```
**********************************************************
    1938 Sun Aug  9 12:47:29 2015
new/usr/src/cmd/cmd-inet/usr.bin/netstat/Makefile
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 2009 Sun Microsystems, Inc.  All rights reserved.
  23 # Use is subject to license terms.
  24 #
  25 # Copyright (c) 1990 Mentat Inc.
  26 #
  27 # cmd/cmd-inet/usr.bin/netstat/Makefile

  29 PROG=           netstat

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

  34 include ../../../Makefile.cmd
  35 include ../../Makefile.cmd-inet

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

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

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

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

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

  53 .KEEP_STATE:

  55 all: $(PROG) $(NPROG)

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

  59 $(PROG): $(OBJS)
  60         $(LINK.c) $(OBJS) -o $@ $(LDLIBS)
```



```
  61         $(POST_PROCESS)

  63 %.o : $(STATCOMMONDIR)/%.c
  64         $(COMPILE.c) -o $@ $<
  65         $(POST_PROCESS_O)

  67 install: all $(ROOTPROG)

  69 clean:
  70         $(RM) $(OBJS)

  72 lint:   lint_SRCS

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

```
**********************************************************
  204530 Sun Aug  9 12:47:31 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(uint32_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_node_list_hdr_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_node_list_hdr_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_node_list_hdr_t        *cph;
#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 *)cph +
4901                             cph->cph_tot_size)) {
4902                                 aptr = v4a == NULL ? NULL : *v4a++;
4903                                 /* LINTED: (note 1) */
4904                                 cph = (conn_pid_node_list_hdr_t *)
4905                                     ((char *)tp + tcpConnEntrySize);
4906                                 print_hdr_once_v4 = tcp_report_item_v4(tp,
4907                                     cph, 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 *)cph +
4915                             cph->cph_tot_size)) {
4916                                 aptr = v6a == NULL ? NULL : *v6a++;
4917                                 /* LINTED: (note 1) */
4918                                 cph = (conn_pid_node_list_hdr_t *)
4919                                     ((char *)tp6 + tcp6ConnEntrySize);
4920                                 print_hdr_once_v6 = tcp_report_item_v6(tp6,
4921                                     cph, 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,
4937     conn_pid_node_list_hdr_t * cph, boolean_t first,
4294 tcp_report_item_v4(const mib2_tcpConnEntry_t *tp, boolean_t first,
4938     const mib2_transportMLPEntry_t *attr)
4939 {
4940         /*
4941          * lname and fname below are for the hostname as well as the portname
4942          * There is no limit on portname length so we assume MAXHOSTNAMELEN
4943          * as the limit
4944          */
4945         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4946         char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];


4949 #endif /* ! codereview */
4950         if (!(Aflag || tp->tcpConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
4951                 return (first); /* Nothing to print */

4953         if (first) {
4954                 (void) printf(v4compat ? tcp_hdr_v4_compat : tcp_hdr_v4);
```

```
4955                if (Uflag)
4956                        (void) printf(Vflag ? tcp_hdr_v4_pid_verbose :
4957                                tcp_hdr_v4_pid);
4958                else
4959                        (void) printf(Vflag ? tcp_hdr_v4_verbose :
4960                                tcp_hdr_v4_normal);
4305                (void) printf(Vflag ? tcp_hdr_v4_verbose : tcp_hdr_v4_normal);
4961        }

4963        if ((!Uflag) && Vflag) {
4308        if (Vflag) {
4964                (void) printf("%-20s\n%-20s %5u %08x %08x %5u %08x %08x "
4965                    "%5u %5u %s\n",
4966                    pr_ap(tp->tcpConnLocalAddress,
4967                    tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4968                    pr_ap(tp->tcpConnRemAddress,
4969                    tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4970                    tp->tcpConnEntryInfo.ce_swnd,
4971                    tp->tcpConnEntryInfo.ce_snxt,
4972                    tp->tcpConnEntryInfo.ce_suna,
4973                    tp->tcpConnEntryInfo.ce_rwnd,
4974                    tp->tcpConnEntryInfo.ce_rnxt,
4975                    tp->tcpConnEntryInfo.ce_rack,
4976                    tp->tcpConnEntryInfo.ce_rto,
4977                    tp->tcpConnEntryInfo.ce_mss,
4978                    mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4979        } else if ((!Uflag) && (!Vflag)) {
4324        } else {
4980                int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
4981                    (int)tp->tcpConnEntryInfo.ce_suna - 1;
4982                int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
4983                    (int)tp->tcpConnEntryInfo.ce_rack;

4985                (void) printf("%-20s %-20s %5u %6d %5u %6d %s\n",
4986                    pr_ap(tp->tcpConnLocalAddress,
4987                    tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4988                    pr_ap(tp->tcpConnRemAddress,
4989                    tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4990                    tp->tcpConnEntryInfo.ce_swnd,
4991                    (sq >= 0) ? sq : 0,
4992                    tp->tcpConnEntryInfo.ce_rwnd,
4993                    (rq >= 0) ? rq : 0,
4994                    mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4995        } else if (Uflag && Vflag) {
4996                int i = 0;
4997                conn_pid_node_t *cpn = cph->cph_cpns;
4998                proc_info_t *pinfo;

5000                do {
5001                        int pid = (cph->cph_pn_cnt)?cpn->cpn_pid:0;
5002                        pinfo = get_proc_info(cpn->cpn_pid);

5004                        (void) printf("%-20s\n%-20s %7u %08x %08x %7u %08x %08x
5005                            "%5u %5u %-11s %-8.8s %6u %s\n",
5006                            pr_ap(tp->tcpConnLocalAddress,
5007                            tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
5008                            pr_ap(tp->tcpConnRemAddress,
5009                            tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
5010                            tp->tcpConnEntryInfo.ce_swnd,
5011                            tp->tcpConnEntryInfo.ce_snxt,
5012                            tp->tcpConnEntryInfo.ce_suna,
5013                            tp->tcpConnEntryInfo.ce_rwnd,
5014                            tp->tcpConnEntryInfo.ce_rnxt,
5015                            tp->tcpConnEntryInfo.ce_rack,
5016                            tp->tcpConnEntryInfo.ce_rto,
5017                            tp->tcpConnEntryInfo.ce_mss,
```



```
5018                            mitcp_state(tp->tcpConnEntryInfo.ce_state, attr),
5019                            pinfo->pr_user, pid, pinfo->pr_psargs);
5020                        i++; cpn++;
5021                } while (i < cph->cph_pn_cnt);
5022        } else if (Uflag && (!Vflag)) {
5023                int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
5024                    (int)tp->tcpConnEntryInfo.ce_suna - 1;
5025                int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
5026                   (int)tp->tcpConnEntryInfo.ce_rack;
5027                int i = 0;
5028                conn_pid_node_t *cpn = cph->cph_cpns;
5029                proc_info_t *pinfo;

5031                do {
5032                        int pid = (cph->cph_pn_cnt)?cpn->cpn_pid:0;
5033                        pinfo = get_proc_info(cpn->cpn_pid);

5035                        (void) printf("%-20s %-20s %-8.8s %6u %-13.13s %7u %6d %
5036                            pr_ap(tp->tcpConnLocalAddress,
5037                            tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
5038                            pr_ap(tp->tcpConnRemAddress,
5039                            tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
5040                            pinfo->pr_user, pid, pinfo->pr_fname,
5041                            tp->tcpConnEntryInfo.ce_swnd,
5042                            (sq >= 0) ? sq : 0,
5043                            tp->tcpConnEntryInfo.ce_rwnd,
5044                            (rq >= 0) ? rq : 0,
5045                            mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));

5047                        i++; cpn++;
5048                } while (i < cph->cph_pn_cnt);
5049 #endif /* ! codereview */
5050        }

5052        print_transport_label(attr);

5054        return (B_FALSE);
5055 }

5057 static boolean_t
5058 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6,
5059     conn_pid_node_list_hdr_t *cph, boolean_t first,
4340 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6, boolean_t first,
5060     const mib2_transportMLPEntry_t *attr)
5061 {
5062        /*
5063         * lname and fname below are for the hostname as well as the portname
5064         * There is no limit on portname length so we assume MAXHOSTNAMELEN
5065         * as the limit
5066         */
5067        char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5068        char    fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5069        char    ifname[LIFNAMSIZ + 1];
5070        char    *ifnamep;

5072        if (!(Aflag || tp6->tcp6ConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
5073                return (first); /* Nothing to print */

5075        if (first) {
5076                (void) printf(tcp_hdr_v6);
5077                if (Uflag)
5078                        (void) printf(Vflag ? tcp_hdr_v6_pid_verbose :
5079                                tcp_hdr_v6_pid);
5080                else
5081                        (void) printf(Vflag ? tcp_hdr_v6_verbose :
5082                                tcp_hdr_v6_normal);
```

```
4358                (void) printf(Vflag ? tcp_hdr_v6_verbose : tcp_hdr_v6_normal);
5083        }

5085        ifnamep = (tp6->tcp6ConnIfIndex != 0) ?
5086            if_indextoname(tp6->tcp6ConnIfIndex, ifname) : NULL;
5087        if (ifnamep == NULL)
5088                ifnamep = "";

5090        if ((!Uflag) && Vflag) {
4366        if (Vflag) {
5091                (void) printf("%-33s\n%-33s %5u %08x %08x %5u %08x %08x "
5092                    "%5u %5u %-11s %s\n",
5093                    pr_ap6(&tp6->tcp6ConnLocalAddress,
5094                    tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
5095                    pr_ap6(&tp6->tcp6ConnRemAddress,
5096                    tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5097                    tp6->tcp6ConnEntryInfo.ce_swnd,
5098                    tp6->tcp6ConnEntryInfo.ce_snxt,
5099                    tp6->tcp6ConnEntryInfo.ce_suna,
5100                    tp6->tcp6ConnEntryInfo.ce_rwnd,
5101                    tp6->tcp6ConnEntryInfo.ce_rnxt,
5102                    tp6->tcp6ConnEntryInfo.ce_rack,
5103                    tp6->tcp6ConnEntryInfo.ce_rto,
5104                    tp6->tcp6ConnEntryInfo.ce_mss,
5105                    mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5106                    ifnamep);
5107        } else if ((!Uflag) && (!Vflag)) {
4383        } else {
5108                int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
5109                    (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
5110                int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
5111                    (int)tp6->tcp6ConnEntryInfo.ce_rack;

5113                (void) printf("%-33s %-33s %5u %6d %5u %6d %-11s %s\n",
5114                    pr_ap6(&tp6->tcp6ConnLocalAddress,
5115                    tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
5116                    pr_ap6(&tp6->tcp6ConnRemAddress,
5117                    tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5118                    tp6->tcp6ConnEntryInfo.ce_swnd,
5119                    (sq >= 0) ? sq : 0,
5120                    tp6->tcp6ConnEntryInfo.ce_rwnd,
5121                    (rq >= 0) ? rq : 0,
5122                    mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5123                    ifnamep);
5124        } else if (Uflag && Vflag) {
5125                int i = 0;
5126                conn_pid_node_t *cpn = cph->cph_cpns;
5127                proc_info_t *pinfo;

5129                do {
5130                        int pid = (cph->cph_pn_cnt)?cpn->cpn_pid:0;
5131                        pinfo = get_proc_info(cpn->cpn_pid);

5133                        (void) printf("%-33s\n%-33s %7u %08x %08x %7u %08x %08x
5134                            "%5u %5u %-11s %-5.5s %-8.8s %6u %s\n",
5135                            pr_ap6(&tp6->tcp6ConnLocalAddress,
5136                            tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)
5137                            pr_ap6(&tp6->tcp6ConnRemAddress,
5138                            tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5139                            tp6->tcp6ConnEntryInfo.ce_swnd,
5140                            tp6->tcp6ConnEntryInfo.ce_snxt,
5141                            tp6->tcp6ConnEntryInfo.ce_suna,
5142                            tp6->tcp6ConnEntryInfo.ce_rwnd,
5143                            tp6->tcp6ConnEntryInfo.ce_rnxt,
5144                            tp6->tcp6ConnEntryInfo.ce_rack,
5145                            tp6->tcp6ConnEntryInfo.ce_rto,
```



```
5146                            tp6->tcp6ConnEntryInfo.ce_mss,
5147                            mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5148                            ifnamep, pinfo->pr_user, pid, pinfo->pr_psargs);
5149                        i++; cpn++;
5150                } while (i < cph->cph_pn_cnt);
5151        } else if (Uflag && (!Vflag)) {
5152                int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
5153                    (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
5154                int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
5155                    (int)tp6->tcp6ConnEntryInfo.ce_rack;
5156                int i = 0;
5157                conn_pid_node_t *cpn = cph->cph_cpns;
5158                proc_info_t *pinfo;

5160                do {
5161                        int pid = (cph->cph_pn_cnt)?cpn->cpn_pid:0;
5162                        pinfo = get_proc_info(cpn->cpn_pid);

5164                        (void) printf("%-33s %-33s %-8.8s %6u %-14.14s %7d %6u %
5165                            pr_ap6(&tp6->tcp6ConnLocalAddress,
5166                            tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)
5167                            pr_ap6(&tp6->tcp6ConnRemAddress,
5168                            tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5169                            pinfo->pr_user, pid, pinfo->pr_fname,
5170                            tp6->tcp6ConnEntryInfo.ce_swnd,
5171                            (sq >= 0) ? sq : 0,
5172                            tp6->tcp6ConnEntryInfo.ce_rwnd,
5173                            (rq >= 0) ? rq : 0,
5174                            mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5175                            ifnamep);

5177                        i++; cpn++;
5178                } while (i < cph->cph_pn_cnt);
5179 #endif /* ! codereview */
5180        }

5182        print_transport_label(attr);

5184        return (B_FALSE);
5185 }

5187 /* ------------------------------ UDP_REPORT------------------------------ */

5189 static boolean_t udp_report_item_v4(const mib2_udpEntry_t *ude,
5190     conn_pid_node_list_hdr_t *cph, boolean_t first,
5191     const mib2_transportMLPEntry_t *attr);
4400     boolean_t first, const mib2_transportMLPEntry_t *attr);
5192 static boolean_t udp_report_item_v6(const mib2_udp6Entry_t *ude6,
5193     conn_pid_node_list_hdr_t *cph, boolean_t first,
5194     const mib2_transportMLPEntry_t *attr);
4402     boolean_t first, const mib2_transportMLPEntry_t *attr);

5196 static const char udp_hdr_v4[] =
5197 "   Local Address        Remote Address      State\n"
5198 "-------------------- -------------------- ----------\n";
5199 static const char udp_hdr_v4_pid[] =
5200 "   Local Address        Remote Address      User    Pid   "
5201 "   Command       State\n"
5202 "-------------------- -------------------- -------- ------ "
5203 "-------------- ----------\n";
5204 static const char udp_hdr_v4_pid_verbose[] =
5205 "   Local Address        Remote Address      User    Pid     State    "
5206 "   Command\n"
5207 "-------------------- -------------------- -------- ------ ---------- "
5208 "---------------\n";
5209 #endif /* ! codereview */
```

```
5211 static const char udp_hdr_v6[] =
5212 "   Local Address                        Remote Address                "
5213 "  State      If\n"
5214 "-------------------------------- -------------------------------- "
5215 "---------- -----\n";
5216 static const char udp_hdr_v6_pid[] =
5217 "   Local Address                        Remote Address                "
5218 "  User    Pid      Command       State      If\n"
5219 "-------------------------------- -------------------------------- "
5220 "-------- ------ -------------- ---------- -----\n";
5221 static const char udp_hdr_v6_pid_verbose[] =
5222 "   Local Address                        Remote Address                "
5223 "  User    Pid     State      If     Command\n"
5224 "-------------------------------- -------------------------------- "
5225 "-------- ------ ---------- ----- ---------------\n";

5227 #endif /* ! codereview */

5229 static void
5230 udp_report(const mib_item_t *item)
5231 {
5232         int                             jtemp = 0;
5233         boolean_t                       print_hdr_once_v4 = B_TRUE;
5234         boolean_t                       print_hdr_once_v6 = B_TRUE;
5235         mib2_udpEntry_t                 *ude;
5236         mib2_udp6Entry_t                *ude6;
5237         mib2_transportMLPEntry_t        **v4_attrs, **v6_attrs;
5238         mib2_transportMLPEntry_t        **v4a, **v6a;
5239         mib2_transportMLPEntry_t        *aptr;
5240         conn_pid_node_list_hdr_t        *cph;
5241 #endif /* ! codereview */

5243         if (!protocol_selected(IPPROTO_UDP))
5244                 return;

5246         /*
5247          * Preparation pass: the kernel returns separate entries for UDP
5248          * connection table entries and Multilevel Port attributes.  We loop
5249          * through the attributes first and set up an array for each address
5250          * family.
5251          */
5252         v4_attrs = family_selected(AF_INET) && RSECflag ?
5253             gather_attrs(item, MIB2_UDP, MIB2_UDP_ENTRY, udpEntrySize) : NULL;
5254         v6_attrs = family_selected(AF_INET6) && RSECflag ?
5255             gather_attrs(item, MIB2_UDP6, MIB2_UDP6_ENTRY, udp6EntrySize) :
5256             NULL;

5258         v4a = v4_attrs;
5259         v6a = v6_attrs;
5260         /* 'for' loop 1: */
5261         for (; item; item = item->next_item) {
5262                 if (Xflag) {
5263                         (void) printf("\n--- Entry %d ---\n", ++jtemp);
5264                         (void) printf("Group = %d, mib_id = %d, "
5265                             "length = %d, valp = 0x%p\n",
5266                             item->group, item->mib_id,
5267                             item->length, item->valp);
5268                 }
5269                 if (!((item->group == MIB2_UDP &&
5270                     item->mib_id == MIB2_UDP_ENTRY) ||
5271                     (item->group == MIB2_UDP6 &&
5272                     item->mib_id == MIB2_UDP6_ENTRY) ||
5273                     (item->group == MIB2_UDP &&
5274                     item->mib_id == EXPER_XPORT_PROC_INFO) ||
5275                     (item->group == MIB2_UDP6 &&
```



```
5276                     item->mib_id == EXPER_XPORT_PROC_INFO)))
4407                     item->mib_id == MIB2_UDP6_ENTRY)))
5277                         continue; /* 'for' loop 1 */

5279                 if (item->group == MIB2_UDP && !family_selected(AF_INET))
5280                         continue; /* 'for' loop 1 */
5281                 else if (item->group == MIB2_UDP6 && !family_selected(AF_INET6))
5282                         continue; /* 'for' loop 1 */

5284                 /*      xxx.xxx.xxx.xxx,pppp  sss... */
5285                 if ((!Uflag) && item->group == MIB2_UDP &&
5286                     item->mib_id == MIB2_UDP_ENTRY) {
4416                 if (item->group == MIB2_UDP) {
5287                         for (ude = (mib2_udpEntry_t *)item->valp;
5288                             (char *)ude < (char *)item->valp + item->length;
5289                             /* LINTED: (note 1) */
5290                             ude = (mib2_udpEntry_t *)((char *)ude +
5291                             udpEntrySize)) {
5292                                 aptr = v4a == NULL ? NULL : *v4a++;
5293                                 print_hdr_once_v4 = udp_report_item_v4(ude,
5294                                     NULL, print_hdr_once_v4, aptr);
4424                                     print_hdr_once_v4, aptr);
5295                         }
5296                 } else if ((!Uflag) && item->group == MIB2_UDP6 &&
5297                     item->mib_id == MIB2_UDP6_ENTRY) {
4426                 } else {
5298                         for (ude6 = (mib2_udp6Entry_t *)item->valp;
5299                             (char *)ude6 < (char *)item->valp + item->length;
5300                             /* LINTED: (note 1) */
5301                             ude6 = (mib2_udp6Entry_t *)((char *)ude6 +
5302                             udp6EntrySize)) {
5303                                 aptr = v6a == NULL ? NULL : *v6a++;
5304                                 print_hdr_once_v6 = udp_report_item_v6(ude6,
5305                                     NULL, print_hdr_once_v6, aptr);
5306                         }
5307                 } else if ((Uflag) && item->group == MIB2_UDP &&
5308                     item->mib_id == EXPER_XPORT_PROC_INFO) {
5309                         for (ude = (mib2_udpEntry_t *)item->valp;
5310                             (char *)ude < (char *)item->valp + item->length;
5311                             /* LINTED: (note 1) */
5312                             ude = (mib2_udpEntry_t *)((char *)cph +
5313                             cph->cph_tot_size)) {
5314                                 aptr = v4a == NULL ? NULL : *v4a++;
5315                                 /* LINTED: (note 1) */
5316                                 cph = (conn_pid_node_list_hdr_t *)
5317                                     ((char *)ude + udpEntrySize);
5318                                 print_hdr_once_v4 = udp_report_item_v4(ude,
5319                                     cph, print_hdr_once_v4, aptr);
5320                         }
5321                 } else if ((Uflag) && item->group == MIB2_UDP6 &&
5322                     item->mib_id == EXPER_XPORT_PROC_INFO) {
5323                         for (ude6 = (mib2_udp6Entry_t *)item->valp;
5324                             (char *)ude6 < (char *)item->valp + item->length;
5325                             /* LINTED: (note 1) */
5326                             ude6 = (mib2_udp6Entry_t *)((char *)cph +
5327                             cph->cph_tot_size)) {
5328                                 aptr = v6a == NULL ? NULL : *v6a++;
5329                                 /* LINTED: (note 1) */
5330                                 cph = (conn_pid_node_list_hdr_t *)
5331                                     ((char *)ude6 + udp6EntrySize);
5332                                 print_hdr_once_v6 = udp_report_item_v6(ude6,
5333                                     cph, print_hdr_once_v6, aptr);
4434                                     print_hdr_once_v6, aptr);
5334                         }
5335                 }
5336         } /* 'for' loop 1 ends */
```

```
5337         (void) fflush(stdout);

5339         if (v4_attrs != NULL)
5340                 free(v4_attrs);
5341         if (v6_attrs != NULL)
5342                 free(v6_attrs);
5343 }

5345 static boolean_t
5346 udp_report_item_v4(const mib2_udpEntry_t *ude, conn_pid_node_list_hdr_t *cph,
5347     boolean_t first, const mib2_transportMLPEntry_t *attr)
4447 udp_report_item_v4(const mib2_udpEntry_t *ude, boolean_t first,
4448     const mib2_transportMLPEntry_t *attr)
5348 {
5349         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5350                         /* hostname + portname */

