa_info, B_TRUE, km_flags) != NULL) {
8579                         pattr_hcksum_t *hck = (pattr_hcksum_t *)pa_info.buf;

8581                         hck->hcksum_start_offset = start;
8582                         hck->hcksum_stuff_offset = stuff;
8583                         hck->hcksum_end_offset = end;
8584                         hck->hcksum_cksum_val.inet_cksum = (uint16_t)value;
8585                         hck->hcksum_flags = flags;
8586                 } else {
8587                         rc = -1;
8588                 }
8589         }
8590         return (rc);
8591 }

8593 void
8594 hcksum_retrieve(mblk_t *mp, multidata_t *mmd, pdesc_t *pd,
8595     uint32_t *start, uint32_t *stuff, uint32_t *end,
8596     uint32_t *value, uint32_t *flags)
8597 {
8598         ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_MULTIDATA);
8599         if (mp->b_datap->db_type == M_DATA) {
8600                 if (flags != NULL) {
8601                         *flags = DB_CKSUMFLAGS(mp) & HCK_FLAGS;
8602                         if ((*flags & (HCK_PARTIALCKSUM |
8603                             HCK_FULLCKSUM)) != 0) {
8604                                 if (value != NULL)
8605                                         *value = (uint32_t)DB_CKSUM16(mp);
8606                                 if ((*flags & HCK_PARTIALCKSUM) != 0) {
8607                                         if (start != NULL)
8608                                                 *start =
8609                                                     (uint32_t)DB_CKSUMSTART(mp);
8610                                         if (stuff != NULL)
8611                                                 *stuff =
8612                                                     (uint32_t)DB_CKSUMSTUFF(mp);
8613                                         if (end != NULL)
8614                                                 *end =
8615                                                     (uint32_t)DB_CKSUMEND(mp);
8616                                 }
8617                         }
8618                 }
8619         } else {
8620                 pattrinfo_t hck_attr = {PATTR_HCKSUM};
```



```
8622                 ASSERT(mmd != NULL);

8624                 /* get hardware checksum attribute */
8625                 if (mmd_getpattr(mmd, pd, &hck_attr) != NULL) {
8626                         pattr_hcksum_t *hck = (pattr_hcksum_t *)hck_attr.buf;

8628                         ASSERT(hck_attr.len >= sizeof (pattr_hcksum_t));
8629                         if (flags != NULL)
8630                                 *flags = hck->hcksum_flags;
8631                         if (start != NULL)
8632                                 *start = hck->hcksum_start_offset;
8633                         if (stuff != NULL)
8634                                 *stuff = hck->hcksum_stuff_offset;
8635                         if (end != NULL)
8636                                 *end = hck->hcksum_end_offset;
8637                         if (value != NULL)
8638                                 *value = (uint32_t)
8639                                     hck->hcksum_cksum_val.inet_cksum;
8640                 }
8641         }
8642 }

8644 void
8645 lso_info_set(mblk_t *mp, uint32_t mss, uint32_t flags)
8646 {
8647         ASSERT(DB_TYPE(mp) == M_DATA);
8648         ASSERT((flags & ~HW_LSO_FLAGS) == 0);

8650         /* Set the flags */
8651         DB_LSOFLAGS(mp) |= flags;
8652         DB_LSOMSS(mp) = mss;
8653 }

8655 void
8656 lso_info_cleanup(mblk_t *mp)
8657 {
8658         ASSERT(DB_TYPE(mp) == M_DATA);

8660         /* Clear the flags */
8661         DB_LSOFLAGS(mp) &= ~HW_LSO_FLAGS;
8662         DB_LSOMSS(mp) = 0;
8663 }

8665 /*
8666  * Checksum buffer *bp for len bytes with psum partial checksum,
8667  * or 0 if none, and return the 16 bit partial checksum.
8668  */
8669 unsigned
8670 bcksum(uchar_t *bp, int len, unsigned int psum)
8671 {
8672         int odd = len & 1;
8673         extern unsigned int ip_ocsum();

8675         if (((intptr_t)bp & 1) == 0 && !odd) {
8676                 /*
8677                  * Bp is 16 bit aligned and len is multiple of 16 bit word.
8678                  */
8679                 return (ip_ocsum((ushort_t *)bp, len >> 1, psum));
8680         }
8681         if (((intptr_t)bp & 1) != 0) {
8682                 /*
8683                  * Bp isn't 16 bit aligned.
8684                  */
8685                 unsigned int tsum;
```

```
#ifdef _LITTLE_ENDIAN
                psum += *bp;
#else
                psum += *bp << 8;
#endif
                len--;
                bp++;
                tsum = ip_ocsum((ushort_t *)bp, len >> 1, 0);
                psum += (tsum << 8) & 0xffff | (tsum >> 8);
                if (len & 1) {
                        bp += len - 1;
#ifdef _LITTLE_ENDIAN
                        psum += *bp << 8;
#else
                        psum += *bp;
#endif
                }
        } else {
                /*
                 * Bp is 16 bit aligned.
                 */
                psum = ip_ocsum((ushort_t *)bp, len >> 1, psum);
                if (odd) {
                        bp += len - 1;
#ifdef _LITTLE_ENDIAN
                        psum += *bp;
#else
                        psum += *bp << 8;
#endif
                }
        }
        /*
         * Normalize psum to 16 bits before returning the new partial
         * checksum. The max psum value before normalization is 0x3FDFE.
         */
        return ((psum >> 16) + (psum & 0xFFFF));
}

boolean_t
is_vmloaned_mblk(mblk_t *mp, multidata_t *mmd, pdesc_t *pd)
{
        boolean_t rc;

        ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_MULTIDATA);
        if (DB_TYPE(mp) == M_DATA) {
                rc = (((mp)->b_datap->db_struioflag & STRUIO_ZC) != 0);
        } else {
                pattrinfo_t zcopy_attr = {PATTR_ZCOPY};

                ASSERT(mmd != NULL);
                rc = (mmd_getpattr(mmd, pd, &zcopy_attr) != NULL);
        }
        return (rc);
}

void
freemsgchain(mblk_t *mp)
{
        mblk_t  *next;

        while (mp != NULL) {
                next = mp->b_next;
                mp->b_next = NULL;

                freemsg(mp);
                mp = next;
```



```
        }
}

mblk_t *
copymsgchain(mblk_t *mp)
{
        mblk_t  *nmp = NULL;
        mblk_t  **nmpp = &nmp;

        for (; mp != NULL; mp = mp->b_next) {
                if ((*nmpp = copymsg(mp)) == NULL) {
                        freemsgchain(nmp);
                        return (NULL);
                }

                nmpp = &((*nmpp)->b_next);
        }

        return (nmp);
}

/* NOTE: Do not add code after this point. */
#undef QLOCK

/*
 * Replacement for QLOCK macro for those that can't use it.
 */
kmutex_t *
QLOCK(queue_t *q)
{
        return (&(q)->q_lock);
}

/*
 * Dummy runqueues/queuerun functions functions for backwards compatibility.
 */
#undef runqueues
void
runqueues(void)
{
}

#undef queuerun
void
queuerun(void)
{
}

/*
 * Initialize the STR stack instance, which tracks autopush and persistent
 * links.
 */
/* ARGSUSED */
static void *
str_stack_init(netstackid_t stackid, netstack_t *ns)
{
        str_stack_t     *ss;
        int i;

        ss = (str_stack_t *)kmem_zalloc(sizeof (*ss), KM_SLEEP);
        ss->ss_netstack = ns;

        /*
         * set up autopush
         */
        sad_initspace(ss);
```

```
8820         /*
8821          * set up mux_node structures.
8822          */
8823         ss->ss_devcnt = devcnt; /* In case it should change before free */
8824         ss->ss_mux_nodes = kmem_zalloc((sizeof (struct mux_node) *
8825             ss->ss_devcnt), KM_SLEEP);
8826         for (i = 0; i < ss->ss_devcnt; i++)
8827                 ss->ss_mux_nodes[i].mn_imaj = i;
8828         return (ss);
8829 }

8831 /*
8832  * Note: run at zone shutdown and not destroy so that the PLINKs are
8833  * gone by the time other cleanup happens from the destroy callbacks.
8834  */
8835 static void
8836 str_stack_shutdown(netstackid_t stackid, void *arg)
8837 {
8838         str_stack_t *ss = (str_stack_t *)arg;
8839         int i;
8840         cred_t *cr;

8842         cr = zone_get_kcred(netstackid_to_zoneid(stackid));
8843         ASSERT(cr != NULL);

8845         /* Undo all the I_PLINKs for this zone */
8846         for (i = 0; i < ss->ss_devcnt; i++) {
8847                 struct mux_edge         *ep;
8848                 ldi_handle_t            lh;
8849                 ldi_ident_t             li;
8850                 int                     ret;
8851                 int                     rval;
8852                 dev_t                   rdev;

8854                 ep = ss->ss_mux_nodes[i].mn_outp;
8855                 if (ep == NULL)
8856                         continue;
8857                 ret = ldi_ident_from_major((major_t)i, &li);
8858                 if (ret != 0) {
8859                         continue;
8860                 }
8861                 rdev = ep->me_dev;
8862                 ret = ldi_open_by_dev(&rdev, OTYP_CHR, FREAD|FWRITE,
8863                     cr, &lh, li);
8864                 if (ret != 0) {
8865                         ldi_ident_release(li);
8866                         continue;
8867                 }

8869                 ret = ldi_ioctl(lh, I_PUNLINK, (intptr_t)MUXID_ALL, FKIOCTL,
8870                     cr, &rval);
8871                 if (ret) {
8872                         (void) ldi_close(lh, FREAD|FWRITE, cr);
8873                         ldi_ident_release(li);
8874                         continue;
8875                 }
8876                 (void) ldi_close(lh, FREAD|FWRITE, cr);

8878                 /* Close layered handles */
8879                 ldi_ident_release(li);
8880         }
8881         crfree(cr);

8883         sad_freespace(ss);
```



```
8885         kmem_free(ss->ss_mux_nodes, sizeof (struct mux_node) * ss->ss_devcnt);
8886         ss->ss_mux_nodes = NULL;
8887 }

8889 /*
8890  * Free the structure; str_stack_shutdown did the other cleanup work.
8891  */
8892 /* ARGSUSED */
8893 static void
8894 str_stack_fini(netstackid_t stackid, void *arg)
8895 {
8896         str_stack_t     *ss = (str_stack_t *)arg;

8898         kmem_free(ss, sizeof (*ss));
8899 }
```

```
**********************************************************
   22101 Mon Aug 17 21:08:08 2015
new/usr/src/uts/common/sys/Makefile
XXXX adding PID information to netstat output
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2013, Joyent, Inc. All rights reserved.
# Copyright 2013 Garrett D'Amore <garrett@damore.org>
#

include $(SRC)/uts/Makefile.uts

FILEMODE=644

#
#       Note that the following headers are present in the kernel but
#       neither installed or shipped as part of the product:
#               cpuid_drv.h:            Private interface for cpuid consumers
#               unix_bb_info.h:         Private interface to kcov
#

i386_HDRS=                      \
        agp/agpamd64gart_io.h   \
        agp/agpdefs.h           \
        agp/agpgart_impl.h      \
        agp/agpmaster_io.h      \
        agp/agptarget_io.h      \
        agpgart.h               \
        asy.h                   \
        fd_debug.h              \
        fdc.h                   \
        fdmedia.h               \
        mouse.h                 \
        ucode.h

sparc_HDRS=                     \
        mouse.h                 \
        scsi/targets/ssddef.h   \
        $(MDESCHDRS)

# Generated headers
GENHDRS=                        \
        priv_const.h            \
        priv_names.h            \
        usb/usbdevs.h
```



```
CHKHDRS=                        \
        acpi_drv.h              \
        acct.h                  \
        acctctl.h               \
        acl.h                   \
        acl_impl.h              \
        aggr.h                  \
        aggr_impl.h             \
        aio.h                   \
        aio_impl.h              \
        aio_req.h               \
        aiocb.h                 \
        ascii.h                 \
        asynch.h                \
        atomic.h                \
        attr.h                  \
        audio.h                 \
        audioio.h               \
        autoconf.h              \
        auxv.h                  \
        auxv_386.h              \
        auxv_SPARC.h            \
        avl.h                   \
        avl_impl.h              \
        bitmap.h                \
        bitset.h                \
        bl.h                    \
        blkdev.h                \
        bofi.h                  \
        bofi_impl.h             \
        bpp_io.h                \
        bootstat.h              \
        brand.h                 \
        buf.h                   \
        bufmod.h                \
        bustypes.h              \
        byteorder.h             \
        callb.h                 \
        callo.h                 \
        cap_util.h              \
        cpucaps.h               \
        cpucaps_impl.h          \
        ccompile.h              \
        cdio.h                  \
        cladm.h                 \
        class.h                 \
        clconf.h                \
        clock_impl.h            \
        cmlb.h                  \
        cmn_err.h               \
        compress.h              \
        condvar.h               \
        condvar_impl.h          \
        conf.h                  \
        consdev.h               \
        console.h               \
        consplat.h              \
        vt.h                    \
        vtdaemon.h              \
        kd.h                    \
        contract.h              \
        contract_impl.h         \
        copyops.h               \
        core.h                  \
        corectl.h               \
```

```
128         cpc_impl.h              \
129         cpc_pcbe.h              \
130         cpr.h                   \
131         cpupart.h               \
132         cpuvar.h                \
133         crc32.h                 \
134         cred.h                  \
135         cred_impl.h             \
136         pidnode.h               \
137 #endif /* ! codereview */
138         crtctl.h                \
139         cryptmod.h              \
140         csiioctl.h              \
141         ctf.h                   \
142         ctfs.h                  \
143         ctfs_impl.h             \
144         ctf_api.h               \
145         ctype.h                 \
146         cyclic.h                \
147         cyclic_impl.h           \
148         dacf.h                  \
149         dacf_impl.h             \
150         damap.h                 \
151         damap_impl.h            \
152         dc_ki.h                 \
153         ddi.h                   \
154         ddifm.h                 \
155         ddifm_impl.h            \
156         ddi_hp.h                \
157         ddi_hp_impl.h           \
158         ddi_intr.h              \
159         ddi_intr_impl.h         \
160         ddi_impldefs.h          \
161         ddi_implfuncs.h         \
162         ddi_obsolete.h          \
163         ddi_periodic.h          \
164         ddidevmap.h             \
165         ddidmareq.h             \
166         ddimapreq.h             \
167         ddipropdefs.h           \
168         dditypes.h              \
169         debug.h                 \
170         des.h                   \
171         devctl.h                \
172         devcache.h              \
173         devcache_impl.h         \
174         devfm.h                 \
175         devid_cache.h           \
176         devinfo_impl.h          \
177         devops.h                \
178         devpolicy.h             \
179         devpoll.h               \
180         dirent.h                \
181         disp.h                  \
182         dkbad.h                 \
183         dkio.h                  \
184         dklabel.h               \
185         dl.h                    \
186         dlpi.h                  \
187         dld.h                   \
188         dld_impl.h              \
189         dld_ioc.h               \
190         dls.h                   \
191         dls_mgmt.h              \
192         dls_impl.h              \
193         dma_i8237A.h            \
```



```
194         dnlc.h                  \
195         door.h                  \
196         door_data.h             \
197         door_impl.h             \
198         dtrace.h                \
199         dtrace_impl.h           \
200         dumpadm.h               \
201         dumphdr.h               \
202         ecppsys.h               \
203         ecppio.h                \
204         ecppreg.h               \
205         ecppvar.h               \
206         efi_partition.h         \
207         elf.h                   \
208         elf_386.h               \
209         elf_SPARC.h             \
210         elf_notes.h             \
211         elf_amd64.h             \
212         elftypes.h              \
213         emul64.h                \
214         emul64cmd.h             \
215         emul64var.h             \
216         epm.h                   \
217         errno.h                 \
218         errorq.h                \
219         errorq_impl.h           \
220         esunddi.h               \
221         ethernet.h              \
222         euc.h                   \
223         eucioctl.h              \
224         exacct.h                \
225         exacct_catalog.h        \
226         exacct_impl.h           \
227         exec.h                  \
228         exechdr.h               \
229         extdirent.h             \
230         fault.h                 \
231         fasttrap.h              \
232         fasttrap_impl.h         \
233         fbio.h                  \
234         fbuf.h                  \
235         fcntl.h                 \
236         fct.h                   \
237         fct_defines.h           \
238         fctio.h                 \
239         fdbuffer.h              \
240         fdio.h                  \
241         feature_tests.h         \
242         fem.h                   \
243         file.h                  \
244         filio.h                 \
245         flock.h                 \
246         flock_impl.h            \
247         fork.h                  \
248         fss.h                   \
249         fsspriocntl.h           \
250         fsid.h                  \
251         fssnap.h                \
252         fssnap_if.h             \
253         fstyp.h                 \
254         ftrace.h                \
255         fx.h                    \
256         fxpriocntl.h            \
257         gfs.h                   \
258         gld.h                   \
259         gldpriv.h               \
```

```
260         group.h                 \
261         hdio.h                  \
262         hook.h                  \
263         hook_event.h            \
264         hook_impl.h             \
265         hwconf.h                \
266         ia.h                    \
267         iapriocntl.h            \
268         ibpart.h                \
269         id32.h                  \
270         idmap.h                 \
271         ieeefp.h                \
272         id_space.h              \
273         instance.h              \
274         int_const.h             \
275         int_fmtio.h             \
276         int_limits.h            \
277         int_types.h             \
278         inttypes.h              \
279         ioccom.h                \
280         ioctl.h                 \
281         ipc.h                   \
282         ipc_impl.h              \
283         ipc_rctl.h              \
284         ipd.h                   \
285         ipmi.h                  \
286         isa_defs.h              \
287         iscsi_authclient.h      \
288         iscsi_authclientglue.h  \
289         iscsi_protocol.h        \
290         jioctl.h                \
291         kbd.h                   \
292         kbdreg.h                \
293         kbio.h                  \
294         kcpc.h                  \
295         kdi.h                   \
296         kdi_impl.h              \
297         kiconv.h                \
298         kiconv_big5_utf8.h      \
299         kiconv_cck_common.h     \
300         kiconv_cp950hkscs_utf8.h        \
301         kiconv_emea1.h          \
302         kiconv_emea2.h          \
303         kiconv_euckr_utf8.h     \
304         kiconv_euctw_utf8.h     \
305         kiconv_gb18030_utf8.h   \
306         kiconv_gb2312_utf8.h    \
307         kiconv_hkscs_utf8.h     \
308         kiconv_ja.h             \
309         kiconv_ja_jis_to_unicode.h      \
310         kiconv_ja_unicode_to_jis.h      \
311         kiconv_ko.h             \
312         kiconv_latin1.h         \
313         kiconv_sc.h             \
314         kiconv_tc.h             \
315         kiconv_uhc_utf8.h       \
316         kiconv_utf8_big5.h      \
317         kiconv_utf8_cp950hkscs.h        \
318         kiconv_utf8_euckr.h     \
319         kiconv_utf8_euctw.h     \
320         kiconv_utf8_gb18030.h   \
321         kiconv_utf8_gb2312.h    \
322         kiconv_utf8_hkscs.h     \
323         kiconv_utf8_uhc.h       \
324         kidmap.h                \
325         klpd.h                  \
```



```
326         klwp.h                  \
327         kmdb.h                  \
328         kmem.h                  \
329         kmem_impl.h             \
330         kobj.h                  \
331         kobj_impl.h             \
332         ksocket.h               \
333         kstat.h                 \
334         kstr.h                  \
335         ksyms.h                 \
336         ksynch.h                \
337         ldterm.h                \
338         lgrp.h                  \
339         lgrp_user.h             \
340         libc_kernel.h           \
341         link.h                  \
342         list.h                  \
343         list_impl.h             \
344         llc1.h                  \
345         loadavg.h               \
346         lock.h                  \
347         lockfs.h                \
348         lockstat.h              \
349         lofi.h                  \
350         log.h                   \
351         logindmux.h             \
352         logindmux_impl.h        \
353         lwp.h                   \
354         lwp_timer_impl.h        \
355         lwp_upimutex_impl.h     \
356         lpif.h                  \
357         mac.h                   \
358         mac_client.h            \
359         mac_client_impl.h       \
360         mac_ether.h             \
361         mac_flow.h              \
362         mac_flow_impl.h         \
363         mac_impl.h              \
364         mac_provider.h          \
365         mac_soft_ring.h         \
366         mac_stat.h              \
367         machelf.h               \
368         map.h                   \
369         md4.h                   \
370         md5.h                   \
371         md5_consts.h            \
372         mdi_impldefs.h          \
373         mem.h                   \
374         mem_config.h            \
375         memlist.h               \
376         mkdev.h                 \
377         mhd.h                   \
378         mii.h                   \
379         miiregs.h               \
380         mixer.h                 \
381         mman.h                  \
382         mmapobj.h               \
383         mntent.h                \
384         mntio.h                 \
385         mnttab.h                \
386         modctl.h                \
387         mode.h                  \
388         model.h                 \
389         modhash.h               \
390         modhash_impl.h          \
391         mount.h                 \
```

```
392         mouse.h                 \
393         msacct.h                \
394         msg.h                   \
395         msg_impl.h              \
396         msio.h                  \
397         msreg.h                 \
398         mtio.h                  \
399         multidata.h             \
400         multidata_impl.h        \
401         mutex.h                 \
402         nbmlock.h               \
403         ndifm.h                 \
404         ndi_impldefs.h          \
405         net80211.h              \
406         net80211_crypto.h       \
407         net80211_ht.h           \
408         net80211_proto.h        \
409         netconfig.h             \
410         neti.h                  \
411         netstack.h              \
412         nexusdefs.h             \
413         note.h                  \
414         nvpair.h                \
415         nvpair_impl.h           \
416         objfs.h                 \
417         objfs_impl.h            \
418         ontrap.h                \
419         open.h                  \
420         openpromio.h            \
421         panic.h                 \
422         param.h                 \
423         pathconf.h              \
424         pathname.h              \
425         pattr.h                 \
426         queue.h                 \
427         serializer.h            \
428         pbio.h                  \
429         pccard.h                \
430         pci.h                   \
431         pcie.h                  \
432         pci_impl.h              \
433         pci_tools.h             \
434         pcmcia.h                \
435         pctypes.h               \
436         pfmod.h                 \
437         pg.h                    \
438         pghw.h                  \
439         physmem.h               \
440         pkp_hash.h              \
441         pm.h                    \
442         policy.h                \
443         poll.h                  \
444         poll_impl.h             \
445         pool.h                  \
446         pool_impl.h             \
447         pool_pset.h             \
448         port.h                  \
449         port_impl.h             \
450         port_kernel.h           \
451         portif.h                \
452         ppmio.h                 \
453         pppt_ic_if.h            \
454         pppt_ioctl.h            \
455         priocntl.h              \
456         priv.h                  \
457         priv_impl.h             \
```



```
458         prnio.h                 \
459         proc.h                  \
460         processor.h             \
461         procfs.h                \
462         procset.h               \
463         project.h               \
464         protosw.h               \
465         prsystm.h               \
466         pset.h                  \
467         pshot.h                 \
468         ptem.h                  \
469         ptms.h                  \
470         ptyvar.h                \
471         raidioctl.h             \
472         ramdisk.h               \
473         random.h                \
474         rctl.h                  \
475         rctl_impl.h             \
476         rds.h                   \
477         reboot.h                \
478         refstr.h                \
479         refstr_impl.h           \
480         resource.h              \
481         rlioctl.h               \
482         rt.h                    \
483         rtpriocntl.h            \
484         rwlock.h                \
485         rwlock_impl.h           \
486         rwstlock.h              \
487         sad.h                   \
488         schedctl.h              \
489         sdt.h                   \
490         select.h                \
491         sem.h                   \
492         sem_impl.h              \
493         sema_impl.h             \
494         semaphore.h             \
495         sendfile.h              \
496         ser_sync.h              \
497         session.h               \
498         sha1.h                  \
499         sha1_consts.h           \
500         sha2.h                  \
501         sha2_consts.h           \
502         share.h                 \
503         shm.h                   \
504         shm_impl.h              \
505         sid.h                   \
506         siginfo.h               \
507         signal.h                \
508         sleepq.h                \
509         smbios.h                \
510         smbios_impl.h           \
511         sobject.h               \
512         socket.h                \
513         socket_impl.h           \
514         socket_proto.h          \
515         socketvar.h             \
516         sockfilter.h            \
517         sockio.h                \
518         soundcard.h             \
519         squeue.h                \
520         squeue_impl.h           \
521         srn.h                   \
522         sservice.h              \
523         stat.h                  \
```

```
524         statfs.h                \
525         statvfs.h               \
526         stdbool.h               \
527         stdint.h                \
528         stermio.h               \
529         stmf.h                  \
530         stmf_defines.h          \
531         stmf_ioctl.h            \
532         stmf_sbd_ioctl.h        \
533         stream.h                \
534         strft.h                 \
535         strlog.h                \
536         strmdep.h               \
537         stropts.h               \
538         strredir.h              \
539         strstat.h               \
540         strsubr.h               \
541         strsun.h                \
542         strtty.h                \
543         sunddi.h                \
544         sunldi.h                \
545         sunldi_impl.h           \
546         sunmdi.h                \
547         sunndi.h                \
548         sunos_dhcp_class.h      \
549         sunpm.h                 \
550         suntpi.h                \
551         suntty.h                \
552         swap.h                  \
553         synch.h                 \
554         sysdc.h                 \
555         sysdc_impl.h            \
556         syscall.h               \
557         sysconf.h               \
558         sysconfig.h             \
559         sysevent.h              \
560         sysevent_impl.h         \
561         sysinfo.h               \
562         syslog.h                \
563         sysmacros.h             \
564         sysmsg_impl.h           \
565         systeminfo.h            \
566         systm.h                 \
567         task.h                  \
568         taskq.h                 \
569         taskq_impl.h            \
570         t_kuser.h               \
571         t_lock.h                \
572         telioctl.h              \
573         termio.h                \
574         termios.h               \
575         termiox.h               \
576         thread.h                \
577         ticlts.h                \
578         ticots.h                \
579         ticotsord.h             \
580         tihdr.h                 \
581         time.h                  \
582         time_impl.h             \
583         time_std_impl.h         \
584         timeb.h                 \
585         timer.h                 \
586         times.h                 \
587         timex.h                 \
588         timod.h                 \
589         tirdwr.h                \
```



```
590         tiuser.h                \
591         tl.h                    \
592         tnf.h                   \
593         tnf_com.h               \
594         tnf_probe.h             \
595         tnf_writer.h            \
596         todio.h                 \
597         tpicommon.h             \
598         ts.h                    \
599         tspriocntl.h            \
600         ttcompat.h              \
601         ttold.h                 \
602         tty.h                   \
603         ttychars.h              \
604         ttydev.h                \
605         tuneable.h              \
606         turnstile.h             \
607         types.h                 \
608         types32.h               \
609         tzfile.h                \
610         u8_textprep.h           \
611         u8_textprep_data.h      \
612         uadmin.h                \
613         ucred.h                 \
614         uio.h                   \
615         ulimit.h                \
616         un.h                    \
617         unistd.h                \
618         user.h                  \
619         ustat.h                 \
620         utime.h                 \
621         utsname.h               \
622         utssys.h                \
623         uuid.h                  \
624         va_impl.h               \
625         va_list.h               \
626         var.h                   \
627         varargs.h               \
628         vfs.h                   \
629         vfs_opreg.h             \
630         vfstab.h                \
631         vgareg.h                \
632         videodev2.h             \
633         visual_io.h             \
634         vlan.h                  \
635         vm.h                    \
636         vm_usage.h              \
637         vmem.h                  \
638         vmem_impl.h             \
639         vmsystm.h               \
640         vnic.h                  \
641         vnic_impl.h             \
642         vnode.h                 \
643         vscan.h                 \
644         vtoc.h                  \
645         vtrace.h                \
646         vuid_event.h            \
647         vuid_wheel.h            \
648         vuid_queue.h            \
649         vuid_state.h            \
650         vuid_store.h            \
651         wait.h                  \
652         waitq.h                 \
653         wanboot_impl.h          \
654         watchpoint.h            \
655         winlockio.h             \
```

```
656         zcons.h                 \
657         zone.h                  \
658         xti_inet.h              \
659         xti_osi.h               \
660         xti_xtiopt.h            \
661         zmod.h

663 HDRS=                           \
664         $(GENHDRS)              \
665         $(CHKHDRS)

667 AUDIOHDRS=                      \
668         ac97.h                  \
669         audio_common.h          \
670         audio_driver.h          \
671         audio_oss.h             \
672         g711.h

674 AVHDRS=                        \
675         iec61883.h

677 BSCHDRS=                       \
678         bscbus.h                \
679         bscv_impl.h             \
680         lom_ebuscodes.h         \
681         lom_io.h                \
682         lom_priv.h              \
683         lombus.h

685 MDESCHDRS=                     \
686         mdesc.h                 \
687         mdesc_impl.h

689 CPUDRVHDRS=                    \
690         cpudrv.h

692 CRYPTOHDRS=                    \
693         elfsign.h               \
694         ioctl.h                 \
695         ioctladmin.h            \
696         common.h                \
697         impl.h                  \
698         spi.h                   \
699         api.h                   \
700         ops_impl.h              \
701         sched_impl.h

703 DCAMHDRS=                      \
704         dcam1394_io.h

706 IBHDRS=                  \
707         ib_types.h      \
708         ib_pkt_hdrs.h

710 IBTLHDRS=                \
711         ibtl_types.h    \
712         ibtl_status.h   \
713         ibti.h          \
714         ibti_cm.h       \
715         ibci.h          \
716         ibti_common.h   \
717         ibvti.h         \
718         ibtl_ci_types.h

720 IBTLIMPLHDRS=            \
721         ibtl_util.h
```



```
723 IBNEXHDRS=              \
724         ibnex_devctl.h

726 IBMFHDRS=               \
727         ibmf.h          \
728         ibmf_msg.h      \
729         ibmf_saa.h      \
730         ibmf_utils.h

732 IBMGTHDRS=              \
733         ib_dm_attr.h    \
734         ib_mad.h        \
735         sm_attr.h       \
736         sa_recs.h

738 IBDHDRS=                \
739         ibd.h

741 OFHDRS=                 \
742         ofa_solaris.h   \
743         ofed_kernel.h

745 RDMAHDRS=               \
746         ib_addr.h       \
747         ib_user_mad.h   \
748         ib_user_sa.h    \
749         ib_user_verbs.h \
750         ib_verbs.h      \
751         rdma_cm.h       \
752         rdma_user_cm.h

754 SOL_UVERBSHDRS=         \
755         sol_uverbs.h    \
756         sol_uverbs2ucma.h       \
757         sol_uverbs_comp.h       \
758         sol_uverbs_hca.h        \
759         sol_uverbs_qp.h         \
760         sol_uverbs_event.h

762 SOL_UMADHDRS=           \
763         sol_umad.h

765 SOL_UCMAHDRS=           \
766         sol_ucma.h      \
767         sol_rdma_user_cm.h

769 SOL_OFSHDRS=            \
770         sol_cma.h       \
771         sol_ib_cma.h    \
772         sol_ofs_common.h        \
773         sol_kverb_impl.h

775 TAVORHDRS=              \
776         tavor_ioctl.h

778 HERMONHDRS=             \
779         hermon_ioctl.h

781 MLNXHDRS=              \
782         mlnx_umap.h

784 IDMHDRS=                \
785         idm.h           \
786         idm_impl.h      \
787         idm_so.h        \
```

```
788         idm_text.h      \
789         idm_transport.h \
790         idm_conn_sm.h

792 ISCSITHDRS=              \
793         radius_packet.h         \
794         radius_protocol.h       \
795         chap.h                  \
796         isns_protocol.h         \
797         iscsi_if.h              \
798         iscsit_common.h

800 ISOHDRS=                        \
801         signal_iso.h

803 DERIVED_LVMHDRS=         \
804         md_mdiox.h              \
805         md_basic.h              \
806         mdmed.h                 \
807         md_mhdx.h               \
808         mdmn_commd.h

810 LVMHDRS=                        \
811         md_convert.h            \
812         md_crc.h                \
813         md_hotspares.h          \
814         md_mddb.h               \
815         md_mirror.h             \
816         md_mirror_shared.h      \
817         md_names.h              \
818         md_notify.h             \
819         md_raid.h               \
820         md_rename.h             \
821         md_sp.h                 \
822         md_stripe.h             \
823         md_trans.h              \
824         mdio.h                  \
825         mdvar.h

827 ALL_LVMHDRS=                    \
828         $(LVMHDRS)              \
829         $(DERIVED_LVMHDRS)

831 FMHDRS=                         \
832         protocol.h              \
833         util.h

835 FMFSHDRS=                       \
836         zfs.h

838 FMIOHDRS=                       \
839         ddi.h                   \
840         disk.h                  \
841         pci.h                   \
842         scsi.h                  \
843         sun4upci.h              \
844         opl_mc_fm.h

846 FSHDRS=                         \
847         autofs.h                \
848         cachefs_dir.h           \
849         cachefs_dlog.h          \
850         cachefs_filegrp.h       \
851         cachefs_fs.h            \
852         cachefs_fscache.h       \
853         cachefs_ioctl.h         \
```



```
854         cachefs_log.h           \
855         decomp.h                \
856         dv_node.h               \
857         sdev_impl.h             \
858         fifonode.h              \
859         hsfs_isospec.h          \
860         hsfs_node.h             \
861         hsfs_rrip.h             \
862         hsfs_spec.h             \
863         hsfs_susp.h             \
864         lofs_info.h             \
865         lofs_node.h             \
866         mntdata.h               \
867         namenode.h              \
868         pc_dir.h                \
869         pc_fs.h                 \
870         pc_label.h              \
871         pc_node.h               \
872         pxfs_ki.h               \
873         snode.h                 \
874         swapnode.h              \
875         tmp.h                   \
876         tmpnode.h               \
877         udf_inode.h             \
878         udf_volume.h            \
879         ufs_acl.h               \
880         ufs_bio.h               \
881         ufs_filio.h             \
882         ufs_fs.h                \
883         ufs_fsdir.h             \
884         ufs_inode.h             \
885         ufs_lockfs.h            \
886         ufs_log.h               \
887         ufs_mount.h             \
888         ufs_panic.h             \
889         ufs_prot.h              \
890         ufs_quota.h             \
891         ufs_snap.h              \
892         ufs_trans.h             \
893         zfs.h                   \
894         zut.h

896 SCSIHDRS=               \
897         scsi.h          \
898         scsi_address.h  \
899         scsi_ctl.h      \
900         scsi_fm.h       \
901         scsi_params.h   \
902         scsi_pkt.h      \
903         scsi_resource.h \
904         scsi_types.h    \
905         scsi_watch.h

907 SCSICONFHDRS=           \
908         autoconf.h      \
909         device.h

911 SCSIGENHDRS=            \
912         commands.h      \
913         dad_mode.h      \
914         inquiry.h       \
915         message.h       \
916         mode.h          \
917         persist.h       \
918         sense.h         \
919         sff_frames.h    \
```

```
920         smp_frames.h    \
921         status.h

923 SCSIIMPLHDRS=           \
924         commands.h      \
925         inquiry.h       \
926         mode.h          \
927         scsi_reset_notify.h \
928         scsi_sas.h      \
929         sense.h         \
930         services.h      \
931         smp_transport.h \
932         spc3_types.h    \
933         status.h        \
934         transport.h     \
935         types.h         \
936         uscsi.h         \
937         usmp.h

939 SCSITARGETSHDRS=        \
940         ses.h           \
941         sesio.h         \
942         sgendef.h       \
943         stdef.h         \
944         sddef.h         \
945         smp.h

947 SCSIADHDRS=

949 SCSICADHDRS=

951 SCSIISCSIHDRS=          \
952         iscsi_door.h    \
953         iscsi_if.h

955 SCSIVHCIHDRS=           \
956         scsi_vhci.h     \
957         mpapi_impl.h    \
958         mpapi_scsi_vhci.h

960 SDCARDHDRS=             \
961         sda.h           \
962         sda_impl.h      \
963         sda_ioctl.h

965 FC4HDRS=                       \
966         fc_transport.h  \
967         linkapp.h       \
968         fc.h            \
969         fcp.h           \
970         fcal_transport.h \
971         fcal.h          \
972         fcal_linkapp.h  \
973         fcio.h

975 FCHDRS=                 \
976         fc.h            \
977         fcio.h          \
978         fc_types.h      \
979         fc_appif.h

981 FCIMPLHDRS=             \
982         fc_error.h      \
983         fcph.h

985 FCULPHDRS=              \
```



```
986         fcp_util.h      \
987         fcsm.h

989 SATAGENHDRS=            \
990         sata_hba.h      \
991         sata_defs.h     \
992         sata_cfgadm.h

994 SYSEVENTHDRS=           \
995         ap_driver.h     \
996         dev.h           \
997         domain.h        \
998         dr.h            \
999         env.h           \
1000        eventdefs.h     \
1001        ipmp.h          \
1002        pwrctl.h        \
1003        svm.h           \
1004        vrrp.h

1006 CONTRACTHDRS=          \
1007        process.h       \
1008        process_impl.h  \
1009        device.h        \
1010        device_impl.h

1012 USBHDRS=        \
1013        usba.h  \
1014        usbai.h

1016 USBAUDHDRS=           \
1017        usb_audio.h

1019 USBHUBDHDRS=          \
1020        hub.h           \
1021        hubd_impl.h

1023 USBHIDHDRS=           \
1024        hid.h

1026 USBMSHDRS=            \
1027        usb_bulkonly.h  \
1028        usb_cbi.h

1030 USBPRNHDRS=          \
1031        usb_printer.h

1033 USBCDCHDRS=           \
1034        usb_cdc.h

1036 USBVIDHDRS=          \
1037        usbvc.h

1039 USBWCMHDRS=           \
1040        usbwcm.h

1042 UGENHDRS=          \
1043        usb_ugen.h

1045 HOTPLUGHDRS=    \
1046        hpcsvc.h        \
1047        hpctrl.h

1049 HOTPLUGPCIHDRS= \
1050        pcicfg.h        \
1051        pcihp.h
```

```
1053 RSMHDRS= \
1054         rsm.h   \
1055         rsm_common.h    \
1056         rsmapi_common.h \
1057         rsmpi.h \
1058         rsmpi_driver.h  \
1059         rsmka_path_int.h

1061 TSOLHDRS=               \
1062         label.h         \
1063         label_macro.h   \
1064         priv.h          \
1065         tndb.h          \
1066         tsyscall.h

1068 I1394HDRS=              \
1069         cmd1394.h       \
1070         id1394.h        \
1071         ieee1212.h      \
1072         ieee1394.h      \
1073         ixl1394.h       \
1074         s1394_impl.h    \
1075         t1394.h

1077 # "cmdk" headers used on sparc
1078 SDKTPHDRS=              \
1079         dadkio.h        \
1080         fdisk.h

1082 # "cmdk" headers used on i386
1083 DKTPHDRS=               \
1084         altsctr.h       \
1085         bbh.h           \
1086         cm.h            \
1087         cmdev.h         \
1088         cmdk.h          \
1089         cmpkt.h         \
1090         controller.h    \
1091         dadev.h         \
1092         dadk.h          \
1093         dadkio.h        \
1094         fctypes.h       \
1095         fdisk.h         \
1096         flowctrl.h      \
1097         gda.h           \
1098         quetypes.h      \
1099         queue.h         \
1100         tgcom.h         \
1101         tgdk.h

1103 # "pc" header files used on i386
1104 PCHDRS=                 \
1105         avintr.h        \
1106         dma_engine.h    \
1107         i8272A.h        \
1108         pcic_reg.h      \
1109         pcic_var.h      \
1110         pic.h           \
1111         pit.h           \
1112         rtc.h

1114 NXGEHDRS=                       \
1115         nxge.h                  \
1116         nxge_common.h           \
1117         nxge_common_impl.h      \
```



```
1118         nxge_defs.h             \
1119         nxge_hw.h               \
1120         nxge_impl.h             \
1121         nxge_ipp.h              \
1122         nxge_ipp_hw.h           \
1123         nxge_mac.h              \
1124         nxge_mac_hw.h           \
1125         nxge_fflp.h             \
1126         nxge_fflp_hw.h          \
1127         nxge_mii.h              \
1128         nxge_rxdma.h            \
1129         nxge_rxdma_hw.h         \
1130         nxge_txc.h              \
1131         nxge_txc_hw.h           \
1132         nxge_txdma.h            \
1133         nxge_txdma_hw.h         \
1134         nxge_virtual.h          \
1135         nxge_espc.h

1137 include Makefile.syshdrs

1139 dcam/%.check:   dcam/%.h
1140         $(DOT_H_CHECK)

1142 CHECKHDRS=                                                      \
1143         $($(MACH)_HDRS:%.h=%.check)                             \
1144         $(AUDIOHDRS:%.h=audio/%.check)                          \
1145         $(AVHDRS:%.h=av/%.check)                                \
1146         $(BSCHDRS:%.h=%.check)                                  \
1147         $(CHKHDRS:%.h=%.check)                                  \
1148         $(CPUDRVHDRS:%.h=%.check)                               \
1149         $(CRYPTOHDRS:%.h=crypto/%.check)                        \
1150         $(DCAMHDRS:%.h=dcam/%.check)                            \
1151         $(FC4HDRS:%.h=fc4/%.check)                              \
1152         $(FCHDRS:%.h=fibre-channel/%.check)                     \
1153         $(FCIMPLHDRS:%.h=fibre-channel/impl/%.check)            \
1154         $(FCULPHDRS:%.h=fibre-channel/ulp/%.check)              \
1155         $(IBHDRS:%.h=ib/%.check)                                \
1156         $(IBDHDRS:%.h=ib/clients/ibd/%.check)                   \
1157         $(IBTLHDRS:%.h=ib/ibtl/%.check)                         \
1158         $(IBTLIMPLHDRS:%.h=ib/ibtl/impl/%.check)                \
1159         $(IBNEXHDRS:%.h=ib/ibnex/%.check)               \
1160         $(IBMGTHDRS:%.h=ib/mgt/%.check)                         \
1161         $(IBMFHDRS:%.h=ib/mgt/ibmf/%.check)                     \
1162         $(OFHDRS:%.h=ib/clients/of/%.check)                             \
1163         $(RDMAHDRS:%.h=ib/clients/of/rdma/%.check)                      \
1164         $(SOL_UVERBSHDRS:%.h=ib/clients/of/sol_uverbs/%.check)  \
1165         $(SOL_UCMAHDRS:%.h=ib/clients/of/sol_ucma/%.check)      \
1166         $(SOL_OFSHDRS:%.h=ib/clients/of/sol_ofs/%.check)        \
1167         $(TAVORHDRS:%.h=ib/adapters/tavor/%.check)              \
1168         $(HERMONHDRS:%.h=ib/adapters/hermon/%.check)            \
1169         $(MLNXHDRS:%.h=ib/adapters/%.check)                     \
1170         $(IDMHDRS:%.h=idm/%.check)                              \
1171         $(ISCSIHDRS:%.h=iscsi/%.check)                          \
1172         $(ISCSITHDRS:%.h=iscsit/%.check)                                \
1173         $(ISOHDRS:%.h=iso/%.check)                              \
1174         $(FMHDRS:%.h=fm/%.check)                                \
1175         $(FMFSHDRS:%.h=fm/fs/%.check)                           \
1176         $(FMIOHDRS:%.h=fm/io/%.check)                           \
1177         $(FSHDRS:%.h=fs/%.check)                                \
1178         $(LVMHDRS:%.h=lvm/%.check)                              \
1179         $(SCSIHDRS:%.h=scsi/%.check)                            \
1180         $(SCSIADHDRS:%.h=scsi/adapters/%.check)                 \
1181         $(SCSICONFHDRS:%.h=scsi/conf/%.check)                   \
1182         $(SCSIIMPLHDRS:%.h=scsi/impl/%.check)                   \
1183         $(SCSIISCSIHDRS:%.h=scsi/adapters/%.check)              \
```

```
1184         $(SCSIGENHDRS:%.h=scsi/generic/%.check)         \
1185         $(SCSITARGETSHDRS:%.h=scsi/targets/%.check)     \
1186         $(SCSIVHCIHDRS:%.h=scsi/adapters/%.check)       \
1187         $(SATAGENHDRS:%.h=sata/%.check)                 \
1188         $(SDCARDHDRS:%.h=sdcard/%.check)                \
1189         $(SYSEVENTHDRS:%.h=sysevent/%.check)            \
1190         $(CONTRACTHDRS:%.h=contract/%.check)            \
1191         $(USBAUDHDRS:%.h=usb/clients/audio/%.check)   \
1192         $(USBHUBDHDRS:%.h=usb/hubd/%.check)             \
1193         $(USBHIDHDRS:%.h=usb/clients/hid/%.check)       \
1194         $(USBMSHDRS:%.h=usb/clients/mass_storage/%.check)       \
1195         $(USBPRNHDRS:%.h=usb/clients/printer/%.check)   \
1196         $(USBCDCHDRS:%.h=usb/clients/usbcdc/%.check)    \
1197         $(USBVIDHDRS:%.h=usb/clients/video/usbvc/%.check)   \
1198         $(USBWCMHDRS:%.h=usb/clients/usbinput/usbwcm/%.check)   \
1199         $(UGENHDRS:%.h=usb/clients/ugen/%.check)   \
1200         $(USBHDRS:%.h=usb/%.check)                      \
1201         $(I1394HDRS:%.h=1394/%.check)                   \
1202         $(RSMHDRS:%.h=rsm/%.check)                      \
1203         $(TSOLHDRS:%.h=tsol/%.check)                    \
1204         $(NXGEHDRS:%.h=nxge/%.check)


1207 .KEEP_STATE:

1209 .PARALLEL:                      \
1210         $(CHECKHDRS)            \
1211         $(ROOTHDRS)             \
1212         $(ROOTAUDHDRS)          \
1213         $(ROOTAVHDRS)           \
1214         $(ROOTCRYPTOHDRS)       \
1215         $(ROOTDCAMHDRS)         \
1216         $(ROOTISOHDRS)          \
1217         $(ROOTIDMHDRS)          \
1218         $(ROOTISCSIHDRS)        \
1219         $(ROOTISCSITHDRS)       \
1220         $(ROOTFC4HDRS)          \
1221         $(ROOTFCHDRS)           \
1222         $(ROOTFCIMPLHDRS)       \
1223         $(ROOTFCULPHDRS)        \
1224         $(ROOTFMHDRS)           \
1225         $(ROOTFMIOHDRS)         \
1226         $(ROOTFMFSHDRS)         \
1227         $(ROOTFSHDRS)           \
1228         $(ROOTIBDHDRS)          \
1229         $(ROOTIBHDRS)           \
1230         $(ROOTIBTLHDRS)         \
1231         $(ROOTIBTLIMPLHDRS)     \
1232         $(ROOTIBNEXHDRS)        \
1233         $(ROOTIBMGTHDRS)        \
1234         $(ROOTIBMFHDRS)         \
1235         $(ROOTOFHDRS)           \
1236         $(ROOTRDMAHDRS)         \
1237         $(ROOTSOL_OFSHDRS)      \
1238         $(ROOTSOL_UMADHDRS)     \
1239         $(ROOTSOL_UVERBSHDRS)   \
1240         $(ROOTSOL_UCMAHDRS)     \
1241         $(ROOTTAVORHDRS)        \
1242         $(ROOTHERMONHDRS)       \
1243         $(ROOTMLNXHDRS)         \
1244         $(ROOTLVMHDRS)          \
1245         $(ROOTSCSIHDRS)         \
1246         $(ROOTSCSIADHDRS)       \
1247         $(ROOTSCSICONFHDRS)     \
1248         $(ROOTSCSIISCSIHDRS)    \
1249         $(ROOTSCSIGENHDRS)      \
```



```
1250         $(ROOTSCSIIMPLHDRS)     \
1251         $(ROOTSCSIVHCIHDRS)     \
1252         $(ROOTSDCARDHDRS)       \
1253         $(ROOTSYSEVENTHDRS)     \
1254         $(ROOTCONTRACTHDRS)     \
1255         $(ROOTUSBHDRS)          \
1256         $(ROOTUWBHDRS)          \
1257         $(ROOTUWBAHDRS)         \
1258         $(ROOTUSBAUDHDRS)       \
1259         $(ROOTUSBHUBDHDRS)      \
1260         $(ROOTUSBHIDHDRS)       \
1261         $(ROOTUSBHRCHDRS)       \
1262         $(ROOTUSBMSHDRS)        \
1263         $(ROOTUSBPRNHDRS)       \
1264         $(ROOTUSBCDCHDRS)       \
1265         $(ROOTUSBVIDHDRS)       \
1266         $(ROOTUSBWCMHDRS)       \
1267         $(ROOTUGENHDRS)         \
1268         $(ROOT1394HDRS)         \
1269         $(ROOTHOTPLUGHDRS)      \
1270         $(ROOTHOTPLUGPCIHDRS)   \
1271         $(ROOTRSMHDRS)          \
1272         $(ROOTTSOLHDRS)         \
1273         $($(MACH)_ROOTHDRS)


1276 install_h:                      \
1277         $(ROOTDIRS)             \
1278         LVMDERIVED_H            \
1279         .WAIT                   \
1280         $(ROOTHDRS)             \
1281         $(ROOTAUDHDRS)          \
1282         $(ROOTAVHDRS)           \
1283         $(ROOTCRYPTOHDRS)       \
1284         $(ROOTDCAMHDRS)         \
1285         $(ROOTISOHDRS)          \
1286         $(ROOTIDMHDRS)          \
1287         $(ROOTISCSIHDRS)        \
1288         $(ROOTISCSITHDRS)       \
1289         $(ROOTFC4HDRS)          \
1290         $(ROOTFCHDRS)           \
1291         $(ROOTFCIMPLHDRS)       \
1292         $(ROOTFCULPHDRS)        \
1293         $(ROOTFMHDRS)           \
1294         $(ROOTFMFSHDRS)         \
1295         $(ROOTFMIOHDRS)         \
1296         $(ROOTFSHDRS)           \
1297         $(ROOTIBDHDRS)          \
1298         $(ROOTIBHDRS)           \
1299         $(ROOTIBTLHDRS)         \
1300         $(ROOTIBTLIMPLHDRS)     \
1301         $(ROOTIBNEXHDRS)        \
1302         $(ROOTIBMGTHDRS)        \
1303         $(ROOTIBMFHDRS)         \
1304         $(ROOTOFHDRS)           \
1305         $(ROOTRDMAHDRS)         \
1306         $(ROOTSOL_OFSHDRS)      \
1307         $(ROOTSOL_UMADHDRS)     \
1308         $(ROOTSOL_UVERBSHDRS)   \
1309         $(ROOTSOL_UCMAHDRS)     \
1310         $(ROOTTAVORHDRS)        \
1311         $(ROOTHERMONHDRS)       \
1312         $(ROOTMLNXHDRS)         \
1313         $(ROOTLVMHDRS)          \
1314         $(ROOTSCSIHDRS)         \
1315         $(ROOTSCSIADHDRS)       \
```

```
1316         $(ROOTSCSIISCSIHDRS)    \
1317         $(ROOTSCSICONFHDRS)     \
1318         $(ROOTSCSIGENHDRS)      \
1319         $(ROOTSCSIIMPLHDRS)     \
1320         $(ROOTSCSIVHCIHDRS)     \
1321         $(ROOTSDCARDHDRS)       \
1322         $(ROOTSYSEVENTHDRS)     \
1323         $(ROOTCONTRACTHDRS)     \
1324         $(ROOTUWBHDRS)          \
1325         $(ROOTUWBAHDRS)         \
1326         $(ROOTUSBHDRS)          \
1327         $(ROOTUSBAUDHDRS)       \
1328         $(ROOTUSBHUBDHDRS)      \
1329         $(ROOTUSBHIDHDRS)       \
1330         $(ROOTUSBHRCHDRS)       \
1331         $(ROOTUSBMSHDRS)        \
1332         $(ROOTUSBPRNHDRS)       \
1333         $(ROOTUSBCDCHDRS)       \
1334         $(ROOTUSBVIDHDRS)       \
1335         $(ROOTUSBWCMHDRS)       \
1336         $(ROOTUGENHDRS)         \
1337         $(ROOT1394HDRS)         \
1338         $(ROOTHOTPLUGHDRS)      \
1339         $(ROOTHOTPLUGPCIHDRS)   \
1340         $(ROOTRSMHDRS)          \
1341         $(ROOTTSOLHDRS)         \
1342         $($(MACH)_ROOTHDRS)

1344 all_h: $(GENHDRS)

1346 priv_const.h: $(PRIVS_AWK) $(PRIVS_DEF)
1347         $(NAWK) -f $(PRIVS_AWK) < $(PRIVS_DEF) -v privhfile=$@

1349 priv_names.h: $(PRIVS_AWK) $(PRIVS_DEF)
1350         $(NAWK) -f $(PRIVS_AWK) < $(PRIVS_DEF) -v pubhfile=$@

1352 usb/usbdevs.h: $(USBDEVS_AWK) $(USBDEVS_DATA)
1353         $(NAWK) -f $(USBDEVS_AWK) $(USBDEVS_DATA) -H > $@

1355 LVMDERIVED_H:
1356         cd $(SRC)/uts/common/sys/lvm; pwd; $(MAKE) all_h

1358 clean:
1359         $(RM) $(GENHDRS)

1361 clobber: clean
1362         cd $(SRC)/uts/common/sys/lvm; pwd; $(MAKE) clobber

1364 check:  $(CHECKHDRS)

1366 FRC:
```

```
**********************************************************
   11457 Mon Aug 17 21:08:08 2015
new/usr/src/uts/common/sys/fcntl.h
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/* Copyright (c) 2013, OmniTI Computer Consulting, Inc. All rights reserved. */

#ifndef _SYS_FCNTL_H
#define _SYS_FCNTL_H

#include <sys/feature_tests.h>

#include <sys/types.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Flag values accessible to open(2) and fcntl(2)
 * The first five can only be set (exclusively) by open(2).
 */
#define O_RDONLY        0
#define O_WRONLY        1
#define O_RDWR          2
#define O_SEARCH        0x200000
#define O_EXEC          0x400000
#if defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE)
```



```
#define O_NDELAY        0x04    /* non-blocking I/O */
#endif /* defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE) */
#define O_APPEND        0x08    /* append (writes guaranteed at the end) */
#if defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE) || \
        (_POSIX_C_SOURCE > 2) || defined(_XOPEN_SOURCE)
#define O_SYNC          0x10    /* synchronized file update option */
#define O_DSYNC         0x40    /* synchronized data update option */
#define O_RSYNC         0x8000  /* synchronized file update option */
                                /* defines read/write file integrity */
#endif /* defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE) ... */
#define O_NONBLOCK      0x80    /* non-blocking I/O (POSIX) */
#ifdef  _LARGEFILE_SOURCE
#define O_LARGEFILE     0x2000
#endif

/*
 * Flag values accessible only to open(2).
 */
#define O_CREAT         0x100   /* open with file create (uses third arg) */
#define O_TRUNC         0x200   /* open with truncation */
#define O_EXCL          0x400   /* exclusive open */
#define O_NOCTTY        0x800   /* don't allocate controlling tty (POSIX) */
#define O_XATTR         0x4000  /* extended attribute */
#define O_NOFOLLOW      0x20000 /* don't follow symlinks */
#define O_NOLINKS       0x40000 /* don't allow multiple hard links */
#define O_CLOEXEC       0x800000        /* set the close-on-exec flag */