5352         if (!(Aflag || ude->udpEntryInfo.ue_state >= MIB2_UDP_connected))
5353                 return (first); /* Nothing to print */

5355         if (first) {
5356                 (void) printf(v4compat ? "\nUDP\n" : "\nUDP: IPv4\n");

5358                 if (Uflag)
5359                         (void) printf(Vflag ? udp_hdr_v4_pid_verbose :
5360                             udp_hdr_v4_pid);
5361                 else
5362 #endif /* ! codereview */
5363                         (void) printf(udp_hdr_v4);

5365 #endif /* ! codereview */
5366                 first = B_FALSE;
5367         }

5369         (void) printf("%-20s %-20s ",
4458         (void) printf("%-20s ",
5370             pr_ap(ude->udpLocalAddress, ude->udpLocalPort, "udp",
5371             lname, sizeof (lname)),
4460             lname, sizeof (lname)));
4461         (void) printf("%-20s %s\n",
5372             ude->udpEntryInfo.ue_state == MIB2_UDP_connected ?
5373             pr_ap(ude->udpEntryInfo.ue_RemoteAddress,
5374             ude->udpEntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5375             "");
5376         if (!Uflag) {
5377                 (void) printf("%s\n",
4465             "",
5378                     miudp_state(ude->udpEntryInfo.ue_state, attr));
5379         } else {
5380                 int i = 0;
5381                 conn_pid_node_t *cpn = cph->cph_cpns;
5382                 proc_info_t *pinfo;

5384                 do {
5385                         int pid = (cph->cph_pn_cnt)?cpn->cpn_pid:0;
5386                         pinfo = get_proc_info(cpn->cpn_pid);
5387                         (void) printf("%-8.8s %6u ", pinfo->pr_user, pid);

5389                         if (Vflag) {
5390                                 (void) printf("%-10.10s %s\n",
5391                                     miudp_state(ude->udpEntryInfo.ue_state,
5392                                     attr),
5393                                     pinfo->pr_psargs);
5394                         } else {
5395                                 (void) printf("%-14.14s %s\n", pinfo->pr_fname,
5396                                   miudp_state(ude->udpEntryInfo.ue_state,
```



```
5397                                     attr));
5398                         }
5399                         i++; cpn++;
5400                 } while (i < cph->cph_pn_cnt);
5401         }
5402 #endif /* ! codereview */

5404         print_transport_label(attr);

5406         return (first);
5407 }

5409 static boolean_t
5410 udp_report_item_v6(const mib2_udp6Entry_t *ude6, conn_pid_node_list_hdr_t *cph,
5411     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)
5412 {
5413         char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5414                         /* hostname + portname */
5415         char    ifname[LIFNAMSIZ + 1];
5416         const char *ifnamep;

5418         if (!(Aflag || ude6->udp6EntryInfo.ue_state >= MIB2_UDP_connected))
5419                 return (first); /* Nothing to print */

5421         if (first) {
5422                 (void) printf("\nUDP: IPv6\n");

5424                 if (Uflag)
5425                         (void) printf(Vflag ? udp_hdr_v6_pid_verbose :
5426                             udp_hdr_v6_pid);
5427                 else
5428 #endif /* ! codereview */
5429                         (void) printf(udp_hdr_v6);

5431 #endif /* ! codereview */
5432                 first = B_FALSE;
5433         }

5435         ifnamep = (ude6->udp6IfIndex != 0) ?
5436             if_indextoname(ude6->udp6IfIndex, ifname) : NULL;

5438         (void) printf("%-33s %-33s ",
4480         (void) printf("%-33s ",
5439             pr_ap6(&ude6->udp6LocalAddress,
5440             ude6->udp6LocalPort, "udp", lname, sizeof (lname)),
4482             ude6->udp6LocalPort, "udp", lname, sizeof (lname)));
4483         (void) printf("%-33s %-10s %s\n",
5441             ude6->udp6EntryInfo.ue_state == MIB2_UDP_connected ?
5442             pr_ap6(&ude6->udp6EntryInfo.ue_RemoteAddress,
5443             ude6->udp6EntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5444             "");
5445         if (!Uflag) {
5446                 (void) printf("%-10s %s\n",
4487             "",
5447                     miudp_state(ude6->udp6EntryInfo.ue_state, attr),
5448                     ifnamep == NULL ? "" : ifnamep);
5449         } else {
5450                 int i = 0;
5451                 conn_pid_node_t *cpn = cph->cph_cpns;
5452                 proc_info_t *pinfo;

5454                 do {
5455                         int pid = (cph->cph_pn_cnt)?cpn->cpn_pid:0;
5456                         pinfo = get_proc_info(cpn->cpn_pid);
```

```
5457                        (void) printf("%-8.8s %6u ", pinfo->pr_user, pid);

5459                        if (Vflag) {
5460                                (void) printf("%-10.10s %-5.5s %s\n",
5461                                    miudp_state(ude6->udp6EntryInfo.ue_state,
5462                                    attr),
5463                                    ifnamep == NULL ? "" : ifnamep,
5464                                    pinfo->pr_psargs);
5465                        } else {
5466                                (void) printf("%-14.14s %-10.10s %s\n",
5467                                    pinfo->pr_fname,
5468                                    miudp_state(ude6->udp6EntryInfo.ue_state,
5469                                    attr),
5470                                    ifnamep == NULL ? "" : ifnamep);
5471                        }
5472                        i++; cpn++;
5473                } while (i < cph->cph_pn_cnt);
5474        }
5475 #endif /* ! codereview */

5477        print_transport_label(attr);

5479        return (first);
5480 }

5482 /* ------------------------------ SCTP_REPORT------------------------------ */

5484 static const char sctp_hdr[] =
5485 "\nSCTP:";
5486 static const char sctp_hdr_normal[] =
5487 "        Local Address                 Remote Address          "
5488 "Swind  Send-Q Rwind  Recv-Q StrsI/O  State\n"
5489 "------------------------------ ------------------------------ "
5490 "------ ------ ------ ------ ------- -----------";
5491 static const char sctp_hdr_pid[] =
5492 "        Local Address                 Remote Address          "
5493 "Swind  Send-Q Rwind  Recv-Q StrsI/O   User    Pid      Command      State\n"
5494 "------------------------------ ------------------------------ ------ "
5495 "------ ------ ------ ------ -------- ------ -------------- -----------";
5496 static const char sctp_hdr_pid_verbose[] =
5497 "        Local Address                 Remote Address          "
5498 "Swind  Send-Q Rwind  Recv-Q StrsI/O   User    Pid    State         Command\n"
5499 "------------------------------ ------------------------------ ------ "
5500 "------ ------ ------ ------ -------- ------ ----------- --------------";
5501 #endif /* ! codereview */

5503 static const char *
5504 nssctp_state(int state, const mib2_transportMLPEntry_t *attr)
5505 {
5506        static char sctpsbuf[50];
5507        const char *cp;

5509        switch (state) {
5510        case MIB2_SCTP_closed:
5511                cp = "CLOSED";
5512                break;
5513        case MIB2_SCTP_cookieWait:
5514                cp = "COOKIE_WAIT";
5515                break;
5516        case MIB2_SCTP_cookieEchoed:
5517                cp = "COOKIE_ECHOED";
5518                break;
5519        case MIB2_SCTP_established:
5520                cp = "ESTABLISHED";
5521                break;
5522        case MIB2_SCTP_shutdownPending:
```



```
5523                cp = "SHUTDOWN_PENDING";
5524                break;
5525        case MIB2_SCTP_shutdownSent:
5526                cp = "SHUTDOWN_SENT";
5527                break;
5528        case MIB2_SCTP_shutdownReceived:
5529                cp = "SHUTDOWN_RECEIVED";
5530                break;
5531        case MIB2_SCTP_shutdownAckSent:
5532                cp = "SHUTDOWN_ACK_SENT";
5533                break;
5534        case MIB2_SCTP_listen:
5535                cp = "LISTEN";
5536                break;
5537        default:
5538                (void) snprintf(sctpsbuf, sizeof (sctpsbuf),
5539                    "UNKNOWN STATE(%d)", state);
5540                cp = sctpsbuf;
5541                break;
5542        }

5544        if (RSECflag && attr != NULL && attr->tme_flags != 0) {
5545                if (cp != sctpsbuf) {
5546                        (void) strlcpy(sctpsbuf, cp, sizeof (sctpsbuf));
5547                        cp = sctpsbuf;
5548                }
5549                if (attr->tme_flags & MIB2_TMEF_PRIVATE)
5550                        (void) strlcat(sctpsbuf, " P", sizeof (sctpsbuf));
5551                if (attr->tme_flags & MIB2_TMEF_SHARED)
5552                        (void) strlcat(sctpsbuf, " S", sizeof (sctpsbuf));
5553        }

5555        return (cp);
5556 }

5558 static const mib2_sctpConnRemoteEntry_t *
5559 sctp_getnext_rem(const mib_item_t **itemp,
5560     const mib2_sctpConnRemoteEntry_t *current, uint32_t associd)
5561 {
5562        const mib_item_t *item = *itemp;
5563        const mib2_sctpConnRemoteEntry_t        *sre;

5565        for (; item != NULL; item = item->next_item, current = NULL) {
5566                if (!(item->group == MIB2_SCTP &&
5567                    item->mib_id == MIB2_SCTP_CONN_REMOTE)) {
5568                        continue;
5569                }

5571                if (current != NULL) {
5572                        /* LINTED: (note 1) */
5573                        sre = (const mib2_sctpConnRemoteEntry_t *)
5574                            ((const char *)current + sctpRemoteEntrySize);
5575                } else {
5576                        sre = item->valp;
5577                }
5578                for (; (char *)sre < (char *)item->valp + item->length;
5579                    /* LINTED: (note 1) */
5580                    sre = (const mib2_sctpConnRemoteEntry_t *)
5581                    ((const char *)sre + sctpRemoteEntrySize)) {
5582                        if (sre->sctpAssocId != associd) {
5583                                continue;
5584                        }
5585                        *itemp = item;
5586                        return (sre);
5587                }
5588        }
```

```
5589         *itemp = NULL;
5590         return (NULL);
5591 }

5593 static const mib2_sctpConnLocalEntry_t *
5594 sctp_getnext_local(const mib_item_t **itemp,
5595     const mib2_sctpConnLocalEntry_t *current, uint32_t associd)
5596 {
5597         const mib_item_t *item = *itemp;
5598         const mib2_sctpConnLocalEntry_t *sle;

5600         for (; item != NULL; item = item->next_item, current = NULL) {
5601                 if (!(item->group == MIB2_SCTP &&
5602                     item->mib_id == MIB2_SCTP_CONN_LOCAL)) {
5603                         continue;
5604                 }

5606                 if (current != NULL) {
5607                         /* LINTED: (note 1) */
5608                         sle = (const mib2_sctpConnLocalEntry_t *)
5609                             ((const char *)current + sctpLocalEntrySize);
5610                 } else {
5611                         sle = item->valp;
5612                 }
5613                 for (; (char *)sle < (char *)item->valp + item->length;
5614                     /* LINTED: (note 1) */
5615                     sle = (const mib2_sctpConnLocalEntry_t *)
5616                     ((const char *)sle + sctpLocalEntrySize)) {
5617                         if (sle->sctpAssocId != associd) {
5618                                 continue;
5619                         }
5620                         *itemp = item;
5621                         return (sle);
5622                 }
5623         }
5624         *itemp = NULL;
5625         return (NULL);
5626 }

5628 static void
5629 sctp_pr_addr(int type, char *name, int namelen, const in6_addr_t *addr,
5630     int port)
5631 {
5632         ipaddr_t        v4addr;
5633         in6_addr_t      v6addr;

5635         /*
5636          * Address is either a v4 mapped or v6 addr. If
5637          * it's a v4 mapped, convert to v4 before
5638          * displaying.
5639          */
5640         switch (type) {
5641         case MIB2_SCTP_ADDR_V4:
5642                 /* v4 */
5643                 v6addr = *addr;

5645                 IN6_V4MAPPED_TO_IPADDR(&v6addr, v4addr);
5646                 if (port > 0) {
5647                         (void) pr_ap(v4addr, port, "sctp", name, namelen);
5648                 } else {
5649                         (void) pr_addr(v4addr, name, namelen);
5650                 }
5651                 break;

5653         case MIB2_SCTP_ADDR_V6:
5654                 /* v6 */
```



```
5655                 if (port > 0) {
5656                         (void) pr_ap6(addr, port, "sctp", name, namelen);
5657                 } else {
5658                         (void) pr_addr6(addr, name, namelen);
5659                 }
5660                 break;

5662         default:
5663                 (void) snprintf(name, namelen, "<unknown addr type>");
5664                 break;
5665         }
5666 }

5668 static boolean_t
5669 sctp_conn_report_item(const mib_item_t *head, conn_pid_node_list_hdr_t * cph,
5670     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,
5671     const mib2_transportMLPEntry_t *attr)
5672 {
5673         char            lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5674         char            fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5675         const mib2_sctpConnRemoteEntry_t        *sre = NULL;
5676         const mib2_sctpConnLocalEntry_t *sle = NULL;
5677         const mib_item_t *local = head;
5678         const mib_item_t *remote = head;
5679         uint32_t        id = sp->sctpAssocId;
5680         boolean_t       printfirst = B_TRUE;

5682         if (print_sctp_hdr == B_TRUE) {
5683                 (void) puts(sctp_hdr);
5684                 if (Uflag)
5685                         (void) puts(Vflag? sctp_hdr_pid_verbose: sctp_hdr_pid);
5686                 else
5687                         (void) puts(sctp_hdr_normal);

5689                 print_sctp_hdr = B_FALSE;
5690         }

5692 #endif /* ! codereview */
5693         sctp_pr_addr(sp->sctpAssocRemPrimAddrType, fname, sizeof (fname),
5694             &sp->sctpAssocRemPrimAddr, sp->sctpAssocRemPort);
5695         sctp_pr_addr(sp->sctpAssocRemPrimAddrType, lname, sizeof (lname),
5696             &sp->sctpAssocLocPrimAddr, sp->sctpAssocLocalPort);

5698         if (Uflag) {
5699                 int i = 0;
5700                 conn_pid_node_t *cpn = cph->cph_cpns;
5701                 proc_info_t *pinfo;

5703                 do {
5704                         int pid = (cph->cph_pn_cnt)?cpn->cpn_pid:0;
5705                         pinfo = get_proc_info(cpn->cpn_pid);
5706                         (void) printf("%-31s %-31s %6u %6d %6u %6d %3d/%-3d %-8.
5707                             lname, fname,
5708                             sp->sctpConnEntryInfo.ce_swnd,
5709                             sp->sctpConnEntryInfo.ce_sendq,
5710                             sp->sctpConnEntryInfo.ce_rwnd,
5711                             sp->sctpConnEntryInfo.ce_recvq,
5712                             sp->sctpAssocInStreams,
5713                             sp->sctpAssocOutStreams,
5714                             pinfo->pr_user, pid);

5716                         if (Vflag) {
5717                                 (void) printf("%-11.11s %s\n",
5718                                     nssctp_state(sp->sctpAssocState, attr),
```

```
5719                                    pinfo->pr_psargs);
5720                            } else {
5721                                   (void) printf("%-14.14s %s\n",
5722                                       pinfo->pr_fname,
5723                                       nssctp_state(sp->sctpAssocState, attr));
5724                            }

5726                            i++; cpn++;
5727                    } while (i < cph->cph_pn_cnt);

5729            } else {

5731 #endif /* ! codereview */
5732                    (void) printf("%-31s %-31s %6u %6d %6u %6d %3d/%-3d %s\n",
5733                        lname, fname,
5734                        sp->sctpConnEntryInfo.ce_swnd,
5735                        sp->sctpConnEntryInfo.ce_sendq,
5736                        sp->sctpConnEntryInfo.ce_rwnd,
5737                        sp->sctpConnEntryInfo.ce_recvq,
5738                        sp->sctpAssocInStreams, sp->sctpAssocOutStreams,
5739                        nssctp_state(sp->sctpAssocState, attr));
5740            }
5741 #endif /* ! codereview */

5743            print_transport_label(attr);

5745            if (!Vflag) {
5746                    return (print_sctp_hdr);
4503                    return;
5747            }

5749            /* Print remote addresses/local addresses on following lines */
5750            while ((sre = sctp_getnext_rem(&remote, sre, id)) != NULL) {
5751                    if (!IN6_ARE_ADDR_EQUAL(&sre->sctpAssocRemAddr,
5752                        &sp->sctpAssocRemPrimAddr)) {
5753                            if (printfirst == B_TRUE) {
5754                                    (void) fputs("\t<Remote: ", stdout);
5755                                    printfirst = B_FALSE;
5756                            } else {
5757                                    (void) fputs(", ", stdout);
5758                            }
5759                            sctp_pr_addr(sre->sctpAssocRemAddrType, fname,
5760                                sizeof (fname), &sre->sctpAssocRemAddr, -1);
5761                            if (sre->sctpAssocRemAddrActive == MIB2_SCTP_ACTIVE) {
5762                                    (void) fputs(fname, stdout);
5763                            } else {
5764                                    (void) printf("(%s)", fname);
5765                            }
5766                    }
5767            }
5768            if (printfirst == B_FALSE) {
5769                    (void) puts(">");
5770                    printfirst = B_TRUE;
5771            }
5772            while ((sle = sctp_getnext_local(&local, sle, id)) != NULL) {
5773                    if (!IN6_ARE_ADDR_EQUAL(&sle->sctpAssocLocalAddr,
5774                        &sp->sctpAssocLocPrimAddr)) {
5775                            if (printfirst == B_TRUE) {
5776                                    (void) fputs("\t<Local: ", stdout);
5777                                    printfirst = B_FALSE;
5778                            } else {
5779                                    (void) fputs(", ", stdout);
5780                            }
5781                            sctp_pr_addr(sle->sctpAssocLocalAddrType, lname,
5782                                sizeof (lname), &sle->sctpAssocLocalAddr, -1);
5783                            (void) fputs(lname, stdout);
```



```
5784                    }
5785            }
5786            if (printfirst == B_FALSE) {
5787                    (void) puts(">");
5788            }

5790            return (print_sctp_hdr);
5791 #endif /* ! codereview */
5792 }

5794 static void
5795 sctp_report(const mib_item_t *item)
5796 {
5797            const mib_item_t                *head;
5798            const mib2_sctpConnEntry_t      *sp;
5799            boolean_t                       print_sctp_hdr_once = B_TRUE;
4546            boolean_t               first = B_TRUE;
5800            mib2_transportMLPEntry_t        **attrs, **aptr;
5801            mib2_transportMLPEntry_t        *attr;
5802            conn_pid_node_list_hdr_t        *cph;
5803 #endif /* ! codereview */

5805            /*
5806             * Preparation pass: the kernel returns separate entries for SCTP
5807             * connection table entries and Multilevel Port attributes.  We loop
5808             * through the attributes first and set up an array for each address
5809             * family.
5810             */
5811            attrs = RSECflag ?
5812                gather_attrs(item, MIB2_SCTP, MIB2_SCTP_CONN, sctpEntrySize) :
5813                NULL;

5815            aptr = attrs;
5816            head = item;
5817            for (; item != NULL; item = item->next_item) {

5819                    if (!((item->group == MIB2_SCTP &&
5820                        item->mib_id == MIB2_SCTP_CONN) ||
5821                        (item->group == MIB2_SCTP &&
5822                        item->mib_id == EXPER_XPORT_PROC_INFO)))
4549                    if (!(item->group == MIB2_SCTP &&
4550                        item->mib_id == MIB2_SCTP_CONN))
5823                            continue;

5825                    if ((!Uflag) && item->group == MIB2_SCTP
5826                        && item->mib_id == MIB2_SCTP_CONN) {
5827 #endif /* ! codereview */
5828                            for (sp = item->valp;
5829                                (char *)sp < (char *)item->valp + item->length;
5830                                /* LINTED: (note 1) */
5831                                sp = (mib2_sctpConnEntry_t *)((char *)sp + sctpEntry
5832                                    if (!(Aflag ||
5833                                      sp->sctpAssocState >= MIB2_SCTP_established))
5834                                          continue;
5835 #endif /* ! codereview */
5836                                    attr = aptr == NULL ? NULL : *aptr++;
5837                                    print_sctp_hdr_once = sctp_conn_report_item(head
5838                                                                print_sctp_hdr_once, sp,
5839                                                                attr);
5840                            }
5841                    } else if ((Uflag) && item->group == MIB2_SCTP &&
5842                        item->mib_id == EXPER_XPORT_PROC_INFO) {
5843                            for (sp = (mib2_sctpConnEntry_t *)item->valp;
5844                                (char *)sp < (char *)item->valp + item->length;
5845                                /* LINTED: (note 1) */
5846                                sp = (mib2_sctpConnEntry_t *)((char *)cph +
```

```
5847                            cph->cph_tot_size)) {
5848                                /* LINTED: (note 1) */
5849                                cph = (conn_pid_node_list_hdr_t *)
5850                                        ((char *)sp + sctpEntrySize);
5851                                if (!(Aflag ||
5852                                  sp->sctpAssocState >= MIB2_SCTP_established))
5853                                        continue;
5854                                attr = aptr == NULL ? NULL : *aptr++;
5855                                print_sctp_hdr_once =
5856                                    sctp_conn_report_item(head, cph,
5857                                        print_sctp_hdr_once, sp, attr);
4553                        if (Aflag ||
4554                            sp->sctpAssocState >= MIB2_SCTP_established) {
4555                                if (first == B_TRUE) {
4556                                        (void) puts(sctp_hdr);
4557                                        (void) puts(sctp_hdr_normal);
4558                                        first = B_FALSE;
4559                                }
4560                                sctp_conn_report_item(head, sp, attr);
5858                        }
5859                }
5860        }
5861        if (attrs != NULL)
5862                free(attrs);
5863 }
```

unchanged_portion_omitted

```
6844 /*
6845  * get proc info (psinfo_t) given pid. It doesn't return NULL.
6846  */

6848 proc_info_t *
6849 get_proc_info(uint32_t pid)
6850 {
6851         static uint32_t saved_pid = 0;
6852         static proc_info_t saved_proc_info;
6853         static proc_info_t unknown_proc_info = {"<unknown>","",""};
6854         static psinfo_t pinfo;
6855         char path[128];
6856         int fd;

6858         /* hardcode pid = 0 */
6859         if (pid == 0) {
6860                 saved_proc_info.pr_user = "root";
6861                 saved_proc_info.pr_fname = "sched";
6862                 saved_proc_info.pr_psargs = "sched";
6863                 saved_pid = 0;
6864                 return &saved_proc_info;
6865         }