/*
 * fcntl(2) requests
 *
 * N.B.: values are not necessarily assigned sequentially below.
 */
#define F_DUPFD         0       /* Duplicate fildes */
#define F_GETFD         1       /* Get fildes flags */
#define F_SETFD         2       /* Set fildes flags */
#define F_GETFL         3       /* Get file flags */
#define F_GETXFL        45      /* Get file flags including open-only flags */
#define F_SETFL         4       /* Set file flags */

/*
 * Applications that read /dev/mem must be built like the kernel.  A
 * new symbol "_KMEMUSER" is defined for this purpose.
 */
#if defined(_KERNEL) || defined(_KMEMUSER)
#define F_O_GETLK       5       /* SVR3 Get file lock (need for rfs, across */
                                /* the wire compatibility */
/* clustering: lock id contains both per-node sysid and node id */
#define SYSIDMASK               0x0000ffff
#define GETSYSID(id)            (id & SYSIDMASK)
#define NODEIDMASK              0xffff0000
#define BITS_IN_SYSID           16
#define GETNLMID(sysid)         ((int)(((uint_t)(sysid) & NODEIDMASK) >> \
                                    BITS_IN_SYSID))

/* Clustering: Macro used for PXFS locks */
#define GETPXFSID(sysid)        ((int)(((uint_t)(sysid) & NODEIDMASK) >> \
                                    BITS_IN_SYSID))
#endif  /* defined(_KERNEL) */

#define F_CHKFL         8       /* Unused */
#define F_DUP2FD        9       /* Duplicate fildes at third arg */
#define F_DUP2FD_CLOEXEC        36      /* Like F_DUP2FD with O_CLOEXEC set */
                                /* EINVAL is fildes matches arg1 */
#define F_DUPFD_CLOEXEC 37      /* Like F_DUPFD with O_CLOEXEC set */

#define F_ISSTREAM      13      /* Is the file desc. a stream ? */
```

```
128 #define F_PRIV          15      /* Turn on private access to file */
129 #define F_NPRIV         16      /* Turn off private access to file */
130 #define F_QUOTACTL      17      /* UFS quota call */
131 #define F_BLOCKS        18      /* Get number of BLKSIZE blocks allocated */
132 #define F_BLKSIZE       19      /* Get optimal I/O block size */
133 /*
134  * Numbers 20-22 have been removed and should not be reused.
135  */
136 #define F_GETOWN        23      /* Get owner (socket emulation) */
137 #define F_SETOWN        24      /* Set owner (socket emulation) */
138 #define F_REVOKE        25      /* Object reuse revoke access to file desc. */

140 #define F_HASREMOTELOCKS 26     /* Does vp have NFS locks; private to lock */
141                                 /* manager */

143 /*
144  * Commands that refer to flock structures.  The argument types differ between
145  * the large and small file environments; therefore, the #defined values must
146  * as well.
147  * The NBMAND forms are private and should not be used.
148  */

150 #if defined(_LP64) || _FILE_OFFSET_BITS == 32
151 /* "Native" application compilation environment */
152 #define F_SETLK         6       /* Set file lock */
153 #define F_SETLKW        7       /* Set file lock and wait */
154 #define F_ALLOCSP       10      /* Allocate file space */
155 #define F_FREESP        11      /* Free file space */
156 #define F_GETLK         14      /* Get file lock */
157 #define F_SETLK_NBMAND  42      /* private */
158 #else
159 /* ILP32 large file application compilation environment version */
160 #define F_SETLK         34      /* Set file lock */
161 #define F_SETLKW        35      /* Set file lock and wait */
162 #define F_ALLOCSP       28      /* Alllocate file space */
163 #define F_FREESP        27      /* Free file space */
164 #define F_GETLK         33      /* Get file lock */
165 #define F_SETLK_NBMAND  44      /* private */
166 #endif /* _LP64 || _FILE_OFFSET_BITS == 32 */

168 #define F_ASSOCI_PID    292929
169 #define F_DASSOC_PID    303030

171 #endif /* ! codereview */
172 #if     defined(_LARGEFILE64_SOURCE)

174 #if !defined(_LP64) || defined(_KERNEL)
175 /*
176  * transitional large file interface version
177  * These are only valid in a 32 bit application compiled with large files
178  * option, for source compatibility, the 64-bit versions are mapped back
179  * to the native versions.
180  */
181 #define F_SETLK64       34      /* Set file lock */
182 #define F_SETLKW64      35      /* Set file lock and wait */
183 #define F_ALLOCSP64     28      /* Allocate file space */
184 #define F_FREESP64      27      /* Free file space */
185 #define F_GETLK64       33      /* Get file lock */
186 #define F_SETLK64_NBMAND        44      /* private */
187 #else
188 #define F_SETLK64       6       /* Set file lock */
189 #define F_SETLKW64      7       /* Set file lock and wait */
190 #define F_ALLOCSP64     10      /* Allocate file space */
191 #define F_FREESP64      11      /* Free file space */
192 #define F_GETLK64       14      /* Get file lock */
193 #define F_SETLK64_NBMAND        42      /* private */
```



```
194 #endif /* !_LP64 || _KERNEL */

196 #endif /* _LARGEFILE64_SOURCE */

198 #define F_SHARE         40      /* Set a file share reservation */
199 #define F_UNSHARE       41      /* Remove a file share reservation */
200 #define F_SHARE_NBMAND  43      /* private */

202 #define F_BADFD         46      /* Create Poison FD */

204 /*
205  * File segment locking set data type - information passed to system by user.
206  */

208 /* regular version, for both small and large file compilation environment */
209 typedef struct flock {
210         short   l_type;
211         short   l_whence;
212         off_t   l_start;
213         off_t   l_len;          /* len == 0 means until end of file */
214         int     l_sysid;
215         pid_t   l_pid;
216         long    l_pad[4];               /* reserve area */
217 } flock_t;

219 #if defined(_SYSCALL32)

221 /* Kernel's view of ILP32 flock structure */

223 typedef struct flock32 {
224         int16_t l_type;
225         int16_t l_whence;
226         off32_t l_start;
227         off32_t l_len;          /* len == 0 means until end of file */
228         int32_t l_sysid;
229         pid32_t l_pid;
230         int32_t l_pad[4];               /* reserve area */
231 } flock32_t;

233 #endif /* _SYSCALL32 */

235 /* transitional large file interface version */

237 #if     defined(_LARGEFILE64_SOURCE)

239 typedef struct flock64 {
240         short   l_type;
241         short   l_whence;
242         off64_t l_start;
243         off64_t l_len;          /* len == 0 means until end of file */
244         int     l_sysid;
245         pid_t   l_pid;
246         long    l_pad[4];               /* reserve area */
247 } flock64_t;

249 #if defined(_SYSCALL32)

251 /* Kernel's view of ILP32 flock64 */

253 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
254 #pragma pack(4)
255 #endif

257 typedef struct flock64_32 {
258         int16_t l_type;
259         int16_t l_whence;
```

```
260         off64_t l_start;
261         off64_t l_len;          /* len == 0 means until end of file */
262         int32_t l_sysid;
263         pid32_t l_pid;
264         int32_t l_pad[4];               /* reserve area */
265 } flock64_32_t;

267 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
268 #pragma pack()
269 #endif

271 /* Kernel's view of LP64 flock64 */

273 typedef struct flock64_64 {
274         int16_t l_type;
275         int16_t l_whence;
276         off64_t l_start;
277         off64_t l_len;          /* len == 0 means until end of file */
278         int32_t l_sysid;
279         pid32_t l_pid;
280         int64_t l_pad[4];               /* reserve area */
281 } flock64_64_t;

283 #endif  /* _SYSCALL32 */

285 #endif /* _LARGEFILE64_SOURCE */

287 #if defined(_KERNEL) || defined(_KMEMUSER)
288 /* SVr3 flock type; needed for rfs across the wire compatibility */
289 typedef struct o_flock {
290         int16_t l_type;
291         int16_t l_whence;
292         int32_t l_start;
293         int32_t l_len;          /* len == 0 means until end of file */
294         int16_t l_sysid;
295         int16_t l_pid;
296 } o_flock_t;
297 #endif  /* defined(_KERNEL) */

299 /*
300  * File segment locking types.
301  */
302 #define F_RDLCK         01      /* Read lock */
303 #define F_WRLCK         02      /* Write lock */
304 #define F_UNLCK         03      /* Remove lock(s) */
305 #define F_UNLKSYS       04      /* remove remote locks for a given system */

307 /*
308  * POSIX constants
309  */

311 /* Mask for file access modes */
312 #define O_ACCMODE       (O_SEARCH | O_EXEC | 0x3)
313 #define FD_CLOEXEC      1       /* close on exec flag */

315 /*
316  * DIRECTIO
317  */
318 #if defined(__EXTENSIONS__) || !defined(__XOPEN_OR_POSIX)
319 #define DIRECTIO_OFF    (0)
320 #define DIRECTIO_ON     (1)

322 /*
323  * File share reservation type
324  */
325 typedef struct fshare {
```



```
326         short   f_access;
327         short   f_deny;
328         int     f_id;
329 } fshare_t;

331 /*
332  * f_access values
333  */
334 #define F_RDACC         0x1     /* Read-only share access */
335 #define F_WRACC         0x2     /* Write-only share access */
336 #define F_RWACC         0x3     /* Read-Write share access */
337 #define F_RMACC         0x4     /* private flag: Delete share access */
338 #define F_MDACC         0x20    /* private flag: Metadata share access */

340 /*
341  * f_deny values
342  */
343 #define F_NODNY         0x0     /* Don't deny others access */
344 #define F_RDDNY         0x1     /* Deny others read share access */
345 #define F_WRDNY         0x2     /* Deny others write share access */
346 #define F_RWDNY         0x3     /* Deny others read or write share access */
347 #define F_RMDNY         0x4     /* private flag: Deny delete share access */
348 #define F_COMPAT        0x8     /* Set share to old DOS compatibility mode */
349 #define F_MANDDNY       0x10    /* private flag: mandatory enforcement */
350 #endif /* defined(__EXTENSIONS__) || !defined(__XOPEN_OR_POSIX) */

352 /*
353  * Special flags for functions such as openat(), fstatat()....
354  */
355 #if !defined(__XOPEN_OR_POSIX) || defined(_ATFILE_SOURCE) || \
356         defined(__EXTENSIONS__)
357         /* || defined(_XPG7) */
358 #define AT_FDCWD                        0xffd19553
359 #define AT_SYMLINK_NOFOLLOW             0x1000
360 #define AT_SYMLINK_FOLLOW               0x2000  /* only for linkat() */
361 #define AT_REMOVEDIR                    0x1
362 #define _AT_TRIGGER                     0x2
363 #define AT_EACCESS                      0x4     /* use EUID/EGID for access */
364 #endif

366 #if !defined(__XOPEN_OR_POSIX) || defined(_XPG6) || defined(__EXTENSIONS__)
367 /* advice for posix_fadvise */
368 #define POSIX_FADV_NORMAL       0
369 #define POSIX_FADV_RANDOM       1
370 #define POSIX_FADV_SEQUENTIAL   2
371 #define POSIX_FADV_WILLNEED     3
372 #define POSIX_FADV_DONTNEED     4
373 #define POSIX_FADV_NOREUSE      5
374 #endif

376 #ifdef  __cplusplus
377 }
378 #endif

380 #endif  /* _SYS_FCNTL_H */
```

```
**********************************************************
    7037 Mon Aug 17 21:08:08 2015
new/usr/src/uts/common/sys/file.h
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

  79 /* f_flag */

  81 #define FOPEN           0xffffffff
  82 #define FREAD           0x01    /* <sys/aiocb.h> LIO_READ must be identical */
  83 #define FWRITE          0x02    /* <sys/aiocb.h> LIO_WRITE must be identical */
  84 #define FNDELAY         0x04
  85 #define FAPPEND         0x08
  86 #define FSYNC           0x10    /* file (data+inode) integrity while writing */
  87 #define FREVOKED        0x20    /* Object reuse Revoked file */
  88 #define FDSYNC          0x40    /* file data only integrity while writing */
  89 #define FNONBLOCK       0x80

  91 #define FMASK           0xa0ff  /* all flags that can be changed by F_SETFL */

  93 /* open-only modes */

  95 #define FCREAT          0x0100
  96 #define FTRUNC          0x0200
  97 #define FEXCL           0x0400
  98 #define FASYNC          0x1000  /* asyncio in progress pseudo flag */
  99 #define FOFFMAX         0x2000  /* large file */
 100 #define FXATTR          0x4000  /* open as extended attribute */
 101 #define FNOCTTY         0x0800
 102 #define FRSYNC          0x8000  /* sync read operations at same level of */
 103                                 /* integrity as specified for writes by */
 104                                 /* FSYNC and FDSYNC flags */

 106 #define FNODSYNC        0x10000 /* fsync pseudo flag */

 108 #define FNOFOLLOW       0x20000 /* don't follow symlinks */
 109 #define FNOLINKS        0x40000 /* don't allow multiple hard links */
 110 #define FIGNORECASE     0x80000 /* request case-insensitive lookups */
 111 #define FXATTRDIROPEN   0x100000  /* only opening hidden attribute directory */

 113 /* f_flag2 (open-only) */

 115 #define FSEARCH         0x200000        /* O_SEARCH = 0x200000 */
 116 #define FEXEC           0x400000        /* O_EXEC = 0x400000 */

 118 #define FCLOEXEC        0x800000        /* O_CLOEXEC = 0x800000 */

 120 #ifdef _KERNEL

 122 /*
 123  * Fake flags for driver ioctl calls to inform them of the originating
 124  * process' model.  See <sys/model.h>
 125  *
 126  * Part of the Solaris 2.6+ DDI/DKI
 127  */
 128 #define FMODELS DATAMODEL_MASK  /* Note: 0x0ff00000 */
 129 #define FILP32  DATAMODEL_ILP32
 130 #define FLP64   DATAMODEL_LP64
 131 #define FNATIVE DATAMODEL_NATIVE

 133 /*
 134  * Large Files: The macro gets the offset maximum (refer to LFS API doc)
 135  * corresponding to a file descriptor. We had the choice of storing
 136  * this value in file descriptor. Right now we only have two
 137  * offset maximums one if MAXOFF_T and other is MAXOFFSET_T. It is
```



```
 138  * inefficient to store these two values in a separate member in
 139  * file descriptor. To avoid wasting spaces we define this macro.
 140  * The day there are more than two offset maximum we may want to
 141  * rewrite this macro.
 142  */

 144 #define OFFSET_MAX(fd)  ((fd->f_flag & FOFFMAX) ? MAXOFFSET_T : MAXOFF32_T)

 146 /*
 147  * Fake flag => internal ioctl call for layered drivers.
 148  * Note that this flag deliberately *won't* fit into
 149  * the f_flag field of a file_t.
 150  *
 151  * Part of the Solaris 2.x DDI/DKI.
 152  */
 153 #define FKIOCTL         0x80000000      /* ioctl addresses are from kernel */

 155 /*
 156  * Fake flag => this time to specify that the open(9E)
 157  * comes from another part of the kernel, not userland.
 158  *
 159  * Part of the Solaris 2.x DDI/DKI.
 160  */
 161 #define FKLYR           0x40000000      /* layered driver call */

 163 #endif  /* _KERNEL */

 165 /* miscellaneous defines */

 167 #ifndef L_SET
 168 #define L_SET   0       /* for lseek */
 169 #endif /* L_SET */

 171 #if defined(_KERNEL)

 173 /*
 174  * Routines dealing with user per-open file flags and
 175  * user open files.
 176  */
 177 struct proc;    /* forward reference for function prototype */
 178 struct vnodeops;
 179 struct vattr;

 181 extern file_t *getf(int);
 182 extern void releasef(int);
 183 extern void areleasef(int, uf_info_t *);
 184 #ifndef _BOOT
 185 extern void closeall(uf_info_t *);
 186 #endif
 187 extern void flist_fork(proc_t *, proc_t *);
 187 extern void flist_fork(uf_info_t *, uf_info_t *);
 188 extern int closef(file_t *);
 189 extern int closeandsetf(int, file_t *);
 190 extern int ufalloc_file(int, file_t *);
 191 extern int ufalloc(int);
 192 extern int ufcanalloc(struct proc *, uint_t);
 193 extern int falloc(struct vnode *, int, file_t **, int *);
 194 extern void finit(void);
 195 extern void unfalloc(file_t *);
 196 extern void setf(int, file_t *);
 197 extern int f_getfd_error(int, int *);
 198 extern char f_getfd(int);
 199 extern int f_setfd_error(int, int);
 200 extern void f_setfd(int, char);
 201 extern int f_getfl(int, int *);
 202 extern int f_badfd(int, int *, int);
```

```
 203 extern int fassign(struct vnode **, int, int *);
 204 extern void fcnt_add(uf_info_t *, int);
 205 extern void close_exec(uf_info_t *);
 206 extern void clear_stale_fd(void);
 207 extern void clear_active_fd(int);
 208 extern void free_afd(afd_t *afd);
 209 extern int fgetstartvp(int, char *, struct vnode **);
 210 extern int fsetattrat(int, char *, int, struct vattr *);
 211 extern int fisopen(struct vnode *);
 212 extern void delfpollinfo(int);
 213 extern void addfpollinfo(int);
 214 extern int sock_getfasync(struct vnode *);
 215 extern int files_can_change_zones(void);
 216 #ifdef DEBUG
 217 /* The following functions are only used in ASSERT()s */
 218 extern void checkwfdlist(struct vnode *, fpollinfo_t *);
 219 extern void checkfpollinfo(void);
 220 extern int infpollinfo(int);
 221 #endif  /* DEBUG */

 223 #endif  /* defined(_KERNEL) */

 225 #ifdef  __cplusplus
 226 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
    1912 Mon Aug 17 21:08:09 2015
new/usr/src/uts/common/sys/list.h
XXXX adding PID information to netstat output
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_LIST_H
#define _SYS_LIST_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/list_impl.h>

#ifdef  __cplusplus
extern "C" {
#endif

typedef struct list_node list_node_t;
typedef struct list list_t;

void list_create(list_t *, size_t, size_t);
void list_destroy(list_t *);

void list_insert_after(list_t *, void *, void *);
void list_insert_before(list_t *, void *, void *);
void list_insert_head(list_t *, void *);
void list_insert_tail(list_t *, void *);
void list_remove(list_t *, void *);
void *list_remove_head(list_t *);
void *list_remove_tail(list_t *);
void list_move_tail(list_t *, list_t *);

void *list_head(list_t *);
void *list_tail(list_t *);
void *list_next(list_t *, void *);
void *list_prev(list_t *, void *);
int list_is_empty(list_t *);
ulong_t list_numnodes(list_t *);
#endif /* ! codereview */

void list_link_init(list_node_t *);
void list_link_replace(list_node_t *, list_node_t *);
```



```
int list_link_active(list_node_t *);

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_LIST_H */
```

```
**********************************************************
    1353 Mon Aug 17 21:08:09 2015
new/usr/src/uts/common/sys/list_impl.h
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

 43 struct list {
 44         size_t  list_size;
 45         size_t  list_offset;
 46         ulong_t list_numnodes;
 47 #endif /* ! codereview */
 48         struct list_node list_head;
 49 };

 51 #ifdef  __cplusplus
 52 }
 53 #endif

 55 #endif  /* _SYS_LIST_IMPL_H */
```

```
**********************************************************
    1243 Mon Aug 17 21:08:09 2015
new/usr/src/uts/common/sys/pidnode.h
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2015 Mohamed A. Khalfella <khalfella@gmail.com>
 */

#ifndef _SYS_PIDNODE_H
#define _SYS_PIDNODE_H

#include <sys/list.h>

#ifdef  __cplusplus
extern "C" {
#endif

#define CONN_PID_INFO_MGC       0x5A7A0B1D

#define CONN_PID_INFO_NON       0
#define CONN_PID_INFO_SOC       1
#define CONN_PID_INFO_XTI       2

typedef struct conn_pid_info_s {
        uint16_t        cpi_contents;   /* CONN_PID_INFO_* */
        uint32_t        cpi_magic;      /* CONN_PID_INFO_MGC */
        uint32_t        cpi_pids_cnt;   /* # of elements in cpi_pids */
        uint32_t        cpi_tot_size;   /* total size of hdr + pids */
        pid_t           cpi_pids[1];    /* varialbe length array of pids */
} conn_pid_info_t;

#if defined(_KERNEL)

typedef struct pid_node_s {
        list_node_t     pn_ref_link;
        uint32_t        pn_count;
        pid_t           pn_pid;
} pid_node_t;

#endif /* defined(_KERNEL) */

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_PIDNODE_H */
#endif /* ! codereview */
```

```
**********************************************************
    8333 Mon Aug 17 21:08:09 2015
new/usr/src/uts/common/sys/socket_proto.h
XXXX adding PID information to netstat output
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#ifndef _SYS_SOCKET_PROTO_H_
#define _SYS_SOCKET_PROTO_H_

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/socket.h>
#include <sys/pidnode.h>
#endif /* ! codereview */

/*
 * Generation count
 */
typedef uint64_t sock_connid_t;

#define SOCK_CONNID_INIT(id) {  \
        (id) = 0;               \
}
#define SOCK_CONNID_BUMP(id)            (++(id))
#define SOCK_CONNID_LT(id1, id2)        ((int64_t)((id1)-(id2)) < 0)

/* Socket protocol properties */
struct sock_proto_props {
        uint_t sopp_flags;              /* options to set */
        ushort_t sopp_wroff;            /* write offset */
        ssize_t sopp_txhiwat;           /* tx hi water mark */
        ssize_t sopp_txlowat;           /* tx lo water mark */
        ssize_t sopp_rxhiwat;           /* recv high water mark */
        ssize_t sopp_rxlowat;           /* recv low water mark */
        ssize_t sopp_maxblk;            /* maximum message block size */
        ssize_t sopp_maxpsz;            /* maximum packet size */
        ssize_t sopp_minpsz;            /* minimum packet size */
        ushort_t sopp_tail;             /* space available at the end */
        uint_t  sopp_zcopyflag;         /* zero copy flag */
        boolean_t sopp_oobinline;       /* OOB inline */
        uint_t sopp_rcvtimer;           /* delayed recv notification (time) */
```



```
        uint32_t sopp_rcvthresh;        /* delayed recv notification (bytes) */
        socklen_t sopp_maxaddrlen;      /* maximum size of protocol address */
        boolean_t sopp_loopback;        /* loopback connection */
};

/* flags to determine which socket options are set */
#define SOCKOPT_WROFF           0x0001  /* set write offset */
#define SOCKOPT_RCVHIWAT        0x0002  /* set read side high water */
#define SOCKOPT_RCVLOWAT        0x0004  /* set read side high water */
#define SOCKOPT_MAXBLK          0x0008  /* set maximum message block size */
#define SOCKOPT_TAIL            0x0010  /* set the extra allocated space */
#define SOCKOPT_ZCOPY           0x0020  /* set/unset zero copy for sendfile */
#define SOCKOPT_MAXPSZ          0x0040  /* set maxpsz for protocols */
#define SOCKOPT_OOBINLINE       0x0080  /* set oob inline processing */
#define SOCKOPT_RCVTIMER        0x0100
#define SOCKOPT_RCVTHRESH       0x0200
#define SOCKOPT_MAXADDRLEN      0x0400  /* set max address length */
#define SOCKOPT_MINPSZ          0x0800  /* set minpsz for protocols */
#define SOCKOPT_LOOPBACK        0x1000  /* set loopback */

#define IS_SO_OOB_INLINE(so)    ((so)->so_proto_props.sopp_oobinline)

#ifdef _KERNEL

struct T_capability_ack;

typedef struct sock_upcalls_s sock_upcalls_t;
typedef struct sock_downcalls_s sock_downcalls_t;

/*
 * Upcall and downcall handle for sockfs and transport layer.
 */
typedef struct __sock_upper_handle *sock_upper_handle_t;
typedef struct __sock_lower_handle *sock_lower_handle_t;

struct sock_downcalls_s {
        void    (*sd_activate)(sock_lower_handle_t, sock_upper_handle_t,
                    sock_upcalls_t *, int, cred_t *);
        int     (*sd_accept)(sock_lower_handle_t, sock_lower_handle_t,
                    sock_upper_handle_t, cred_t *);
        int     (*sd_bind)(sock_lower_handle_t, struct sockaddr *, socklen_t,
                    cred_t *);
        int     (*sd_listen)(sock_lower_handle_t, int, cred_t *);
        int     (*sd_connect)(sock_lower_handle_t, const struct sockaddr *,
                    socklen_t, sock_connid_t *, cred_t *);
        int     (*sd_getpeername)(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
        int     (*sd_getsockname)(sock_lower_handle_t, struct sockaddr *,
                    socklen_t *, cred_t *);
        int     (*sd_getsockopt)(sock_lower_handle_t, int, int, void *,
                    socklen_t *, cred_t *);
        int     (*sd_setsockopt)(sock_lower_handle_t, int, int, const void *,
                    socklen_t, cred_t *);
        int     (*sd_send)(sock_lower_handle_t, mblk_t *, struct nmsghdr *,
                    cred_t *);
        int     (*sd_send_uio)(sock_lower_handle_t, uio_t *, struct nmsghdr *,
                    cred_t *);
        int     (*sd_recv_uio)(sock_lower_handle_t, uio_t *, struct nmsghdr *,
                    cred_t *);
        short   (*sd_poll)(sock_lower_handle_t, short, int, cred_t *);
        int     (*sd_shutdown)(sock_lower_handle_t, int, cred_t *);
        void    (*sd_clr_flowctrl)(sock_lower_handle_t);
        int     (*sd_ioctl)(sock_lower_handle_t, int, intptr_t, int,
                    int32_t *, cred_t *);
        int     (*sd_close)(sock_lower_handle_t, int, cred_t *);
};
```

```
129 typedef sock_lower_handle_t (*so_proto_create_func_t)(int, int, int,
130     sock_downcalls_t **, uint_t *, int *, int, cred_t *);

132 typedef struct sock_quiesce_arg {
133         mblk_t *soqa_exdata_mp;
134         mblk_t *soqa_urgmark_mp;
135 } sock_quiesce_arg_t;
136 typedef mblk_t *(*so_proto_quiesced_cb_t)(sock_upper_handle_t,
137     sock_quiesce_arg_t *, struct T_capability_ack *, struct sockaddr *,
138     socklen_t, struct sockaddr *, socklen_t, short);
139 typedef int (*so_proto_fallback_func_t)(sock_lower_handle_t, queue_t *,
140     boolean_t, so_proto_quiesced_cb_t, sock_quiesce_arg_t *);