6867         if (pid == saved_pid)
6868                 return &saved_proc_info;
6869         if ((snprintf(path, 128, "/proc/%u/psinfo",pid) > 0) &&
6870             ((fd = open(path, O_RDONLY)) != -1)) {
6871                 if (read(fd, &pinfo, sizeof(pinfo)) == sizeof(pinfo)){
6872                         saved_proc_info.pr_user = get_username(pinfo.pr_uid);
6873                         saved_proc_info.pr_fname = pinfo.pr_fname;
6874                         saved_proc_info.pr_psargs = pinfo.pr_psargs;
6875                         saved_pid = pid;
6876                         (void) close(fd);
6877                         return &saved_proc_info;
6878                 } else {
6879                         (void) close(fd);
6880                 }
6881         }
```



```
6883         return (&unknown_proc_info);
6884 }

6886 /*
6887  * get username given uid. It doesn't return NULL.
6888  */

6890 static char *
6891 get_username(uid_t u)
6892 {
6893         static uid_t saved_uid = UINT_MAX;
6894         static char  saved_username[128];
6895         struct passwd *pw = NULL;
6896         if (u == UINT_MAX)
6897                 return "<unknown>";
6898         if (u == saved_uid && saved_username[0] != '\0')
6899                 return  (saved_username);
6900         setpwent();
6901         if ((pw = getpwuid(u)) != NULL)
6902                 (void) strlcpy(saved_username, pw->pw_name, 128);
6903         else
6904                 (void) snprintf(saved_username, 128, "%u", u);
6905         saved_uid = u;
6906         return saved_username;
6907 }

6909 /*
6910 #endif /* ! codereview */
6911  * print the usage line
6912  */
6913 static void
6914 usage(char *cmdname)
6915 {
6916         (void) fprintf(stderr, "usage: %s [-anuv] [-f address_family] "
5548         (void) fprintf(stderr, "usage: %s [-anv] [-f address_family] "
6917             "[-T d|u]\n", cmdname);
6918         (void) fprintf(stderr, "       %s [-n] [-f address_family] "
6919             "[-P protocol] [-T d|u] [-g | -p | -s [interval [count]]]\n",
6920             cmdname);
6921         (void) fprintf(stderr, "       %s -m [-v] [-T d|u] "
6922             "[interval [count]]\n", cmdname);
6923         (void) fprintf(stderr, "       %s -i [-I interface] [-an] "
6924             "[-f address_family] [-T d|u] [interval [count]]\n", cmdname);
6925         (void) fprintf(stderr, "       %s -r [-anv] "
6926             "[-f address_family|filter] [-T d|u]\n", cmdname);
6927         (void) fprintf(stderr, "       %s -M [-ns] [-f address_family] "
6928             "[-T d|u]\n", cmdname);
6929         (void) fprintf(stderr, "       %s -D [-I interface] "
6930             "[-f address_family] [-T d|u]\n", cmdname);
6931         exit(EXIT_FAILURE);
6932 }

6934 /*
6935  * fatal: print error message to stderr and
6936  * call exit(errcode)
6937  */
6938 /*PRINTFLIKE2*/
6939 static void
6940 fatal(int errcode, char *format, ...)
6941 {
6942         va_list argp;

6944         if (format == NULL)
6945                 return;

6947         va_start(argp, format);
```

```
6948         (void) vfprintf(stderr, format, argp);
6949         va_end(argp);

6951         exit(errcode);
6952 }


6955 /* -------------------UNIX Domain Sockets Report--------------------------- */


6958 #define NO_ADDR         "                                       "
6959 #define SO_PAIR         " (socketpair)                          "

6961 static char             *typetoname(t_scalar_t);
6962 static boolean_t        uds_report_item(struct sockinfo *, boolean_t);


6965 static char uds_hdr[] = "\nActive UNIX domain sockets\n";

6967 static char uds_hdr_normal[] =
6968 " Type       Local Adress                           "
6969 " Remote Address\n"
6970 "---------- ---------------------------------------- "
6971 "---------------------------------------\n";

6973 static char uds_hdr_pid[] =
6974 " Type       User     Pid     Command       "
6975 " Local Address                         "
6976 " Remote Address\n"
6977 "---------- -------- ------ -------------- "
6978 "--------------------------------------- "
6979 "---------------------------------------\n";
6980 static char uds_hdr_pid_verbose[] =
6981 " Type       User     Pid    Local Address                           "
6982 " Remote Address                        Command\n"
6983 "---------- -------- ------ ---------------------------------------- "
6984 "--------------------------------------- --------------\n";

6986 /*
6987  * Print a summary of connections related to a unix protocol.
6988  */
6989 static void
6990 uds_report(kstat_ctl_t  *kc)
6991 {
6992         int             i;
6993         kstat_t         *ksp;
6994         struct sockinfo *psi;           /* ptr to current sockinfo      */
6995         boolean_t       print_uds_hdr_once = B_TRUE;

6997         if (kc == NULL) {       /* sanity check.                        */
6998                 fail(0, "uds_report: No kstat");
6999                 exit(3);
7000         }

7002         /* find the sockfs kstat:                                       */
7003         if ((ksp = kstat_lookup(kc, "sockfs", 0, "sock_unix_list")) ==
7004             (kstat_t *)NULL) {
7005                 fail(0, "kstat_data_lookup failed\n");
7006         }

7008         if (kstat_read(kc, ksp, NULL) == -1) {
7009                 fail(0, "kstat_read failed for sock_unix_list\n");
7010         }

7012         if (ksp->ks_ndata == 0) {
7013                 return;                 /* no AF_UNIX sockets found     */
```



```
7014         }

7016         /*
7017          * Having ks_data set with ks_data == NULL shouldn't happen;
7018          * If it does, the sockfs kstat is seriously broken.
7019          */
7020         if ((psi = ksp->ks_data) == NULL) {
7021                 fail(0, "uds_report: no kstat data\n");
7022         }

7024         /* for each sockinfo structure, display what we need:           */
7025         for (i = 0; i < ksp->ks_ndata; i++) {

7027                 /* process this entry */
7028                 print_uds_hdr_once = uds_report_item(psi, print_uds_hdr_once);

7030                 /* if si_size didn't get filled in, then we're done     */
7031                 if (psi->si_size == 0 ||
7032                     !IS_P2ALIGNED(psi->si_size, sizeof (psi))) {
7033                         break;
7034                 }

7036                 /* point to the next sockinfo in the array */
7037                 /* LINTED: (note 1) */
7038                 psi = (struct sockinfo *)(((char *)psi) + psi->si_size);
7039         }
7040 }

7042 static boolean_t
7043 uds_report_item(struct sockinfo *psi, boolean_t first)
7044 {
7045         int             i = 0;
7046         conn_pid_node_t *cpn;
7047         proc_info_t     *pinfo;
7048         char            *laddr, *raddr;

7050         if(first) {
7051                 (void) printf("%s", uds_hdr);
7052                 if (Uflag)
7053                         (void) printf("%s", Vflag?uds_hdr_pid_verbose:
7054                             uds_hdr_pid);
7055                 else
7056                         (void) printf("%s", uds_hdr_normal);

7058                 first = B_FALSE;
7059         }

7061         cpn = psi->si_pns;

7063         do {
7064                 int pid = (psi->si_pn_cnt)?cpn->cpn_pid:0;
7065                 pinfo = get_proc_info(cpn->cpn_pid);
7066                 raddr = laddr = NO_ADDR;

7068                 /* laddr.soa_sa:                                        */
7069                 if ((psi->si_state & SS_ISBOUND) &&
7070                     strlen(psi->si_laddr_sun_path) != 0 &&
7071                     psi->si_laddr_soa_len != 0) {
7072                         if (psi->si_faddr_noxlate) {
7073                                 laddr = SO_PAIR;
7074                         } else {
7075                                 if (psi->si_laddr_soa_len >
7076                                     sizeof (psi->si_laddr_family))
7077                                         laddr = psi->si_laddr_sun_path;
7078                         }
7079                 }
```

```
7081                /* faddr.soa_sa:                                        */
7082                if ((psi->si_state & SS_ISCONNECTED) &&
7083                    strlen(psi->si_faddr_sun_path) != 0 &&
7084                    psi->si_faddr_soa_len != 0) {

7086                        if (psi->si_faddr_noxlate) {
7087                                raddr = SO_PAIR;
7088                        } else {
7089                                if (psi->si_faddr_soa_len >
7090                                    sizeof (psi->si_faddr_family))
7091                                        raddr = psi->si_faddr_sun_path;
7092                        }
7093                }

7095                if (Uflag && Vflag) {
7096                        (void) printf("%-10.10s %-8.8s %6u "
7097                                "%-39.39s %-39.39s %s\n",
7098                                typetoname(psi->si_serv_type), pinfo->pr_user,
7099                                pid, laddr, raddr, pinfo->pr_psargs);
7100                } else if (Uflag && (!Vflag)) {
7101                        (void) printf("%-10.10s %-8.8s %6u %-14.14s"
7102                                "%-39.39s %-39.39s\n",
7103                                typetoname(psi->si_serv_type), pinfo->pr_user,
7104                                pid, pinfo->pr_fname, laddr, raddr);
7105                } else {
7106                        (void) printf("%-10.10s %s %s\n",
7107                                typetoname(psi->si_serv_type), laddr, raddr);
7108                }

7110                i++; cpn++;
7111        } while (i < psi->si_pn_cnt);

7113        return (first);
7114 }

7116 static char *
7117 typetoname(t_scalar_t type)
7118 {
7119        switch (type) {
7120        case T_CLTS:
7121                return ("dgram");

7123        case T_COTS:
7124                return ("stream");

7126        case T_COTS_ORD:
7127                return ("stream-ord");

7129        default:
7130                return ("");
7131        }
7132 #endif /* ! codereview */
7133 }
```

```
**********************************************************
    6005 Sun Aug  9 12:47:34 2015
new/usr/src/common/list/list.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_

253 uint64_t
254 list_size(list_t *list)
255 {
256         size_t sz = 0;
257         list_node_t *node;

259         node = &list->list_head;
260         while (node->list_next != &list->list_head) {
261                 sz++;
262                 node = node->list_next;
263         }
264         return (sz);
265 }
266 #endif /* ! codereview */
```

```
**********************************************************
   90098 Sun Aug  9 12:47:35 2015
new/usr/src/pkg/manifests/system-header.mf
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) 2012 by Delphix. All rights reserved.
# Copyright 2012 Nexenta Systems, Inc. All rights reserved.
# Copyright 2014 Garrett D'Amore <garrett@damore.org>
#

set name=pkg.fmri value=pkg:/system/header@$(PKGVERS)
set name=pkg.description \
    value="SunOS C/C++ header files for general development of software"
set name=pkg.summary value="SunOS Header Files"
set name=info.classification value=org.opensolaris.category.2008:System/Core
set name=variant.arch value=$(ARCH)
dir path=usr group=sys
dir path=usr/include
$(i386_ONLY)dir path=usr/include/$(ARCH64)
$(i386_ONLY)dir path=usr/include/$(ARCH64)/sys
dir path=usr/include/arpa
dir path=usr/include/asm
dir path=usr/include/ast
dir path=usr/include/bsm
dir path=usr/include/dat
dir path=usr/include/des
dir path=usr/include/gssapi
dir path=usr/include/hal
$(i386_ONLY)dir path=usr/include/ia32
$(i386_ONLY)dir path=usr/include/ia32/sys
dir path=usr/include/inet
dir path=usr/include/inet/kssl
dir path=usr/include/ipp
dir path=usr/include/ipp/ipgpc
dir path=usr/include/iso
dir path=usr/include/kerberosv5
dir path=usr/include/libpolkit
dir path=usr/include/net
dir path=usr/include/netinet
dir path=usr/include/nfs
dir path=usr/include/protocols
dir path=usr/include/rpc
dir path=usr/include/rpcsvc
```



```
dir path=usr/include/sasl
dir path=usr/include/scsi
dir path=usr/include/scsi/plugins
dir path=usr/include/scsi/plugins/ses
dir path=usr/include/scsi/plugins/ses/framework
dir path=usr/include/scsi/plugins/ses/vendor
dir path=usr/include/scsi/plugins/smp
dir path=usr/include/scsi/plugins/smp/engine
dir path=usr/include/scsi/plugins/smp/framework
dir path=usr/include/security
dir path=usr/include/sharefs
dir path=usr/include/sys
dir path=usr/include/sys/av
dir path=usr/include/sys/contract
dir path=usr/include/sys/crypto
dir path=usr/include/sys/dktp
dir path=usr/include/sys/fc4
dir path=usr/include/sys/fm
dir path=usr/include/sys/fm/cpu
dir path=usr/include/sys/fm/fs
dir path=usr/include/sys/fm/io
$(sparc_ONLY)dir path=usr/include/sys/fpu
dir path=usr/include/sys/fs
dir path=usr/include/sys/hotplug
dir path=usr/include/sys/hotplug/pci
dir path=usr/include/sys/ib
dir path=usr/include/sys/ib/adapters
dir path=usr/include/sys/ib/adapters/hermon
dir path=usr/include/sys/ib/adapters/tavor
dir path=usr/include/sys/ib/clients
dir path=usr/include/sys/ib/clients/ibd
dir path=usr/include/sys/ib/clients/of
dir path=usr/include/sys/ib/clients/of/rdma
dir path=usr/include/sys/ib/clients/of/sol_ofs
dir path=usr/include/sys/ib/clients/of/sol_ucma
dir path=usr/include/sys/ib/clients/of/sol_umad
dir path=usr/include/sys/ib/clients/of/sol_uverbs
dir path=usr/include/sys/ib/ibnex
dir path=usr/include/sys/ib/ibtl
dir path=usr/include/sys/ib/ibtl/impl
dir path=usr/include/sys/ib/mgt
dir path=usr/include/sys/ib/mgt/ibmf
dir path=usr/include/sys/iso
dir path=usr/include/sys/lvm
dir path=usr/include/sys/proc
dir path=usr/include/sys/rsm
$(i386_ONLY)dir path=usr/include/sys/sata group=sys
dir path=usr/include/sys/scsi
dir path=usr/include/sys/scsi/adapters
dir path=usr/include/sys/scsi/conf
dir path=usr/include/sys/scsi/generic
dir path=usr/include/sys/scsi/impl
dir path=usr/include/sys/scsi/targets
dir path=usr/include/sys/sysevent
dir path=usr/include/sys/tsol
dir path=usr/include/tsol
dir path=usr/include/uuid
$(sparc_ONLY)dir path=usr/include/v7
$(sparc_ONLY)dir path=usr/include/v7/sys
$(sparc_ONLY)dir path=usr/include/v9
$(sparc_ONLY)dir path=usr/include/v9/sys
dir path=usr/include/vm
dir path=usr/platform group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,A70 group=sys
$(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300 group=sys
$(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
```

```
**********************************************************
   87207 Sun Aug  9 12:47:37 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 curproc to the pid list associated with that file */
1788         if (fp->f_vnode != NULL)
1789                 (void) VOP_IOCTL(fp->f_vnode, F_FORKED, (intptr_t)curproc, FKIOC
1790                         kcred, NULL, NULL);

1792 #endif /* ! codereview */
1793         /* Fill in the attributes */
1794         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
1795                 vp = fp->f_vnode;
1796         if (vp && vp->v_type == VDOOR) {
1797                 if (VTOD(vp)->door_target == curproc)
1798                         attributes |= DOOR_LOCAL;
1799                 attributes |= VTOD(vp)->door_flags & DOOR_ATTR_MASK;
1800                 dp->d_data.d_desc.d_id = VTOD(vp)->door_index;
1801         }
1802         dp->d_attributes = attributes;
1803         return (0);
1804 }

1806 /*
1807  * Return an available thread for this server.  A NULL return value indicates
1808  * that either:
1809  *      The door has been revoked, or
1810  *      a signal was received.
1811  * The two conditions can be differentiated using DOOR_INVALID(dp).
1812  */
1813 static kthread_t *
1814 door_get_server(door_node_t *dp)
1815 {
1816         kthread_t **ktp;
1817         kthread_t *server_t;
1818         door_pool_t *pool;
1819         door_server_t *st;
1820         int signalled;

1822         disp_lock_t *tlp;
1823         cpu_t *cp;

1825         ASSERT(MUTEX_HELD(&door_knob));
```



```
1827         if (dp->door_flags & DOOR_PRIVATE)
1828                 pool = &dp->door_servers;
1829         else
1830                 pool = &dp->door_target->p_server_threads;

1832         for (;;) {
1833                 /*
1834                  * We search the thread pool, looking for a server thread
1835                  * ready to take an invocation (i.e. one which is still
1836                  * sleeping on a shuttle object).  If none are available,
1837                  * we sleep on the pool's CV, and will be signaled when a
1838                  * thread is added to the pool.
1839                  *
1840                  * This relies on the fact that once a thread in the thread
1841                  * pool wakes up, it *must* remove and add itself to the pool
1842                  * before it can receive door calls.
1843                  */
1844                 if (DOOR_INVALID(dp))
1845                         return (NULL);  /* Target has become invalid */

1847                 for (ktp = &pool->dp_threads;
1848                     (server_t = *ktp) != NULL;
1849                     ktp = &st->d_servers) {
1850                         st = DOOR_SERVER(server_t->t_door);

1852                         thread_lock(server_t);
1853                         if (server_t->t_state == TS_SLEEP &&
1854                             SOBJ_TYPE(server_t->t_sobj_ops) == SOBJ_SHUTTLE)
1855                                 break;
1856                         thread_unlock(server_t);
1857                 }
1858                 if (server_t != NULL)
1859                         break;          /* we've got a live one! */

1861                 if (!cv_wait_sig_swap_core(&pool->dp_cv, &door_knob,
1862                     &signalled)) {
1863                         /*
1864                          * If we were signaled and the door is still
1865                          * valid, pass the signal on to another waiter.
1866                          */
1867                         if (signalled && !DOOR_INVALID(dp))
1868                                 cv_signal(&pool->dp_cv);
1869                         return (NULL);  /* Got a signal */
1870                 }
1871         }

1873         /*
1874          * We've got a thread_lock()ed thread which is still on the
1875          * shuttle.  Take it off the list of available server threads
1876          * and mark it as ONPROC.  We are committed to resuming this
1877          * thread now.
1878          */
1879         tlp = server_t->t_lockp;
1880         cp = CPU;

1882         *ktp = st->d_servers;
1883         st->d_servers = NULL;
1884         /*
1885          * Setting t_disp_queue prevents erroneous preemptions
1886          * if this thread is still in execution on another processor
1887          */
1888         server_t->t_disp_queue = cp->cpu_disp;
1889         CL_ACTIVE(server_t);
1890         /*
1891          * We are calling thread_onproc() instead of
1892          * THREAD_ONPROC() because compiler can reorder
```

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

1901 /*
1902  * Put a server thread back in the pool.
1903  */
1904 static void
1905 door_release_server(door_node_t *dp, kthread_t *t)
1906 {
1907         door_server_t *st = DOOR_SERVER(t->t_door);
1908         door_pool_t *pool;

1910         ASSERT(MUTEX_HELD(&door_knob));
1911         st->d_active = NULL;
1912         st->d_caller = NULL;
1913         st->d_layout_done = 0;
1914         if (dp && (dp->door_flags & DOOR_PRIVATE)) {
1915                 ASSERT(dp->door_target == NULL ||
1916                     dp->door_target == ttoproc(t));
1917                 pool = &dp->door_servers;
1918         } else {
1919                 pool = &ttoproc(t)->p_server_threads;
1920         }

1922         st->d_servers = pool->dp_threads;
1923         pool->dp_threads = t;

1925         /* If someone is waiting for a server thread, wake him up */
1926         cv_signal(&pool->dp_cv);
1927 }

1929 /*
1930  * Remove a server thread from the pool if present.
1931  */
1932 static void
1933 door_server_exit(proc_t *p, kthread_t *t)
1934 {
1935         door_pool_t *pool;
1936         kthread_t **next;
1937         door_server_t *st = DOOR_SERVER(t->t_door);

1939         ASSERT(MUTEX_HELD(&door_knob));
1940         if (st->d_pool != NULL) {
1941                 ASSERT(st->d_pool->door_flags & DOOR_PRIVATE);
1942                 pool = &st->d_pool->door_servers;
1943         } else {
1944                 pool = &p->p_server_threads;
1945         }

1947         next = &pool->dp_threads;
1948         while (*next != NULL) {
1949                 if (*next == t) {
1950                         *next = DOOR_SERVER(t->t_door)->d_servers;
1951                         return;
1952                 }
1953                 next = &(DOOR_SERVER((*next)->t_door)->d_servers);
1954         }
1955 }

1957 /*
1958  * Lookup the door descriptor. Caller must call releasef when finished
```



```
1959  * with associated door.
1960  */
1961 static door_node_t *
1962 door_lookup(int did, file_t **fpp)
1963 {
1964         vnode_t *vp;
1965         file_t *fp;

1967         ASSERT(MUTEX_NOT_HELD(&door_knob));
1968         if ((fp = getf(did)) == NULL)
1969                 return (NULL);
1970         /*
1971          * Use the underlying vnode (we may be namefs mounted)
1972          */
1973         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
1974                 vp = fp->f_vnode;

1976         if (vp == NULL || vp->v_type != VDOOR) {
1977                 releasef(did);
1978                 return (NULL);
1979         }

1981         if (fpp)
1982                 *fpp = fp;

1984         return (VTOD(vp));
1985 }

1987 /*
1988  * The current thread is exiting, so clean up any pending
1989  * invocation details
1990  */
1991 void
1992 door_slam(void)
1993 {
1994         door_node_t *dp;
1995         door_data_t *dt;
1996         door_client_t *ct;
1997         door_server_t *st;

1999         /*
2000          * If we are an active door server, notify our
2001          * client that we are exiting and revoke our door.
2002          */
2003         if ((dt = door_my_data(0)) == NULL)
2004                 return;
2005         ct = DOOR_CLIENT(dt);
2006         st = DOOR_SERVER(dt);

2008         mutex_enter(&door_knob);
2009         for (;;) {
2010                 if (DOOR_T_HELD(ct))
2011                         cv_wait(&ct->d_cv, &door_knob);
2012                 else if (DOOR_T_HELD(st))
2013                         cv_wait(&st->d_cv, &door_knob);
2014                 else
2015                         break;                  /* neither flag is set */
2016         }
2017         curthread->t_door = NULL;
2018         if ((dp = st->d_active) != NULL) {
2019                 kthread_t *t = st->d_caller;
2020                 proc_t *p = curproc;

2022                 /* Revoke our door if the process is exiting */
2023                 if (dp->door_target == p && (p->p_flag & SEXITING)) {
2024                         door_list_delete(dp);
```