142 /*
143  * These functions return EOPNOTSUPP and are intended for the sockfs
144  * developer that doesn't wish to supply stubs for every function themselves.
145  */
146 extern int sock_accept_notsupp(sock_lower_handle_t, sock_lower_handle_t,
147     sock_upper_handle_t, cred_t *);
148 extern int sock_bind_notsupp(sock_lower_handle_t, struct sockaddr *,
149     socklen_t, cred_t *);
150 extern int sock_listen_notsupp(sock_lower_handle_t, int, cred_t *);
151 extern int sock_connect_notsupp(sock_lower_handle_t,
152     const struct sockaddr *, socklen_t, sock_connid_t *, cred_t *);
153 extern int sock_getpeername_notsupp(sock_lower_handle_t, struct sockaddr *,
154     socklen_t *, cred_t *);
155 extern int sock_getsockname_notsupp(sock_lower_handle_t, struct sockaddr *,
156     socklen_t *, cred_t *);
157 extern int sock_getsockopt_notsupp(sock_lower_handle_t, int, int, void *,
158     socklen_t *, cred_t *);
159 extern int sock_setsockopt_notsupp(sock_lower_handle_t, int, int,
160     const void *, socklen_t, cred_t *);
161 extern int sock_send_notsupp(sock_lower_handle_t, mblk_t *,
162     struct nmsghdr *, cred_t *);
163 extern int sock_send_uio_notsupp(sock_lower_handle_t, uio_t *,
164     struct nmsghdr *, cred_t *);
165 extern int sock_recv_uio_notsupp(sock_lower_handle_t, uio_t *,
166     struct nmsghdr *, cred_t *);
167 extern short sock_poll_notsupp(sock_lower_handle_t, short, int, cred_t *);
168 extern int sock_shutdown_notsupp(sock_lower_handle_t, int, cred_t *);
169 extern void sock_clr_flowctrl_notsupp(sock_lower_handle_t);
170 extern int sock_ioctl_notsupp(sock_lower_handle_t, int, intptr_t, int,
171     int32_t *, cred_t *);
172 extern int sock_close_notsupp(sock_lower_handle_t, int, cred_t *);

174 /*
175  * Upcalls and related information
176  */

178 /*
179  * su_opctl() actions
180  */
181 typedef enum sock_opctl_action {
182         SOCK_OPCTL_ENAB_ACCEPT = 0,
183         SOCK_OPCTL_SHUT_SEND,
184         SOCK_OPCTL_SHUT_RECV
185 } sock_opctl_action_t;

187 struct sock_upcalls_s {
188         sock_upper_handle_t (*su_newconn)(sock_upper_handle_t,
189             sock_lower_handle_t, sock_downcalls_t *, cred_t *, pid_t,
190             sock_upcalls_t **);
191         void    (*su_connected)(sock_upper_handle_t, sock_connid_t, cred_t *,
192             pid_t);
193         int     (*su_disconnected)(sock_upper_handle_t, sock_connid_t, int);
```



```
194         void    (*su_opctl)(sock_upper_handle_t, sock_opctl_action_t,
195             uintptr_t);
196         ssize_t (*su_recv)(sock_upper_handle_t, mblk_t *, size_t, int,
197                     int *, boolean_t *);
198         void    (*su_set_proto_props)(sock_upper_handle_t,
199                     struct sock_proto_props *);
200         void    (*su_txq_full)(sock_upper_handle_t, boolean_t);
201         void    (*su_signal_oob)(sock_upper_handle_t, ssize_t);
202         void    (*su_zcopy_notify)(sock_upper_handle_t);
203         void    (*su_set_error)(sock_upper_handle_t, int);
204         void    (*su_closed)(sock_upper_handle_t);
205         mblk_t *        (*su_get_sock_pid_mblk)(sock_upper_handle_t);
206 #endif /* ! codereview */
207 };

209 #define SOCK_UC_VERSION         sizeof (sock_upcalls_t)
210 #define SOCK_DC_VERSION         sizeof (sock_downcalls_t)

212 #define SOCKET_RECVHIWATER      (48 * 1024)
213 #define SOCKET_RECVLOWATER      1024

215 #define SOCKET_NO_RCVTIMER      0
216 #define SOCKET_TIMER_INTERVAL   50

218 #endif /* _KERNEL */

220 #ifdef  __cplusplus
221 }
222 #endif

224 #endif /* _SYS_SOCKET_PROTO_H_ */
```

```
**********************************************************
   35588 Mon Aug 17 21:08:09 2015
new/usr/src/uts/common/sys/socketvar.h
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1996, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */
/*
 * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
 */

#ifndef _SYS_SOCKETVAR_H
#define _SYS_SOCKETVAR_H

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/t_lock.h>
#include <sys/cred.h>
#include <sys/pidnode.h>
#endif /* ! codereview */
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/param.h>
#include <sys/zone.h>
#include <sys/sdt.h>
#include <sys/modctl.h>
#include <sys/atomic.h>
#include <sys/socket.h>
#include <sys/ksocket.h>
#include <sys/kstat.h>
```



```
#ifdef _KERNEL
#include <sys/vfs_opreg.h>
#endif

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Internal representation of the address used to represent addresses
 * in the loopback transport for AF_UNIX. While the sockaddr_un is used
 * as the sockfs layer address for AF_UNIX the pathnames contained in
 * these addresses are not unique (due to relative pathnames) thus can not
 * be used in the transport.
 *
 * The transport level address consists of a magic number (used to separate the
 * name space for specific and implicit binds). For a specific bind
 * this is followed by a "vnode *" which ensures that all specific binds
 * have a unique transport level address. For implicit binds the latter
 * part of the address is a byte string (of the same length as a pointer)
 * that is assigned by the loopback transport.
 *
 * The uniqueness assumes that the loopback transport has a separate namespace
 * for sockets in order to avoid name conflicts with e.g. TLI use of the
 * same transport.
 */
struct so_ux_addr {
        void    *soua_vp;       /* vnode pointer or assigned by tl */
        uint_t  soua_magic;     /* See below */
};

#define SOU_MAGIC_EXPLICIT      0x75787670      /* "uxvp" */
#define SOU_MAGIC_IMPLICIT      0x616e6f6e      /* "anon" */

struct sockaddr_ux {
        sa_family_t             sou_family;     /* AF_UNIX */
        struct so_ux_addr       sou_addr;
};

#if defined(_KERNEL) || defined(_KMEMUSER)

#include <sys/socket_proto.h>

typedef struct sonodeops sonodeops_t;
typedef struct sonode sonode_t;

struct sodirect_s;

/*
 * The sonode represents a socket. A sonode never exist in the file system
 * name space and can not be opened using open() - only the socket, socketpair
 * and accept calls create sonodes.
 *
 * The locking of sockfs uses the so_lock mutex plus the SOLOCKED and
 * SOREADLOCKED flags in so_flag. The mutex protects all the state in the
 * sonode. It is expected that the underlying transport protocol serializes
 * socket operations, so sockfs will not normally not single-thread
 * operations. However, certain sockets, including TPI based ones, can only
 * handle one control operation at a time. The SOLOCKED flag is used to
 * single-thread operations from sockfs users to prevent e.g. multiple bind()
 * calls to operate on the same sonode concurrently. The SOREADLOCKED flag is
 * used to ensure that only one thread sleeps in kstrgetmsg for a given
 * sonode. This is needed to ensure atomic operation for things like
 * MSG_WAITALL.
 *
 * The so_fallback_rwlock is used to ensure that for sockets that can
```

```
 * fall back to TPI, the fallback is not initiated until all pending
 * operations have completed.
 *
 * Note that so_lock is sometimes held across calls that might go to sleep
 * (kmem_alloc and soallocproto*). This implies that no other lock in
 * the system should be held when calling into sockfs; from the system call
 * side or from strrput (in case of TPI based sockets). If locks are held
 * while calling into sockfs the system might hang when running low on memory.
 */
struct sonode {
        struct  vnode   *so_vnode;      /* vnode associated with this sonode */

        sonodeops_t     *so_ops;        /* operations vector for this sonode */
        void            *so_priv;       /* sonode private data */

        krwlock_t       so_fallback_rwlock;
        kmutex_t        so_lock;        /* protects sonode fields */

        kcondvar_t      so_state_cv;    /* synchronize state changes */
        kcondvar_t      so_single_cv;   /* wait due to SOLOCKED */
        kcondvar_t      so_read_cv;     /* wait due to SOREADLOCKED */

        /* These fields are protected by so_lock */

        uint_t          so_state;       /* internal state flags SS_*, below */
        uint_t          so_mode;        /* characteristics on socket. SM_* */
        ushort_t        so_flag;        /* flags, see below */
        int             so_count;       /* count of opened references */

        sock_connid_t   so_proto_connid; /* protocol generation number */

        ushort_t        so_error;       /* error affecting connection */

        struct sockparams *so_sockparams;       /* vnode or socket module */
        /* Needed to recreate the same socket for accept */
        short   so_family;
        short   so_type;
        short   so_protocol;
        short   so_version;             /* From so_socket call */

        /* Accept queue */
        kmutex_t        so_acceptq_lock;        /* protects accept queue */
        list_t          so_acceptq_list;        /* pending conns */
        list_t          so_acceptq_defer;       /* deferred conns */
        list_node_t     so_acceptq_node;        /* acceptq list node */
        unsigned int    so_acceptq_len;         /* # of conns (both lists) */
        unsigned int    so_backlog;             /* Listen backlog */
        kcondvar_t      so_acceptq_cv;          /* wait for new conn. */
        struct sonode   *so_listener;           /* parent socket */

        /* Options */
        short   so_options;             /* From socket call, see socket.h */
        struct linger   so_linger;      /* SO_LINGER value */
#define so_sndbuf       so_proto_props.sopp_txhiwat     /* SO_SNDBUF value */
#define so_sndlowat     so_proto_props.sopp_txlowat     /* tx low water mark */
#define so_rcvbuf       so_proto_props.sopp_rxhiwat     /* SO_RCVBUF value */
#define so_rcvlowat     so_proto_props.sopp_rxlowat     /* rx low water mark */
#define so_max_addr_len so_proto_props.sopp_maxaddrlen
#define so_minpsz       so_proto_props.sopp_minpsz
#define so_maxpsz       so_proto_props.sopp_maxpsz

        int     so_xpg_rcvbuf;          /* SO_RCVBUF value for XPG4 socket */
        clock_t so_sndtimeo;            /* send timeout */
        clock_t so_rcvtimeo;            /* recv timeout */

        mblk_t  *so_oobmsg;             /* outofline oob data */
```



```
        ssize_t so_oobmark;             /* offset of the oob data */

        pid_t   so_pgrp;                /* pgrp for signals */

        cred_t          *so_peercred;   /* connected socket peer cred */
        pid_t           so_cpid;        /* connected socket peer cached pid */
        zoneid_t        so_zoneid;      /* opener's zoneid */

        struct pollhead so_poll_list;   /* common pollhead */
        short           so_pollev;      /* events that should be generated */

        /* Receive */
        unsigned int    so_rcv_queued;  /* # bytes on both rcv lists */
        mblk_t          *so_rcv_q_head; /* processing/copyout rcv queue */
        mblk_t          *so_rcv_q_last_head;
        mblk_t          *so_rcv_head;   /* protocol prequeue */
        mblk_t          *so_rcv_last_head;      /* last mblk in b_next chain */
        kcondvar_t      so_rcv_cv;      /* wait for data */
        uint_t          so_rcv_wanted;  /* # of bytes wanted by app */
        timeout_id_t    so_rcv_timer_tid;

#define so_rcv_thresh   so_proto_props.sopp_rcvthresh
#define so_rcv_timer_interval so_proto_props.sopp_rcvtimer

        kcondvar_t      so_snd_cv;      /* wait for snd buffers */
        uint32_t
                so_snd_qfull: 1,        /* Transmit full */
                so_rcv_wakeup: 1,
                so_snd_wakeup: 1,
                so_not_str: 1,  /* B_TRUE if not streams based socket */
                so_pad_to_bit_31: 28;

        /* Communication channel with protocol */
        sock_lower_handle_t     so_proto_handle;
        sock_downcalls_t        *so_downcalls;

        struct sock_proto_props so_proto_props; /* protocol settings */
        boolean_t               so_flowctrld;   /* Flow controlled */
        uint_t                  so_copyflag;    /* Copy related flag */
        kcondvar_t              so_copy_cv;     /* Copy cond variable */

        /* kernel sockets */
        ksocket_callbacks_t     so_ksock_callbacks;
        void                    *so_ksock_cb_arg;       /* callback argument */
        kcondvar_t              so_closing_cv;

        /* != NULL for sodirect enabled socket */
        struct sodirect_s       *so_direct;

        /* socket filters */
        uint_t                  so_filter_active;       /* # of active fil */
        uint_t                  so_filter_tx;           /* pending tx ops */
        struct sof_instance     *so_filter_top;         /* top of stack */
        struct sof_instance     *so_filter_bottom;      /* bottom of stack */
        clock_t                 so_filter_defertime;    /* time when deferred */

        /* pid list */
        list_t                  so_pid_list;
        kmutex_t                so_pid_list_lock;
#endif /* ! codereview */
};

#define SO_HAVE_DATA(so)                                                \
        /*                                                              \
         * For the (tid == 0) case we must check so_rcv_{q_,}head       \
         * rather than (so_rcv_queued > 0), since the latter does not   \
```

```
260          * take into account mblks with only control/name information.  \
261          */                                                             \
262         ((so)->so_rcv_timer_tid == 0 && ((so)->so_rcv_head != NULL ||   \
263         (so)->so_rcv_q_head != NULL)) ||                                \
264         ((so)->so_state & SS_CANTRCVMORE)

266 /*
267  * Events handled by the protocol (in case sd_poll is set)
268  */
269 #define SO_PROTO_POLLEV         (POLLIN|POLLRDNORM|POLLRDBAND)


272 #endif /* _KERNEL || _KMEMUSER */

274 /* flags */
275 #define SOMOD           0x0001          /* update socket modification time */
276 #define SOACC           0x0002          /* update socket access time */

278 #define SOLOCKED        0x0010          /* use to serialize open/closes */
279 #define SOREADLOCKED    0x0020          /* serialize kstrgetmsg calls */
280 #define SOCLONE         0x0040          /* child of clone driver */
281 #define SOASYNC_UNBIND  0x0080          /* wait for ACK of async unbind */

283 #define SOCK_IS_NONSTR(so)      ((so)->so_not_str)

285 /*
286  * Socket state bits.
287  */
288 #define SS_ISCONNECTED          0x00000001 /* socket connected to a peer */
289 #define SS_ISCONNECTING         0x00000002 /* in process, connecting to peer */
290 #define SS_ISDISCONNECTING      0x00000004 /* in process of disconnecting */
291 #define SS_CANTSENDMORE         0x00000008 /* can't send more data to peer */

293 #define SS_CANTRCVMORE          0x00000010 /* can't receive more data */
294 #define SS_ISBOUND              0x00000020 /* socket is bound */
295 #define SS_NDELAY               0x00000040 /* FNDELAY non-blocking */
296 #define SS_NONBLOCK             0x00000080 /* O_NONBLOCK non-blocking */

298 #define SS_ASYNC                0x00000100 /* async i/o notify */
299 #define SS_ACCEPTCONN           0x00000200 /* listen done */
300 /*      unused                  0x00000400 */   /* was SS_HASCONNIND */
301 #define SS_SAVEDEOR             0x00000800 /* Saved MSG_EOR rcv side state */

303 #define SS_RCVATMARK            0x00001000 /* at mark on input */
304 #define SS_OOBPEND              0x00002000 /* OOB pending or present - poll */
305 #define SS_HAVEOOBDATA          0x00004000 /* OOB data present */
306 #define SS_HADOOBDATA           0x00008000 /* OOB data consumed */
307 #define SS_CLOSING              0x00010000 /* in process of closing */

309 #define SS_FIL_DEFER            0x00020000 /* filter deferred notification */
310 #define SS_FILOP_OK             0x00040000 /* socket can attach filters */
311 #define SS_FIL_RCV_FLOWCTRL     0x00080000 /* filter asserted rcv flow ctrl */
312 #define SS_FIL_SND_FLOWCTRL     0x00100000 /* filter asserted snd flow ctrl */
313 #define SS_FIL_STOP             0x00200000 /* no more filter actions */

315 #define SS_SODIRECT             0x00400000 /* transport supports sodirect */

317 #define SS_SENTLASTREADSIG      0x01000000 /* last rx signal has been sent */
318 #define SS_SENTLASTWRITESIG     0x02000000 /* last tx signal has been sent */

320 #define SS_FALLBACK_DRAIN       0x20000000 /* data was/is being drained */
321 #define SS_FALLBACK_PENDING     0x40000000 /* fallback is pending */
322 #define SS_FALLBACK_COMP        0x80000000 /* fallback has completed */


325 /* Set of states when the socket can't be rebound */
```



```
326 #define SS_CANTREBIND   (SS_ISCONNECTED|SS_ISCONNECTING|SS_ISDISCONNECTING|\
327                             SS_CANTSENDMORE|SS_CANTRCVMORE|SS_ACCEPTCONN)

329 /*
330  * Sockets that can fall back to TPI must ensure that fall back is not
331  * initiated while a thread is using a socket.
332  */
333 #define SO_BLOCK_FALLBACK(so, fn)                                       \
334         ASSERT(MUTEX_NOT_HELD(&(so)->so_lock));                         \
335         rw_enter(&(so)->so_fallback_rwlock, RW_READER);                 \
336         if ((so)->so_state & (SS_FALLBACK_COMP|SS_FILOP_OK)) {          \
337                 if ((so)->so_state & SS_FALLBACK_COMP) {                \
338                         rw_exit(&(so)->so_fallback_rwlock);             \
339                         return (fn);                                    \
340                 } else {                                                \
341                         mutex_enter(&(so)->so_lock);                    \
342                         (so)->so_state &= ~SS_FILOP_OK;                 \
343                         mutex_exit(&(so)->so_lock);                     \
344                 }                                                       \
345         }

347 #define SO_UNBLOCK_FALLBACK(so) {                               \
348         rw_exit(&(so)->so_fallback_rwlock);                     \
349 }

351 #define SO_SND_FLOWCTRLD(so)    \
352         ((so)->so_snd_qfull || (so)->so_state & SS_FIL_SND_FLOWCTRL)

354 /* Poll events */
355 #define SO_POLLEV_IN            0x1     /* POLLIN wakeup needed */
356 #define SO_POLLEV_ALWAYS        0x2     /* wakeups */

358 /*
359  * Characteristics of sockets. Not changed after the socket is created.
360  */
361 #define SM_PRIV                 0x001   /* privileged for broadcast, raw... */
362 #define SM_ATOMIC               0x002   /* atomic data transmission */
363 #define SM_ADDR                 0x004   /* addresses given with messages */
364 #define SM_CONNREQUIRED         0x008   /* connection required by protocol */

366 #define SM_FDPASSING            0x010   /* passes file descriptors */
367 #define SM_EXDATA               0x020   /* Can handle T_EXDATA_REQ */
368 #define SM_OPTDATA              0x040   /* Can handle T_OPTDATA_REQ */
369 #define SM_BYTESTREAM           0x080   /* Byte stream - can use M_DATA */

371 #define SM_ACCEPTOR_ID          0x100   /* so_acceptor_id is valid */

373 #define SM_KERNEL               0x200   /* kernel socket */

375 /* The modes below are only for non-streams sockets */
376 #define SM_ACCEPTSUPP           0x400   /* can handle accept() */
377 #define SM_SENDFILESUPP         0x800   /* Private: proto supp sendfile  */

379 /*
380  * Socket versions. Used by the socket library when calling _so_socket().
381  */
382 #define SOV_STREAM      0       /* Not a socket - just a stream */
383 #define SOV_DEFAULT     1       /* Select based on so_default_version */
384 #define SOV_SOCKSTREAM  2       /* Socket plus streams operations */
385 #define SOV_SOCKBSD     3       /* Socket with no streams operations */
386 #define SOV_XPG4_2      4       /* Xnet socket */

388 #if defined(_KERNEL) || defined(_KMEMUSER)

390 /*
391  * sonode create and destroy functions.
```

```
 */
typedef struct sonode *(*so_create_func_t)(struct sockparams *,
    int, int, int, int, int, int *, cred_t *);
typedef void (*so_destroy_func_t)(struct sonode *);

/* STREAM device information */
typedef struct sdev_info {
        char    *sd_devpath;
        int     sd_devpathlen; /* Is 0 if sp_devpath is a static string */
        vnode_t *sd_vnode;
} sdev_info_t;

#define SOCKMOD_VERSION_1       1
#define SOCKMOD_VERSION         2

/* name of the TPI pseudo socket module */
#define SOTPI_SMOD_NAME         "socktpi"

typedef struct __smod_priv_s {
        so_create_func_t        smodp_sock_create_func;
        so_destroy_func_t       smodp_sock_destroy_func;
        so_proto_fallback_func_t smodp_proto_fallback_func;
        const char              *smodp_fallback_devpath_v4;
        const char              *smodp_fallback_devpath_v6;
} __smod_priv_t;

/*
 * Socket module register information
 */
typedef struct smod_reg_s {
        int             smod_version;
        char            *smod_name;
        size_t          smod_uc_version;
        size_t          smod_dc_version;
        so_proto_create_func_t  smod_proto_create_func;

        /* __smod_priv_data must be NULL */
        __smod_priv_t   *__smod_priv;
} smod_reg_t;

/*
 * Socket module information
 */
typedef struct smod_info {
        int             smod_version;
        char            *smod_name;
        uint_t          smod_refcnt;            /* # of entries */
        size_t          smod_uc_version;        /* upcall version */
        size_t          smod_dc_version;        /* down call version */
        so_proto_create_func_t  smod_proto_create_func;
        so_proto_fallback_func_t smod_proto_fallback_func;
        const char              *smod_fallback_devpath_v4;
        const char              *smod_fallback_devpath_v6;
        so_create_func_t        smod_sock_create_func;
        so_destroy_func_t       smod_sock_destroy_func;
        list_node_t     smod_node;
} smod_info_t;

typedef struct sockparams_stats {
        kstat_named_t   sps_nfallback;  /* # of fallbacks to TPI */
        kstat_named_t   sps_nactive;    /* # of active sockets */
        kstat_named_t   sps_ncreate;    /* total # of created sockets */
} sockparams_stats_t;

/*
 * sockparams
```



```
 *
 * Used for mapping family/type/protocol to a socket module or STREAMS device
 */
struct sockparams {
        /*
         * The family, type, protocol, sdev_info and smod_name are
         * set when the entry is created, and they will never change
         * thereafter.
         */
        int             sp_family;
        int             sp_type;
        int             sp_protocol;

        sdev_info_t     sp_sdev_info;   /* STREAM device */
        char            *sp_smod_name;  /* socket module name */

        kmutex_t        sp_lock;        /* lock for refcnt and smod_info */
        uint64_t        sp_refcnt;      /* entry reference count */
        smod_info_t     *sp_smod_info;  /* socket module */

        sockparams_stats_t sp_stats;
        kstat_t         *sp_kstat;

        /*
         * The entries below are only modified while holding
         * sockconf_lock as a writer.
         */
        int             sp_flags;       /* see below */
        list_node_t     sp_node;

        list_t          sp_auto_filters; /* list of automatic filters */
        list_t          sp_prog_filters; /* list of programmatic filters */
};

struct sof_entry;

typedef struct sp_filter {
        struct sof_entry *spf_filter;
        list_node_t     spf_node;
} sp_filter_t;


/*
 * sockparams flags
 */
#define SOCKPARAMS_EPHEMERAL    0x1     /* temp. entry, not on global list */

extern void sockparams_init(void);
extern struct sockparams *sockparams_hold_ephemeral_bydev(int, int, int,
    const char *, int, int *);
extern struct sockparams *sockparams_hold_ephemeral_bymod(int, int, int,
    const char *, int, int *);
extern void sockparams_ephemeral_drop_last_ref(struct sockparams *);

extern struct sockparams *sockparams_create(int, int, int, char *, char *, int,
    int, int, int *);
extern void     sockparams_destroy(struct sockparams *);
extern int      sockparams_add(struct sockparams *);
extern int      sockparams_delete(int, int, int);
extern int      sockparams_new_filter(struct sof_entry *);
extern void     sockparams_filter_cleanup(struct sof_entry *);
extern int      sockparams_copyout_socktable(uintptr_t);

extern void smod_init(void);
extern void smod_add(smod_info_t *);
extern int smod_register(const smod_reg_t *);
```

```
524 extern int smod_unregister(const char *);
525 extern smod_info_t *smod_lookup_byname(const char *);

527 #define SOCKPARAMS_HAS_DEVICE(sp)                                       \
528         ((sp)->sp_sdev_info.sd_devpath != NULL)

530 /* Increase the smod_info_t reference count */
531 #define SMOD_INC_REF(smodp) {                                           \
532         ASSERT((smodp) != NULL);                                        \
533         DTRACE_PROBE1(smodinfo__inc__ref, struct smod_info *, (smodp)); \
534         atomic_inc_uint(&(smodp)->smod_refcnt);                         \
535 }

537 /*
538  * Decreace the socket module entry reference count.
539  * When no one mapping to the entry, we try to unload the module from the
540  * kernel. If the module can't unload, just leave the module entry with
541  * a zero refcnt.
542  */
543 #define SMOD_DEC_REF(smodp, modname) {                                  \
544         ASSERT((smodp) != NULL);                                        \
545         ASSERT((smodp)->smod_refcnt != 0);                              \
546         atomic_dec_uint(&(smodp)->smod_refcnt);                         \
547         /*                                                              \
548          * No need to atomically check the return value because the     \
549          * socket module framework will verify that no one is using     \
550          * the module before unloading. Worst thing that can happen     \
551          * here is multiple calls to mod_remove_by_name(), which is OK. \
552          */                                                             \
553         if ((smodp)->smod_refcnt == 0)                                  \
554                 (void) mod_remove_by_name(modname);                     \
555 }

557 /* Increase the reference count */
558 #define SOCKPARAMS_INC_REF(sp) {                                        \
559         ASSERT((sp) != NULL);                                           \
560         DTRACE_PROBE1(sockparams__inc__ref, struct sockparams *, (sp)); \
561         mutex_enter(&(sp)->sp_lock);                                    \
562         (sp)->sp_refcnt++;                                              \
563         ASSERT((sp)->sp_refcnt != 0);                                   \
564         mutex_exit(&(sp)->sp_lock);                                     \
565 }

567 /*
568  * Decrease the reference count.
569  *
570  * If the sockparams is ephemeral, then the thread dropping the last ref
571  * count will destroy the entry.
572  */
573 #define SOCKPARAMS_DEC_REF(sp) {                                        \
574         ASSERT((sp) != NULL);                                           \
575         DTRACE_PROBE1(sockparams__dec__ref, struct sockparams *, (sp)); \
576         mutex_enter(&(sp)->sp_lock);                                    \
577         ASSERT((sp)->sp_refcnt > 0);                                    \
578         if ((sp)->sp_refcnt == 1) {                                     \
579                 if ((sp)->sp_flags & SOCKPARAMS_EPHEMERAL) {            \
580                         mutex_exit(&(sp)->sp_lock);                     \
581                         sockparams_ephemeral_drop_last_ref((sp));       \
582                 } else {                                                \
583                         (sp)->sp_refcnt--;                              \
584                         if ((sp)->sp_smod_info != NULL) {               \
585                                 SMOD_DEC_REF((sp)->sp_smod_info,        \
586                                     (sp)->sp_smod_name);                \
587                         }                                               \
588                         (sp)->sp_smod_info = NULL;                      \
589                         mutex_exit(&(sp)->sp_lock);                     \
```



```
590                 }                                                       \
591         } else {                                                        \
592                 (sp)->sp_refcnt--;                                      \
593                 mutex_exit(&(sp)->sp_lock);                             \
594         }                                                               \
595 }

597 /*
598  * Used to traverse the list of AF_UNIX sockets to construct the kstat
599  * for netstat(1m).
600  */
601 struct socklist {
602         kmutex_t        sl_lock;
603         struct sonode   *sl_list;
604 };

606 extern struct socklist socklist;
607 /*
608  * ss_full_waits is the number of times the reader thread
609  * waits when the queue is full and ss_empty_waits is the number
610  * of times the consumer thread waits when the queue is empty.
611  * No locks for these as they are just indicators of whether
612  * disk or network or both is slow or fast.
613  */
614 struct sendfile_stats {
615         uint32_t ss_file_cached;
616         uint32_t ss_file_not_cached;
617         uint32_t ss_full_waits;
618         uint32_t ss_empty_waits;
619         uint32_t ss_file_segmap;
620 };