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

2033                if (t != NULL) {
2034                        /*
2035                         * Let the caller know we are gone
2036                         */
2037                        DOOR_CLIENT(t->t_door)->d_error = DOOR_EXIT;
2038                        thread_lock(t);
2039                        if (t->t_state == TS_SLEEP &&
2040                            SOBJ_TYPE(t->t_sobj_ops) == SOBJ_SHUTTLE)
2041                                setrun_locked(t);
2042                        thread_unlock(t);
2043                }
2044        }
2045        mutex_exit(&door_knob);
2046        if (st->d_pool)
2047                door_unbind_thread(st->d_pool); /* Implicit door_unbind */
2048        kmem_free(dt, sizeof (door_data_t));
2049 }

2051 /*
2052  * Set DOOR_REVOKED for all doors of the current process. This is called
2053  * on exit before all lwp's are being terminated so that door calls will
2054  * return with an error.
2055  */
2056 void
2057 door_revoke_all()
2058 {
2059        door_node_t *dp;
2060        proc_t *p = ttoproc(curthread);

2062        mutex_enter(&door_knob);
2063        for (dp = p->p_door_list; dp != NULL; dp = dp->door_list) {
2064                ASSERT(dp->door_target == p);
2065                dp->door_flags |= DOOR_REVOKED;
2066                if (dp->door_flags & DOOR_PRIVATE)
2067                        cv_broadcast(&dp->door_servers.dp_cv);
2068        }
2069        cv_broadcast(&p->p_server_threads.dp_cv);
2070        mutex_exit(&door_knob);
2071 }

2073 /*
2074  * The process is exiting, and all doors it created need to be revoked.
2075  */
2076 void
2077 door_exit(void)
2078 {
2079        door_node_t *dp;
2080        proc_t *p = ttoproc(curthread);

2082        ASSERT(p->p_lwpcnt == 1);
2083        /*
2084         * Walk the list of active doors created by this process and
2085         * revoke them all.
2086         */
2087        mutex_enter(&door_knob);
2088        for (dp = p->p_door_list; dp != NULL; dp = dp->door_list) {
2089                dp->door_target = NULL;
2090                dp->door_flags |= DOOR_REVOKED;
```



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

2098        /* Clean up the unref list */
2099        while ((dp = p->p_unref_list) != NULL) {
2100                p->p_unref_list = dp->door_ulist;
2101                dp->door_ulist = NULL;
2102                mutex_exit(&door_knob);
2103                VN_RELE(DTOV(dp));
2104                mutex_enter(&door_knob);
2105        }
2106        mutex_exit(&door_knob);
2107 }


2110 /*
2111  * The process is executing forkall(), and we need to flag threads that
2112  * are bound to a door in the child.  This will make the child threads
2113  * return an error to door_return unless they call door_unbind first.
2114  */
2115 void
2116 door_fork(kthread_t *parent, kthread_t *child)
2117 {
2118        door_data_t *pt = parent->t_door;
2119        door_server_t *st = DOOR_SERVER(pt);
2120        door_data_t *dt;

2122        ASSERT(MUTEX_NOT_HELD(&door_knob));
2123        if (pt != NULL && (st->d_pool != NULL || st->d_invbound)) {
2124                /* parent thread is bound to a door */
2125                dt = child->t_door =
2126                    kmem_zalloc(sizeof (door_data_t), KM_SLEEP);
2127                DOOR_SERVER(dt)->d_invbound = 1;
2128        }
2129 }

2131 /*
2132  * Deliver queued unrefs to appropriate door server.
2133  */
2134 static int
2135 door_unref(void)
2136 {
2137        door_node_t     *dp;
2138        static door_arg_t unref_args = { DOOR_UNREF_DATA, 0, 0, 0, 0, 0 };
2139        proc_t *p = ttoproc(curthread);

2141        /* make sure there's only one unref thread per process */
2142        mutex_enter(&door_knob);
2143        if (p->p_unref_thread) {
2144                mutex_exit(&door_knob);
2145                return (set_errno(EALREADY));
2146        }
2147        p->p_unref_thread = 1;
2148        mutex_exit(&door_knob);

2150        (void) door_my_data(1);                 /* create info, if necessary */

2152        for (;;) {
2153                mutex_enter(&door_knob);

2155                /* Grab a queued request */
2156                while ((dp = p->p_unref_list) == NULL) {
```

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

2172                (void) door_upcall(DTOV(dp), &unref_args, NULL, SIZE_MAX, 0);

2174                if (unref_args.rbuf != 0) {
2175                        kmem_free(unref_args.rbuf, unref_args.rsize);
2176                        unref_args.rbuf = NULL;
2177                        unref_args.rsize = 0;
2178                }

2180                mutex_enter(&door_knob);
2181                ASSERT(dp->door_flags & DOOR_UNREF_ACTIVE);
2182                dp->door_flags &= ~DOOR_UNREF_ACTIVE;
2183                mutex_exit(&door_knob);
2184                VN_RELE(DTOV(dp));
2185        }
2186 }


2189 /*
2190  * Deliver queued unrefs to kernel door server.
2191  */
2192 /* ARGSUSED */
2193 static void
2194 door_unref_kernel(caddr_t arg)
2195 {
2196         door_node_t     *dp;
2197         static door_arg_t unref_args = { DOOR_UNREF_DATA, 0, 0, 0, 0, 0 };
2198         proc_t *p = ttoproc(curthread);
2199         callb_cpr_t cprinfo;

2201         /* should only be one of these */
2202         mutex_enter(&door_knob);
2203         if (p->p_unref_thread) {
2204                 mutex_exit(&door_knob);
2205                 return;
2206         }
2207         p->p_unref_thread = 1;
2208         mutex_exit(&door_knob);

2210         (void) door_my_data(1);         /* make sure we have a door_data_t */

2212         CALLB_CPR_INIT(&cprinfo, &door_knob, callb_generic_cpr, "door_unref");
2213         for (;;) {
2214                 mutex_enter(&door_knob);
2215                 /* Grab a queued request */
2216                 while ((dp = p->p_unref_list) == NULL) {
2217                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
2218                         cv_wait(&p->p_unref_cv, &door_knob);
2219                         CALLB_CPR_SAFE_END(&cprinfo, &door_knob);
2220                 }
2221                 p->p_unref_list = dp->door_ulist;
2222                 dp->door_ulist = NULL;
```



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

2226                (*(dp->door_pc))(dp->door_data, &unref_args, NULL, NULL, NULL);

2228                mutex_enter(&door_knob);
2229                ASSERT(dp->door_flags & DOOR_UNREF_ACTIVE);
2230                dp->door_flags &= ~DOOR_UNREF_ACTIVE;
2231                mutex_exit(&door_knob);
2232                VN_RELE(DTOV(dp));
2233        }
2234 }


2237 /*
2238  * Queue an unref invocation for processing for the current process
2239  * The door may or may not be revoked at this point.
2240  */
2241 void
2242 door_deliver_unref(door_node_t *d)
2243 {
2244         struct proc *server = d->door_target;

2246         ASSERT(MUTEX_HELD(&door_knob));
2247         ASSERT(d->door_active == 0);

2249         if (server == NULL)
2250                 return;
2251         /*
2252          * Create a lwp to deliver unref calls if one isn't already running.
2253          *
2254          * A separate thread is used to deliver unrefs since the current
2255          * thread may be holding resources (e.g. locks) in user land that
2256          * may be needed by the unref processing. This would cause a
2257          * deadlock.
2258          */
2259         if (d->door_flags & DOOR_UNREF_MULTI) {
2260                 /* multiple unrefs */
2261                 d->door_flags &= ~DOOR_DELAY;
2262         } else {
2263                 /* Only 1 unref per door */
2264                 d->door_flags &= ~(DOOR_UNREF|DOOR_DELAY);
2265         }
2266         mutex_exit(&door_knob);

2268         /*
2269          * Need to bump the vnode count before putting the door on the
2270          * list so it doesn't get prematurely released by door_unref.
2271          */
2272         VN_HOLD(DTOV(d));

2274         mutex_enter(&door_knob);
2275         /* is this door already on the unref list? */
2276         if (d->door_flags & DOOR_UNREF_MULTI) {
2277                 door_node_t *dp;
2278                 for (dp = server->p_unref_list; dp != NULL;
2279                     dp = dp->door_ulist) {
2280                         if (d == dp) {
2281                                 /* already there, don't need to add another */
2282                                 mutex_exit(&door_knob);
2283                                 VN_RELE(DTOV(d));
2284                                 mutex_enter(&door_knob);
2285                                 return;
2286                         }
2287                 }
2288         }
```

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

2295 /*
2296  * The callers buffer isn't big enough for all of the data/fd's. Allocate
2297  * space in the callers address space for the results and copy the data
2298  * there.
2299  *
2300  * For EOVERFLOW, we must clean up the server's door descriptors.
2301  */
2302 static int
2303 door_overflow(
2304         kthread_t       *caller,
2305         caddr_t         data_ptr,       /* data location */
2306         size_t          data_size,      /* data size */
2307         door_desc_t     *desc_ptr,      /* descriptor location */
2308         uint_t          desc_num)       /* descriptor size */
2309 {
2310         proc_t *callerp = ttoproc(caller);
2311         struct as *as = callerp->p_as;
2312         door_client_t *ct = DOOR_CLIENT(caller->t_door);
2313         caddr_t addr;                   /* Resulting address in target */
2314         size_t  rlen;                   /* Rounded len */
2315         size_t  len;
2316         uint_t  i;
2317         size_t  ds = desc_num * sizeof (door_desc_t);

2319         ASSERT(MUTEX_NOT_HELD(&door_knob));
2320         ASSERT(DOOR_T_HELD(ct) || ct->d_kernel);

2322         /* Do initial overflow check */
2323         if (!ufcanalloc(callerp, desc_num))
2324                 return (EMFILE);

2326         /*
2327          * Allocate space for this stuff in the callers address space
2328          */
2329         rlen = roundup(data_size + ds, PAGESIZE);
2330         as_rangelock(as);
2331         map_addr_proc(&addr, rlen, 0, 1, as->a_userlimit, ttoproc(caller), 0);
2332         if (addr == NULL ||
2333             as_map(as, addr, rlen, segvn_create, zfod_argsp) != 0) {
2334                 /* No virtual memory available, or anon mapping failed */
2335                 as_rangeunlock(as);
2336                 if (!ct->d_kernel && desc_num > 0) {
2337                         int error = door_release_fds(desc_ptr, desc_num);
2338                         if (error)
2339                                 return (error);
2340                 }
2341                 return (EOVERFLOW);
2342         }
2343         as_rangeunlock(as);

2345         if (ct->d_kernel)
2346                 goto out;

2348         if (data_size != 0) {
2349                 caddr_t src = data_ptr;
2350                 caddr_t saddr = addr;

2352                 /* Copy any data */
2353                 len = data_size;
2354                 while (len != 0) {
```



```
2355                         int     amount;
2356                         int     error;

2358                         amount = len > PAGESIZE ? PAGESIZE : len;
2359                         if ((error = door_copy(as, src, saddr, amount)) != 0) {
2360                                 (void) as_unmap(as, addr, rlen);
2361                                 return (error);
2362                         }
2363                         saddr += amount;
2364                         src += amount;
2365                         len -= amount;
2366                 }
2367         }
2368         /* Copy any fd's */
2369         if (desc_num != 0) {
2370                 door_desc_t     *didpp, *start;
2371                 struct file     **fpp;
2372                 int             fpp_size;

2374                 start = didpp = kmem_alloc(ds, KM_SLEEP);
2375                 if (copyin_nowatch(desc_ptr, didpp, ds)) {
2376                         kmem_free(start, ds);
2377                         (void) as_unmap(as, addr, rlen);
2378                         return (EFAULT);
2379                 }

2381                 fpp_size = desc_num * sizeof (struct file *);
2382                 if (fpp_size > ct->d_fpp_size) {
2383                         /* make more space */
2384                         if (ct->d_fpp_size)
2385                                 kmem_free(ct->d_fpp, ct->d_fpp_size);
2386                         ct->d_fpp_size = fpp_size;
2387                         ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2388                 }
2389                 fpp = ct->d_fpp;

2391                 for (i = 0; i < desc_num; i++) {
2392                         struct file *fp;
2393                         int fd = didpp->d_data.d_desc.d_descriptor;

2395                         if (!(didpp->d_attributes & DOOR_DESCRIPTOR) ||
2396                             (fp = getf(fd)) == NULL) {
2397                                 /* close translated references */
2398                                 door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2399                                 /* close untranslated references */
2400                                 door_fd_rele(didpp, desc_num - i, 0);
2401                                 kmem_free(start, ds);
2402                                 (void) as_unmap(as, addr, rlen);
2403                                 return (EINVAL);
2404                         }
2405                         mutex_enter(&fp->f_tlock);
2406                         fp->f_count++;
2407                         mutex_exit(&fp->f_tlock);

2409                         *fpp = fp;
2410                         releasef(fd);

2412                         if (didpp->d_attributes & DOOR_RELEASE) {
2413                                 /* release passed reference */
2414                                 (void) closeandsetf(fd, NULL);
2415                         }

2417                         fpp++; didpp++;
2418                 }
2419                 kmem_free(start, ds);
2420         }
```

```
2422 out:
2423         ct->d_overflow = 1;
2424         ct->d_args.rbuf = addr;
2425         ct->d_args.rsize = rlen;
2426         return (0);
2427 }

2429 /*
2430  * Transfer arguments from the client to the server.
2431  */
2432 static int
2433 door_args(kthread_t *server, int is_private)
2434 {
2435         door_server_t *st = DOOR_SERVER(server->t_door);
2436         door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2437         uint_t  ndid;
2438         size_t  dsize;
2439         int     error;

2441         ASSERT(DOOR_T_HELD(st));
2442         ASSERT(MUTEX_NOT_HELD(&door_knob));

2444         ndid = ct->d_args.desc_num;
2445         if (ndid > door_max_desc)
2446                 return (E2BIG);

2448         /*
2449          * Get the stack layout, and fail now if it won't fit.
2450          */
2451         error = door_layout(server, ct->d_args.data_size, ndid, is_private);
2452         if (error != 0)
2453                 return (error);

2455         dsize = ndid * sizeof (door_desc_t);
2456         if (ct->d_args.data_size != 0) {
2457                 if (ct->d_args.data_size <= door_max_arg) {
2458                         /*
2459                          * Use a 2 copy method for small amounts of data
2460                          *
2461                          * Allocate a little more than we need for the
2462                          * args, in the hope that the results will fit
2463                          * without having to reallocate a buffer
2464                          */
2465                         ASSERT(ct->d_buf == NULL);
2466                         ct->d_bufsize = roundup(ct->d_args.data_size,
2467                             DOOR_ROUND);
2468                         ct->d_buf = kmem_alloc(ct->d_bufsize, KM_SLEEP);
2469                         if (copyin_nowatch(ct->d_args.data_ptr,
2470                             ct->d_buf, ct->d_args.data_size) != 0) {
2471                                 kmem_free(ct->d_buf, ct->d_bufsize);
2472                                 ct->d_buf = NULL;
2473                                 ct->d_bufsize = 0;
2474                                 return (EFAULT);
2475                         }
2476                 } else {
2477                         struct as       *as;
2478                         caddr_t         src;
2479                         caddr_t         dest;
2480                         size_t          len = ct->d_args.data_size;
2481                         uintptr_t       base;

2483                         /*
2484                          * Use a 1 copy method
2485                          */
2486                         as = ttoproc(server)->p_as;
```



```
2487                         src = ct->d_args.data_ptr;

2489                         dest = st->d_layout.dl_datap;
2490                         base = (uintptr_t)dest;

2492                         /*
2493                          * Copy data directly into server.  We proceed
2494                          * downward from the top of the stack, to mimic
2495                          * normal stack usage. This allows the guard page
2496                          * to stop us before we corrupt anything.
2497                          */
2498                         while (len != 0) {
2499                                 uintptr_t start;
2500                                 uintptr_t end;
2501                                 uintptr_t offset;
2502                                 size_t  amount;

2504                                 /*
2505                                  * Locate the next part to copy.
2506                                  */
2507                                 end = base + len;
2508                                 start = P2ALIGN(end - 1, PAGESIZE);

2510                                 /*
2511                                  * if we are on the final (first) page, fix
2512                                  * up the start position.
2513                                  */
2514                                 if (P2ALIGN(base, PAGESIZE) == start)
2515                                         start = base;

2517                                 offset = start - base;  /* the copy offset */
2518                                 amount = end - start;   /* # bytes to copy */

2520                                 ASSERT(amount > 0 && amount <= len &&
2521                                     amount <= PAGESIZE);

2523                                 error = door_copy(as, src + offset,
2524                                     dest + offset, amount);
2525                                 if (error != 0)
2526                                         return (error);
2527                                 len -= amount;
2528                         }
2529                 }
2530         }
2531         /*
2532          * Copyin the door args and translate them into files
2533          */
2534         if (ndid != 0) {
2535                 door_desc_t     *didpp;
2536                 door_desc_t     *start;
2537                 struct file     **fpp;

2539                 start = didpp = kmem_alloc(dsize, KM_SLEEP);

2541                 if (copyin_nowatch(ct->d_args.desc_ptr, didpp, dsize)) {
2542                         kmem_free(start, dsize);
2543                         return (EFAULT);
2544                 }
2545                 ct->d_fpp_size = ndid * sizeof (struct file *);
2546                 ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2547                 fpp = ct->d_fpp;
2548                 while (ndid--) {
2549                         struct file *fp;
2550                         int fd = didpp->d_data.d_desc.d_descriptor;

2552                         /* We only understand file descriptors as passed objs */
```

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

2570                        *fpp = fp;
2571                        releasef(fd);

2573                        if (didpp->d_attributes & DOOR_RELEASE) {
2574                                /* release passed reference */
2575                                (void) closeandsetf(fd, NULL);
2576                        }

2578                        fpp++; didpp++;
2579                }
2580                kmem_free(start, dsize);
2581        }
2582        return (0);
2583 }

2585 /*
2586  * Transfer arguments from a user client to a kernel server.  This copies in
2587  * descriptors and translates them into door handles.  It doesn't touch the
2588  * other data, letting the kernel server deal with that (to avoid needing
2589  * to copy the data twice).
2590  */
2591 static int
2592 door_translate_in(void)
2593 {
2594        door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2595        uint_t  ndid;

2597        ASSERT(MUTEX_NOT_HELD(&door_knob));
2598        ndid = ct->d_args.desc_num;
2599        if (ndid > door_max_desc)
2600                return (E2BIG);
2601        /*
2602         * Copyin the door args and translate them into door handles.
2603         */
2604        if (ndid != 0) {
2605                door_desc_t     *didpp;
2606                door_desc_t     *start;
2607                size_t          dsize = ndid * sizeof (door_desc_t);
2608                struct file     *fp;

2610                start = didpp = kmem_alloc(dsize, KM_SLEEP);

2612                if (copyin_nowatch(ct->d_args.desc_ptr, didpp, dsize)) {
2613                        kmem_free(start, dsize);
2614                        return (EFAULT);
2615                }
2616                while (ndid--) {
2617                        vnode_t *vp;
2618                        int fd = didpp->d_data.d_desc.d_descriptor;
```



```
2620                        /*
2621                         * We only understand file descriptors as passed objs
2622                         */
2623                        if ((didpp->d_attributes & DOOR_DESCRIPTOR) &&
2624                            (fp = getf(fd)) != NULL) {
2625                                didpp->d_data.d_handle = FTODH(fp);
2626                                /* Hold the door */
2627                                door_ki_hold(didpp->d_data.d_handle);

2629                                releasef(fd);

2631                                if (didpp->d_attributes & DOOR_RELEASE) {
2632                                        /* release passed reference */
2633                                        (void) closeandsetf(fd, NULL);
2634                                }

2636                                if (VOP_REALVP(fp->f_vnode, &vp, NULL))
2637                                        vp = fp->f_vnode;

2639                                /* Set attributes */
2640                                didpp->d_attributes = DOOR_HANDLE |
2641                                    (VTOD(vp)->door_flags & DOOR_ATTR_MASK);
2642                        } else {
2643                                /* close translated references */
2644                                door_fd_close(start, didpp - start);
2645                                /* close untranslated references */
2646                                door_fd_rele(didpp, ndid + 1, 0);
2647                                kmem_free(start, dsize);
2648                                return (EINVAL);
2649                        }
2650                        didpp++;
2651                }
2652                ct->d_args.desc_ptr = start;
2653        }
2654        return (0);
2655 }

2657 /*
2658  * Translate door arguments from kernel to user.  This copies the passed
2659  * door handles.  It doesn't touch other data.  It is used by door_upcall,
2660  * and for data returned by a door_call to a kernel server.
2661  */
2662 static int
2663 door_translate_out(void)
2664 {
2665        door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2666        uint_t  ndid;

2668        ASSERT(MUTEX_NOT_HELD(&door_knob));
2669        ndid = ct->d_args.desc_num;
2670        if (ndid > door_max_desc) {
2671                door_fd_rele(ct->d_args.desc_ptr, ndid, 1);
2672                return (E2BIG);
2673        }
2674        /*
2675         * Translate the door args into files
2676         */
2677        if (ndid != 0) {
2678                door_desc_t     *didpp = ct->d_args.desc_ptr;
2679                struct file     **fpp;

2681                ct->d_fpp_size = ndid * sizeof (struct file *);
2682                fpp = ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2683                while (ndid--) {
2684                        struct file *fp = NULL;
```

```
2685                        int fd = -1;

2687                        /*
2688                         * We understand file descriptors and door
2689                         * handles as passed objs.
2690                         */
2691                        if (didpp->d_attributes & DOOR_DESCRIPTOR) {
2692                                fd = didpp->d_data.d_desc.d_descriptor;
2693                                fp = getf(fd);
2694                        } else if (didpp->d_attributes & DOOR_HANDLE)
2695                                fp = DHTOF(didpp->d_data.d_handle);
2696                        if (fp != NULL) {
2697                                /* Hold the fp */
2698                                mutex_enter(&fp->f_tlock);
2699                                fp->f_count++;
2700                                mutex_exit(&fp->f_tlock);

2702                                *fpp = fp;
2703                                if (didpp->d_attributes & DOOR_DESCRIPTOR)
2704                                        releasef(fd);
2705                                if (didpp->d_attributes & DOOR_RELEASE) {
2706                                        /* release passed reference */
2707                                        if (fd >= 0)
2708                                                (void) closeandsetf(fd, NULL);
2709                                        else
2710                                                (void) closef(fp);
2711                                }
2712                        } else {
2713                                /* close translated references */
2714                                door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2715                                /* close untranslated references */
2716                                door_fd_rele(didpp, ndid + 1, 1);
2717                                kmem_free(ct->d_fpp, ct->d_fpp_size);
2718                                ct->d_fpp = NULL;
2719                                ct->d_fpp_size = 0;
2720                                return (EINVAL);
2721                        }
2722                        fpp++; didpp++;
2723                }
2724        }
2725        return (0);
2726 }

2728 /*
2729  * Move the results from the server to the client
2730  */
2731 static int
2732 door_results(kthread_t *caller, caddr_t data_ptr, size_t data_size,
2733                 door_desc_t *desc_ptr, uint_t desc_num)
2734 {
2735        door_client_t   *ct = DOOR_CLIENT(caller->t_door);
2736        door_upcall_t   *dup = ct->d_upcall;
2737        size_t          dsize;
2738        size_t          rlen;
2739        size_t          result_size;

2741        ASSERT(DOOR_T_HELD(ct));
2742        ASSERT(MUTEX_NOT_HELD(&door_knob));

2744        if (ct->d_noresults)
2745                return (E2BIG);         /* No results expected */

2747        if (desc_num > door_max_desc)
2748                return (E2BIG);         /* Too many descriptors */

2750        dsize = desc_num * sizeof (door_desc_t);
```