622 /*
623  * A single sendfile request is represented by snf_req.
624  */
625 typedef struct snf_req {
626         struct snf_req  *sr_next;
627         mblk_t          *sr_mp_head;
628         mblk_t          *sr_mp_tail;
629         kmutex_t        sr_lock;
630         kcondvar_t      sr_cv;
631         uint_t          sr_qlen;
632         int             sr_hiwat;
633         int             sr_lowat;
634         int             sr_operation;
635         struct vnode    *sr_vp;
636         file_t          *sr_fp;
637         ssize_t         sr_maxpsz;
638         u_offset_t      sr_file_off;
639         u_offset_t      sr_file_size;
640 #define SR_READ_DONE    0x80000000
641         int             sr_read_error;
642         int             sr_write_error;
643 } snf_req_t;

645 /* A queue of sendfile requests */
646 struct sendfile_queue {
647         snf_req_t       *snfq_req_head;
648         snf_req_t       *snfq_req_tail;
649         kmutex_t        snfq_lock;
650         kcondvar_t      snfq_cv;
651         int             snfq_svc_threads;       /* # of service threads */
652         int             snfq_idle_cnt;          /* # of idling threads */
653         int             snfq_max_threads;
654         int             snfq_req_cnt;           /* Number of requests */
655 };
```

```
#define READ_OP                 1
#define SNFQ_TIMEOUT            (60 * 5 * hz)   /* 5 minutes */

/* Socket network operations switch */
struct sonodeops {
        int     (*sop_init)(struct sonode *, struct sonode *, cred_t *,
                    int);
        int     (*sop_accept)(struct sonode *, int, cred_t *, struct sonode **);
        int     (*sop_bind)(struct sonode *, struct sockaddr *, socklen_t,
                    int, cred_t *);
        int     (*sop_listen)(struct sonode *, int, cred_t *);
        int     (*sop_connect)(struct sonode *, struct sockaddr *,
                    socklen_t, int, int, cred_t *);
        int     (*sop_recvmsg)(struct sonode *, struct msghdr *,
                    struct uio *, cred_t *);
        int     (*sop_sendmsg)(struct sonode *, struct msghdr *,
                    struct uio *, cred_t *);
        int     (*sop_sendmblk)(struct sonode *, struct msghdr *, int,
                    cred_t *, mblk_t **);
        int     (*sop_getpeername)(struct sonode *, struct sockaddr *,
                    socklen_t *, boolean_t, cred_t *);
        int     (*sop_getsockname)(struct sonode *, struct sockaddr *,
                    socklen_t *, cred_t *);
        int     (*sop_shutdown)(struct sonode *, int, cred_t *);
        int     (*sop_getsockopt)(struct sonode *, int, int, void *,
                    socklen_t *, int, cred_t *);
        int     (*sop_setsockopt)(struct sonode *, int, int, const void *,
                    socklen_t, cred_t *);
        int     (*sop_ioctl)(struct sonode *, int, intptr_t, int,
                    cred_t *, int32_t *);
        int     (*sop_poll)(struct sonode *, short, int, short *,
                    struct pollhead **);
        int     (*sop_close)(struct sonode *, int, cred_t *);
};

#define SOP_INIT(so, flag, cr, flags)   \
        ((so)->so_ops->sop_init((so), (flag), (cr), (flags)))
#define SOP_ACCEPT(so, fflag, cr, nsop) \
        ((so)->so_ops->sop_accept((so), (fflag), (cr), (nsop)))
#define SOP_BIND(so, name, namelen, flags, cr)  \
        ((so)->so_ops->sop_bind((so), (name), (namelen), (flags), (cr)))
#define SOP_LISTEN(so, backlog, cr)     \
        ((so)->so_ops->sop_listen((so), (backlog), (cr)))
#define SOP_CONNECT(so, name, namelen, fflag, flags, cr)        \
        ((so)->so_ops->sop_connect((so), (name), (namelen), (fflag), (flags), \
        (cr)))
#define SOP_RECVMSG(so, msg, uiop, cr)  \
        ((so)->so_ops->sop_recvmsg((so), (msg), (uiop), (cr)))
#define SOP_SENDMSG(so, msg, uiop, cr)  \
        ((so)->so_ops->sop_sendmsg((so), (msg), (uiop), (cr)))
#define SOP_SENDMBLK(so, msg, size, cr, mpp)    \
        ((so)->so_ops->sop_sendmblk((so), (msg), (size), (cr), (mpp)))
#define SOP_GETPEERNAME(so, addr, addrlen, accept, cr)  \
        ((so)->so_ops->sop_getpeername((so), (addr), (addrlen), (accept), (cr)))
#define SOP_GETSOCKNAME(so, addr, addrlen, cr)  \
        ((so)->so_ops->sop_getsockname((so), (addr), (addrlen), (cr)))
#define SOP_SHUTDOWN(so, how, cr)       \
        ((so)->so_ops->sop_shutdown((so), (how), (cr)))
#define SOP_GETSOCKOPT(so, level, optionname, optval, optlenp, flags, cr) \
        ((so)->so_ops->sop_getsockopt((so), (level), (optionname),      \
            (optval), (optlenp), (flags), (cr)))
#define SOP_SETSOCKOPT(so, level, optionname, optval, optlen, cr)       \
        ((so)->so_ops->sop_setsockopt((so), (level), (optionname),      \
            (optval), (optlen), (cr)))
#define SOP_IOCTL(so, cmd, arg, mode, cr, rvalp)        \
```



```
        ((so)->so_ops->sop_ioctl((so), (cmd), (arg), (mode), (cr), (rvalp)))
#define SOP_POLL(so, events, anyyet, reventsp, phpp) \
        ((so)->so_ops->sop_poll((so), (events), (anyyet), (reventsp), (phpp)))
#define SOP_CLOSE(so, flag, cr) \
        ((so)->so_ops->sop_close((so), (flag), (cr)))

#endif /* defined(_KERNEL) || defined(_KMEMUSER) */

#ifdef _KERNEL

#define ISALIGNED_cmsghdr(addr) \
                (((uintptr_t)(addr) & (_CMSG_HDR_ALIGNMENT - 1)) == 0)

#define ROUNDUP_cmsglen(len) \
        (((len) + _CMSG_HDR_ALIGNMENT - 1) & ~(_CMSG_HDR_ALIGNMENT - 1))

#define IS_NON_STREAM_SOCK(vp) \
        ((vp)->v_type == VSOCK && (vp)->v_stream == NULL)
/*
 * Macros that operate on struct cmsghdr.
 * Used in parsing msg_control.
 * The CMSG_VALID macro does not assume that the last option buffer is padded.
 */
#define CMSG_NEXT(cmsg)                                                 \
        (struct cmsghdr *)((uintptr_t)(cmsg) +                          \
            ROUNDUP_cmsglen((cmsg)->cmsg_len))
#define CMSG_CONTENT(cmsg)      (&((cmsg)[1]))
#define CMSG_CONTENTLEN(cmsg)   ((cmsg)->cmsg_len - sizeof (struct cmsghdr))
#define CMSG_VALID(cmsg, start, end)                                    \
        (ISALIGNED_cmsghdr(cmsg) &&                                     \
        ((uintptr_t)(cmsg) >= (uintptr_t)(start)) &&                    \
        ((uintptr_t)(cmsg) < (uintptr_t)(end)) &&                       \
        ((ssize_t)(cmsg)->cmsg_len >= sizeof (struct cmsghdr)) &&       \
        ((uintptr_t)(cmsg) + (cmsg)->cmsg_len <= (uintptr_t)(end)))

/*
 * Maximum size of any argument that is copied in (addresses, options,
 * access rights). MUST be at least MAXPATHLEN + 3.
 * BSD and SunOS 4.X limited this to MLEN or MCLBYTES.
 */
#define SO_MAXARGSIZE   8192

/*
 * Convert between vnode and sonode
 */
#define VTOSO(vp)       ((struct sonode *)((vp)->v_data))
#define SOTOV(sp)       ((sp)->so_vnode)

/*
 * Internal flags for sobind()
 */
#define _SOBIND_REBIND          0x01    /* Bind to existing local address */
#define _SOBIND_UNSPEC          0x02    /* Bind to unspecified address */
#define _SOBIND_LOCK_HELD       0x04    /* so_excl_lock held by caller */
#define _SOBIND_NOXLATE         0x08    /* No addr translation for AF_UNIX */
#define _SOBIND_XPG4_2          0x10    /* xpg4.2 semantics */
#define _SOBIND_SOCKBSD         0x20    /* BSD semantics */
#define _SOBIND_LISTEN          0x40    /* Make into SS_ACCEPTCONN */
#define _SOBIND_SOCKETPAIR      0x80    /* Internal flag for so_socketpair() */
                                        /* to enable listen with backlog = 1 */

/*
 * Internal flags for sounbind()
 */
#define _SOUNBIND_REBIND        0x01    /* Don't clear fields - will rebind */
```

```
788 /*
789  * Internal flags for soconnect()
790  */
791 #define _SOCONNECT_NOXLATE      0x01    /* No addr translation for AF_UNIX */
792 #define _SOCONNECT_DID_BIND     0x02    /* Unbind when connect fails */
793 #define _SOCONNECT_XPG4_2       0x04    /* xpg4.2 semantics */

795 /*
796  * Internal flags for sodisconnect()
797  */
798 #define _SODISCONNECT_LOCK_HELD 0x01    /* so_excl_lock held by caller */

800 /*
801  * Internal flags for sotpi_getsockopt().
802  */
803 #define _SOGETSOCKOPT_XPG4_2    0x01    /* xpg4.2 semantics */

805 /*
806  * Internal flags for soallocproto*()
807  */
808 #define _ALLOC_NOSLEEP          0       /* Don't sleep for memory */
809 #define _ALLOC_INTR             1       /* Sleep until interrupt */
810 #define _ALLOC_SLEEP            2       /* Sleep forever */

812 /*
813  * Internal structure for handling AF_UNIX file descriptor passing
814  */
815 struct fdbuf {
816         int             fd_size;        /* In bytes, for kmem_free */
817         int             fd_numfd;       /* Number of elements below */
818         char            *fd_ebuf;       /* Extra buffer to free  */
819         int             fd_ebuflen;
820         frtn_t          fd_frtn;
821         struct file     *fd_fds[1];     /* One or more */
822 };
823 #define FDBUF_HDRSIZE   (sizeof (struct fdbuf) - sizeof (struct file *))

825 /*
826  * Variable that can be patched to set what version of socket socket()
827  * will create.
828  */
829 extern int so_default_version;

831 #ifdef DEBUG
832 /* Turn on extra testing capabilities */
833 #define SOCK_TEST
834 #endif /* DEBUG */

836 #ifdef DEBUG
837 char    *pr_state(uint_t, uint_t);
838 char    *pr_addr(int, struct sockaddr *, t_uscalar_t);
839 int     so_verify_oobstate(struct sonode *);
840 #endif /* DEBUG */

842 /*
843  * DEBUG macros
844  */
845 #if defined(DEBUG)
846 #define SOCK_DEBUG

848 extern int sockdebug;
849 extern int sockprinterr;

851 #define eprint(args)    printf args
852 #define eprintso(so, args) \
853 { if (sockprinterr && ((so)->so_options & SO_DEBUG)) printf args; }
```



```
854 #define eprintline(error)                                       \
855 {                                                               \
856         if (error != EINTR && (sockprinterr || sockdebug > 0))  \
857                 printf("socket error %d: line %d file %s\n",    \
858                         (error), __LINE__, __FILE__);           \
859 }

861 #define eprintsoline(so, error)                                 \
862 { if (sockprinterr && ((so)->so_options & SO_DEBUG))            \
863         printf("socket(%p) error %d: line %d file %s\n",        \
864                 (void *)(so), (error), __LINE__, __FILE__);     \
865 }
866 #define dprint(level, args)     { if (sockdebug > (level)) printf args; }
867 #define dprintso(so, level, args) \
868 { if (sockdebug > (level) && ((so)->so_options & SO_DEBUG)) printf args; }

870 #else /* define(DEBUG) */

872 #define eprint(args)            {}
873 #define eprintso(so, args)      {}
874 #define eprintline(error)       {}
875 #define eprintsoline(so, error) {}
876 #define dprint(level, args)     {}
877 #define dprintso(so, level, args) {}

879 #endif /* defined(DEBUG) */

881 extern struct vfsops                    sock_vfsops;
882 extern struct vnodeops                  *socket_vnodeops;
883 extern const struct fs_operation_def    socket_vnodeops_template[];

885 extern dev_t                            sockdev;

887 extern krwlock_t                        sockconf_lock;

889 /*
890  * sockfs functions
891  */
892 extern int      sock_getmsg(vnode_t *, struct strbuf *, struct strbuf *,
893                         uchar_t *, int *, int, rval_t *);
894 extern int      sock_putmsg(vnode_t *, struct strbuf *, struct strbuf *,
895                         uchar_t, int, int);
896 extern int      sogetvp(char *, vnode_t **, int);
897 extern int      sockinit(int, char *);
898 extern int      solookup(int, int, int, struct sockparams **);
899 extern void     so_lock_single(struct sonode *);
900 extern void     so_unlock_single(struct sonode *, int);
901 extern int      so_lock_read(struct sonode *, int);
902 extern int      so_lock_read_intr(struct sonode *, int);
903 extern void     so_unlock_read(struct sonode *);
904 extern void     *sogetoff(mblk_t *, t_uscalar_t, t_uscalar_t, uint_t);
905 extern void     so_getopt_srcaddr(void *, t_uscalar_t,
906                         void **, t_uscalar_t *);
907 extern int      so_getopt_unix_close(void *, t_uscalar_t);
908 extern void     fdbuf_free(struct fdbuf *);
909 extern mblk_t   *fdbuf_allocmsg(int, struct fdbuf *);
910 extern int      fdbuf_create(void *, int, struct fdbuf **);
911 extern void     so_closefds(void *, t_uscalar_t, int, int);
912 extern int      so_getfdopt(void *, t_uscalar_t, int, void **, int *);
913 t_uscalar_t     so_optlen(void *, t_uscalar_t, int);
914 extern void     so_cmsg2opt(void *, t_uscalar_t, int, mblk_t *);
915 extern t_uscalar_t
916                 so_cmsglen(mblk_t *, void *, t_uscalar_t, int);
917 extern int      so_opt2cmsg(mblk_t *, void *, t_uscalar_t, int,
918                         void *, t_uscalar_t);
919 extern void     soisconnecting(struct sonode *);
```

```
extern void     soisconnected(struct sonode *);
extern void     soisdisconnected(struct sonode *, int);
extern void     socantsendmore(struct sonode *);
extern void     socantrcvmore(struct sonode *);
extern void     soseterror(struct sonode *, int);
extern int      sogeterr(struct sonode *, boolean_t);
extern int      sowaitconnected(struct sonode *, int, int);

extern ssize_t  soreadfile(file_t *, uchar_t *, u_offset_t, int *, size_t);
extern void     *sock_kstat_init(zoneid_t);
extern void     sock_kstat_fini(zoneid_t, void *);
extern struct sonode *getsonode(int, int *, file_t **);
/*
 * Function wrappers (mostly around the sonode switch) for
 * backward compatibility.
 */
extern int      soaccept(struct sonode *, int, struct sonode **);
extern int      sobind(struct sonode *, struct sockaddr *, socklen_t,
                    int, int);
extern int      solisten(struct sonode *, int);
extern int      soconnect(struct sonode *, struct sockaddr *, socklen_t,
                    int, int);
extern int      sorecvmsg(struct sonode *, struct nmsghdr *, struct uio *);
extern int      sosendmsg(struct sonode *, struct nmsghdr *, struct uio *);
extern int      soshutdown(struct sonode *, int);
extern int      sogetsockopt(struct sonode *, int, int, void *, socklen_t *,
                    int);
extern int      sosetsockopt(struct sonode *, int, int, const void *,
                    t_uscalar_t);

extern struct sonode    *socreate(struct sockparams *, int, int, int, int,
                        int *);

extern int      so_copyin(const void *, void *, size_t, int);
extern int      so_copyout(const void *, void *, size_t, int);

#endif

/*
 * Internal structure for obtaining sonode information from the socklist.
 * These types match those corresponding in the sonode structure.
 * This is not a published interface, and may change at any time. It is
 * used for passing information bak up to the kstat consumers. By converting
 * kernel addresses to strings, we should be able to pass information from
 * the kernel to userland regardless of n-bit kernel we are using.
 * This is not a published interface, and may change at any time.
 */

#define ADRSTRLEN (2 * sizeof (uint64_t) + 1)

#endif /* ! codereview */
struct sockinfo {
        uint_t          si_size;                /* real length of this struct */
        short           si_family;
        short           si_type;
        ushort_t        si_flag;
        uint_t          si_state;
        uint_t          si_ux_laddr_sou_magic;
        uint_t          si_ux_faddr_sou_magic;
        t_scalar_t      si_serv_type;
        t_uscalar_t     si_laddr_soa_len;
        t_uscalar_t     si_faddr_soa_len;
        uint16_t        si_laddr_family;
        uint16_t        si_faddr_family;
        char            si_laddr_sun_path[MAXPATHLEN + 1]; /* NULL terminated */
        char            si_faddr_sun_path[MAXPATHLEN + 1];
```



```
        boolean_t       si_faddr_noxlate;
        zoneid_t        si_szoneid;
        char            si_son_straddr[ADRSTRLEN];
        char            si_lvn_straddr[ADRSTRLEN];
        char            si_fvn_straddr[ADRSTRLEN];
        uint_t          si_pn_cnt;
        pid_t           si_pids[1];
#endif /* ! codereview */
};

/*
 * Subcodes for sockconf() system call
 */
#define SOCKCONFIG_ADD_SOCK             0
#define SOCKCONFIG_REMOVE_SOCK          1
#define SOCKCONFIG_ADD_FILTER           2
#define SOCKCONFIG_REMOVE_FILTER        3
#define SOCKCONFIG_GET_SOCKTABLE        4

/*
 * Data structures for configuring socket filters.
 */

/*
 * Placement hint for automatic filters
 */
typedef enum {
        SOF_HINT_NONE,
        SOF_HINT_TOP,
        SOF_HINT_BOTTOM,
        SOF_HINT_BEFORE,
        SOF_HINT_AFTER
} sof_hint_t;

/*
 * Socket tuple. Used by sockconfig_filter_props to list socket
 * types of interest.
 */
typedef struct sof_socktuple {
        int     sofst_family;
        int     sofst_type;
        int     sofst_protocol;
} sof_socktuple_t;

/*
 * Socket filter properties used by sockconfig() system call.
 */
struct sockconfig_filter_props {
        char            *sfp_modname;
        boolean_t       sfp_autoattach;
        sof_hint_t      sfp_hint;
        char            *sfp_hintarg;
        uint_t          sfp_socktuple_cnt;
        sof_socktuple_t *sfp_socktuple;
};

/*
 * Data structures for the in-kernel socket configuration table.
 */
typedef struct sockconfig_socktable_entry {
        int             se_family;
        int             se_type;
        int             se_protocol;
        int             se_refcnt;
        int             se_flags;
        char            se_modname[MODMAXNAMELEN];
```

```
1051         char            se_strdev[MAXPATHLEN];
1052 } sockconfig_socktable_entry_t;

1054 typedef struct sockconfig_socktable {
1055         uint_t          num_of_entries;
1056         sockconfig_socktable_entry_t *st_entries;
1057 } sockconfig_socktable_t;

1059 #ifdef  _SYSCALL32

1061 typedef struct sof_socktuple32 {
1062         int32_t sofst_family;
1063         int32_t sofst_type;
1064         int32_t sofst_protocol;
1065 } sof_socktuple32_t;

1067 struct sockconfig_filter_props32 {
1068         caddr32_t       sfp_modname;
1069         boolean_t       sfp_autoattach;
1070         sof_hint_t      sfp_hint;
1071         caddr32_t       sfp_hintarg;
1072         uint32_t        sfp_socktuple_cnt;
1073         caddr32_t       sfp_socktuple;
1074 };

1076 typedef struct sockconfig_socktable32 {
1077         uint_t          num_of_entries;
1078         caddr32_t       st_entries;
1079 } sockconfig_socktable32_t;

1081 #endif  /* _SYSCALL32 */

1083 #define SOCKMOD_PATH    "socketmod"     /* dir where sockmods are stored */

1085 #ifdef  __cplusplus
1086 }
1087 #endif

1089 #endif  /* _SYS_SOCKETVAR_H */
```

```
**********************************************************
   48786 Mon Aug 17 21:08:10 2015
new/usr/src/uts/common/sys/strsubr.h
XXXX adding PID information to netstat output
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_STRSUBR_H
#define _SYS_STRSUBR_H

/*
 * WARNING:
 * Everything in this file is private, belonging to the
 * STREAMS subsystem.  The only guarantee made about the
 * contents of this file is that if you include it, your
 * code will not port to the next release.
 */
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/kstat.h>
#include <sys/uio.h>
#include <sys/proc.h>
#include <sys/netstack.h>
#include <sys/modhash.h>
#include <sys/pidnode.h>
#endif /* ! codereview */

#ifdef __cplusplus
extern "C" {
#endif

/*
 * In general, the STREAMS locks are disjoint; they are only held
 * locally, and not simultaneously by a thread.  However, module
 * code, including at the stream head, requires some locks to be
 * acquired in order for its safety.
 *      1. Stream level claim.  This prevents the value of q_next
 *              from changing while module code is executing.
 *      2. Queue level claim.  This prevents the value of q_ptr
```



```
 *              from changing while put or service code is executing.
 *              In addition, it provides for queue single-threading
 *              for QPAIR and PERQ MT-safe modules.
 *      3. Stream head lock.  May be held by the stream head module
 *              to implement a read/write/open/close monitor.
 *         Note: that the only types of twisted stream supported are
 *         the pipe and transports which have read and write service
 *         procedures on both sides of the twist.
 *      4. Queue lock.  May be acquired by utility routines on
 *              behalf of a module.
 */

/*
 * In general, sd_lock protects the consistency of the stdata
 * structure.  Additionally, it is used with sd_monitor
 * to implement an open/close monitor.  In particular, it protects
 * the following fields:
 *      sd_iocblk
 *      sd_flag
 *      sd_copyflag
 *      sd_iocid
 *      sd_iocwait
 *      sd_sidp
 *      sd_pgidp
 *      sd_wroff
 *      sd_tail
 *      sd_rerror
 *      sd_werror
 *      sd_pushcnt
 *      sd_sigflags
 *      sd_siglist
 *      sd_pollist
 *      sd_mark
 *      sd_closetime
 *      sd_wakeq
 *      sd_maxblk
 *
 * The following fields are modified only by the allocator, which
 * has exclusive access to them at that time:
 *      sd_wrq
 *      sd_strtab
 *
 * The following field is protected by the overlying file system
 * code, guaranteeing single-threading of opens:
 *      sd_vnode
 *
 * Stream-level locks should be acquired before any queue-level locks
 *      are acquired.
 *
 * The stream head write queue lock(sd_wrq) is used to protect the
 * fields qn_maxpsz and qn_minpsz because freezestr() which is
 * necessary for strqset() only gets the queue lock.
 */

/*
 * Function types for the parameterized stream head.
 * The msgfunc_t takes the parameters:
 *      msgfunc(vnode_t *vp, mblk_t *mp, strwakeup_t *wakeups,
 *              strsigset_t *firstmsgsigs, strsigset_t *allmsgsigs,
 *              strpollset_t *pollwakeups);
 * It returns an optional message to be processed by the stream head.
 *
 * The parameters for errfunc_t are:
 *      errfunc(vnode *vp, int ispeek, int *clearerr);
 * It returns an errno and zero if there was no pending error.
 */
```

```
typedef uint_t  strwakeup_t;
typedef uint_t  strsigset_t;
typedef short   strpollset_t;
typedef uintptr_t callbparams_id_t;
typedef mblk_t  *(*msgfunc_t)(vnode_t *, mblk_t *, strwakeup_t *,
                        strsigset_t *, strsigset_t *, strpollset_t *);
typedef int     (*errfunc_t)(vnode_t *, int, int *);

/*
 * Per stream sd_lock in putnext may be replaced by per cpu stream_putlocks
 * each living in a separate cache line. putnext/canputnext grabs only one of
 * stream_putlocks while strlock() (called on behalf of insertq()/removeq())
 * acquires all stream_putlocks. Normally stream_putlocks are only employed
 * for highly contended streams that have SQ_CIPUT queues in the critical path
 * (e.g. NFS/UDP stream).
 *
 * stream_putlocks are dynamically assigned to stdata structure through
 * sd_ciputctrl pointer possibly when a stream is already in use. Since
 * strlock() uses stream_putlocks only under sd_lock acquiring sd_lock when
 * assigning stream_putlocks to the stream ensures synchronization with
 * strlock().
 *
 * For lock ordering purposes stream_putlocks are treated as the extension of
 * sd_lock and are always grabbed right after grabbing sd_lock and released
 * right before releasing sd_lock except putnext/canputnext where only one of
 * stream_putlocks locks is used and where it is the first lock to grab.
 */

typedef struct ciputctrl_str {
        union _ciput_un {
                uchar_t pad[64];
                struct _ciput_str {
                        kmutex_t        ciput_lck;
                        ushort_t        ciput_cnt;
                } ciput_str;
        } ciput_un;
} ciputctrl_t;

#define ciputctrl_lock  ciput_un.ciput_str.ciput_lck
#define ciputctrl_count ciput_un.ciput_str.ciput_cnt

/*
 * Header for a stream: interface to rest of system.
 *
 * NOTE: While this is a consolidation-private structure, some unbundled and
 *       third-party products inappropriately make use of some of the fields.
 *       As such, please take care to not gratuitously change any offsets of
 *       existing members.
 */
typedef struct stdata {
        struct queue    *sd_wrq;        /* write queue */
        struct msgb     *sd_iocblk;     /* return block for ioctl */
        struct vnode    *sd_vnode;      /* pointer to associated vnode */
        struct streamtab *sd_strtab;    /* pointer to streamtab for stream */
        uint_t          sd_flag;        /* state/flags */
        uint_t          sd_iocid;       /* ioctl id */
        struct pid      *sd_sidp;       /* controlling session info */
        struct pid      *sd_pgidp;      /* controlling process group info */
        ushort_t        sd_tail;        /* reserved space in written mblks */
        ushort_t        sd_wroff;       /* write offset */
        int             sd_rerror;      /* error to return on read ops */
        int             sd_werror;      /* error to return on write ops */
        int             sd_pushcnt;     /* number of pushes done on stream */
        int             sd_sigflags;    /* logical OR of all siglist events */
        struct strsig   *sd_siglist;    /* pid linked list to rcv SIGPOLL sig */
        struct pollhead sd_pollist;     /* list of all pollers to wake up */
```