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

2761        if (dup != NULL) {
2762                if (desc_num > dup->du_max_descs)
2763                        return (EMFILE);

2765                if (data_size > dup->du_max_data)
2766                        return (E2BIG);

2768                /*
2769                 * Handle upcalls
2770                 */
2771                if (ct->d_args.rbuf == NULL || ct->d_args.rsize < result_size) {
2772                        /*
2773                         * If there's no return buffer or the buffer is too
2774                         * small, allocate a new one.  The old buffer (if it
2775                         * exists) will be freed by the upcall client.
2776                         */
2777                        if (result_size > door_max_upcall_reply)
2778                                return (E2BIG);
2779                        ct->d_args.rsize = result_size;
2780                        ct->d_args.rbuf = kmem_alloc(result_size, KM_SLEEP);
2781                }
2782                ct->d_args.data_ptr = ct->d_args.rbuf;
2783                if (data_size != 0 &&
2784                    copyin_nowatch(data_ptr, ct->d_args.data_ptr,
2785                    data_size) != 0)
2786                        return (EFAULT);
2787        } else if (result_size > ct->d_args.rsize) {
2788                return (door_overflow(caller, data_ptr, data_size,
2789                    desc_ptr, desc_num));
2790        } else if (data_size != 0) {
2791                if (data_size <= door_max_arg) {
2792                        /*
2793                         * Use a 2 copy method for small amounts of data
2794                         */
2795                        if (ct->d_buf == NULL) {
2796                                ct->d_bufsize = data_size;
2797                                ct->d_buf = kmem_alloc(ct->d_bufsize, KM_SLEEP);
2798                        } else if (ct->d_bufsize < data_size) {
2799                                kmem_free(ct->d_buf, ct->d_bufsize);
2800                                ct->d_bufsize = data_size;
2801                                ct->d_buf = kmem_alloc(ct->d_bufsize, KM_SLEEP);
2802                        }
2803                        if (copyin_nowatch(data_ptr, ct->d_buf, data_size) != 0)
2804                                return (EFAULT);
2805                } else {
2806                        struct as *as = ttoproc(caller)->p_as;
2807                        caddr_t dest = ct->d_args.rbuf;
2808                        caddr_t src = data_ptr;
2809                        size_t  len = data_size;

2811                        /* Copy data directly into client */
2812                        while (len != 0) {
2813                                uint_t  amount;
2814                                uint_t  max;
2815                                uint_t  off;
2816                                int     error;
```

```
2818                                off = (uintptr_t)dest & PAGEOFFSET;
2819                                if (off)
2820                                        max = PAGESIZE - off;
2821                                else
2822                                        max = PAGESIZE;
2823                                amount = len > max ? max : len;
2824                                error = door_copy(as, src, dest, amount);
2825                                if (error != 0)
2826                                        return (error);
2827                                dest += amount;
2828                                src += amount;
2829                                len -= amount;
2830                        }
2831                }
2832        }

2834        /*
2835         * Copyin the returned door ids and translate them into door_node_t
2836         */
2837        if (desc_num != 0) {
2838                door_desc_t *start;
2839                door_desc_t *didpp;
2840                struct file **fpp;
2841                size_t  fpp_size;
2842                uint_t  i;

2844                /* First, check if we would overflow client */
2845                if (!ufcanalloc(ttoproc(caller), desc_num))
2846                        return (EMFILE);

2848                start = didpp = kmem_alloc(dsize, KM_SLEEP);
2849                if (copyin_nowatch(desc_ptr, didpp, dsize)) {
2850                        kmem_free(start, dsize);
2851                        return (EFAULT);
2852                }
2853                fpp_size = desc_num * sizeof (struct file *);
2854                if (fpp_size > ct->d_fpp_size) {
2855                        /* make more space */
2856                        if (ct->d_fpp_size)
2857                                kmem_free(ct->d_fpp, ct->d_fpp_size);
2858                        ct->d_fpp_size = fpp_size;
2859                        ct->d_fpp = kmem_alloc(fpp_size, KM_SLEEP);
2860                }
2861                fpp = ct->d_fpp;

2863                for (i = 0; i < desc_num; i++) {
2864                        struct file *fp;
2865                        int fd = didpp->d_data.d_desc.d_descriptor;

2867                        /* Only understand file descriptor results */
2868                        if (!(didpp->d_attributes & DOOR_DESCRIPTOR) ||
2869                            (fp = getf(fd)) == NULL) {
2870                                /* close translated references */
2871                                door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2872                                /* close untranslated references */
2873                                door_fd_rele(didpp, desc_num - i, 0);
2874                                kmem_free(start, dsize);
2875                                return (EINVAL);
2876                        }

2878                        mutex_enter(&fp->f_tlock);
2879                        fp->f_count++;
2880                        mutex_exit(&fp->f_tlock);

2882                        *fpp = fp;
```



```
2883                        releasef(fd);

2885                        if (didpp->d_attributes & DOOR_RELEASE) {
2886                                /* release passed reference */
2887                                (void) closeandsetf(fd, NULL);
2888                        }

2890                        fpp++; didpp++;
2891                }
2892                kmem_free(start, dsize);
2893        }
2894        return (0);
2895 }

2897 /*
2898  * Close all the descriptors.
2899  */
2900 static void
2901 door_fd_close(door_desc_t *d, uint_t n)
2902 {
2903        uint_t  i;

2905        ASSERT(MUTEX_NOT_HELD(&door_knob));
2906        for (i = 0; i < n; i++) {
2907                if (d->d_attributes & DOOR_DESCRIPTOR) {
2908                        (void) closeandsetf(
2909                            d->d_data.d_desc.d_descriptor, NULL);
2910                } else if (d->d_attributes & DOOR_HANDLE) {
2911                        door_ki_rele(d->d_data.d_handle);
2912                }
2913                d++;
2914        }
2915 }

2917 /*
2918  * Close descriptors that have the DOOR_RELEASE attribute set.
2919  */
2920 void
2921 door_fd_rele(door_desc_t *d, uint_t n, int from_kernel)
2922 {
2923        uint_t  i;

2925        ASSERT(MUTEX_NOT_HELD(&door_knob));
2926        for (i = 0; i < n; i++) {
2927                if (d->d_attributes & DOOR_RELEASE) {
2928                        if (d->d_attributes & DOOR_DESCRIPTOR) {
2929                                (void) closeandsetf(
2930                                    d->d_data.d_desc.d_descriptor, NULL);
2931                        } else if (from_kernel &&
2932                            (d->d_attributes & DOOR_HANDLE)) {
2933                                door_ki_rele(d->d_data.d_handle);
2934                        }
2935                }
2936                d++;
2937        }
2938 }

2940 /*
2941  * Copy descriptors into the kernel so we can release any marked
2942  * DOOR_RELEASE.
2943  */
2944 int
2945 door_release_fds(door_desc_t *desc_ptr, uint_t ndesc)
2946 {
2947        size_t dsize;
2948        door_desc_t *didpp;
```

```
2949         uint_t desc_num;

2951         ASSERT(MUTEX_NOT_HELD(&door_knob));
2952         ASSERT(ndesc != 0);

2954         desc_num = MIN(ndesc, door_max_desc);

2956         dsize = desc_num * sizeof (door_desc_t);
2957         didpp = kmem_alloc(dsize, KM_SLEEP);

2959         while (ndesc > 0) {
2960                 uint_t count = MIN(ndesc, desc_num);

2962                 if (copyin_nowatch(desc_ptr, didpp,
2963                     count * sizeof (door_desc_t))) {
2964                         kmem_free(didpp, dsize);
2965                         return (EFAULT);
2966                 }
2967                 door_fd_rele(didpp, count, 0);

2969                 ndesc -= count;
2970                 desc_ptr += count;
2971         }
2972         kmem_free(didpp, dsize);
2973         return (0);
2974 }

2976 /*
2977  * Decrement ref count on all the files passed
2978  */
2979 static void
2980 door_fp_close(struct file **fp, uint_t n)
2981 {
2982         uint_t  i;

2984         ASSERT(MUTEX_NOT_HELD(&door_knob));

2986         for (i = 0; i < n; i++)
2987                 (void) closef(fp[i]);
2988 }

2990 /*
2991  * Copy data from 'src' in current address space to 'dest' in 'as' for 'len'
2992  * bytes.
2993  *
2994  * Performs this using 1 mapin and 1 copy operation.
2995  *
2996  * We really should do more than 1 page at a time to improve
2997  * performance, but for now this is treated as an anomalous condition.
2998  */
2999 static int
3000 door_copy(struct as *as, caddr_t src, caddr_t dest, uint_t len)
3001 {
3002         caddr_t kaddr;
3003         caddr_t rdest;
3004         uint_t  off;
3005         page_t  **pplist;
3006         page_t  *pp = NULL;
3007         int     error = 0;

3009         ASSERT(len <= PAGESIZE);
3010         off = (uintptr_t)dest & PAGEOFFSET;     /* offset within the page */
3011         rdest = (caddr_t)((uintptr_t)dest &
3012             (uintptr_t)PAGEMASK);       /* Page boundary */
3013         ASSERT(off + len <= PAGESIZE);
```



```
3015         /*
3016          * Lock down destination page.
3017          */
3018         if (as_pagelock(as, &pplist, rdest, PAGESIZE, S_WRITE))
3019                 return (E2BIG);
3020         /*
3021          * Check if we have a shadow page list from as_pagelock. If not,
3022          * we took the slow path and have to find our page struct the hard
3023          * way.
3024          */
3025         if (pplist == NULL) {
3026                 pfn_t   pfnum;

3028                 /* MMU mapping is already locked down */
3029                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3030                 pfnum = hat_getpfnum(as->a_hat, rdest);
3031                 AS_LOCK_EXIT(as, &as->a_lock);

3033                 /*
3034                  * TODO: The pfn step should not be necessary - need
3035                  * a hat_getpp() function.
3036                  */
3037                 if (pf_is_memory(pfnum)) {
3038                         pp = page_numtopp_nolock(pfnum);
3039                         ASSERT(pp == NULL || PAGE_LOCKED(pp));
3040                 } else
3041                         pp = NULL;
3042                 if (pp == NULL) {
3043                         as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3044                         return (E2BIG);
3045                 }
3046         } else {
3047                 pp = *pplist;
3048         }
3049         /*
3050          * Map destination page into kernel address
3051          */
3052         if (kpm_enable)
3053                 kaddr = (caddr_t)hat_kpm_mapin(pp, (struct kpme *)NULL);
3054         else
3055                 kaddr = (caddr_t)ppmapin(pp, PROT_READ | PROT_WRITE,
3056                     (caddr_t)-1);

3058         /*
3059          * Copy from src to dest
3060          */
3061         if (copyin_nowatch(src, kaddr + off, len) != 0)
3062                 error = EFAULT;
3063         /*
3064          * Unmap destination page from kernel
3065          */
3066         if (kpm_enable)
3067                 hat_kpm_mapout(pp, (struct kpme *)NULL, kaddr);
3068         else
3069                 ppmapout(kaddr);
3070         /*
3071          * Unlock destination page
3072          */
3073         as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3074         return (error);
3075 }

3077 /*
3078  * General kernel upcall using doors
3079  *      Returns 0 on success, errno for failures.
3080  *      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 ...)
                 */
                mutex_exit(&door_knob);         /* May block in ISSIG */
```

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

3236                                st->d_active = NULL;
3237                                st->d_caller = NULL;
3238                                if (!(dp->door_flags & DOOR_NO_CANCEL)) {
3239                                        DOOR_T_HOLD(st);
3240                                        mutex_exit(&door_knob);

3242                                        mutex_enter(&p->p_lock);
3243                                        sigtoproc(p, server_thread, SIGCANCEL);
3244                                        mutex_exit(&p->p_lock);

3246                                        mutex_enter(&door_knob);
3247                                        DOOR_T_RELEASE(st);
3248                                }
3249                        }
3250                } else {
3251                        /*
3252                         * Return from stop(), server exit...
3253                         *
3254                         * Note that the server could have done a
3255                         * door_return while the client was in stop state
3256                         * (ISSIG), in which case the error condition
3257                         * is updated by the server.
3258                         */
3259                        mutex_enter(&door_knob);
3260                        if (ct->d_error == DOOR_WAIT) {
3261                                /* Still waiting for a reply */
3262                                shuttle_swtch(&door_knob);
3263                                mutex_enter(&door_knob);
3264                                if (lwp)
3265                                        lwp->lwp_asleep = 0;
3266                                goto    shuttle_return;
3267                        } else if (ct->d_error == DOOR_EXIT) {
3268                                /* Server exit */
3269                                error = EINTR;
3270                        } else {
3271                                /* Server did a door_return during ISSIG */
3272                                error = ct->d_error;
3273                        }
3274                }
3275                /*
3276                 * Can't exit if the server is currently copying
3277                 * results for me
3278                 */
```



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

3282                /*
3283                 * Find out if results were successfully copied.
3284                 */
3285                if (ct->d_error == 0)
3286                        gotresults = 1;
3287        }
3288        if (lwp) {
3289                lwp->lwp_asleep = 0;            /* /proc */
3290                lwp->lwp_sysabort = 0;          /* /proc */
3291        }
3292        if (--dp->door_active == 0 && (dp->door_flags & DOOR_DELAY))
3293                door_deliver_unref(dp);
3294        mutex_exit(&door_knob);

3296        /*
3297         * Translate returned doors (if any)
3298         */

3300        if (ct->d_noresults)
3301                goto out;

3303        if (error) {
3304                /*
3305                 * If server returned results successfully, then we've
3306                 * been interrupted and may need to clean up.
3307                 */
3308                if (gotresults) {
3309                        ASSERT(error == EINTR);
3310                        door_fp_close(ct->d_fpp, ct->d_args.desc_num);
3311                }
3312                goto out;
3313        }

3315        if (ct->d_args.desc_num) {
3316                struct file     **fpp;
3317                door_desc_t     *didpp;
3318                vnode_t         *vp;
3319                uint_t          n = ct->d_args.desc_num;

3321                didpp = ct->d_args.desc_ptr = (door_desc_t *)(ct->d_args.rbuf +
3322                    roundup(ct->d_args.data_size, sizeof (door_desc_t)));
3323                fpp = ct->d_fpp;

3325                while (n--) {
3326                        struct file *fp;

3328                        fp = *fpp;
3329                        if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3330                                vp = fp->f_vnode;

3332                        didpp->d_attributes = DOOR_HANDLE |
3333                            (VTOD(vp)->door_flags & DOOR_ATTR_MASK);
3334                        didpp->d_data.d_handle = FTODH(fp);

3336                        fpp++; didpp++;
3337                }
3338        }

3340        /* on return data is in rbuf */
3341        *param = ct->d_args;            /* structure assignment */

3343 out:
3344        kmem_free(dup, sizeof (*dup));
```

```
3346         if (ct->d_fpp) {
3347                 kmem_free(ct->d_fpp, ct->d_fpp_size);
3348                 ct->d_fpp = NULL;
3349                 ct->d_fpp_size = 0;
3350         }

3352         ct->d_upcall = NULL;
3353         ct->d_noresults = 0;
3354         ct->d_buf = NULL;
3355         ct->d_bufsize = 0;
3356         return (error);
3357 }

3359 /*
3360  * Add a door to the per-process list of active doors for which the
3361  * process is a server.
3362  */
3363 static void
3364 door_list_insert(door_node_t *dp)
3365 {
3366         proc_t *p = dp->door_target;

3368         ASSERT(MUTEX_HELD(&door_knob));
3369         dp->door_list = p->p_door_list;
3370         p->p_door_list = dp;
3371 }

3373 /*
3374  * Remove a door from the per-process list of active doors.
3375  */
3376 void
3377 door_list_delete(door_node_t *dp)
3378 {
3379         door_node_t **pp;

3381         ASSERT(MUTEX_HELD(&door_knob));
3382         /*
3383          * Find the door in the list.  If the door belongs to another process,
3384          * it's OK to use p_door_list since that process can't exit until all
3385          * doors have been taken off the list (see door_exit).
3386          */
3387         pp = &(dp->door_target->p_door_list);
3388         while (*pp != dp)
3389                 pp = &((*pp)->door_list);

3391         /* found it, take it off the list */
3392         *pp = dp->door_list;
3393 }


3396 /*
3397  * External kernel interfaces for doors.  These functions are available
3398  * outside the doorfs module for use in creating and using doors from
3399  * within the kernel.
3400  */

3402 /*
3403  * door_ki_upcall invokes a user-level door server from the kernel, with
3404  * the credentials associated with curthread.
3405  */
3406 int
3407 door_ki_upcall(door_handle_t dh, door_arg_t *param)
3408 {
3409         return (door_ki_upcall_limited(dh, param, NULL, SIZE_MAX, UINT_MAX));
3410 }
```



```
3412 /*
3413  * door_ki_upcall_limited invokes a user-level door server from the
3414  * kernel with the given credentials and reply limits.  If the "cred"
3415  * argument is NULL, uses the credentials associated with current
3416  * thread.  max_data limits the maximum length of the returned data (the
3417  * client will get E2BIG if they go over), and max_desc limits the
3418  * number of returned descriptors (the client will get EMFILE if they
3419  * go over).
3420  */
3421 int
3422 door_ki_upcall_limited(door_handle_t dh, door_arg_t *param, struct cred *cred,
3423     size_t max_data, uint_t max_desc)
3424 {
3425         file_t *fp = DHTOF(dh);
3426         vnode_t *realvp;

3428         if (VOP_REALVP(fp->f_vnode, &realvp, NULL))
3429                 realvp = fp->f_vnode;
3430         return (door_upcall(realvp, param, cred, max_data, max_desc));
3431 }

3433 /*
3434  * Function call to create a "kernel" door server.  A kernel door
3435  * server provides a way for a user-level process to invoke a function
3436  * in the kernel through a door_call.  From the caller's point of
3437  * view, a kernel door server looks the same as a user-level one
3438  * (except the server pid is 0).  Unlike normal door calls, the
3439  * kernel door function is invoked via a normal function call in the
3440  * same thread and context as the caller.
3441  */
3442 int
3443 door_ki_create(void (*pc_cookie)(), void *data_cookie, uint_t attributes,
3444     door_handle_t *dhp)
3445 {
3446         int err;
3447         file_t *fp;

3449         /* no DOOR_PRIVATE */
3450         if ((attributes & ~DOOR_KI_CREATE_MASK) ||
3451             (attributes & (DOOR_UNREF | DOOR_UNREF_MULTI)) ==
3452             (DOOR_UNREF | DOOR_UNREF_MULTI))
3453                 return (EINVAL);

3455         err = door_create_common(pc_cookie, data_cookie, attributes,
3456             1, NULL, &fp);
3457         if (err == 0 && (attributes & (DOOR_UNREF | DOOR_UNREF_MULTI)) &&
3458             p0.p_unref_thread == 0) {
3459                 /* need to create unref thread for process 0 */
3460                 (void) thread_create(NULL, 0, door_unref_kernel, NULL, 0, &p0,
3461                     TS_RUN, minclsyspri);
3462         }
3463         if (err == 0) {
3464                 *dhp = FTODH(fp);
3465         }
3466         return (err);
3467 }

3469 void
3470 door_ki_hold(door_handle_t dh)
3471 {
3472         file_t *fp = DHTOF(dh);

3474         mutex_enter(&fp->f_tlock);
3475         fp->f_count++;
3476         mutex_exit(&fp->f_tlock);
```

```
3477 }

3479 void
3480 door_ki_rele(door_handle_t dh)
3481 {
3482         file_t *fp = DHTOF(dh);

3484         (void) closef(fp);
3485 }

3487 int
3488 door_ki_open(char *pathname, door_handle_t *dhp)
3489 {
3490         file_t *fp;
3491         vnode_t *vp;
3492         int err;

3494         if ((err = lookupname(pathname, UIO_SYSSPACE, FOLLOW, NULL, &vp)) != 0)
3495                 return (err);
3496         if (err = VOP_OPEN(&vp, FREAD, kcred, NULL)) {
3497                 VN_RELE(vp);
3498                 return (err);
3499         }
3500         if (vp->v_type != VDOOR) {
3501                 VN_RELE(vp);
3502                 return (EINVAL);
3503         }
3504         if ((err = falloc(vp, FREAD | FWRITE, &fp, NULL)) != 0) {
3505                 VN_RELE(vp);
3506                 return (err);
3507         }
3508         /* falloc returns with f_tlock held on success */
3509         mutex_exit(&fp->f_tlock);
3510         *dhp = FTODH(fp);
3511         return (0);
3512 }

3514 int
3515 door_ki_info(door_handle_t dh, struct door_info *dip)
3516 {
3517         file_t *fp = DHTOF(dh);
3518         vnode_t *vp;

3520         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3521                 vp = fp->f_vnode;
3522         if (vp->v_type != VDOOR)
3523                 return (EINVAL);
3524         door_info_common(VTOD(vp), dip, fp);
3525         return (0);
3526 }