```
        struct msgb     *sd_mark;       /* "marked" message on read queue */
        clock_t         sd_closetime;   /* time to wait to drain q in close */
        kmutex_t        sd_lock;        /* protect head consistency */
        kcondvar_t      sd_monitor;     /* open/close/push/pop monitor */
        kcondvar_t      sd_iocmonitor;  /* ioctl single-threading */
        kcondvar_t      sd_refmonitor;  /* sd_refcnt monitor */
        ssize_t         sd_qn_minpsz;   /* These two fields are a performance */
        ssize_t         sd_qn_maxpsz;   /* enhancements, cache the values in */
                                        /* the stream head so we don't have */
                                        /* to ask the module below the stream */
                                        /* head to get this information. */
        struct stdata   *sd_mate;       /* pointer to twisted stream mate */
        kthread_id_t    sd_freezer;     /* thread that froze stream */
        kmutex_t        sd_reflock;     /* Protects sd_refcnt */
        int             sd_refcnt;      /* number of claimstr */
        uint_t          sd_wakeq;       /* strwakeq()'s copy of sd_flag */
        struct queue    *sd_struiordq;  /* sync barrier struio() read queue */
        struct queue    *sd_struiowrq;  /* sync barrier struio() write queue */
        char            sd_struiodnak;  /* defer NAK of M_IOCTL by rput() */
        struct msgb     *sd_struionak;  /* pointer M_IOCTL mblk(s) to NAK */
        caddr_t         sd_t_audit_data; /* For audit purposes only */
        ssize_t         sd_maxblk;      /* maximum message block size */
        uint_t          sd_rput_opt;    /* options/flags for strrput */
        uint_t          sd_wput_opt;    /* options/flags for write/putmsg */
        uint_t          sd_read_opt;    /* options/flags for strread */
        msgfunc_t       sd_rprotofunc;  /* rput M_*PROTO routine */
        msgfunc_t       sd_rputdatafunc; /* read M_DATA routine */
        msgfunc_t       sd_rmiscfunc;   /* rput routine (non-data/proto) */
        msgfunc_t       sd_wputdatafunc; /* wput M_DATA routine */
        errfunc_t       sd_rderrfunc;   /* read side error callback */
        errfunc_t       sd_wrerrfunc;   /* write side error callback */
        /*
         * support for low contention concurrent putnext.
         */
        ciputctrl_t     *sd_ciputctrl;
        uint_t          sd_nciputctrl;

        int             sd_anchor;      /* position of anchor in stream */
        /*
         * Service scheduling at the stream head.
         */
        kmutex_t        sd_qlock;
        struct queue    *sd_qhead;      /* Head of queues to be serviced. */
        struct queue    *sd_qtail;      /* Tail of queues to be serviced. */
        void            *sd_servid;     /* Service ID for bckgrnd schedule */
        ushort_t        sd_svcflags;    /* Servicing flags */
        short           sd_nqueues;     /* Number of queues in the list */
        kcondvar_t      sd_qcv;         /* Waiters for qhead to become empty */
        kcondvar_t      sd_zcopy_wait;
        uint_t          sd_copyflag;    /* copy-related flags */
        zoneid_t        sd_anchorzone;  /* Allow removal from same zone only */
        struct msgb     *sd_cmdblk;     /* reply from _I_CMD */
        list_t          sd_pid_list;
        kmutex_t        sd_pid_list_lock;
#endif /* ! codereview */
} stdata_t;

/*
 * stdata servicing flags.
 */
#define STRS_WILLSERVICE        0x01
#define STRS_SCHEDULED          0x02

#define STREAM_NEEDSERVICE(stp) ((stp)->sd_qhead != NULL)

/*
```

```
 * stdata flag field defines
 */
#define IOCWAIT         0x00000001      /* Someone is doing an ioctl */
#define RSLEEP          0x00000002      /* Someone wants to read/recv msg */
#define WSLEEP          0x00000004      /* Someone wants to write */
#define STRPRI          0x00000008      /* An M_PCPROTO is at stream head */
#define STRHUP          0x00000010      /* Device has vanished */
#define STWOPEN         0x00000020      /* waiting for 1st open */
#define STPLEX          0x00000040      /* stream is being multiplexed */
#define STRISTTY        0x00000080      /* stream is a terminal */
#define STRGETINPROG    0x00000100      /* (k)strgetmsg is running */
#define IOCWAITNE       0x00000200      /* STR_NOERROR ioctl running */
#define STRDERR         0x00000400      /* fatal read error from M_ERROR */
#define STWRERR         0x00000800      /* fatal write error from M_ERROR */
#define STRDERRNONPERSIST 0x00001000    /* nonpersistent read errors */
#define STWRERRNONPERSIST 0x00002000    /* nonpersistent write errors */
#define STRCLOSE        0x00004000      /* wait for a close to complete */
#define SNDMREAD        0x00008000      /* used for read notification */
#define OLDNDELAY       0x00010000      /* use old TTY semantics for */
                                        /* NDELAY reads and writes */
        /*              0x00020000         unused */
        /*              0x00040000         unused */
#define STRTOSTOP       0x00080000      /* block background writes */
#define STRCMDWAIT      0x00100000      /* someone is doing an _I_CMD */
        /*              0x00200000         unused */
#define STRMOUNT        0x00400000      /* stream is mounted */
#define STRNOTATMARK    0x00800000      /* Not at mark (when empty read q) */
#define STRDELIM        0x01000000      /* generate delimited messages */
#define STRATMARK       0x02000000      /* At mark (due to MSGMARKNEXT) */
#define STZCNOTIFY      0x04000000      /* wait for zerocopy mblk to be acked */
#define STRPLUMB        0x08000000      /* push/pop pending */
#define STREOF          0x10000000      /* End-of-file indication */
#define STREOPENFAIL    0x20000000      /* indicates if re-open has failed */
#define STRMATE         0x40000000      /* this stream is a mate */
#define STRHASLINKS     0x80000000      /* I_LINKs under this stream */

/*
 * Copy-related flags (sd_copyflag), set by SO_COPYOPT.
 */
#define STZCVMSAFE      0x00000001      /* safe to borrow file (segmapped) */
                                        /* pages instead of bcopy */
#define STZCVMUNSAFE    0x00000002      /* unsafe to borrow file pages */
#define STRCOPYCACHED   0x00000004      /* copy should NOT bypass cache */

/*
 * Options and flags for strrput (sd_rput_opt)
 */
#define SR_POLLIN       0x00000001      /* pollwakeup needed for band0 data */
#define SR_SIGALLDATA   0x00000002      /* Send SIGPOLL for all M_DATA */
#define SR_CONSOL_DATA  0x00000004      /* Consolidate M_DATA onto q_last */
#define SR_IGN_ZEROLEN  0x00000008      /* Ignore zero-length M_DATA */

/*
 * Options and flags for strwrite/strputmsg (sd_wput_opt)
 */
#define SW_SIGPIPE      0x00000001      /* Send SIGPIPE for write error */
#define SW_RECHECK_ERR  0x00000002      /* Recheck errors in strwrite loop */
#define SW_SNDZERO      0x00000004      /* send 0-length msg down pipe/FIFO */

/*
 * Options and flags for strread (sd_read_opt)
 */
#define RD_MSGDIS       0x00000001      /* read msg discard */
#define RD_MSGNODIS     0x00000002      /* read msg no discard */
#define RD_PROTDAT      0x00000004      /* read M_[PC]PROTO contents as data */
#define RD_PROTDIS      0x00000008      /* discard M_[PC]PROTO blocks and */
```



```
                                        /* retain data blocks */
/*
 * Flags parameter for strsetrputhooks() and strsetwputhooks().
 * These flags define the interface for setting the above internal
 * flags in sd_rput_opt and sd_wput_opt.
 */
#define SH_CONSOL_DATA  0x00000001      /* Consolidate M_DATA onto q_last */
#define SH_SIGALLDATA   0x00000002      /* Send SIGPOLL for all M_DATA */
#define SH_IGN_ZEROLEN  0x00000004      /* Drop zero-length M_DATA */

#define SH_SIGPIPE      0x00000100      /* Send SIGPIPE for write error */
#define SH_RECHECK_ERR  0x00000200      /* Recheck errors in strwrite loop */

/*
 * Each queue points to a sync queue (the inner perimeter) which keeps
 * track of the number of threads that are inside a given queue (sq_count)
 * and also is used to implement the asynchronous putnext
 * (by queuing messages if the queue can not be entered.)
 *
 * Messages are queued on sq_head/sq_tail including deferred qwriter(INNER)
 * messages. The sq_head/sq_tail list is a singly-linked list with
 * b_queue recording the queue and b_prev recording the function to
 * be called (either the put procedure or a qwriter callback function.)
 *
 * The sq_count counter tracks the number of threads that are
 * executing inside the perimeter or (in the case of outer perimeters)
 * have some work queued for them relating to the perimeter. The sq_rmqcount
 * counter tracks the subset which are in removeq() (usually invoked from
 * qprocsoff(9F)).
 *
 * In addition a module writer can declare that the module has an outer
 * perimeter (by setting D_MTOUTPERIM) in which case all inner perimeter
 * syncq's for the module point (through sq_outer) to an outer perimeter
 * syncq. The outer perimeter consists of the doubly linked list (sq_onext and
 * sq_oprev) linking all the inner perimeter syncq's with out outer perimeter
 * syncq. This is used to implement qwriter(OUTER) (an asynchronous way of
 * getting exclusive access at the outer perimeter) and outer_enter/exit
 * which are used by the framework to acquire exclusive access to the outer
 * perimeter during open and close of modules that have set D_MTOUTPERIM.
 *
 * In the inner perimeter case sq_save is available for use by machine
 * dependent code. sq_head/sq_tail are used to queue deferred messages on
 * the inner perimeter syncqs and to queue become_writer requests on the
 * outer perimeter syncqs.
 *
 * Note: machine dependent optimized versions of putnext may depend
 * on the order of sq_flags and sq_count (so that they can e.g.
 * read these two fields in a single load instruction.)
 *
 * Per perimeter SQLOCK/sq_count in putnext/put may be replaced by per cpu
 * sq_putlocks/sq_putcounts each living in a separate cache line. Obviously
 * sq_putlock[x] protects sq_putcount[x]. putnext/put routine will grab only 1
 * of sq_putlocks and update only 1 of sq_putcounts. strlock() and many
 * other routines in strsubr.c and ddi.c will grab all sq_putlocks (as well as
 * SQLOCK) and figure out the count value as the sum of sq_count and all of
 * sq_putcounts. The idea is to make critical fast path -- putnext -- much
 * faster at the expense of much less often used slower path like
 * strlock(). One known case where entersq/strlock is executed pretty often is
 * SpecWeb but since IP is SQ_CIOC and socket TCP/IP stream is nextless
 * there's no need to grab multiple sq_putlocks and look at sq_putcounts. See
 * strsubr.c for more comments.
 *
 * Note regular SQLOCK and sq_count are still used in many routines
 * (e.g. entersq(), rwnext()) in the same way as before sq_putlocks were
 * introduced.
 *
```

```
392  * To understand when all sq_putlocks need to be held and all sq_putcounts
393  * need to be added up one needs to look closely at putnext code. Basically if
394  * a routine like e.g. wait_syncq() needs to be sure that perimeter is empty
395  * all sq_putlocks/sq_putcounts need to be held/added up. On the other hand
396  * there's no need to hold all sq_putlocks and count all sq_putcounts in
397  * routines like leavesq()/dropsq() and etc. since the are usually exit
398  * counterparts of entersq/outer_enter() and etc. which have already either
399  * prevented put entry poins from executing or did not care about put
400  * entrypoints. entersq() doesn't need to care about sq_putlocks/sq_putcounts
401  * if the entry point has a shared access since put has the highest degree of
402  * concurrency and such entersq() does not intend to block out put
403  * entrypoints.
404  *
405  * Before sq_putcounts were introduced the standard way to wait for perimeter
406  * to become empty was:
407  *
408  *      mutex_enter(SQLOCK(sq));
409  *      while (sq->sq_count > 0) {
410  *              sq->sq_flags |= SQ_WANTWAKEUP;
411  *              cv_wait(&sq->sq_wait, SQLOCK(sq));
412  *      }
413  *      mutex_exit(SQLOCK(sq));
414  *
415  * The new way is:
416  *
417  *      mutex_enter(SQLOCK(sq));
418  *      count = sq->sq_count;
419  *      SQ_PUTLOCKS_ENTER(sq);
420  *      SUM_SQ_PUTCOUNTS(sq, count);
421  *      while (count != 0) {
422  *              sq->sq_flags |= SQ_WANTWAKEUP;
423  *              SQ_PUTLOCKS_EXIT(sq);
424  *              cv_wait(&sq->sq_wait, SQLOCK(sq));
425  *              count = sq->sq_count;
426  *              SQ_PUTLOCKS_ENTER(sq);
427  *              SUM_SQ_PUTCOUNTS(sq, count);
428  *      }
429  *      SQ_PUTLOCKS_EXIT(sq);
430  *      mutex_exit(SQLOCK(sq));
431  *
432  * Note that SQ_WANTWAKEUP is set before dropping SQ_PUTLOCKS. This makes sure
433  * putnext won't skip a wakeup.
434  *
435  * sq_putlocks are treated as the extension of SQLOCK for lock ordering
436  * purposes and are always grabbed right after grabbing SQLOCK and released
437  * right before releasing SQLOCK. This also allows dynamic creation of
438  * sq_putlocks while holding SQLOCK (by making sq_ciputctrl non null even when
439  * the stream is already in use). Only in putnext one of sq_putlocks
440  * is grabbed instead of SQLOCK. putnext return path remembers what counter it
441  * incremented and decrements the right counter on its way out.
442  */

443 struct syncq {
444         kmutex_t        sq_lock;        /* atomic access to syncq */
445         uint16_t        sq_count;       /* # threads inside */
446         uint16_t        sq_flags;       /* state and some type info */
447         /*
448          * Distributed syncq scheduling
449          *  The list of queue's is handled by sq_head and
450          *  sq_tail fields.
451          *
452          *  The list of events is handled by the sq_evhead and sq_evtail
453          *  fields.
454          */
455         queue_t         *sq_head;       /* queue of deferred messages */
456         queue_t         *sq_tail;       /* queue of deferred messages */
```



```
458         mblk_t          *sq_evhead;     /* Event message on the syncq */
459         mblk_t          *sq_evtail;
460         uint_t          sq_nqueues;     /* # of queues on this sq */
461         /*
462          * Concurrency and condition variables
463          */
464         uint16_t        sq_type;        /* type (concurrency) of syncq */
465         uint16_t        sq_rmqcount;    /* # threads inside removeq() */
466         kcondvar_t      sq_wait;        /* block on this sync queue */
467         kcondvar_t      sq_exitwait;    /* waiting for thread to leave the */
468                                         /* inner perimeter */
469         /*
470          * Handling synchronous callbacks such as qtimeout and qbufcall
471          */
472         ushort_t        sq_callbflags;  /* flags for callback synchronization */
473         callbparams_id_t sq_cancelid;   /* id of callback being cancelled */
474         struct callbparams *sq_callbpend;       /* Pending callbacks */

476         /*
477          * Links forming an outer perimeter from one outer syncq and
478          * a set of inner sync queues.
479          */
480         struct syncq    *sq_outer;      /* Pointer to outer perimeter */
481         struct syncq    *sq_onext;      /* Linked list of syncq's making */
482         struct syncq    *sq_oprev;      /* up the outer perimeter. */
483         /*
484          * support for low contention concurrent putnext.
485          */
486         ciputctrl_t     *sq_ciputctrl;
487         uint_t          sq_nciputctrl;
488         /*
489          * Counter for the number of threads wanting to become exclusive.
490          */
491         uint_t          sq_needexcl;
492         /*
493          * These two fields are used for scheduling a syncq for
494          * background processing. The sq_svcflag is protected by
495          * SQLOCK lock.
496          */
497         struct syncq    *sq_next;       /* for syncq scheduling */
498         void *          sq_servid;
499         uint_t          sq_servcount;   /* # pending background threads */
500         uint_t          sq_svcflags;    /* Scheduling flags     */
501         clock_t         sq_tstamp;      /* Time when was enabled */
502         /*
503          * Maximum priority of the queues on this syncq.
504          */
505         pri_t           sq_pri;
506 };
507 typedef struct syncq syncq_t;

509 /*
510  * sync queue scheduling flags (for sq_svcflags).
511  */
512 #define SQ_SERVICE      0x1             /* being serviced */
513 #define SQ_BGTHREAD     0x2             /* awaiting service by bg thread */
514 #define SQ_DISABLED     0x4             /* don't put syncq in service list */

516 /*
517  * FASTPUT bit in sd_count/putcount.
518  */
519 #define SQ_FASTPUT      0x8000
520 #define SQ_FASTMASK     0x7FFF

522 /*
523  * sync queue state flags
```

```
524  */
525 #define SQ_EXCL         0x0001          /* exclusive access to inner */
526                                         /*      perimeter */
527 #define SQ_BLOCKED      0x0002          /* qprocsoff */
528 #define SQ_FROZEN       0x0004          /* freezestr */
529 #define SQ_WRITER       0x0008          /* qwriter(OUTER) pending or running */
530 #define SQ_MESSAGES     0x0010          /* messages on syncq */
531 #define SQ_WANTWAKEUP   0x0020          /* do cv_broadcast on sq_wait */
532 #define SQ_WANTEXWAKEUP 0x0040          /* do cv_broadcast on sq_exitwait */
533 #define SQ_EVENTS       0x0080          /* Events pending */
534 #define SQ_QUEUED       (SQ_MESSAGES | SQ_EVENTS)
535 #define SQ_FLAGMASK     0x00FF

537 /*
538  * Test a queue to see if inner perimeter is exclusive.
539  */
540 #define PERIM_EXCL(q)   ((q)->q_syncq->sq_flags & SQ_EXCL)

542 /*
543  * If any of these flags are set it is not possible for a thread to
544  * enter a put or service procedure. Instead it must either block
545  * or put the message on the syncq.
546  */
547 #define SQ_GOAWAY       (SQ_EXCL|SQ_BLOCKED|SQ_FROZEN|SQ_WRITER|\
548                         SQ_QUEUED)
549 /*
550  * If any of these flags are set it not possible to drain the syncq
551  */
552 #define SQ_STAYAWAY     (SQ_BLOCKED|SQ_FROZEN|SQ_WRITER)

554 /*
555  * Flags to trigger syncq tail processing.
556  */
557 #define SQ_TAIL         (SQ_QUEUED|SQ_WANTWAKEUP|SQ_WANTEXWAKEUP)

559 /*
560  * Syncq types (stored in sq_type)
561  * The SQ_TYPES_IN_FLAGS (ciput) are also stored in sq_flags
562  * for performance reasons. Thus these type values have to be in the low
563  * 16 bits and not conflict with the sq_flags values above.
564  *
565  * Notes:
566  *  - putnext() and put() assume that the put procedures have the highest
567  *    degree of concurrency. Thus if any of the SQ_CI* are set then SQ_CIPUT
568  *    has to be set. This restriction can be lifted by adding code to putnext
569  *    and put that check that sq_count == 0 like entersq does.
570  *  - putnext() and put() does currently not handle !SQ_COPUT
571  *  - In order to implement !SQ_COCB outer_enter has to be fixed so that
572  *    the callback can be cancelled while cv_waiting in outer_enter.
573  *  - If SQ_CISVC needs to be implemented, qprocsoff() needs to wait
574  *    for the currently running services to stop (wait for QINSERVICE
575  *    to go off). disable_svc called from qprcosoff disables only
576  *    services that will be run in future.
577  *
578  * All the SQ_CO flags are set when there is no outer perimeter.
579  */
580 #define SQ_CIPUT        0x0100          /* Concurrent inner put proc */
581 #define SQ_CISVC        0x0200          /* Concurrent inner svc proc */
582 #define SQ_CIOC         0x0400          /* Concurrent inner open/close */
583 #define SQ_CICB         0x0800          /* Concurrent inner callback */
584 #define SQ_COPUT        0x1000          /* Concurrent outer put proc */
585 #define SQ_COSVC        0x2000          /* Concurrent outer svc proc */
586 #define SQ_COOC         0x4000          /* Concurrent outer open/close */
587 #define SQ_COCB         0x8000          /* Concurrent outer callback */

589 /* Types also kept in sq_flags for performance */
```



```
590 #define SQ_TYPES_IN_FLAGS       (SQ_CIPUT)

592 #define SQ_CI           (SQ_CIPUT|SQ_CISVC|SQ_CIOC|SQ_CICB)
593 #define SQ_CO           (SQ_COPUT|SQ_COSVC|SQ_COOC|SQ_COCB)
594 #define SQ_TYPEMASK     (SQ_CI|SQ_CO)

596 /*
597  * Flag combinations passed to entersq and leavesq to specify the type
598  * of entry point.
599  */
600 #define SQ_PUT          (SQ_CIPUT|SQ_COPUT)
601 #define SQ_SVC          (SQ_CISVC|SQ_COSVC)
602 #define SQ_OPENCLOSE    (SQ_CIOC|SQ_COOC)
603 #define SQ_CALLBACK     (SQ_CICB|SQ_COCB)

605 /*
606  * Other syncq types which are not copied into flags.
607  */
608 #define SQ_PERMOD       0x01            /* Syncq is PERMOD */

610 /*
611  * Asynchronous callback qun*** flag.
612  * The mechanism these flags are used in is one where callbacks enter
613  * the perimeter thanks to framework support. To use this mechanism
614  * the q* and qun* flavors of the callback routines must be used.
615  * e.g. qtimeout and quntimeout. The synchronization provided by the flags
616  * avoids deadlocks between blocking qun* routines and the perimeter
617  * lock.
618  */
619 #define SQ_CALLB_BYPASSED       0x01            /* bypassed callback fn */

621 /*
622  * Cancel callback mask.
623  * The mask expands as the number of cancelable callback types grows
624  * Note - separate callback flag because different callbacks have
625  * overlapping id space.
626  */
627 #define SQ_CALLB_CANCEL_MASK    (SQ_CANCEL_TOUT|SQ_CANCEL_BUFCALL)

629 #define SQ_CANCEL_TOUT          0x02            /* cancel timeout request */
630 #define SQ_CANCEL_BUFCALL       0x04            /* cancel bufcall request */

632 typedef struct callbparams {
633         syncq_t         *cbp_sq;
634         void            (*cbp_func)(void *);
635         void            *cbp_arg;
636         callbparams_id_t cbp_id;
637         uint_t          cbp_flags;
638         struct callbparams *cbp_next;
639         size_t          cbp_size;
640 } callbparams_t;

642 typedef struct strbufcall {
643         void            (*bc_func)(void *);
644         void            *bc_arg;
645         size_t          bc_size;
646         bufcall_id_t    bc_id;
647         struct strbufcall *bc_next;
648         kthread_id_t    bc_executor;
649 } strbufcall_t;

651 /*
652  * Structure of list of processes to be sent SIGPOLL/SIGURG signal
653  * on request.  The valid S_* events are defined in stropts.h.
654  */
655 typedef struct strsig {
```

```
656         struct pid      *ss_pidp;       /* pid/pgrp pointer */
657         pid_t           ss_pid;         /* positive pid, negative pgrp */
658         int             ss_events;      /* S_* events */
659         struct strsig   *ss_next;
660 } strsig_t;

662 /*
663  * bufcall list
664  */
665 struct bclist {
666         strbufcall_t    *bc_head;
667         strbufcall_t    *bc_tail;
668 };

670 /*
671  * Structure used to track mux links and unlinks.
672  */
673 struct mux_node {
674         major_t          mn_imaj;       /* internal major device number */
675         uint16_t         mn_indegree;   /* number of incoming edges */
676         struct mux_node *mn_originp;    /* where we came from during search */
677         struct mux_edge *mn_startp;     /* where search left off in mn_outp */
678         struct mux_edge *mn_outp;       /* list of outgoing edges */
679         uint_t           mn_flags;      /* see below */
680 };

682 /*
683  * Flags for mux_nodes.
684  */
685 #define VISITED 1

687 /*
688  * Edge structure - a list of these is hung off the
689  * mux_node to represent the outgoing edges.
690  */
691 struct mux_edge {
692         struct mux_node *me_nodep;      /* edge leads to this node */
693         struct mux_edge *me_nextp;      /* next edge */
694         int              me_muxid;      /* id of link */
695         dev_t            me_dev;        /* dev_t - used for kernel PUNLINK */
696 };

698 /*
699  * Queue info
700  *
701  * The syncq is included here to reduce memory fragmentation
702  * for kernel memory allocators that only allocate in sizes that are
703  * powers of two. If the kernel memory allocator changes this should
704  * be revisited.
705  */
706 typedef struct queinfo {
707         struct queue    qu_rqueue;      /* read queue - must be first */
708         struct queue    qu_wqueue;      /* write queue - must be second */
709         struct syncq    qu_syncq;       /* syncq - must be third */
710 } queinfo_t;

712 /*
713  * Multiplexed streams info
714  */
715 typedef struct linkinfo {
716         struct linkblk  li_lblk;        /* must be first */
717         struct file     *li_fpdown;     /* file pointer for lower stream */
718         struct linkinfo *li_next;       /* next in list */
719         struct linkinfo *li_prev;       /* previous in list */
720 } linkinfo_t;
```



```
722 /*
723  * List of syncq's used by freeezestr/unfreezestr
724  */
725 typedef struct syncql {
726         struct syncql   *sql_next;
727         syncq_t         *sql_sq;
728 } syncql_t;

730 typedef struct sqlist {
731         syncql_t        *sqlist_head;
732         size_t          sqlist_size;            /* structure size in bytes */
733         size_t          sqlist_index;           /* next free entry in array */
734         syncql_t        sqlist_array[4];        /* 4 or more entries */
735 } sqlist_t;

737 typedef struct perdm {
738         struct perdm            *dm_next;
739         syncq_t                 *dm_sq;
740         struct streamtab        *dm_str;
741         uint_t                  dm_ref;
742 } perdm_t;

744 #define NEED_DM(dmp, qflag) \
745         (dmp == NULL && (qflag & (QPERMOD | QMTOUTPERIM)))

747 /*
748  * fmodsw_impl_t is used within the kernel. fmodsw is used by
749  * the modules/drivers. The information is copied from fmodsw
750  * defined in the module/driver into the fmodsw_impl_t structure
751  * during the module/driver initialization.
752  */
753 typedef struct fmodsw_impl      fmodsw_impl_t;

755 struct fmodsw_impl {
756         fmodsw_impl_t           *f_next;
757         char                    f_name[FMNAMESZ + 1];
758         struct streamtab        *f_str;
759         uint32_t                f_qflag;
760         uint32_t                f_sqtype;
761         perdm_t                 *f_dmp;
762         uint32_t                f_ref;
763         uint32_t                f_hits;
764 };

766 typedef enum {
767         FMODSW_HOLD =   0x00000001,
768         FMODSW_LOAD =   0x00000002
769 } fmodsw_flags_t;

771 typedef struct cdevsw_impl {
772         struct streamtab        *d_str;
773         uint32_t                d_qflag;
774         uint32_t                d_sqtype;
775         perdm_t                 *d_dmp;
776 } cdevsw_impl_t;

778 /*
779  * Enumeration of the types of access that can be requested for a
780  * controlling terminal under job control.
781  */
782 enum jcaccess {
783         JCREAD,                 /* read data on a ctty */
784         JCWRITE,                /* write data to a ctty */
785         JCSETP,                 /* set ctty parameters */
786         JCGETP                  /* get ctty parameters */
787 };
```

```
789 struct str_stack {
790         netstack_t      *ss_netstack;   /* Common netstack */

792         kmutex_t        ss_sad_lock;    /* autopush lock */
793         mod_hash_t      *ss_sad_hash;
794         size_t          ss_sad_hash_nchains;
795         struct saddev   *ss_saddev;     /* sad device array */
796         int             ss_sadcnt;      /* number of sad devices */

798         int             ss_devcnt;      /* number of mux_nodes */
799         struct mux_node *ss_mux_nodes;  /* mux info for cycle checking */
800 };
801 typedef struct str_stack str_stack_t;

803 /*
804  * Finding related queues
805  */
806 #define STREAM(q)       ((q)->q_stream)
807 #define SQ(rq)          ((syncq_t *)((rq) + 2))

809 /*
810  * Get the module/driver name for a queue.  Since some queues don't have
811  * q_info structures (e.g., see log_makeq()), fall back to "?".
812  */
813 #define Q2NAME(q) \
814         (((q)->q_qinfo != NULL && (q)->q_qinfo->qi_minfo->mi_idname != NULL) ? \
815         (q)->q_qinfo->qi_minfo->mi_idname : "?")