3528 door_handle_t
3529 door_ki_lookup(int did)
3530 {
3531         file_t *fp;
3532         door_handle_t dh;

3534         /* is the descriptor really a door? */
3535         if (door_lookup(did, &fp) == NULL)
3536                 return (NULL);
3537         /* got the door, put a hold on it and release the fd */
3538         dh = FTODH(fp);
3539         door_ki_hold(dh);
3540         releasef(did);
3541         return (dh);
3542 }
```



```
3544 int
3545 door_ki_setparam(door_handle_t dh, int type, size_t val)
3546 {
3547         file_t *fp = DHTOF(dh);
3548         vnode_t *vp;

3550         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3551                 vp = fp->f_vnode;
3552         if (vp->v_type != VDOOR)
3553                 return (EINVAL);
3554         return (door_setparam_common(VTOD(vp), 1, type, val));
3555 }

3557 int
3558 door_ki_getparam(door_handle_t dh, int type, size_t *out)
3559 {
3560         file_t *fp = DHTOF(dh);
3561         vnode_t *vp;

3563         if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3564                 vp = fp->f_vnode;
3565         if (vp->v_type != VDOOR)
3566                 return (EINVAL);
3567         return (door_getparam_common(VTOD(vp), type, out));
3568 }
```

```
**********************************************************
   17129 Sun Aug  9 12:47:39 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, proc_t *p)
689 {
690         pid_node_t      *pn;

692         /*
693          * don’t add pid = 0 to the list. this  is usually the case
694          * with sockets created as the result of accepting a new connection.
695          * These sockets are created by the kernel, we will catch them once
```

```
696          * the process accepts them.
697          */

699         if (p->p_pidp->pid_id == 0)
700                 return;

702         mutex_enter(&so->so_pid_list_lock);
703         pn = list_head(&so->so_pid_list);
704         while (pn != NULL && pn->pn_pid != p->p_pidp->pid_id) {
705                 pn = list_next(&so->so_pid_list, pn);
706         }

708         if (pn != NULL) {
709                 pn->pn_count++;
710         } else {
711                 pn = kmem_zalloc(sizeof (*pn), KM_SLEEP);
712                 list_link_init(&pn->pn_ref_link);
713                 pn->pn_pid = p->p_pidp->pid_id;
714                 pn->pn_count = 1;
715                 list_insert_tail(&so->so_pid_list, pn);
716         }
717         mutex_exit(&so->so_pid_list_lock);
718 }

720 void
721 sonode_remove_pid(struct sonode *so, proc_t *p)
722 {
723         pid_node_t *pn;

725         mutex_enter(&so->so_pid_list_lock);
726         pn = list_head(&so->so_pid_list);
727         while (pn != NULL && pn->pn_pid != p->p_pidp->pid_id) {
728                 pn = list_next(&so->so_pid_list, pn);
729         }

731         if (pn != NULL) {
732                 if (pn->pn_count > 1)
733                                 pn->pn_count--;
734                 else {
735                         list_remove(&so->so_pid_list, pn);
736                         kmem_free(pn, sizeof (*pn));
737                 }
738         }
739         mutex_exit(&so->so_pid_list_lock);
740 #endif /* ! codereview */
741 }
```

```
**********************************************************
    9840 Sun Aug  9 12:47:40 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 conn_pid_node_list_hdr_t *so_get_sock_pid_list(sock_upper_handle_t sock_handle);
 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 *, proc_t *);
239 extern void     sonode_remove_pid(struct sonode *, proc_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_ */
```

```
**********************************************************
   48937 Sun Aug  9 12:47:42 2015
new/usr/src/uts/common/fs/sockfs/sockcommon_sops.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) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
 */

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

#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/debug.h>
#include <sys/cmn_err.h>

#include <sys/stropts.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/fcntl.h>
#endif /* ! codereview */

#define _SUN_TPI_VERSION        2
#include <sys/tihdr.h>
#include <sys/sockio.h>
#include <sys/kmem_impl.h>

#include <sys/strsubr.h>
#include <sys/strsun.h>
#include <sys/ddi.h>
#include <netinet/in.h>
#include <inet/ip.h>

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

#include <sys/socket_proto.h>

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



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

extern int xnet_skip_checks;
extern int xnet_check_print;

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


/*ARGSUSED*/
int
so_accept_notsupp(struct sonode *lso, int fflag,
    struct cred *cr, struct sonode **nsop)
{
        return (EOPNOTSUPP);
}

/*ARGSUSED*/
int
so_listen_notsupp(struct sonode *so, int backlog, struct cred *cr)
{
        return (EOPNOTSUPP);
}

/*ARGSUSED*/
int
so_getsockname_notsupp(struct sonode *so, struct sockaddr *sa,
    socklen_t *len, struct cred *cr)
{
        return (EOPNOTSUPP);
}

/*ARGSUSED*/
int
so_getpeername_notsupp(struct sonode *so, struct sockaddr *addr,
    socklen_t *addrlen, boolean_t accept, struct cred *cr)
{
        return (EOPNOTSUPP);
}

/*ARGSUSED*/
int
so_shutdown_notsupp(struct sonode *so, int how, struct cred *cr)
{
        return (EOPNOTSUPP);
}

/*ARGSUSED*/
int
so_sendmblk_notsupp(struct sonode *so, struct msghdr *msg, int fflag,
    struct cred *cr, mblk_t **mpp)
{
        return (EOPNOTSUPP);
}

/*
 * Generic Socket Ops
 */

/* ARGSUSED */
int
so_init(struct sonode *so, struct sonode *pso, struct cred *cr, int flags)
{
        return (socket_init_common(so, pso, flags, cr));
}

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                         sonode_insert_pid(nso, curproc);
363 #endif /* ! codereview */
364                 }
365         }

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

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

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

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

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

```
392                 return (EOPNOTSUPP);
393         }
395         if ((so->so_mode & SM_ATOMIC) &&
396             uiop->uio_resid > so->so_proto_props.sopp_maxpsz &&
397             so->so_proto_props.sopp_maxpsz != -1) {
398                 SO_UNBLOCK_FALLBACK(so);
399                 return (EMSGSIZE);
400         }
402         /*
403          * For atomic sends we will only do one iteration.
404          */
405         do {
406                 if (so->so_state & SS_CANTSENDMORE) {
407                         error = EPIPE;
408                         break;
409                 }
411                 if (so->so_error != 0) {
412                         mutex_enter(&so->so_lock);
413                         error = sogeterr(so, B_TRUE);
414                         mutex_exit(&so->so_lock);
415                         if (error != 0)
416                                 break;
417                 }
419                 /*
420                  * Send down OOB messages even if the send path is being
421                  * flow controlled (assuming the protocol supports OOB data).
422                  */
423                 if (flags & MSG_OOB) {
424                         if ((so->so_mode & SM_EXDATA) == 0) {
425                                 error = EOPNOTSUPP;
426                                 break;
427                         }
428                 } else if (SO_SND_FLOWCTRLD(so)) {
429                         /*
430                          * Need to wait until the protocol is ready to receive
431                          * more data for transmission.
432                          */
433                         if ((error = so_snd_wait_qnotfull(so, dontblock)) != 0)
434                                 break;
435                 }
437                 /*
438                  * Time to send data to the protocol. We either copy the
439                  * data into mblks or pass the uio directly to the protocol.
440                  * We decide what to do based on the available down calls.
441                  */
442                 if (so->so_downcalls->sd_send_uio != NULL) {
443                         error = (*so->so_downcalls->sd_send_uio)
444                             (so->so_proto_handle, uiop, msg, cr);
445                         if (error != 0)
446                                 break;
447                 } else {
448                         /* save the resid in case of failure */
449                         orig_resid = uiop->uio_resid;
451                         if ((mp = socopyinuio(uiop,
452                             so->so_proto_props.sopp_maxpsz,
453                             so->so_proto_props.sopp_wroff,
454                             so->so_proto_props.sopp_maxblk,
455                             so->so_proto_props.sopp_tail, &error)) == NULL) {
456                                 break;
457                         }
```



```
458                         ASSERT(uiop->uio_resid >= 0);
460                         if (so->so_filter_active > 0 &&
461                             ((mp = SOF_FILTER_DATA_OUT(so, mp, msg, cr,
462                             &error)) == NULL)) {
463                                 if (error != 0)
464                                         break;
465                                 continue;
466                         }
467                         error = (*so->so_downcalls->sd_send)
468                             (so->so_proto_handle, mp, msg, cr);
469                         if (error != 0) {
470                                 /*
471                                  * The send failed. We do not have to free the
472                                  * mblks, because that is the protocol's
473                                  * responsibility. However, uio_resid must
474                                  * remain accurate, so adjust that here.
475                                  */
476                                 uiop->uio_resid = orig_resid;
477                                         break;
478                         }
479                 }
480         } while (uiop->uio_resid > 0);
482         SO_UNBLOCK_FALLBACK(so);
484         return (error);
485 }
487 int
488 so_sendmblk_impl(struct sonode *so, struct nmsghdr *msg, int fflag,
489     struct cred *cr, mblk_t **mpp, sof_instance_t *fil,
490     boolean_t fil_inject)
491 {
492         int error;
493         boolean_t dontblock;
494         size_t size;
495         mblk_t *mp = *mpp;
497         if (so->so_downcalls->sd_send == NULL)
498                 return (EOPNOTSUPP);
500         error = 0;
501         dontblock = (msg->msg_flags & MSG_DONTWAIT) ||
502             (fflag & (FNONBLOCK|FNDELAY));
503         size = msgdsize(mp);
505         if ((so->so_mode & SM_ATOMIC) &&
506             size > so->so_proto_props.sopp_maxpsz &&
507             so->so_proto_props.sopp_maxpsz != -1) {
508                 SO_UNBLOCK_FALLBACK(so);
509                 return (EMSGSIZE);
510         }
512         while (mp != NULL) {
513                 mblk_t *nmp, *last_mblk;
514                 size_t mlen;
516                 if (so->so_state & SS_CANTSENDMORE) {
517                         error = EPIPE;
518                         break;
519                 }
520                 if (so->so_error != 0) {
521                         mutex_enter(&so->so_lock);
522                         error = sogeterr(so, B_TRUE);
523                         mutex_exit(&so->so_lock);
```

```
524                         if (error != 0)
525                                 break;
526                 }
527                 /* Socket filters are not flow controlled */
528                 if (SO_SND_FLOWCTRLD(so) && !fil_inject) {
529                         /*
530                          * Need to wait until the protocol is ready to receive
531                          * more data for transmission.
532                          */
533                         if ((error = so_snd_wait_qnotfull(so, dontblock)) != 0)
534                                 break;
535                 }

537                 /*
538                  * We only allow so_maxpsz of data to be sent down to
539                  * the protocol at time.
540                  */
541                 mlen = MBLKL(mp);
542                 nmp = mp->b_cont;
543                 last_mblk = mp;
544                 while (nmp != NULL) {
545                         mlen += MBLKL(nmp);
546                         if (mlen > so->so_proto_props.sopp_maxpsz) {
547                                 last_mblk->b_cont = NULL;
548                                 break;
549                         }
550                         last_mblk = nmp;
551                         nmp = nmp->b_cont;
552                 }

554                 if (so->so_filter_active > 0 &&
555                     (mp = SOF_FILTER_DATA_OUT_FROM(so, fil, mp, msg,
556                     cr, &error)) == NULL) {
557                         *mpp = mp = nmp;
558                         if (error != 0)
559                                 break;
560                         continue;
561                 }
562                 error = (*so->so_downcalls->sd_send)
563                     (so->so_proto_handle, mp, msg, cr);
564                 if (error != 0) {
565                         /*
566                          * The send failed. The protocol will free the mblks
567                          * that were sent down. Let the caller deal with the
568                          * rest.
569                          */
570                         *mpp = nmp;
571                         break;
572                 }

574                 *mpp = mp = nmp;
575         }
576         /* Let the filter know whether the protocol is flow controlled */
577         if (fil_inject && error == 0 && SO_SND_FLOWCTRLD(so))
578                 error = ENOSPC;

580         return (error);
581 }

583 #pragma inline(so_sendmblk_impl)

585 int
586 so_sendmblk(struct sonode *so, struct nmsghdr *msg, int fflag,
587     struct cred *cr, mblk_t **mpp)
588 {
589         int error;
```



```
591         SO_BLOCK_FALLBACK(so, SOP_SENDMBLK(so, msg, fflag, cr, mpp));

593         if ((so->so_mode & SM_SENDFILESUPP) == 0) {
594                 SO_UNBLOCK_FALLBACK(so);
595                 return (EOPNOTSUPP);
596         }

598         error = so_sendmblk_impl(so, msg, fflag, cr, mpp, so->so_filter_top,
599             B_FALSE);

601         SO_UNBLOCK_FALLBACK(so);

603         return (error);
604 }

606 int
607 so_shutdown(struct sonode *so, int how, struct cred *cr)
608 {
609         int error;

611         SO_BLOCK_FALLBACK(so, SOP_SHUTDOWN(so, how, cr));

613         /*
614          * SunOS 4.X has no check for datagram sockets.
615          * 5.X checks that it is connected (ENOTCONN)
616          * X/Open requires that we check the connected state.
617          */
618         if (!(so->so_state & SS_ISCONNECTED)) {
619                 if (!xnet_skip_checks) {
620                         error = ENOTCONN;
621                         if (xnet_check_print) {
622                                 printf("sockfs: X/Open shutdown check "
623                                     "caused ENOTCONN\n");
624                         }
625                 }
626                 goto done;
627         }

629         if (so->so_filter_active == 0 ||
630             (error = sof_filter_shutdown(so, &how, cr)) < 0)
631                 error = ((*so->so_downcalls->sd_shutdown)(so->so_proto_handle,
632                     how, cr));

634         /*
635          * Protocol agreed to shutdown. We need to flush the
636          * receive buffer if the receive side is being shutdown.
637          */
638         if (error == 0 && how != SHUT_WR) {
639                 mutex_enter(&so->so_lock);
640                 /* wait for active reader to finish */
641                 (void) so_lock_read(so, 0);

643                 so_rcv_flush(so);

645                 so_unlock_read(so);
646                 mutex_exit(&so->so_lock);
647         }

649 done:
650         SO_UNBLOCK_FALLBACK(so);
651         return (error);
652 }

654 int
655 so_getsockname(struct sonode *so, struct sockaddr *addr,
```

```
656     socklen_t *addrlen, struct cred *cr)
657 {
658         int error;

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

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

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

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

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

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

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

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

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

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

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



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

783         SO_UNBLOCK_FALLBACK(so);
784         return (error);
785 }

787 int
```

```
788 so_setsockopt(struct sonode *so, int level, int option_name,
789     const void *optval, socklen_t optlen, struct cred *cr)
790 {
791         int error = 0;
792         struct timeval tl;
793         const void *opt = optval;
795         if (level == SOL_FILTER)
796                 return (sof_setsockopt(so, option_name, optval, optlen, cr));
798         SO_BLOCK_FALLBACK(so,
799             SOP_SETSOCKOPT(so, level, option_name, optval, optlen, cr));
801         /* X/Open requires this check */
802         if (so->so_state & SS_CANTSENDMORE && !xnet_skip_checks) {
803                 SO_UNBLOCK_FALLBACK(so);
804                 if (xnet_check_print)
805                         printf("sockfs: X/Open setsockopt check => EINVAL\n");
806                 return (EINVAL);
807         }
809         if (so->so_filter_active > 0 &&
810             (error = sof_filter_setsockopt(so, level, option_name,
811             (void *)optval, &optlen, cr)) >= 0)
812                 goto done;
814         if (level == SOL_SOCKET) {
815                 switch (option_name) {
816                 case SO_RCVTIMEO:
817                 case SO_SNDTIMEO: {
818                         /*
819                          * We pass down these two options to protocol in order
820                          * to support some third part protocols which need to
821                          * know them. For those protocols which don't care
822                          * these two options, simply return 0.
823                          */
824                         clock_t t_usec;
826                         if (get_udatamodel() == DATAMODEL_NONE ||
827                             get_udatamodel() == DATAMODEL_NATIVE) {
828                                 if (optlen != sizeof (struct timeval)) {
829                                         error = EINVAL;
830                                         goto done;
831                                 }
832                                 bcopy((struct timeval *)optval, &tl,
833                                     sizeof (struct timeval));
834                         } else {
835                                 if (optlen != sizeof (struct timeval32)) {
836                                         error = EINVAL;
837                                         goto done;
838                                 }
839                                 TIMEVAL32_TO_TIMEVAL(&tl,
840                                     (struct timeval32 *)optval);
841                         }
842                         opt = &tl;
843                         optlen = sizeof (tl);
844                         t_usec = tl.tv_sec * 1000 * 1000 + tl.tv_usec;
845                         mutex_enter(&so->so_lock);
846                         if (option_name == SO_RCVTIMEO)
847                                 so->so_rcvtimeo = drv_usectohz(t_usec);
848                         else
849                                 so->so_sndtimeo = drv_usectohz(t_usec);
850                         mutex_exit(&so->so_lock);
851                         break;
852                 }
853                 case SO_RCVBUF:
```



```
854                         /*
855                          * XXX XPG 4.2 applications retrieve SO_RCVBUF from
856                          * sockfs since the transport might adjust the value
857                          * and not return exactly what was set by the
858                          * application.
859                          */
860                         so->so_xpg_rcvbuf = *(int32_t *)optval;
861                         break;
862                 }
863         }
864         error = (*so->so_downcalls->sd_setsockopt)
865             (so->so_proto_handle, level, option_name, opt, optlen, cr);
866 done:
867         SO_UNBLOCK_FALLBACK(so);
868         return (error);
869 }
871 int
872 so_ioctl(struct sonode *so, int cmd, intptr_t arg, int mode,
873     struct cred *cr, int32_t *rvalp)
874 {
875         int error = 0;
877         SO_BLOCK_FALLBACK(so, SOP_IOCTL(so, cmd, arg, mode, cr, rvalp));
879         /*
880          * If there is a pending error, return error
881          * This can happen if a non blocking operation caused an error.
882          */
883         if (so->so_error != 0) {
884                 mutex_enter(&so->so_lock);
885                 error = sogeterr(so, B_TRUE);
886                 mutex_exit(&so->so_lock);
887                 if (error != 0)
888                         goto done;
889         }
891         /*
892          * calling strioc can result in the socket falling back to TPI,
893          * if that is supported.
894          */
895         if ((so->so_filter_active == 0 ||
896             (error = sof_filter_ioctl(so, cmd, arg, mode,
897             rvalp, cr)) < 0) &&
898             (error = socket_ioctl_common(so, cmd, arg, mode, cr, rvalp)) < 0 &&
899             (error = socket_strioc_common(so, cmd, arg, mode, cr, rvalp)) < 0) {
900                 error = (*so->so_downcalls->sd_ioctl)(so->so_proto_handle,
901                     cmd, arg, mode, rvalp, cr);
902         }
904 done:
905         SO_UNBLOCK_FALLBACK(so);
907         return (error);
908 }
910 int
911 so_poll(struct sonode *so, short events, int anyyet, short *reventsp,
912     struct pollhead **phpp)
913 {
914         int state = so->so_state, mask;
915         *reventsp = 0;
917         /*
918          * In sockets the errors are represented as input/output events
919          */
```

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



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

```
1052          * rather a failure to connect.
1053          */
1054         connect_failed = !(so->so_state & SS_ISCONNECTED);

1056         so->so_proto_connid = id;
1057         soisdisconnected(so, error);
1058         so_notify_disconnected(so, connect_failed, error);

1060         return (0);
1061 }

1063 void
1064 so_opctl(sock_upper_handle_t sock_handle, sock_opctl_action_t action,
1065     uintptr_t arg)
1066 {
1067         struct sonode *so = (struct sonode *)sock_handle;

1069         switch (action) {
1070         case SOCK_OPCTL_SHUT_SEND:
1071                 mutex_enter(&so->so_lock);
1072                 socantsendmore(so);
1073                 so_notify_disconnecting(so);
1074                 break;
1075         case SOCK_OPCTL_SHUT_RECV: {
1076                 mutex_enter(&so->so_lock);
1077                 socantrcvmore(so);
1078                 so_notify_eof(so);
1079                 break;
1080         }
1081         case SOCK_OPCTL_ENAB_ACCEPT:
1082                 mutex_enter(&so->so_lock);
1083                 so->so_state |= SS_ACCEPTCONN;
1084                 so->so_backlog = (unsigned int)arg;
1085                 /*
1086                  * The protocol can stop generating newconn upcalls when
1087                  * the backlog is full, so to make sure the listener does
1088                  * not end up with a queue full of deferred connections
1089                  * we reduce the backlog by one. Thus the listener will
1090                  * start closing deferred connections before the backlog
1091                  * is full.
1092                  */
1093                 if (so->so_filter_active > 0)
1094                         so->so_backlog = MAX(1, so->so_backlog - 1);
1095                 mutex_exit(&so->so_lock);
1096                 break;
1097         default:
1098                 ASSERT(0);
1099                 break;
1100         }
1101 }

1103 void
1104 so_txq_full(sock_upper_handle_t sock_handle, boolean_t qfull)
1105 {
1106         struct sonode *so = (struct sonode *)sock_handle;

1108         if (qfull) {
1109                 so_snd_qfull(so);
1110         } else {
1111                 so_snd_qnotfull(so);
1112                 mutex_enter(&so->so_lock);
1113                 /* so_notify_writable drops so_lock */
1114                 so_notify_writable(so);
1115         }
1116 }
```



```
1118 sock_upper_handle_t
1119 so_newconn(sock_upper_handle_t parenthandle,
1120     sock_lower_handle_t proto_handle, sock_downcalls_t *sock_downcalls,
1121     struct cred *peer_cred, pid_t peer_cpid, sock_upcalls_t **sock_upcallsp)
1122 {
1123         struct sonode   *so = (struct sonode *)parenthandle;
1124         struct sonode   *nso;
1125         int error;

1127         ASSERT(proto_handle != NULL);

1129         if ((so->so_state & SS_ACCEPTCONN) == 0 ||
1130             (so->so_acceptq_len >= so->so_backlog &&
1131             (so->so_filter_active == 0 || !sof_sonode_drop_deferred(so)))) {
1132                         return (NULL);
1133         }

1135         nso = socket_newconn(so, proto_handle, sock_downcalls, SOCKET_NOSLEEP,
1136             &error);
1137         if (nso == NULL)
1138                 return (NULL);

1140         if (peer_cred != NULL) {
1141                 crhold(peer_cred);
1142                 nso->so_peercred = peer_cred;
1143                 nso->so_cpid = peer_cpid;
1144         }
1145         nso->so_listener = so;

1147         /*
1148          * The new socket (nso), proto_handle and sock_upcallsp are all
1149          * valid at this point. But as soon as nso is placed in the accept
1150          * queue that can no longer be assumed (since an accept() thread may
1151          * pull it off the queue and close the socket).
1152          */
1153         *sock_upcallsp = &so_upcalls;

1155         mutex_enter(&so->so_acceptq_lock);
1156         if (so->so_state & (SS_CLOSING|SS_FALLBACK_PENDING|SS_FALLBACK_COMP)) {
1157                 mutex_exit(&so->so_acceptq_lock);
1158                 ASSERT(nso->so_count == 1);
1159                 nso->so_count--;
1160                 nso->so_listener = NULL;
1161                 /* drop proto ref */
1162                 VN_RELE(SOTOV(nso));
1163                 socket_destroy(nso);
1164                 return (NULL);
1165         } else {
1166                 so->so_acceptq_len++;
1167                 if (nso->so_state & SS_FIL_DEFER) {
1168                         list_insert_tail(&so->so_acceptq_defer, nso);
1169                         mutex_exit(&so->so_acceptq_lock);
1170                 } else {
1171                         list_insert_tail(&so->so_acceptq_list, nso);
1172                         cv_signal(&so->so_acceptq_cv);
1173                         mutex_exit(&so->so_acceptq_lock);
1174                         mutex_enter(&so->so_lock);
1175                         so_notify_newconn(so);
1176                 }

1178                 return ((sock_upper_handle_t)nso);
1179         }
1180 }

1182 void
1183 so_set_prop(sock_upper_handle_t sock_handle, struct sock_proto_props *soppp)
```

```
{
        struct sonode *so;

        so = (struct sonode *)sock_handle;

        mutex_enter(&so->so_lock);

        if (soppp->sopp_flags & SOCKOPT_MAXBLK)
                so->so_proto_props.sopp_maxblk = soppp->sopp_maxblk;
        if (soppp->sopp_flags & SOCKOPT_WROFF)
                so->so_proto_props.sopp_wroff = soppp->sopp_wroff;
        if (soppp->sopp_flags & SOCKOPT_TAIL)
                so->so_proto_props.sopp_tail = soppp->sopp_tail;
        if (soppp->sopp_flags & SOCKOPT_RCVHIWAT)
                so->so_proto_props.sopp_rxhiwat = soppp->sopp_rxhiwat;
        if (soppp->sopp_flags & SOCKOPT_RCVLOWAT)
                so->so_proto_props.sopp_rxlowat = soppp->sopp_rxlowat;
        if (soppp->sopp_flags & SOCKOPT_MAXPSZ)
                so->so_proto_props.sopp_maxpsz = soppp->sopp_maxpsz;
        if (soppp->sopp_flags & SOCKOPT_MINPSZ)
                so->so_proto_props.sopp_minpsz = soppp->sopp_minpsz;
        if (soppp->sopp_flags & SOCKOPT_ZCOPY) {
                if (soppp->sopp_zcopyflag & ZCVMSAFE) {
                        so->so_proto_props.sopp_zcopyflag |= STZCVMSAFE;
                        so->so_proto_props.sopp_zcopyflag &= ~STZCVMUNSAFE;
                } else if (soppp->sopp_zcopyflag & ZCVMUNSAFE) {
                        so->so_proto_props.sopp_zcopyflag |= STZCVMUNSAFE;
                        so->so_proto_props.sopp_zcopyflag &= ~STZCVMSAFE;
                }

                if (soppp->sopp_zcopyflag & COPYCACHED) {
                        so->so_proto_props.sopp_zcopyflag |= STRCOPYCACHED;
                }
        }
        if (soppp->sopp_flags & SOCKOPT_OOBINLINE)
                so->so_proto_props.sopp_oobinline = soppp->sopp_oobinline;
        if (soppp->sopp_flags & SOCKOPT_RCVTIMER)
                so->so_proto_props.sopp_rcvtimer = soppp->sopp_rcvtimer;
        if (soppp->sopp_flags & SOCKOPT_RCVTHRESH)
                so->so_proto_props.sopp_rcvthresh = soppp->sopp_rcvthresh;
        if (soppp->sopp_flags & SOCKOPT_MAXADDRLEN)
                so->so_proto_props.sopp_maxaddrlen = soppp->sopp_maxaddrlen;
        if (soppp->sopp_flags & SOCKOPT_LOOPBACK)
                so->so_proto_props.sopp_loopback = soppp->sopp_loopback;

        mutex_exit(&so->so_lock);

        if (so->so_filter_active > 0) {
                sof_instance_t *inst;
                ssize_t maxblk;
                ushort_t wroff, tail;
                maxblk = so->so_proto_props.sopp_maxblk;
                wroff = so->so_proto_props.sopp_wroff;
                tail = so->so_proto_props.sopp_tail;
                for (inst = so->so_filter_bottom; inst != NULL;
                    inst = inst->sofi_prev) {
                        if (SOF_INTERESTED(inst, mblk_prop)) {
                                (*inst->sofi_ops->sofop_mblk_prop)(
                                    (sof_handle_t)inst, inst->sofi_cookie,
                                    &maxblk, &wroff, &tail);
                        }
                }
                mutex_enter(&so->so_lock);
                so->so_proto_props.sopp_maxblk = maxblk;
                so->so_proto_props.sopp_wroff = wroff;
                so->so_proto_props.sopp_tail = tail;
```



```
                mutex_exit(&so->so_lock);
        }
#ifdef DEBUG
        soppp->sopp_flags &= ~(SOCKOPT_MAXBLK | SOCKOPT_WROFF | SOCKOPT_TAIL |
            SOCKOPT_RCVHIWAT | SOCKOPT_RCVLOWAT | SOCKOPT_MAXPSZ |
            SOCKOPT_ZCOPY | SOCKOPT_OOBINLINE | SOCKOPT_RCVTIMER |
            SOCKOPT_RCVTHRESH | SOCKOPT_MAXADDRLEN | SOCKOPT_MINPSZ |
            SOCKOPT_LOOPBACK);
        ASSERT(soppp->sopp_flags == 0);
#endif
}

/* ARGSUSED */
ssize_t
so_queue_msg_impl(struct sonode *so, mblk_t *mp,
    size_t msg_size, int flags, int *errorp,  boolean_t *force_pushp,
    sof_instance_t *filter)
{
        boolean_t force_push = B_TRUE;
        int space_left;
        sodirect_t *sodp = so->so_direct;

        ASSERT(errorp != NULL);
        *errorp = 0;
        if (mp == NULL) {
                if (so->so_downcalls->sd_recv_uio != NULL) {
                        mutex_enter(&so->so_lock);
                        /* the notify functions will drop the lock */
                        if (flags & MSG_OOB)
                                so_notify_oobdata(so, IS_SO_OOB_INLINE(so));
                        else
                                so_notify_data(so, msg_size);
                        return (0);
                }
                ASSERT(msg_size == 0);
                mutex_enter(&so->so_lock);
                goto space_check;
        }

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

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

        if (so->so_filter_active > 0) {
                for (; filter != NULL; filter = filter->sofi_prev) {
                        if (!SOF_INTERESTED(filter, data_in))
                                continue;
                        mp = (*filter->sofi_ops->sofop_data_in)(
                            (sof_handle_t)filter, filter->sofi_cookie, mp,
                            flags, &msg_size);
                        ASSERT(msgdsize(mp) == msg_size);
                        DTRACE_PROBE2(filter__data, (sof_instance_t), filter,
                            (mblk_t *), mp);
                        /* Data was consumed/dropped, just do space check */
```

```
1316                        if (msg_size == 0) {
1317                                mutex_enter(&so->so_lock);
1318                                goto space_check;
1319                        }
1320                }
1321        }

1323        if (flags & MSG_OOB) {
1324                so_queue_oob(so, mp, msg_size);
1325                mutex_enter(&so->so_lock);
1326                goto space_check;
1327        }

1329        if (force_pushp != NULL)
1330                force_push = *force_pushp;

1332        mutex_enter(&so->so_lock);
1333        if (so->so_state & (SS_FALLBACK_DRAIN | SS_FALLBACK_COMP)) {
1334                if (sodp != NULL)
1335                        SOD_DISABLE(sodp);
1336                mutex_exit(&so->so_lock);
1337                *errorp = EOPNOTSUPP;
1338                return (-1);
1339        }
1340        if (so->so_state & (SS_CANTRCVMORE | SS_CLOSING)) {
1341                freemsg(mp);
1342                if (sodp != NULL)
1343                        SOD_DISABLE(sodp);
1344                mutex_exit(&so->so_lock);
1345                return (0);
1346        }

1348        /* process the mblk via I/OAT if capable */
1349        if (sodp != NULL && sodp->sod_enabled) {
1350                if (DB_TYPE(mp) == M_DATA) {
1351                        sod_uioa_mblk_init(sodp, mp, msg_size);
1352                } else {
1353                        SOD_UIOAFINI(sodp);
1354                }
1355        }

1357        if (mp->b_next == NULL) {
1358                so_enqueue_msg(so, mp, msg_size);
1359        } else {
1360                do {
1361                        mblk_t *nmp;

1363                        if ((nmp = mp->b_next) != NULL) {
1364                                mp->b_next = NULL;
1365                        }
1366                        so_enqueue_msg(so, mp, msgdsize(mp));
1367                        mp = nmp;
1368                } while (mp != NULL);
1369        }

1371        space_left = so->so_rcvbuf - so->so_rcv_queued;
1372        if (space_left <= 0) {
1373                so->so_flowctrld = B_TRUE;
1374                *errorp = ENOSPC;
1375                space_left = -1;
1376        }

1378        if (force_push || so->so_rcv_queued >= so->so_rcv_thresh ||
1379            so->so_rcv_queued >= so->so_rcv_wanted) {
1380                SOCKET_TIMER_CANCEL(so);
1381                /*
```



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

1386                if (force_pushp != NULL)
1387                        *force_pushp = B_TRUE;
1388                goto done;
1389        } else if (so->so_rcv_timer_tid == 0) {
1390                /* Make sure the recv push timer is running */
1391                SOCKET_TIMER_START(so);
1392        }

1394 done_unlock:
1395        mutex_exit(&so->so_lock);
1396 done:
1397        return (space_left);

1399 space_check:
1400        space_left = so->so_rcvbuf - so->so_rcv_queued;
1401        if (space_left <= 0) {
1402                so->so_flowctrld = B_TRUE;
1403                *errorp = ENOSPC;
1404                space_left = -1;
1405        }
1406        goto done_unlock;
1407 }

1409 #pragma inline(so_queue_msg_impl)

1411 ssize_t
1412 so_queue_msg(sock_upper_handle_t sock_handle, mblk_t *mp,
1413     size_t msg_size, int flags, int *errorp,  boolean_t *force_pushp)
1414 {
1415        struct sonode *so = (struct sonode *)sock_handle;

1417        return (so_queue_msg_impl(so, mp, msg_size, flags, errorp, force_pushp,
1418            so->so_filter_bottom));
1419 }

1421 /*
1422  * Set the offset of where the oob data is relative to the bytes in
1423  * queued. Also generate SIGURG
1424  */
1425 void
1426 so_signal_oob(sock_upper_handle_t sock_handle, ssize_t offset)
1427 {
1428        struct sonode *so;

1430        ASSERT(offset >= 0);
1431        so = (struct sonode *)sock_handle;
1432        mutex_enter(&so->so_lock);
1433        if (so->so_direct != NULL)
1434                SOD_UIOAFINI(so->so_direct);

1436        /*
1437         * New urgent data on the way so forget about any old
1438         * urgent data.
1439         */
1440        so->so_state &= ~(SS_HAVEOOBDATA|SS_HADOOBDATA);

1442        /*
1443         * Record that urgent data is pending.
1444         */
1445        so->so_state |= SS_OOBPEND;

1447        if (so->so_oobmsg != NULL) {
```

```
1448                 dprintso(so, 1, ("sock: discarding old oob\n"));
1449                 freemsg(so->so_oobmsg);
1450                 so->so_oobmsg = NULL;
1451         }

1453         /*
1454          * set the offset where the urgent byte is
1455          */
1456         so->so_oobmark = so->so_rcv_queued + offset;
1457         if (so->so_oobmark == 0)
1458                 so->so_state |= SS_RCVATMARK;
1459         else
1460                 so->so_state &= ~SS_RCVATMARK;

1462         so_notify_oobsig(so);
1463 }

1465 /*
1466  * Queue the OOB byte
1467  */
1468 static void
1469 so_queue_oob(struct sonode *so, mblk_t *mp, size_t len)
1470 {
1471         mutex_enter(&so->so_lock);
1472         if (so->so_direct != NULL)
1473                 SOD_UIOAFINI(so->so_direct);

1475         ASSERT(mp != NULL);
1476         if (!IS_SO_OOB_INLINE(so)) {
1477                 so->so_oobmsg = mp;
1478                 so->so_state |= SS_HAVEOOBDATA;
1479         } else {
1480                 so_enqueue_msg(so, mp, len);
1481         }

1483         so_notify_oobdata(so, IS_SO_OOB_INLINE(so));
1484 }

1486 int
1487 so_close(struct sonode *so, int flag, struct cred *cr)
1488 {
1489         int error;

1491         /*
1492          * No new data will be enqueued once the CLOSING flag is set.
1493          */
1494         mutex_enter(&so->so_lock);
1495         so->so_state |= SS_CLOSING;
1496         ASSERT(so_verify_oobstate(so));
1497         so_rcv_flush(so);
1498         mutex_exit(&so->so_lock);

1500         if (so->so_filter_active > 0)
1501                 sof_sonode_closing(so);

1503         if (so->so_state & SS_ACCEPTCONN) {
1504                 /*
1505                  * We grab and release the accept lock to ensure that any
1506                  * thread about to insert a socket in so_newconn completes
1507                  * before we flush the queue. Any thread calling so_newconn
1508                  * after we drop the lock will observe the SS_CLOSING flag,
1509                  * which will stop it from inserting the socket in the queue.
1510                  */
1511                 mutex_enter(&so->so_acceptq_lock);
1512                 mutex_exit(&so->so_acceptq_lock);
```



```
1514                 so_acceptq_flush(so, B_TRUE);
1515         }

1517         error = (*so->so_downcalls->sd_close)(so->so_proto_handle, flag, cr);
1518         switch (error) {
1519         default:
1520                 /* Protocol made a synchronous close; remove proto ref */
1521                 VN_RELE(SOTOV(so));
1522                 break;
1523         case EINPROGRESS:
1524                 /*
1525                  * Protocol is in the process of closing, it will make a
1526                  * 'closed' upcall to remove the reference.
1527                  */
1528                 error = 0;
1529                 break;
1530         }

1532         return (error);
1533 }

1535 /*
1536  * Upcall made by the protocol when it's doing an asynchronous close. It
1537  * will drop the protocol's reference on the socket.
1538  */
1539 void
1540 so_closed(sock_upper_handle_t sock_handle)
1541 {
1542         struct sonode *so = (struct sonode *)sock_handle;

1544         VN_RELE(SOTOV(so));
1545 }

1547 conn_pid_node_list_hdr_t *
1548 so_get_sock_pid_list(sock_upper_handle_t sock_handle)
1549 {
1550         int sz, n = 0;
1551         pid_node_t                      *pn;
1552         conn_pid_node_t                 *cpn;
1553         conn_pid_node_list_hdr_t        *cph;
1554         struct sonode                   *so = (struct sonode *)sock_handle;

1556         mutex_enter(&so->so_pid_list_lock);

1558         n = list_size(&so->so_pid_list);
1559         sz = sizeof (conn_pid_node_list_hdr_t);
1560         sz += (n > 1)?((n - 1) * sizeof (conn_pid_node_t)):0;
1561         cph = kmem_zalloc(sz, KM_SLEEP);

1563         cph->cph_magic = CONN_PID_NODE_LIST_HDR_MAGIC;
1564         cph->cph_contents = CONN_PID_NODE_LIST_HDR_SOC;
1565         cph->cph_pn_cnt = n;
1566         cph->cph_tot_size = sz;
1567         cph->cph_flags = 0;
1568         cph->cph_optional1 = 0;
1569         cph->cph_optional2 = 0;

1571         if (cph->cph_pn_cnt > 0) {
1572                 cpn = cph->cph_cpns;
1573                 pn = list_head(&so->so_pid_list);
1574                 while (pn != NULL) {
1575                         PIDNODE2CONNPIDNODE(pn, cpn);
1576                         pn = list_next(&so->so_pid_list, pn);
1577                         cpn++;
1578                 }
1579         }
```

```
1581         mutex_exit(&so->so_pid_list_lock);

1583         return (cph);
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_list
  40         so_closed
1989 };
______unchanged_portion_omitted_
```

```
**********************************************************
   12887 Sun Aug  9 12:47:45 2015
new/usr/src/uts/common/fs/sockfs/sockcommon_vnops.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_


 109 /*
 110  * generic vnode ops
 111  */

 113 /*ARGSUSED*/
 114 static int
 115 socket_vop_open(struct vnode **vpp, int flag, struct cred *cr,
 116     caller_context_t *ct)
 117 {
 118         struct vnode *vp = *vpp;
 119         struct sonode *so = VTOSO(vp);

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

 126         sonode_insert_pid(so, curproc);

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

 132         return (0);
 133 }

 135 /*ARGSUSED*/
 136 static int
 137 socket_vop_close(struct vnode *vp, int flag, int count, offset_t offset,
 138     struct cred *cr, caller_context_t *ct)
 139 {
 140         struct sonode *so;
 141         int error = 0;

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

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

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

 152         if (count > 1) {
 153                 dprint(2, ("socket_vop_close: count %d\n", count));
 154                 return (0);
 155         }

 157         mutex_enter(&so->so_lock);
 158         if (--so->so_count == 0) {
 159                 /*
 160                  * Initiate connection shutdown.
 161                  */
 162                 mutex_exit(&so->so_lock);
 163                 error = socket_close_internal(so, flag, cr);
 164         } else {
 165                 mutex_exit(&so->so_lock);
 166         }
```



```
 168         return (error);
 169 }

 171 /*ARGSUSED2*/
 172 static int
 173 socket_vop_read(struct vnode *vp, struct uio *uiop, int ioflag, struct cred *cr,
 174     caller_context_t *ct)
 175 {
 176         struct sonode *so = VTOSO(vp);
 177         struct nmsghdr lmsg;

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

 182         return (socket_recvmsg(so, &lmsg, uiop, cr));
 183 }

 185 /*ARGSUSED2*/
 186 static int
 187 socket_vop_write(struct vnode *vp, struct uio *uiop, int ioflag,
 188     struct cred *cr, caller_context_t *ct)
 189 {
 190         struct sonode *so = VTOSO(vp);
 191         struct nmsghdr lmsg;

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

 196         if (!(so->so_mode & SM_BYTESTREAM)) {
 197                 /*
 198                  * If the socket is not byte stream set MSG_EOR
 199                  */
 200                 lmsg.msg_flags = MSG_EOR;
 201         }

 203         return (socket_sendmsg(so, &lmsg, uiop, cr));
 204 }

 206 /*ARGSUSED4*/
 207 static int
 208 socket_vop_ioctl(struct vnode *vp, int cmd, intptr_t arg, int mode,
 209     struct cred *cr, int32_t *rvalp, caller_context_t *ct)
 210 {
 211         struct sonode *so = VTOSO(vp);

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

 215         switch (cmd) {
 216                 case F_FORKED: {
 217                         if (cr != kcred)
 218                                 return (-1);
 219                         sonode_insert_pid(so, (proc_t *)arg);
 220                         return (0);
 221                 }

 223                 case F_CLOSED: {
 224                         if (cr != kcred)
 225                                 return (-1);
 226                         sonode_remove_pid(so, (proc_t *)arg);
 227                         return (0);
 228                 }
 229         }
 230 #endif /* ! codereview */

 232         return (socket_ioctl(so, cmd, arg, mode, cr, rvalp));
```

```
233 }

235 /*
236  * Allow any flags. Record FNDELAY and FNONBLOCK so that they can be inherited
237  * from listener to acceptor.
238  */
239 /* ARGSUSED */
240 static int
241 socket_vop_setfl(vnode_t *vp, int oflags, int nflags, cred_t *cr,
242     caller_context_t *ct)
243 {
244         struct sonode *so = VTOSO(vp);
245         int error = 0;

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

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

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

271                 /*
272                  * For non-TPI sockets all we have to do is set/remove the
273                  * SS_ASYNC bit, but for TPI it is more involved. For that
274                  * reason we delegate the job to the protocol's ioctl handler.
275                  */
276                 error = socket_ioctl(so, FIOASYNC, (intptr_t)&async, FKIOCTL,
277                     cr, &rv);
278         }
279         return (error);
280 }


283 /*
284  * Get the made up attributes for the vnode.
285  * 4.3BSD returns the current time for all the timestamps.
286  * 4.4BSD returns 0 for all the timestamps.
287  * Here we use the access and modified times recorded in the sonode.
288  *
289  * Just like in BSD there is not effect on the underlying file system node
290  * bound to an AF_UNIX pathname.
291  *
292  * When sockmod has been popped this will act just like a stream. Since
293  * a socket is always a clone there is no need to inspect the attributes
294  * of the "realvp".
295  */
296 /* ARGSUSED */
297 int
298 socket_vop_getattr(struct vnode *vp, struct vattr *vap, int flags,
```



```
299     struct cred *cr, caller_context_t *ct)
300 {
301         dev_t           fsid;
302         struct sonode   *so;
303         static int      sonode_shift = 0;

305         /*
306          * Calculate the amount of bitshift to a sonode pointer which will
307          * still keep it unique.  See below.
308          */
309         if (sonode_shift == 0)
310                 sonode_shift = highbit(sizeof (struct sonode));
311         ASSERT(sonode_shift > 0);

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

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

339         /*
340          * We need to zero out the va_rdev to avoid some fstats getting
341          * EOVERFLOW.  This also mimics SunOS 4.x and BSD behavior.
342          */
343         vap->va_rdev = (dev_t)0;
344         vap->va_blksize = MAXBSIZE;
345         vap->va_nblocks = btod(vap->va_size);