817 /*
818  * Locking macros
819  */
820 #define QLOCK(q)        (&(q)->q_lock)
821 #define SQLOCK(sq)      (&(sq)->sq_lock)

823 #define STREAM_PUTLOCKS_ENTER(stp) {                                    \
824                 ASSERT(MUTEX_HELD(&(stp)->sd_lock));                    \
825                 if ((stp)->sd_ciputctrl != NULL) {                      \
826                         int i;                                          \
827                         int nlocks = (stp)->sd_nciputctrl;              \
828                         ciputctrl_t *cip = (stp)->sd_ciputctrl;         \
829                         for (i = 0; i <= nlocks; i++) {                 \
830                                 mutex_enter(&cip[i].ciputctrl_lock);    \
831                         }                                               \
832                 }                                                       \
833         }

835 #define STREAM_PUTLOCKS_EXIT(stp) {                                     \
836                 ASSERT(MUTEX_HELD(&(stp)->sd_lock));                    \
837                 if ((stp)->sd_ciputctrl != NULL) {                      \
838                         int i;                                          \
839                         int nlocks = (stp)->sd_nciputctrl;              \
840                         ciputctrl_t *cip = (stp)->sd_ciputctrl;         \
841                         for (i = 0; i <= nlocks; i++) {                 \
842                                 mutex_exit(&cip[i].ciputctrl_lock);     \
843                         }                                               \
844                 }                                                       \
845         }

847 #define SQ_PUTLOCKS_ENTER(sq) {                                         \
848                 ASSERT(MUTEX_HELD(SQLOCK(sq)));                         \
849                 if ((sq)->sq_ciputctrl != NULL) {                       \
850                         int i;                                          \
851                         int nlocks = (sq)->sq_nciputctrl;               \
852                         ciputctrl_t *cip = (sq)->sq_ciputctrl;          \
853                         ASSERT((sq)->sq_type & SQ_CIPUT);               \
```



```
854                         for (i = 0; i <= nlocks; i++) {                 \
855                                 mutex_enter(&cip[i].ciputctrl_lock);    \
856                         }                                               \
857                 }                                                       \
858         }

860 #define SQ_PUTLOCKS_EXIT(sq) {                                          \
861                 ASSERT(MUTEX_HELD(SQLOCK(sq)));                         \
862                 if ((sq)->sq_ciputctrl != NULL) {                       \
863                         int i;                                          \
864                         int nlocks = (sq)->sq_nciputctrl;               \
865                         ciputctrl_t *cip = (sq)->sq_ciputctrl;          \
866                         ASSERT((sq)->sq_type & SQ_CIPUT);               \
867                         for (i = 0; i <= nlocks; i++) {                 \
868                                 mutex_exit(&cip[i].ciputctrl_lock);     \
869                         }                                               \
870                 }                                                       \
871         }

873 #define SQ_PUTCOUNT_SETFAST(sq) {                                       \
874                 ASSERT(MUTEX_HELD(SQLOCK(sq)));                         \
875                 if ((sq)->sq_ciputctrl != NULL) {                       \
876                         int i;                                          \
877                         int nlocks = (sq)->sq_nciputctrl;               \
878                         ciputctrl_t *cip = (sq)->sq_ciputctrl;          \
879                         ASSERT((sq)->sq_type & SQ_CIPUT);               \
880                         for (i = 0; i <= nlocks; i++) {                 \
881                                 mutex_enter(&cip[i].ciputctrl_lock);    \
882                                 cip[i].ciputctrl_count |= SQ_FASTPUT;   \
883                                 mutex_exit(&cip[i].ciputctrl_lock);     \
884                         }                                               \
885                 }                                                       \
886         }

888 #define SQ_PUTCOUNT_CLRFAST(sq) {                                       \
889                 ASSERT(MUTEX_HELD(SQLOCK(sq)));                         \
890                 if ((sq)->sq_ciputctrl != NULL) {                       \
891                         int i;                                          \
892                         int nlocks = (sq)->sq_nciputctrl;               \
893                         ciputctrl_t *cip = (sq)->sq_ciputctrl;          \
894                         ASSERT((sq)->sq_type & SQ_CIPUT);               \
895                         for (i = 0; i <= nlocks; i++) {                 \
896                                 mutex_enter(&cip[i].ciputctrl_lock);    \
897                                 cip[i].ciputctrl_count &= ~SQ_FASTPUT;  \
898                                 mutex_exit(&cip[i].ciputctrl_lock);     \
899                         }                                               \
900                 }                                                       \
901         }


904 #ifdef  DEBUG

906 #define SQ_PUTLOCKS_HELD(sq) {                                          \
907                 ASSERT(MUTEX_HELD(SQLOCK(sq)));                         \
908                 if ((sq)->sq_ciputctrl != NULL) {                       \
909                         int i;                                          \
910                         int nlocks = (sq)->sq_nciputctrl;               \
911                         ciputctrl_t *cip = (sq)->sq_ciputctrl;          \
912                         ASSERT((sq)->sq_type & SQ_CIPUT);               \
913                         for (i = 0; i <= nlocks; i++) {                 \
914                                 ASSERT(MUTEX_HELD(&cip[i].ciputctrl_lock));     \
915                         }                                               \
916                 }                                                       \
917         }

919 #define SUMCHECK_SQ_PUTCOUNTS(sq, countcheck) {                         \
```

```
920                 if ((sq)->sq_ciputctrl != NULL) {                               \
921                         int i;                                                  \
922                         uint_t count = 0;                                       \
923                         int ncounts = (sq)->sq_nciputctrl;                      \
924                         ASSERT((sq)->sq_type & SQ_CIPUT);                       \
925                         for (i = 0; i <= ncounts; i++) {                        \
926                                 count +=                                        \
927                                     (((sq)->sq_ciputctrl[i].ciputctrl_count) & \
928                                     SQ_FASTMASK);                               \
929                         }                                                       \
930                         ASSERT(count == (countcheck));                          \
931                 }                                                               \
932         }

934 #define SUMCHECK_CIPUTCTRL_COUNTS(ciput, nciput, countcheck) {                  \
935                 int i;                                                          \
936                 uint_t count = 0;                                               \
937                 ASSERT((ciput) != NULL);                                        \
938                 for (i = 0; i <= (nciput); i++) {                               \
939                         count += (((ciput)[i].ciputctrl_count) &                \
940                             SQ_FASTMASK);                                       \
941                 }                                                               \
942                 ASSERT(count == (countcheck));                                  \
943         }

945 #else   /* DEBUG */

947 #define SQ_PUTLOCKS_HELD(sq)
948 #define SUMCHECK_SQ_PUTCOUNTS(sq, countcheck)
949 #define SUMCHECK_CIPUTCTRL_COUNTS(sq, nciput, countcheck)

951 #endif  /* DEBUG */

953 #define SUM_SQ_PUTCOUNTS(sq, count) {                                           \
954                 if ((sq)->sq_ciputctrl != NULL) {                               \
955                         int i;                                                  \
956                         int ncounts = (sq)->sq_nciputctrl;                      \
957                         ciputctrl_t *cip = (sq)->sq_ciputctrl;                  \
958                         ASSERT((sq)->sq_type & SQ_CIPUT);                       \
959                         for (i = 0; i <= ncounts; i++) {                        \
960                                 (count) += ((cip[i].ciputctrl_count) &          \
961                                     SQ_FASTMASK);                               \
962                         }                                                       \
963                 }                                                               \
964         }

966 #define CLAIM_QNEXT_LOCK(stp)   mutex_enter(&(stp)->sd_lock)
967 #define RELEASE_QNEXT_LOCK(stp) mutex_exit(&(stp)->sd_lock)

969 /*
970  * syncq message manipulation macros.
971  */
972 /*
973  * Put a message on the queue syncq.
974  * Assumes QLOCK held.
975  */
976 #define SQPUT_MP(qp, mp)                                                \
977         {                                                               \
978                 qp->q_syncqmsgs++;                                      \
979                 if (qp->q_sqhead == NULL) {                             \
980                         qp->q_sqhead = qp->q_sqtail = mp;               \
981                 } else {                                                \
982                         qp->q_sqtail->b_next = mp;                      \
983                         qp->q_sqtail = mp;                              \
984                 }                                                       \
985                 set_qfull(qp);                                          \
```



```
986         }

988 /*
989  * Miscellaneous parameters and flags.
990  */

992 /*
993  * Default timeout in milliseconds for ioctls and close
994  */
995 #define STRTIMOUT 15000

997 /*
998  * Flag values for stream io
999  */
1000 #define WRITEWAIT       0x1     /* waiting for write event */
1001 #define READWAIT        0x2     /* waiting for read event */
1002 #define NOINTR          0x4     /* error is not to be set for signal */
1003 #define GETWAIT         0x8     /* waiting for getmsg event */

1005 /*
1006  * These flags need to be unique for stream io name space
1007  * and copy modes name space.  These flags allow strwaitq
1008  * and strdoioctl to proceed as if signals or errors on the stream
1009  * head have not occurred; i.e. they will be detected by some other
1010  * means.
1011  * STR_NOSIG does not allow signals to interrupt the call
1012  * STR_NOERROR does not allow stream head read, write or hup errors to
1013  * affect the call.  When used with strdoioctl(), if a previous ioctl
1014  * is pending and times out, STR_NOERROR will cause strdoioctl() to not
1015  * return ETIME. If, however, the requested ioctl times out, ETIME
1016  * will be returned (use ic_timout instead)
1017  * STR_PEEK is used to inform strwaitq that the reader is peeking at data
1018  * and that a non-persistent error should not be cleared.
1019  * STR_DELAYERR is used to inform strwaitq that it should not check errors
1020  * after being awoken since, in addition to an error, there might also be
1021  * data queued on the stream head read queue.
1022  */
1023 #define STR_NOSIG       0x10    /* Ignore signals during strdoioctl/strwaitq */
1024 #define STR_NOERROR     0x20    /* Ignore errors during strdoioctl/strwaitq */
1025 #define STR_PEEK        0x40    /* Peeking behavior on non-persistent errors */
1026 #define STR_DELAYERR    0x80    /* Do not check errors on return */

1028 /*
1029  * Copy modes for tty and I_STR ioctls
1030  */
1031 #define U_TO_K  01                      /* User to Kernel */
1032 #define K_TO_K  02                      /* Kernel to Kernel */

1034 /*
1035  * Mux defines.
1036  */
1037 #define LINKNORMAL      0x01            /* normal mux link */
1038 #define LINKPERSIST     0x02            /* persistent mux link */
1039 #define LINKTYPEMASK    0x03            /* bitmask of all link types */
1040 #define LINKCLOSE       0x04            /* unlink from strclose */

1042 /*
1043  * Definitions of Streams macros and function interfaces.
1044  */

1046 /*
1047  * Obsolete queue scheduling macros. They are not used anymore, but still kept
1048  * here for 3-d party modules and drivers who might still use them.
1049  */
1050 #define setqsched()
1051 #define qready()        1
```

```
#ifdef _KERNEL
#define runqueues()
#define queuerun()
#endif

/* compatibility module for style 2 drivers with DR race condition */
#define DRMODNAME       "drcompat"

/*
 * Macros dealing with mux_nodes.
 */
#define MUX_VISIT(X)    ((X)->mn_flags |= VISITED)
#define MUX_CLEAR(X)    ((X)->mn_flags &= (~VISITED)); \
                        ((X)->mn_originp = NULL)
#define MUX_DIDVISIT(X) ((X)->mn_flags & VISITED)


/*
 * Twisted stream macros
 */
#define STRMATED(X)     ((X)->sd_flag & STRMATE)
#define STRLOCKMATES(X) if (&((X)->sd_lock) > &(((X)->sd_mate)->sd_lock)) { \
                                mutex_enter(&((X)->sd_lock)); \
                                mutex_enter(&(((X)->sd_mate)->sd_lock));  \
                        } else {  \
                                mutex_enter(&(((X)->sd_mate)->sd_lock)); \
                                mutex_enter(&((X)->sd_lock)); \
                        }
#define STRUNLOCKMATES(X)       mutex_exit(&((X)->sd_lock)); \
                        mutex_exit(&(((X)->sd_mate)->sd_lock))

#ifdef _KERNEL

extern void strinit(void);
extern int strdoioctl(struct stdata *, struct strioctl *, int, int,
    cred_t *, int *);
extern void strsendsig(struct strsig *, int, uchar_t, int);
extern void str_sendsig(vnode_t *, int, uchar_t, int);
extern void strhup(struct stdata *);
extern int qattach(queue_t *, dev_t *, int, cred_t *, fmodsw_impl_t *,
    boolean_t);
extern int qreopen(queue_t *, dev_t *, int, cred_t *);
extern void qdetach(queue_t *, int, int, cred_t *, boolean_t);
extern void enterq(queue_t *);
extern void leaveq(queue_t *);
extern int putiocd(mblk_t *, caddr_t, int, cred_t *);
extern int getiocd(mblk_t *, caddr_t, int);
extern struct linkinfo *alloclink(queue_t *, queue_t *, struct file *);
extern void lbfree(struct linkinfo *);
extern int linkcycle(stdata_t *, stdata_t *, str_stack_t *);
extern struct linkinfo *findlinks(stdata_t *, int, int, str_stack_t *);
extern queue_t *getendq(queue_t *);
extern int mlink(vnode_t *, int, int, cred_t *, int *, int);
extern int mlink_file(vnode_t *, int, struct file *, cred_t *, int *, int);
extern int munlink(struct stdata *, struct linkinfo *, int, cred_t *, int *,
    str_stack_t *);
extern int munlinkall(struct stdata *, int, cred_t *, int *, str_stack_t *);
extern void mux_addedge(stdata_t *, stdata_t *, int, str_stack_t *);
extern void mux_rmvedge(stdata_t *, int, str_stack_t *);
extern int devflg_to_qflag(struct streamtab *, uint32_t, uint32_t *,
    uint32_t *);
extern void setq(queue_t *, struct qinit *, struct qinit *, perdm_t *,
    uint32_t, uint32_t, boolean_t);
extern perdm_t *hold_dm(struct streamtab *, uint32_t, uint32_t);
extern void rele_dm(perdm_t *);
```



```
extern int strmakectl(struct strbuf *, int32_t, int32_t, mblk_t **);
extern int strmakedata(ssize_t *, struct uio *, stdata_t *, int32_t, mblk_t **);
extern int strmakemsg(struct strbuf *, ssize_t *, struct uio *,
    struct stdata *, int32_t, mblk_t **);
extern int strgetmsg(vnode_t *, struct strbuf *, struct strbuf *, uchar_t *,
    int *, int, rval_t *);
extern int strputmsg(vnode_t *, struct strbuf *, struct strbuf *, uchar_t,
    int flag, int fmode);
extern int strstartplumb(struct stdata *, int, int);
extern void strendplumb(struct stdata *);
extern int stropen(struct vnode *, dev_t *, int, cred_t *);
extern int strclose(struct vnode *, int, cred_t *);
extern int strpoll(register struct stdata *, short, int, short *,
    struct pollhead **);
extern void strclean(struct vnode *);
extern void str_cn_clean();     /* XXX hook for consoles signal cleanup */
extern int strwrite(struct vnode *, struct uio *, cred_t *);
extern int strwrite_common(struct vnode *, struct uio *, cred_t *, int);
extern int strread(struct vnode *, struct uio *, cred_t *);
extern int strioctl(struct vnode *, int, intptr_t, int, int, cred_t *, int *);
extern int strrput(queue_t *, mblk_t *);
extern int strrput_nondata(queue_t *, mblk_t *);
extern mblk_t *strrput_proto(vnode_t *, mblk_t *,
    strwakeup_t *, strsigset_t *, strsigset_t *, strpollset_t *);
extern mblk_t *strrput_misc(vnode_t *, mblk_t *,
    strwakeup_t *, strsigset_t *, strsigset_t *, strpollset_t *);
extern int getiocseqno(void);
extern int strwaitbuf(size_t, int);
extern int strwaitq(stdata_t *, int, ssize_t, int, clock_t, int *);
extern struct stdata *shalloc(queue_t *);
extern void sh_insert_pid(struct stdata *, pid_t);
extern void sh_remove_pid(struct stdata *, pid_t);
extern mblk_t *sh_get_pid_mblk(struct stdata *);
#endif /* ! codereview */
extern void shfree(struct stdata *s);
extern queue_t *allocq(void);
extern void freeq(queue_t *);
extern qband_t *allocband(void);
extern void freeband(qband_t *);
extern void freebs_enqueue(mblk_t *, dblk_t *);
extern void setqback(queue_t *, unsigned char);
extern int strcopyin(void *, void *, size_t, int);
extern int strcopyout(void *, void *, size_t, int);
extern void strsignal(struct stdata *, int, int32_t);
extern clock_t str_cv_wait(kcondvar_t *, kmutex_t *, clock_t, int);
extern void disable_svc(queue_t *);
extern void enable_svc(queue_t *);
extern void remove_runlist(queue_t *);
extern void wait_svc(queue_t *);
extern void backenable(queue_t *, uchar_t);
extern void set_qend(queue_t *);
extern int strgeterr(stdata_t *, int32_t, int);
extern void qenable_locked(queue_t *);
extern mblk_t *getq_noenab(queue_t *, ssize_t);
extern void rmvq_noenab(queue_t *, mblk_t *);
extern void qbackenable(queue_t *, uchar_t);
extern void set_qfull(queue_t *);

extern void strblock(queue_t *);
extern void strunblock(queue_t *);
extern int qclaimed(queue_t *);
extern int straccess(struct stdata *, enum jcaccess);

extern void entersq(syncq_t *, int);
extern void leavesq(syncq_t *, int);
extern void claimq(queue_t *);
```

```
1184 extern void releaseq(queue_t *);
1185 extern void claimstr(queue_t *);
1186 extern void releasestr(queue_t *);
1187 extern void removeq(queue_t *);
1188 extern void insertq(struct stdata *, queue_t *);
1189 extern void drain_syncq(syncq_t *);
1190 extern void qfill_syncq(syncq_t *, queue_t *, mblk_t *);
1191 extern void qdrain_syncq(syncq_t *, queue_t *);
1192 extern int flush_syncq(syncq_t *, queue_t *);
1193 extern void wait_sq_svc(syncq_t *);

1195 extern void outer_enter(syncq_t *, uint16_t);
1196 extern void outer_exit(syncq_t *);
1197 extern void qwriter_inner(queue_t *, mblk_t *, void (*)());
1198 extern void qwriter_outer(queue_t *, mblk_t *, void (*)());

1200 extern callbparams_t *callbparams_alloc(syncq_t *, void (*)(void *),
1201     void *, int);
1202 extern void callbparams_free(syncq_t *, callbparams_t *);
1203 extern void callbparams_free_id(syncq_t *, callbparams_id_t, int32_t);
1204 extern void qcallbwrapper(void *);

1206 extern mblk_t *esballoc_wait(unsigned char *, size_t, uint_t, frtn_t *);
1207 extern mblk_t *esballoca(unsigned char *, size_t, uint_t, frtn_t *);
1208 extern mblk_t *desballoca(unsigned char *, size_t, uint_t, frtn_t *);
1209 extern int do_sendfp(struct stdata *, struct file *, struct cred *);
1210 extern int frozenstr(queue_t *);
1211 extern size_t xmsgsize(mblk_t *);

1213 extern void putnext_tail(syncq_t *, queue_t *, uint32_t);
1214 extern void stream_willservice(stdata_t *);
1215 extern void stream_runservice(stdata_t *);

1217 extern void strmate(vnode_t *, vnode_t *);
1218 extern queue_t *strvp2wq(vnode_t *);
1219 extern vnode_t *strq2vp(queue_t *);
1220 extern mblk_t *allocb_wait(size_t, uint_t, uint_t, int *);
1221 extern mblk_t *allocb_cred(size_t, cred_t *, pid_t);
1222 extern mblk_t *allocb_cred_wait(size_t, uint_t, int *, cred_t *, pid_t);
1223 extern mblk_t *allocb_tmpl(size_t, const mblk_t *);
1224 extern mblk_t *allocb_tryhard(size_t);
1225 extern void mblk_copycred(mblk_t *, const mblk_t *);
1226 extern void mblk_setcred(mblk_t *, cred_t *, pid_t);
1227 extern cred_t *msg_getcred(const mblk_t *, pid_t *);
1228 extern struct ts_label_s *msg_getlabel(const mblk_t *);
1229 extern cred_t *msg_extractcred(mblk_t *, pid_t *);
1230 extern void strpollwakeup(vnode_t *, short);
1231 extern int putnextctl_wait(queue_t *, int);

1233 extern int kstrputmsg(struct vnode *, mblk_t *, struct uio *, ssize_t,
1234     unsigned char, int, int);
1235 extern int kstrgetmsg(struct vnode *, mblk_t **, struct uio *,
1236     unsigned char *, int *, clock_t, rval_t *);

1238 extern void strsetrerror(vnode_t *, int, int, errfunc_t);
1239 extern void strsetwerror(vnode_t *, int, int, errfunc_t);
1240 extern void strseteof(vnode_t *, int);
1241 extern void strflushrq(vnode_t *, int);
1242 extern void strsetrputhooks(vnode_t *, uint_t, msgfunc_t, msgfunc_t);
1243 extern void strsetwputhooks(vnode_t *, uint_t, clock_t);
1244 extern void strsetrwputdatahooks(vnode_t *, msgfunc_t, msgfunc_t);
1245 extern int strwaitmark(vnode_t *);
1246 extern void strsignal_nolock(stdata_t *, int, uchar_t);

1248 struct multidata_s;
1249 struct pdesc_s;
```



```
1250 extern int hcksum_assoc(mblk_t *, struct multidata_s *, struct pdesc_s  *,
1251     uint32_t, uint32_t, uint32_t, uint32_t, uint32_t, int);
1252 extern void hcksum_retrieve(mblk_t *, struct multidata_s *, struct pdesc_s *,
1253     uint32_t *, uint32_t *, uint32_t *, uint32_t *, uint32_t *);
1254 extern void lso_info_set(mblk_t *, uint32_t, uint32_t);
1255 extern void lso_info_cleanup(mblk_t *);
1256 extern unsigned int bcksum(uchar_t *, int, unsigned int);
1257 extern boolean_t is_vmloaned_mblk(mblk_t *, struct multidata_s *,
1258     struct pdesc_s *);

1260 extern int fmodsw_register(const char *, struct streamtab *, int);
1261 extern int fmodsw_unregister(const char *);
1262 extern fmodsw_impl_t *fmodsw_find(const char *, fmodsw_flags_t);
1263 extern void fmodsw_rele(fmodsw_impl_t *);

1265 extern void freemsgchain(mblk_t *);
1266 extern mblk_t *copymsgchain(mblk_t *);

1268 extern mblk_t *mcopyinuio(struct stdata *, uio_t *, ssize_t, ssize_t, int *);

1270 /*
1271  * shared or externally configured data structures
1272  */
1273 extern ssize_t strmsgsz;                /* maximum stream message size */
1274 extern ssize_t strctlsz;                /* maximum size of ctl message */
1275 extern int nstrpush;                    /* maximum number of pushes allowed */

1277 /*
1278  * Bufcalls related variables.
1279  */
1280 extern struct bclist strbcalls;         /* List of bufcalls */
1281 extern kmutex_t strbcall_lock;          /* Protects the list of bufcalls */
1282 extern kcondvar_t strbcall_cv;          /* Signaling when a bufcall is added */
1283 extern kcondvar_t bcall_cv;     /* wait of executing bufcall completes */

1285 extern frtn_t frnop;

1287 extern struct kmem_cache *ciputctrl_cache;
1288 extern int n_ciputctrl;
1289 extern int max_n_ciputctrl;
1290 extern int min_n_ciputctrl;

1292 extern cdevsw_impl_t *devimpl;

1294 /*
1295  * esballoc queue for throttling
1296  */
1297 typedef struct esb_queue {
1298         kmutex_t        eq_lock;
1299         uint_t          eq_len;         /* number of queued messages */
1300         mblk_t          *eq_head;       /* head of queue */
1301         mblk_t          *eq_tail;       /* tail of queue */
1302         uint_t          eq_flags;       /* esballoc queue flags */
1303 } esb_queue_t;

1305 /*
1306  * esballoc flags for queue processing.
1307  */
1308 #define ESBQ_PROCESSING 0x01    /* queue is being processed */
1309 #define ESBQ_TIMER      0x02    /* timer is active */

1311 extern void esballoc_queue_init(void);

1313 #endif  /* _KERNEL */

1315 /*
```

```
1316  * Note: Use of these macros are restricted to kernel/unix and
1317  * intended for the STREAMS framework.
1318  * All modules/drivers should include sys/ddi.h.
1319  *
1320  * Finding related queues
1321  */
1322 #define         _OTHERQ(q)      ((q)->q_flag&QREADR? (q)+1: (q)-1)
1323 #define         _WR(q)          ((q)->q_flag&QREADR? (q)+1: (q))
1324 #define         _RD(q)          ((q)->q_flag&QREADR? (q): (q)-1)
1325 #define         _SAMESTR(q)     (!((q)->q_flag & QEND))

1327 /*
1328  * These are also declared here for modules/drivers that erroneously
1329  * include strsubr.h after ddi.h or fail to include ddi.h at all.
1330  */
1331 extern struct queue *OTHERQ(queue_t *); /* stream.h */
1332 extern struct queue *RD(queue_t *);
1333 extern struct queue *WR(queue_t *);
1334 extern int SAMESTR(queue_t *);

1336 /*
1337  * The following hardware checksum related macros are private
1338  * interfaces that are subject to change without notice.
1339  */
1340 #ifdef _KERNEL
1341 #define DB_CKSUMSTART(mp)       ((mp)->b_datap->db_cksumstart)
1342 #define DB_CKSUMEND(mp)         ((mp)->b_datap->db_cksumend)
1343 #define DB_CKSUMSTUFF(mp)       ((mp)->b_datap->db_cksumstuff)
1344 #define DB_CKSUMFLAGS(mp)       ((mp)->b_datap->db_struioun.cksum.flags)
1345 #define DB_CKSUM16(mp)          ((mp)->b_datap->db_cksum16)
1346 #define DB_CKSUM32(mp)          ((mp)->b_datap->db_cksum32)
1347 #define DB_LSOFLAGS(mp)         ((mp)->b_datap->db_struioun.cksum.flags)
1348 #define DB_LSOMSS(mp)           ((mp)->b_datap->db_struioun.cksum.pad)
1349 #endif  /* _KERNEL */

1351 #ifdef  __cplusplus
1352 }
1353 #endif


1356 #endif  /* _SYS_STRSUBR_H */
```

```
**********************************************************
   21242 Mon Aug 17 21:08:10 2015
new/usr/src/uts/common/syscall/fcntl.c
XXXX adding PID information to netstat output
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013, OmniTI Computer Consulting, Inc. All rights reserved.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * Portions of this source code were derived from Berkeley 4.3 BSD
 * under license from the Regents of the University of California.
 */


#include <sys/param.h>
#include <sys/isa_defs.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/mode.h>
#include <sys/proc.h>
#include <sys/filio.h>
#include <sys/share.h>
#include <sys/debug.h>
#include <sys/rctl.h>
#include <sys/nbmlock.h>

#include <sys/cmn_err.h>

static int flock_check(vnode_t *, flock64_t *, offset_t, offset_t);
static int flock_get_start(vnode_t *, flock64_t *, offset_t, u_offset_t *);
static void fd_too_big(proc_t *);

/*
 * File control.
```



```
 */
int
fcntl(int fdes, int cmd, intptr_t arg)
{
        int iarg;
        int error = 0;
        int retval;
        proc_t *p;
        file_t *fp;
        vnode_t *vp;
        u_offset_t offset;
        u_offset_t start;
        struct vattr vattr;
        int in_crit;
        int flag;
        struct flock sbf;
        struct flock64 bf;
        struct o_flock obf;
        struct flock64_32 bf64_32;
        struct fshare fsh;
        struct shrlock shr;
        struct shr_locowner shr_own;
        offset_t maxoffset;
        model_t datamodel;
        int fdres;

#if defined(_ILP32) && !defined(lint) && defined(_SYSCALL32)
        ASSERT(sizeof (struct flock) == sizeof (struct flock32));
        ASSERT(sizeof (struct flock64) == sizeof (struct flock64_32));
#endif
#if defined(_LP64) && !defined(lint) && defined(_SYSCALL32)
        ASSERT(sizeof (struct flock) == sizeof (struct flock64_64));
        ASSERT(sizeof (struct flock64) == sizeof (struct flock64_64));
#endif

        /*
         * First, for speed, deal with the subset of cases
         * that do not require getf() / releasef().
         */
        switch (cmd) {
        case F_GETFD:
                if ((error = f_getfd_error(fdes, &flag)) == 0)
                        retval = flag;
                goto out;

        case F_SETFD:
                error = f_setfd_error(fdes, (int)arg);
                retval = 0;
                goto out;

        case F_GETFL:
                if ((error = f_getfl(fdes, &flag)) == 0) {
                        retval = (flag & (FMASK | FASYNC));
                        if ((flag & (FSEARCH | FEXEC)) == 0)
                                retval += FOPEN;
                        else
                                retval |= (flag & (FSEARCH | FEXEC));
                }
                goto out;

        case F_GETXFL:
                if ((error = f_getfl(fdes, &flag)) == 0) {
                        retval = flag;
                        if ((flag & (FSEARCH | FEXEC)) == 0)
                                retval += FOPEN;
                }
```

```
128                 goto out;
129
130         case F_BADFD:
131                 if ((error = f_badfd(fdes, &fdres, (int)arg)) == 0)
132                         retval = fdres;
133                 goto out;
134         }
135
136         /*
137          * Second, for speed, deal with the subset of cases that
138          * require getf() / releasef() but do not require copyin.
139          */
140         if ((fp = getf(fdes)) == NULL) {
141                 error = EBADF;
142                 goto out;
143         }
144         iarg = (int)arg;
145
146         switch (cmd) {
147         case F_DUPFD:
148         case F_DUPFD_CLOEXEC:
149                 p = curproc;
150                 if ((uint_t)iarg >= p->p_fno_ctl) {
151                         if (iarg >= 0)
152                                 fd_too_big(p);
153                         error = EINVAL;
154                         goto done;
155                 }
156                 /*
157                  * We need to increment the f_count reference counter
158                  * before allocating a new file descriptor.
159                  * Doing it other way round opens a window for race condition
160                  * with closeandsetf() on the target file descriptor which can
161                  * close the file still referenced by the original
162                  * file descriptor.
163                  */
164                 mutex_enter(&fp->f_tlock);
165                 fp->f_count++;
166                 mutex_exit(&fp->f_tlock);
167                 if ((retval = ufalloc_file(iarg, fp)) == -1) {
168                         /*
169                          * New file descriptor can't be allocated.
170                          * Revert the reference count.
171                          */
172                         mutex_enter(&fp->f_tlock);
173                         fp->f_count--;
174                         mutex_exit(&fp->f_tlock);
175                         error = EMFILE;
176                 } else {
177                         if (cmd == F_DUPFD_CLOEXEC) {
178                                 f_setfd(retval, FD_CLOEXEC);
179                         }
180                 }
181
182                 if (error == 0 && fp->f_vnode != NULL) {
183                         (void) VOP_IOCTL(fp->f_vnode, F_ASSOCI_PID,
184                             (intptr_t)p->p_pidp->pid_id, FKIOCTL, kcred,
185                             NULL, NULL);
186                 }
187
188 #endif /* ! codereview */
189                 goto done;
190
191         case F_DUP2FD_CLOEXEC:
192                 if (fdes == iarg) {
193                         error = EINVAL;
```



```
194                         goto done;
195                 }
196
197                 /*FALLTHROUGH*/
198
199         case F_DUP2FD:
200                 p = curproc;
201                 if (fdes == iarg) {
202                         retval = iarg;
203                 } else if ((uint_t)iarg >= p->p_fno_ctl) {
204                         if (iarg >= 0)
205                                 fd_too_big(p);
206                         error = EBADF;
207                 } else {
208                         /*
209                          * We can't hold our getf(fdes) across the call to
210                          * closeandsetf() because it creates a window for
211                          * deadlock: if one thread is doing dup2(a, b) while
212                          * another is doing dup2(b, a), each one will block
213                          * waiting for the other to call releasef().  The
214                          * solution is to increment the file reference count
215                          * (which we have to do anyway), then releasef(fdes),
216                          * then closeandsetf().  Incrementing f_count ensures
217                          * that fp won't disappear after we call releasef().
218                          * When closeandsetf() fails, we try avoid calling
219                          * closef() because of all the side effects.
220                          */
221                         mutex_enter(&fp->f_tlock);
222                         fp->f_count++;
223                         mutex_exit(&fp->f_tlock);
224                         releasef(fdes);
225
226                         /* assume we have forked successfully */
227                         if (fp->f_vnode != NULL) {
228                                 (void) VOP_IOCTL(fp->f_vnode, F_ASSOCI_PID,
229                                     (intptr_t)p->p_pidp->pid_id, FKIOCTL,
230                                     kcred, NULL, NULL);
231                         }
232
233 #endif /* ! codereview */
234                         if ((error = closeandsetf(iarg, fp)) == 0) {
235                                 if (cmd == F_DUP2FD_CLOEXEC) {
236                                         f_setfd(iarg, FD_CLOEXEC);
237                                 }
238                                 retval = iarg;
239                         } else {
240                                 mutex_enter(&fp->f_tlock);
241                                 if (fp->f_count > 1) {
242                                         fp->f_count--;
243                                         mutex_exit(&fp->f_tlock);
244                                         if (fp->f_vnode != NULL) {
245                                                 (void) VOP_IOCTL(fp->f_vnode,
246                                                     F_DASSOC_PID,
247                                                     (intptr_t)p->p_pidp->pid_id,
248                                                     FKIOCTL, kcred, NULL, NULL);
249                                         }
250
251 #endif /* ! codereview */
252                                 } else {
253                                         mutex_exit(&fp->f_tlock);
254                                         (void) closef(fp);
255                                 }
256                         }
257                         goto out;
258                 }
259                 goto done;
```

```
261         case F_SETFL:
262                 vp = fp->f_vnode;
263                 flag = fp->f_flag;
264                 if ((iarg & (FNONBLOCK|FNDELAY)) == (FNONBLOCK|FNDELAY))
265                         iarg &= ~FNDELAY;
266                 if ((error = VOP_SETFL(vp, flag, iarg, fp->f_cred, NULL)) ==
267                     0) {
268                         iarg &= FMASK;
269                         mutex_enter(&fp->f_tlock);
270                         fp->f_flag &= ~FMASK | (FREAD|FWRITE);
271                         fp->f_flag |= (iarg - FOPEN) & ~(FREAD|FWRITE);
272                         mutex_exit(&fp->f_tlock);
273                 }
274                 retval = 0;
275                 goto done;
276         }
277
278         /*
279          * Finally, deal with the expensive cases.
280          */
281         retval = 0;
282         in_crit = 0;
283         maxoffset = MAXOFF_T;
284         datamodel = DATAMODEL_NATIVE;
285 #if defined(_SYSCALL32_IMPL)
286         if ((datamodel = get_udatamodel()) == DATAMODEL_ILP32)
287                 maxoffset = MAXOFF32_T;
288 #endif
289
290         vp = fp->f_vnode;
291         flag = fp->f_flag;
292         offset = fp->f_offset;
293
294         switch (cmd) {
295         /*
296          * The file system and vnode layers understand and implement
297          * locking with flock64 structures. So here once we pass through
298          * the test for compatibility as defined by LFS API, (for F_SETLK,
299          * F_SETLKW, F_GETLK, F_GETLKW, F_FREESP) we transform
300          * the flock structure to a flock64 structure and send it to the
301          * lower layers. Similarly in case of GETLK the returned flock64
302          * structure is transformed to a flock structure if everything fits
303          * in nicely, otherwise we return EOVERFLOW.
304          */
305
306         case F_GETLK:
307         case F_O_GETLK:
308         case F_SETLK:
309         case F_SETLKW:
310         case F_SETLK_NBMAND:
311
312                 /*
313                  * Copy in input fields only.
314                  */
315
316                 if (cmd == F_O_GETLK) {
317                         if (datamodel != DATAMODEL_ILP32) {
318                                 error = EINVAL;
319                                 break;
320                         }
321
322                         if (copyin((void *)arg, &obf, sizeof (obf))) {
323                                 error = EFAULT;
324                                 break;
325                         }
```



```
326                         bf.l_type = obf.l_type;
327                         bf.l_whence = obf.l_whence;
328                         bf.l_start = (off64_t)obf.l_start;
329                         bf.l_len = (off64_t)obf.l_len;
330                         bf.l_sysid = (int)obf.l_sysid;
331                         bf.l_pid = obf.l_pid;
332                 } else if (datamodel == DATAMODEL_NATIVE) {
333                         if (copyin((void *)arg, &sbf, sizeof (sbf))) {
334                                 error = EFAULT;
335                                 break;
336                         }
337                         /*
338                          * XXX  In an LP64 kernel with an LP64 application
339                          *      there's no need to do a structure copy here
340                          *      struct flock == struct flock64. However,
341                          *      we did it this way to avoid more conditional
342                          *      compilation.
343                          */
344                         bf.l_type = sbf.l_type;
345                         bf.l_whence = sbf.l_whence;
346                         bf.l_start = (off64_t)sbf.l_start;
347                         bf.l_len = (off64_t)sbf.l_len;
348                         bf.l_sysid = sbf.l_sysid;
349                         bf.l_pid = sbf.l_pid;
350                 }
351 #if defined(_SYSCALL32_IMPL)
352                 else {
353                         struct flock32 sbf32;
354                         if (copyin((void *)arg, &sbf32, sizeof (sbf32))) {
355                                 error = EFAULT;
356                                 break;
357                         }
358                         bf.l_type = sbf32.l_type;
359                         bf.l_whence = sbf32.l_whence;
360                         bf.l_start = (off64_t)sbf32.l_start;
361                         bf.l_len = (off64_t)sbf32.l_len;
362                         bf.l_sysid = sbf32.l_sysid;
363                         bf.l_pid = sbf32.l_pid;
364                 }
365 #endif /* _SYSCALL32_IMPL */
366
367                 /*
368                  * 64-bit support: check for overflow for 32-bit lock ops
369                  */
370                 if ((error = flock_check(vp, &bf, offset, maxoffset)) != 0)
371                         break;
372
373                 /*
374                  * Not all of the filesystems understand F_O_GETLK, and
375                  * there's no need for them to know.  Map it to F_GETLK.
376                  */
377                 if ((error = VOP_FRLOCK(vp, (cmd == F_O_GETLK) ? F_GETLK : cmd,
378                     &bf, flag, offset, NULL, fp->f_cred, NULL)) != 0)
379                         break;
380
381                 /*
382                  * If command is GETLK and no lock is found, only
383                  * the type field is changed.
384                  */
385                 if ((cmd == F_O_GETLK || cmd == F_GETLK) &&
386                     bf.l_type == F_UNLCK) {
387                         /* l_type always first entry, always a short */
388                         if (copyout(&bf.l_type, &((struct flock *)arg)->l_type,
389                             sizeof (bf.l_type)))
390                                 error = EFAULT;
391                         break;
```

```
392                 }
393
394                 if (cmd == F_O_GETLK) {
395                         /*
396                          * Return an SVR3 flock structure to the user.
397                          */
398                         obf.l_type = (int16_t)bf.l_type;
399                         obf.l_whence = (int16_t)bf.l_whence;
400                         obf.l_start = (int32_t)bf.l_start;
401                         obf.l_len = (int32_t)bf.l_len;
402                         if (bf.l_sysid > SHRT_MAX || bf.l_pid > SHRT_MAX) {
403                                 /*
404                                  * One or both values for the above fields
405                                  * is too large to store in an SVR3 flock
406                                  * structure.
407                                  */
408                                 error = EOVERFLOW;
409                                 break;
410                         }
411                         obf.l_sysid = (int16_t)bf.l_sysid;
412                         obf.l_pid = (int16_t)bf.l_pid;
413                         if (copyout(&obf, (void *)arg, sizeof (obf)))
414                                 error = EFAULT;
415                 } else if (cmd == F_GETLK) {
416                         /*
417                          * Copy out SVR4 flock.
418                          */
419                         int i;
420
421                         if (bf.l_start > maxoffset || bf.l_len > maxoffset) {
422                                 error = EOVERFLOW;
423                                 break;
424                         }
425
426                         if (datamodel == DATAMODEL_NATIVE) {
427                                 for (i = 0; i < 4; i++)
428                                         sbf.l_pad[i] = 0;
429                                 /*
430                                  * XXX  In an LP64 kernel with an LP64
431                                  *      application there's no need to do a
432                                  *      structure copy here as currently
433                                  *      struct flock == struct flock64.
434                                  *      We did it this way to avoid more
435                                  *      conditional compilation.
436                                  */
437                                 sbf.l_type = bf.l_type;
438                                 sbf.l_whence = bf.l_whence;
439                                 sbf.l_start = (off_t)bf.l_start;
440                                 sbf.l_len = (off_t)bf.l_len;
441                                 sbf.l_sysid = bf.l_sysid;
442                                 sbf.l_pid = bf.l_pid;
443                                 if (copyout(&sbf, (void *)arg, sizeof (sbf)))
444                                         error = EFAULT;
445                         }
446 #if defined(_SYSCALL32_IMPL)
447                         else {
448                                 struct flock32 sbf32;
449                                 if (bf.l_start > MAXOFF32_T ||
450                                     bf.l_len > MAXOFF32_T) {
451                                         error = EOVERFLOW;
452                                         break;
453                                 }
454                                 for (i = 0; i < 4; i++)
455                                         sbf32.l_pad[i] = 0;
456                                 sbf32.l_type = (int16_t)bf.l_type;
457                                 sbf32.l_whence = (int16_t)bf.l_whence;
```



```
458                                 sbf32.l_start = (off32_t)bf.l_start;
459                                 sbf32.l_len = (off32_t)bf.l_len;
460                                 sbf32.l_sysid = (int32_t)bf.l_sysid;
461                                 sbf32.l_pid = (pid32_t)bf.l_pid;
462                                 if (copyout(&sbf32,
463                                     (void *)arg, sizeof (sbf32)))
464                                         error = EFAULT;
465                         }
466 #endif
467                 }
468                 break;
469
470         case F_CHKFL:
471                 /*
472                  * This is for internal use only, to allow the vnode layer
473                  * to validate a flags setting before applying it.  User
474                  * programs can't issue it.
475                  */
476                 error = EINVAL;
477                 break;
478
479         case F_ALLOCSP:
480         case F_FREESP:
481         case F_ALLOCSP64:
482         case F_FREESP64:
483                 /*
484                  * Test for not-a-regular-file (and returning EINVAL)
485                  * before testing for open-for-writing (and returning EBADF).
486                  * This is relied upon by posix_fallocate() in libc.
487                  */
488                 if (vp->v_type != VREG) {
489                         error = EINVAL;
490                         break;
491                 }
492
493                 if ((flag & FWRITE) == 0) {
494                         error = EBADF;
495                         break;
496                 }
497
498                 if (datamodel != DATAMODEL_ILP32 &&
499                     (cmd == F_ALLOCSP64 || cmd == F_FREESP64)) {
500                         error = EINVAL;
501                         break;
502                 }
503
504 #if defined(_ILP32) || defined(_SYSCALL32_IMPL)
505                 if (datamodel == DATAMODEL_ILP32 &&
506                     (cmd == F_ALLOCSP || cmd == F_FREESP)) {
507                         struct flock32 sbf32;
508                         /*
509                          * For compatibility we overlay an SVR3 flock on an SVR4
510                          * flock.  This works because the input field offsets
511                          * in "struct flock" were preserved.
512                          */
513                         if (copyin((void *)arg, &sbf32, sizeof (sbf32))) {
514                                 error = EFAULT;
515                                 break;
516                         } else {
517                                 bf.l_type = sbf32.l_type;
518                                 bf.l_whence = sbf32.l_whence;
519                                 bf.l_start = (off64_t)sbf32.l_start;
520                                 bf.l_len = (off64_t)sbf32.l_len;
521                                 bf.l_sysid = sbf32.l_sysid;
522                                 bf.l_pid = sbf32.l_pid;
523                         }
```

```
524                 }
525 #endif /* _ILP32 || _SYSCALL32_IMPL */

527 #if defined(_LP64)
528                 if (datamodel == DATAMODEL_LP64 &&
529                     (cmd == F_ALLOCSP || cmd == F_FREESP)) {
530                         if (copyin((void *)arg, &bf, sizeof (bf))) {
531                                 error = EFAULT;
532                                 break;
533                         }
534                 }
535 #endif /* defined(_LP64) */

537 #if !defined(_LP64) || defined(_SYSCALL32_IMPL)
538                 if (datamodel == DATAMODEL_ILP32 &&
539                     (cmd == F_ALLOCSP64 || cmd == F_FREESP64)) {
540                         if (copyin((void *)arg, &bf64_32, sizeof (bf64_32))) {
541                                 error = EFAULT;
542                                 break;
543                         } else {
544                                 /*
545                                  * Note that the size of flock64 is different in
546                                  * the ILP32 and LP64 models, due to the l_pad
547                                  * field. We do not want to assume that the
548                                  * flock64 structure is laid out the same in
549                                  * ILP32 and LP64 environments, so we will
550                                  * copy in the ILP32 version of flock64
551                                  * explicitly and copy it to the native
552                                  * flock64 structure.
553                                  */
554                                 bf.l_type = (short)bf64_32.l_type;
555                                 bf.l_whence = (short)bf64_32.l_whence;
556                                 bf.l_start = bf64_32.l_start;
557                                 bf.l_len = bf64_32.l_len;
558                                 bf.l_sysid = (int)bf64_32.l_sysid;
559                                 bf.l_pid = (pid_t)bf64_32.l_pid;
560                         }
561                 }
562 #endif /* !defined(_LP64) || defined(_SYSCALL32_IMPL) */

564                 if (cmd == F_ALLOCSP || cmd == F_FREESP)
565                         error = flock_check(vp, &bf, offset, maxoffset);
566                 else if (cmd == F_ALLOCSP64 || cmd == F_FREESP64)
567                         error = flock_check(vp, &bf, offset, MAXOFFSET_T);
568                 if (error)
569                         break;

571                 if (vp->v_type == VREG && bf.l_len == 0 &&
572                     bf.l_start > OFFSET_MAX(fp)) {
573                         error = EFBIG;
574                         break;
575                 }

577                 /*
578                  * Make sure that there are no conflicting non-blocking
579                  * mandatory locks in the region being manipulated. If
580                  * there are such locks then return EACCES.
581                  */
582                 if ((error = flock_get_start(vp, &bf, offset, &start)) != 0)
583                         break;

585                 if (nbl_need_check(vp)) {
586                         u_offset_t      begin;
587                         ssize_t         length;

589                         nbl_start_crit(vp, RW_READER);
```



```
590                         in_crit = 1;
591                         vattr.va_mask = AT_SIZE;
592                         if ((error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
593                             != 0)
594                                 break;
595                         begin = start > vattr.va_size ? vattr.va_size : start;
596                         length = vattr.va_size > start ? vattr.va_size - start :
597                             start - vattr.va_size;
598                         if (nbl_conflict(vp, NBL_WRITE, begin, length, 0,
599                             NULL)) {
600                                 error = EACCES;
601                                 break;
602                         }
603                 }

605                 if (cmd == F_ALLOCSP64)
606                         cmd = F_ALLOCSP;
607                 else if (cmd == F_FREESP64)
608                         cmd = F_FREESP;

610                 error = VOP_SPACE(vp, cmd, &bf, flag, offset, fp->f_cred, NULL);

612                 break;

614 #if !defined(_LP64) || defined(_SYSCALL32_IMPL)
615         case F_GETLK64:
616         case F_SETLK64:
617         case F_SETLKW64:
618         case F_SETLK64_NBMAND:
619                 /*
620                  * Large Files: Here we set cmd as *LK and send it to
621                  * lower layers. *LK64 is only for the user land.
622                  * Most of the comments described above for F_SETLK
623                  * applies here too.
624                  * Large File support is only needed for ILP32 apps!
625                  */
626                 if (datamodel != DATAMODEL_ILP32) {
627                         error = EINVAL;
628                         break;
629                 }

631                 if (cmd == F_GETLK64)
632                         cmd = F_GETLK;
633                 else if (cmd == F_SETLK64)
634                         cmd = F_SETLK;
635                 else if (cmd == F_SETLKW64)
636                         cmd = F_SETLKW;
637                 else if (cmd == F_SETLK64_NBMAND)
638                         cmd = F_SETLK_NBMAND;

640                 /*
641                  * Note that the size of flock64 is different in the ILP32
642                  * and LP64 models, due to the sucking l_pad field.
643                  * We do not want to assume that the flock64 structure is
644                  * laid out in the same in ILP32 and LP64 environments, so
645                  * we will copy in the ILP32 version of flock64 explicitly
646                  * and copy it to the native flock64 structure.
647                  */

649                 if (copyin((void *)arg, &bf64_32, sizeof (bf64_32))) {
650                         error = EFAULT;
651                         break;
652                 }

654                 bf.l_type = (short)bf64_32.l_type;
655                 bf.l_whence = (short)bf64_32.l_whence;
```

```
656                 bf.l_start = bf64_32.l_start;
657                 bf.l_len = bf64_32.l_len;
658                 bf.l_sysid = (int)bf64_32.l_sysid;
659                 bf.l_pid = (pid_t)bf64_32.l_pid;
660
661                 if ((error = flock_check(vp, &bf, offset, MAXOFFSET_T)) != 0)
662                         break;
663
664                 if ((error = VOP_FRLOCK(vp, cmd, &bf, flag, offset,
665                     NULL, fp->f_cred, NULL)) != 0)
666                         break;
667
668                 if ((cmd == F_GETLK) && bf.l_type == F_UNLCK) {
669                         if (copyout(&bf.l_type, &((struct flock *)arg)->l_type,
670                             sizeof (bf.l_type)))
671                                 error = EFAULT;
672                         break;
673                 }
674
675                 if (cmd == F_GETLK) {
676                         int i;
677
678                         /*
679                          * We do not want to assume that the flock64 structure
680                          * is laid out in the same in ILP32 and LP64
681                          * environments, so we will copy out the ILP32 version
682                          * of flock64 explicitly after copying the native
683                          * flock64 structure to it.
684                          */
685                         for (i = 0; i < 4; i++)
686                                 bf64_32.l_pad[i] = 0;
687                         bf64_32.l_type = (int16_t)bf.l_type;
688                         bf64_32.l_whence = (int16_t)bf.l_whence;
689                         bf64_32.l_start = bf.l_start;
690                         bf64_32.l_len = bf.l_len;
691                         bf64_32.l_sysid = (int32_t)bf.l_sysid;
692                         bf64_32.l_pid = (pid32_t)bf.l_pid;
693                         if (copyout(&bf64_32, (void *)arg, sizeof (bf64_32)))
694                                 error = EFAULT;
695                 }
696                 break;
697 #endif /* !defined(_LP64) || defined(_SYSCALL32_IMPL) */
698
699         case F_SHARE:
700         case F_SHARE_NBMAND:
701         case F_UNSHARE:
702
703                 /*
704                  * Copy in input fields only.
705                  */
706                 if (copyin((void *)arg, &fsh, sizeof (fsh))) {
707                         error = EFAULT;
708                         break;
709                 }
710
711                 /*
712                  * Local share reservations always have this simple form
713                  */
714                 shr.s_access = fsh.f_access;
715                 shr.s_deny = fsh.f_deny;
716                 shr.s_sysid = 0;
717                 shr.s_pid = ttoproc(curthread)->p_pid;
718                 shr_own.sl_pid = shr.s_pid;
719                 shr_own.sl_id = fsh.f_id;
720                 shr.s_own_len = sizeof (shr_own);
721                 shr.s_owner = (caddr_t)&shr_own;
```



```
722                 error = VOP_SHRLOCK(vp, cmd, &shr, flag, fp->f_cred, NULL);
723                 break;
724
725         default:
726                 error = EINVAL;
727                 break;
728         }
729
730         if (in_crit)
731                 nbl_end_crit(vp);
732
733 done:
734         releasef(fdes);
735 out:
736         if (error)
737                 return (set_errno(error));
738         return (retval);
739 }
740
741 int
742 flock_check(vnode_t *vp, flock64_t *flp, offset_t offset, offset_t max)
743 {
744         struct vattr    vattr;
745         int     error;
746         u_offset_t start, end;
747
748         /*
749          * Determine the starting point of the request
750          */
751         switch (flp->l_whence) {
752         case 0:         /* SEEK_SET */
753                 start = (u_offset_t)flp->l_start;
754                 if (start > max)
755                         return (EINVAL);
756                 break;
757         case 1:         /* SEEK_CUR */
758                 if (flp->l_start > (max - offset))
759                         return (EOVERFLOW);
760                 start = (u_offset_t)(flp->l_start + offset);
761                 if (start > max)
762                         return (EINVAL);
763                 break;
764         case 2:         /* SEEK_END */
765                 vattr.va_mask = AT_SIZE;
766                 if (error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
767                         return (error);
768                 if (flp->l_start > (max - (offset_t)vattr.va_size))
769                         return (EOVERFLOW);
770                 start = (u_offset_t)(flp->l_start + (offset_t)vattr.va_size);
771                 if (start > max)
772                         return (EINVAL);
773                 break;
774         default:
775                 return (EINVAL);
776         }
777
778         /*
779          * Determine the range covered by the request.
780          */
781         if (flp->l_len == 0)
782                 end = MAXEND;
783         else if ((offset_t)flp->l_len > 0) {
784                 if (flp->l_len > (max - start + 1))
785                         return (EOVERFLOW);
786                 end = (u_offset_t)(start + (flp->l_len - 1));
787                 ASSERT(end <= max);
```

```
788         } else {
789                 /*
790                  * Negative length; why do we even allow this ?
791                  * Because this allows easy specification of
792                  * the last n bytes of the file.
793                  */
794                 end = start;
795                 start += (u_offset_t)flp->l_len;
796                 (start)++;
797                 if (start > max)
798                         return (EINVAL);
799                 ASSERT(end <= max);
800         }
801         ASSERT(start <= max);
802         if (flp->l_type == F_UNLCK && flp->l_len > 0 &&
803             end == (offset_t)max) {
804                 flp->l_len = 0;
805         }
806         if (start  > end)
807                 return (EINVAL);
808         return (0);
809 }

811 static int
812 flock_get_start(vnode_t *vp, flock64_t *flp, offset_t offset, u_offset_t *start)
813 {
814         struct vattr    vattr;
815         int     error;

817         /*
818          * Determine the starting point of the request. Assume that it is
819          * a valid starting point.
820          */
821         switch (flp->l_whence) {
822         case 0:         /* SEEK_SET */
823                 *start = (u_offset_t)flp->l_start;
824                 break;
825         case 1:         /* SEEK_CUR */
826                 *start = (u_offset_t)(flp->l_start + offset);
827                 break;
828         case 2:         /* SEEK_END */
829                 vattr.va_mask = AT_SIZE;
830                 if (error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
831                         return (error);
832                 *start = (u_offset_t)(flp->l_start + (offset_t)vattr.va_size);
833                 break;
834         default:
835                 return (EINVAL);
836         }

838         return (0);
839 }

841 /*
842  * Take rctl action when the requested file descriptor is too big.
843  */
844 static void
845 fd_too_big(proc_t *p)
846 {
847         mutex_enter(&p->p_lock);
848         (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
849             p->p_rctls, p, RCA_SAFE);
850         mutex_exit(&p->p_lock);
851 }
```