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

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

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

```
366         return (0);
367 }

369 /*
370  * Set attributes.
371  * Just like in BSD there is not effect on the underlying file system node
372  * bound to an AF_UNIX pathname.
373  *
374  * When sockmod has been popped this will act just like a stream. Since
375  * a socket is always a clone there is no need to modify the attributes
376  * of the "realvp".
377  */
378 /* ARGSUSED */
379 int
380 socket_vop_setattr(struct vnode *vp, struct vattr *vap, int flags,
381     struct cred *cr, caller_context_t *ct)
382 {
383         struct sonode *so = VTOSO(vp);

385         /*
386          * If times were changed, and we have a STREAMS socket, then update
387          * the sonode.
388          */
389         if (!SOCK_IS_NONSTR(so)) {
390                 sotpi_info_t *sti = SOTOTPI(so);

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

402         return (0);
403 }

405 /*
406  * Check if user is allowed to access vp. For non-STREAMS based sockets,
407  * there might not be a device attached to the file system. So for those
408  * types of sockets there are no permissions to check.
409  *
410  * XXX Should there be some other mechanism to check access rights?
411  */
412 /*ARGSUSED*/
413 int
414 socket_vop_access(struct vnode *vp, int mode, int flags, struct cred *cr,
415     caller_context_t *ct)
416 {
417         struct sonode *so = VTOSO(vp);

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

427 /*
428  * 4.3BSD and 4.4BSD fail a fsync on a socket with EINVAL.
429  * This code does the same to be compatible and also to not give an
430  * application the impression that the data has actually been "synced"
```



```
431  * to the other end of the connection.
432  */
433 /* ARGSUSED */
434 int
435 socket_vop_fsync(struct vnode *vp, int syncflag, struct cred *cr,
436     caller_context_t *ct)
437 {
438         return (EINVAL);
439 }

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

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

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

457         /*
458          * Drop the temporary hold by vn_rele now
459          */
460         if (--vp->v_count != 0) {
461                 mutex_exit(&vp->v_lock);
462                 return;
463         }
464         mutex_exit(&vp->v_lock);


467         ASSERT(!vn_has_cached_data(vp));

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

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

480 /*
481  * Sockets are not seekable.
482  * (and there is a bug to fix STREAMS to make them fail this as well).
483  */
484 /*ARGSUSED*/
485 int
486 socket_vop_seek(struct vnode *vp, offset_t ooff, offset_t *noffp,
487     caller_context_t *ct)
488 {
489         return (ESPIPE);
490 }

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

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

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

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

```
**********************************************************
   49697 Sun Aug  9 12:47:48 2015
new/usr/src/uts/common/fs/sockfs/socksubr.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) 1995, 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/conf.h>
#include <sys/cred.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/vnode.h>
#include <sys/debug.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <sys/file.h>
#include <sys/open.h>
#include <sys/user.h>
#include <sys/termios.h>
#include <sys/stream.h>
#include <sys/strsubr.h>
#include <sys/strsun.h>
#include <sys/esunddi.h>
#include <sys/flock.h>
#include <sys/modctl.h>
#include <sys/cmn_err.h>
#include <sys/mkdev.h>
#include <sys/pathname.h>
#include <sys/ddi.h>
#include <sys/stat.h>
#include <sys/fs/snode.h>
#include <sys/fs/dv_node.h>
#include <sys/zone.h>

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



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

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

#include <c2/audit.h>

#include <fs/sockfs/nl7c.h>
#include <fs/sockfs/sockcommon.h>
#include <fs/sockfs/sockfilter_impl.h>
#include <fs/sockfs/socktpi.h>
#include <fs/sockfs/socktpi_impl.h>
#include <fs/sockfs/sodirect.h>

/*
 * Macros that operate on struct cmsghdr.
 * The CMSG_VALID macro does not assume that the last option buffer is padded.
 */
#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)))
#define SO_LOCK_WAKEUP_TIME     3000    /* Wakeup time in milliseconds */

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

struct socklist socklist;

struct kmem_cache *socket_cache;

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

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

extern void sendfile_init();

extern void nl7c_init(void);

extern int modrootloaded;

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

/*
```

```
 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;
 758                 /* add curproc to the pid list associated with that file */
 759                 if (fp->f_vnode != NULL)
 760                         (void) VOP_IOCTL(fp->f_vnode, F_FORKED,
 761                             (intptr_t)curproc, FKIOCTL, kcred, NULL, NULL);

 763 #endif /* ! codereview */
 764                 if (AU_AUDITING())
 765                         audit_fdrecv(fd, fp);
 766                 dprint(1, ("fdbuf_extract: [%d] = %d, %p refcnt %d\n",
 767                     i, fd, (void *)fp, fp->f_count));
 768         }
 769         return (0);

 771 cleanup:
 772         /*
 773          * Undo whatever partial work the loop above has done.
 774          */
 775         {
 776                 int j;

 778                 rp = (int *)rights;
 779                 for (j = 0; j < i; j++) {
 780                         dprint(0,
 781                             ("fdbuf_extract: cleanup[%d] = %d\n", j, *rp));
 782                         (void) closeandsetf(*rp++, NULL);
 783                 }
 784         }

 786         return (EMFILE);
 787 }

 789 /*
```

```
 * Insert file descriptors into an fdbuf.
 * Returns a kmem_alloc'ed fdbuf. The fdbuf should be freed
 * by calling fdbuf_free().
 */
int
fdbuf_create(void *rights, int rightslen, struct fdbuf **fdbufp)
{
        int             numfd, i;
        int             *fds;
        struct file     *fp;
        struct fdbuf    *fdbuf;
        int             fdbufsize;

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

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

        fdbufsize = (int)FDBUF_HDRSIZE + (numfd * (int)sizeof (struct file *));
        fdbuf = kmem_alloc(fdbufsize, KM_SLEEP);
        fdbuf->fd_size = fdbufsize;
        fdbuf->fd_numfd = 0;
        fdbuf->fd_ebuf = NULL;
        fdbuf->fd_ebuflen = 0;
        fds = (int *)rights;
        for (i = 0; i < numfd; i++) {
                if ((fp = getf(fds[i])) == NULL) {
                        fdbuf_free(fdbuf);
                        return (EBADF);
                }
                dprint(1, ("fdbuf_create: [%d] = %d, %p refcnt %d\n",
                    i, fds[i], (void *)fp, fp->f_count));
                mutex_enter(&fp->f_tlock);
                fp->f_count++;
                mutex_exit(&fp->f_tlock);
                /*
                 * The maximum alignment for fdbuf (or any option header
                 * and its value) it 4 bytes. On a LP64 kernel, the alignment
                 * is not sufficient for pointers (fd_fds in this case). Since
                 * we just did a kmem_alloc (we get a double word alignment),
                 * we don't need to do anything on the send side (we loose
                 * the double word alignment because fdbuf goes after an
                 * option header (eg T_unitdata_req) which is only 4 byte
                 * aligned). We take care of this when we extract the file
                 * descriptor in fdbuf_extract or fdbuf_free.
                 */
                fdbuf->fd_fds[i] = fp;
                fdbuf->fd_numfd++;
                releasef(fds[i]);
                if (AU_AUDITING())
                        audit_fdsend(fds[i], fp, 0);
        }
        *fdbufp = fdbuf;
        return (0);
}

static int
fdbuf_optlen(int rightslen)
{
        int numfd;

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

        return ((int)FDBUF_HDRSIZE + (numfd * (int)sizeof (struct file *)));
}

static t_uscalar_t
```



```
fdbuf_cmsglen(int fdbuflen)
{
        return (t_uscalar_t)((fdbuflen - FDBUF_HDRSIZE) /
            (int)sizeof (struct file *) * (int)sizeof (int));
}


/*
 * Return non-zero if the mblk and fdbuf are consistent.
 */
static int
fdbuf_verify(mblk_t *mp, struct fdbuf *fdbuf, int fdbuflen)
{
        if (fdbuflen >= FDBUF_HDRSIZE &&
            fdbuflen == fdbuf->fd_size) {
                frtn_t *frp = mp->b_datap->db_frtnp;
                /*
                 * Check that the SO_FILEP portion of the
                 * message has not been modified by
                 * the loopback transport. The sending sockfs generates
                 * a message that is esballoc'ed with the free function
                 * being fdbuf_free() and where free_arg contains the
                 * identical information as the SO_FILEP content.
                 *
                 * If any of these constraints are not satisfied we
                 * silently ignore the option.
                 */
                ASSERT(mp);
                if (frp != NULL &&
                    frp->free_func == fdbuf_free &&
                    frp->free_arg != NULL &&
                    bcmp(frp->free_arg, fdbuf, fdbuflen) == 0) {
                        dprint(1, ("fdbuf_verify: fdbuf %p len %d\n",
                            (void *)fdbuf, fdbuflen));
                        return (1);
                } else {
                        zcmn_err(getzoneid(), CE_WARN,
                            "sockfs: mismatched fdbuf content (%p)",
                            (void *)mp);
                        return (0);
                }
        } else {
                zcmn_err(getzoneid(), CE_WARN,
                    "sockfs: mismatched fdbuf len %d, %d\n",
                    fdbuflen, fdbuf->fd_size);
                return (0);
        }
}

/*
 * When the file descriptors returned by sorecvmsg can not be passed
 * to the application this routine will cleanup the references on
 * the files. Start at startoff bytes into the buffer.
 */
static void
close_fds(void *fdbuf, int fdbuflen, int startoff)
{
        int *fds = (int *)fdbuf;
        int numfd = fdbuflen / (int)sizeof (int);
        int i;

        dprint(1, ("close_fds(%p, %d, %d)\n", fdbuf, fdbuflen, startoff));

        for (i = 0; i < numfd; i++) {
                if (startoff < 0)
                        startoff = 0;
```

```
922                 if (startoff < (int)sizeof (int)) {
923                         /*
924                          * This file descriptor is partially or fully after
925                          * the offset
926                          */
927                         dprint(0,
928                             ("close_fds: cleanup[%d] = %d\n", i, fds[i]));
929                         (void) closeandsetf(fds[i], NULL);
930                 }
931                 startoff -= (int)sizeof (int);
932         }
933 }

935 /*
936  * Close all file descriptors contained in the control part starting at
937  * the startoffset.
938  */
939 void
940 so_closefds(void *control, t_uscalar_t controllen, int oldflg,
941     int startoff)
942 {
943         struct cmsghdr *cmsg;

945         if (control == NULL)
946                 return;

948         if (oldflg) {
949                 close_fds(control, controllen, startoff);
950                 return;
951         }
952         /* Scan control part for file descriptors. */
953         for (cmsg = (struct cmsghdr *)control;
954             CMSG_VALID(cmsg, control, (uintptr_t)control + controllen);
955             cmsg = CMSG_NEXT(cmsg)) {
956                 if (cmsg->cmsg_level == SOL_SOCKET &&
957                     cmsg->cmsg_type == SCM_RIGHTS) {
958                         close_fds(CMSG_CONTENT(cmsg),
959                             (int)CMSG_CONTENTLEN(cmsg),
960                             startoff - (int)sizeof (struct cmsghdr));
961                 }
962                 startoff -= cmsg->cmsg_len;
963         }
964 }

966 /*
967  * Returns a pointer/length for the file descriptors contained
968  * in the control buffer. Returns with *fdlenp == -1 if there are no
969  * file descriptor options present. This is different than there being
970  * a zero-length file descriptor option.
971  * Fail if there are multiple SCM_RIGHT cmsgs.
972  */
973 int
974 so_getfdopt(void *control, t_uscalar_t controllen, int oldflg,
975     void **fdsp, int *fdlenp)
976 {
977         struct cmsghdr *cmsg;
978         void *fds;
979         int fdlen;

981         if (control == NULL) {
982                 *fdsp = NULL;
983                 *fdlenp = -1;
984                 return (0);
985         }

987         if (oldflg) {
```



```
988                 *fdsp = control;
989                 if (controllen == 0)
990                         *fdlenp = -1;
991                 else
992                         *fdlenp = controllen;
993                 dprint(1, ("so_getfdopt: old %d\n", *fdlenp));
994                 return (0);
995         }

997         fds = NULL;
998         fdlen = 0;

1000        for (cmsg = (struct cmsghdr *)control;
1001            CMSG_VALID(cmsg, control, (uintptr_t)control + controllen);
1002            cmsg = CMSG_NEXT(cmsg)) {
1003                if (cmsg->cmsg_level == SOL_SOCKET &&
1004                    cmsg->cmsg_type == SCM_RIGHTS) {
1005                        if (fds != NULL)
1006                                return (EINVAL);
1007                        fds = CMSG_CONTENT(cmsg);
1008                        fdlen = (int)CMSG_CONTENTLEN(cmsg);
1009                        dprint(1, ("so_getfdopt: new %lu\n",
1010                            (size_t)CMSG_CONTENTLEN(cmsg)));
1011                }
1012        }
1013        if (fds == NULL) {
1014                dprint(1, ("so_getfdopt: NONE\n"));
1015                *fdlenp = -1;
1016        } else
1017                *fdlenp = fdlen;
1018        *fdsp = fds;
1019        return (0);
1020 }

1022 /*
1023  * Return the length of the options including any file descriptor options.
1024  */
1025 t_uscalar_t
1026 so_optlen(void *control, t_uscalar_t controllen, int oldflg)
1027 {
1028         struct cmsghdr *cmsg;
1029         t_uscalar_t optlen = 0;
1030         t_uscalar_t len;

1032         if (control == NULL)
1033                 return (0);

1035         if (oldflg)
1036                 return ((t_uscalar_t)(sizeof (struct T_opthdr) +
1037                     fdbuf_optlen(controllen)));

1039         for (cmsg = (struct cmsghdr *)control;
1040             CMSG_VALID(cmsg, control, (uintptr_t)control + controllen);
1041             cmsg = CMSG_NEXT(cmsg)) {
1042                 if (cmsg->cmsg_level == SOL_SOCKET &&
1043                     cmsg->cmsg_type == SCM_RIGHTS) {
1044                         len = fdbuf_optlen((int)CMSG_CONTENTLEN(cmsg));
1045                 } else {
1046                         len = (t_uscalar_t)CMSG_CONTENTLEN(cmsg);
1047                 }
1048                 optlen += (t_uscalar_t)(_TPI_ALIGN_TOPT(len) +
1049                     sizeof (struct T_opthdr));
1050         }
1051         dprint(1, ("so_optlen: controllen %d, flg %d -> optlen %d\n",
1052             controllen, oldflg, optlen));
1053         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,
    void *control, t_uscalar_t controllen)
{
        struct T_opthdr *tohp;
        struct cmsghdr *cmsg;
        struct fdbuf *fdbuf;
```

```
1186         int fdbuflen;
1187         int error;
1188 #if defined(DEBUG) || defined(__lint)
1189         struct cmsghdr *cend = (struct cmsghdr *)
1190             (((uint8_t *)control) + ROUNDUP_cmsglen(controllen));
1191 #endif
1192         cmsg = (struct cmsghdr *)control;

1194         ASSERT(__TPI_TOPT_ISALIGNED(opt));

1196         for (tohp = (struct T_opthdr *)opt;
1197             tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1198             tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1199                 dprint(1, ("so_opt2cmsg: level 0x%x, name %d, len %d\n",
1200                     tohp->level, tohp->name, tohp->len));

1202                 if (tohp->level == SOL_SOCKET &&
1203                     (tohp->name == SO_SRCADDR ||
1204                     tohp->name == SO_UNIX_CLOSE)) {
1205                         continue;
1206                 }
1207                 ASSERT((uintptr_t)cmsg <= (uintptr_t)control + controllen);
1208                 if (tohp->level == SOL_SOCKET && tohp->name == SO_FILEP) {
1209                         fdbuf = (struct fdbuf *)_TPI_TOPT_DATA(tohp);
1210                         fdbuflen = (int)_TPI_TOPT_DATALEN(tohp);

1212                         if (!fdbuf_verify(mp, fdbuf, fdbuflen))
1213                                 return (EPROTO);
1214                         if (oldflg) {
1215                                 error = fdbuf_extract(fdbuf, control,
1216                                     (int)controllen);
1217                                 if (error != 0)
1218                                         return (error);
1219                                 continue;
1220                         } else {
1221                                 int fdlen;

1223                                 fdlen = (int)fdbuf_cmsglen(
1224                                     (int)_TPI_TOPT_DATALEN(tohp));

1226                                 cmsg->cmsg_level = tohp->level;
1227                                 cmsg->cmsg_type = SCM_RIGHTS;
1228                                 cmsg->cmsg_len = (socklen_t)(fdlen +
1229                                     sizeof (struct cmsghdr));

1231                                 error = fdbuf_extract(fdbuf,
1232                                     CMSG_CONTENT(cmsg), fdlen);
1233                                 if (error != 0)
1234                                         return (error);
1235                         }
1236                 } else if (tohp->level == SOL_SOCKET &&
1237                     tohp->name == SCM_TIMESTAMP) {
1238                         timestruc_t *timestamp;

1240                         if (oldflg)
1241                                 continue;

1243                         cmsg->cmsg_level = tohp->level;
1244                         cmsg->cmsg_type = tohp->name;

1246                         timestamp =
1247                             (timestruc_t *)P2ROUNDUP((intptr_t)&tohp[1],
1248                             sizeof (intptr_t));

1250                         if (get_udatamodel() == DATAMODEL_NATIVE) {
1251                                 struct timeval tv;
```



```
1253                                 cmsg->cmsg_len = sizeof (struct timeval) +
1254                                     sizeof (struct cmsghdr);
1255                                 tv.tv_sec = timestamp->tv_sec;
1256                                 tv.tv_usec = timestamp->tv_nsec /
1257                                     (NANOSEC / MICROSEC);
1258                                 /*
1259                                  * on LP64 systems, the struct timeval in
1260                                  * the destination will not be 8-byte aligned,
1261                                  * so use bcopy to avoid alignment trouble
1262                                  */
1263                                 bcopy(&tv, CMSG_CONTENT(cmsg), sizeof (tv));
1264                         } else {
1265                                 struct timeval32 *time32;

1267                                 cmsg->cmsg_len = sizeof (struct timeval32) +
1268                                     sizeof (struct cmsghdr);
1269                                 time32 = (struct timeval32 *)CMSG_CONTENT(cmsg);
1270                                 time32->tv_sec = (time32_t)timestamp->tv_sec;
1271                                 time32->tv_usec =
1272                                     (int32_t)(timestamp->tv_nsec /
1273                                     (NANOSEC / MICROSEC));
1274                         }

1276                 } else {
1277                         if (oldflg)
1278                                 continue;

1280                         cmsg->cmsg_level = tohp->level;
1281                         cmsg->cmsg_type = tohp->name;
1282                         cmsg->cmsg_len = (socklen_t)(_TPI_TOPT_DATALEN(tohp) +
1283                             sizeof (struct cmsghdr));

1285                         /* copy content to control data part */
1286                         bcopy(&tohp[1], CMSG_CONTENT(cmsg),
1287                             CMSG_CONTENTLEN(cmsg));
1288                 }
1289                 /* move to next CMSG structure! */
1290                 cmsg = CMSG_NEXT(cmsg);
1291         }
1292         dprint(1, ("so_opt2cmsg: buf %p len %d; cend %p; final cmsg %p\n",
1293             control, controllen, (void *)cend, (void *)cmsg));
1294         ASSERT(cmsg <= cend);
1295         return (0);
1296 }

1298 /*
1299  * Extract the SO_SRCADDR option value if present.
1300  */
1301 void
1302 so_getopt_srcaddr(void *opt, t_uscalar_t optlen, void **srcp,
1303     t_uscalar_t *srclenp)
1304 {
1305         struct T_opthdr         *tohp;

1307         ASSERT(__TPI_TOPT_ISALIGNED(opt));

1309         ASSERT(srcp != NULL && srclenp != NULL);
1310         *srcp = NULL;
1311         *srclenp = 0;

1313         for (tohp = (struct T_opthdr *)opt;
1314             tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1315             tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1316                 dprint(1, ("so_getopt_srcaddr: level 0x%x, name %d, len %d\n",
1317                     tohp->level, tohp->name, tohp->len));
```

```
1318                 if (tohp->level == SOL_SOCKET &&
1319                     tohp->name == SO_SRCADDR) {
1320                         *srcp = _TPI_TOPT_DATA(tohp);
1321                         *srclenp = (t_uscalar_t)_TPI_TOPT_DATALEN(tohp);
1322                 }
1323         }
1324 }

1326 /*
1327  * Verify if the SO_UNIX_CLOSE option is present.
1328  */
1329 int
1330 so_getopt_unix_close(void *opt, t_uscalar_t optlen)
1331 {
1332         struct T_opthdr         *tohp;

1334         ASSERT(__TPI_TOPT_ISALIGNED(opt));

1336         for (tohp = (struct T_opthdr *)opt;
1337             tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1338             tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1339                 dprint(1,
1340                     ("so_getopt_unix_close: level 0x%x, name %d, len %d\n",
1341                     tohp->level, tohp->name, tohp->len));
1342                 if (tohp->level == SOL_SOCKET &&
1343                     tohp->name == SO_UNIX_CLOSE)
1344                         return (1);
1345         }
1346         return (0);
1347 }

1349 /*
1350  * Allocate an M_PROTO message.
1351  *
1352  * If allocation fails the behavior depends on sleepflg:
1353  *      _ALLOC_NOSLEEP  fail immediately
1354  *      _ALLOC_INTR     sleep for memory until a signal is caught
1355  *      _ALLOC_SLEEP    sleep forever. Don't return NULL.
1356  */
1357 mblk_t *
1358 soallocproto(size_t size, int sleepflg, cred_t *cr)
1359 {
1360         mblk_t  *mp;

1362         /* Round up size for reuse */
1363         size = MAX(size, 64);
1364         if (cr != NULL)
1365                 mp = allocb_cred(size, cr, curproc->p_pid);
1366         else
1367                 mp = allocb(size, BPRI_MED);

1369         if (mp == NULL) {
1370                 int error;      /* Dummy - error not returned to caller */

1372                 switch (sleepflg) {
1373                 case _ALLOC_SLEEP:
1374                         if (cr != NULL) {
1375                                 mp = allocb_cred_wait(size, STR_NOSIG, &error,
1376                                     cr, curproc->p_pid);
1377                         } else {
1378                                 mp = allocb_wait(size, BPRI_MED, STR_NOSIG,
1379                                     &error);
1380                         }
1381                         ASSERT(mp);
1382                         break;
1383                 case _ALLOC_INTR:
```



```
1384                         if (cr != NULL) {
1385                                 mp = allocb_cred_wait(size, 0, &error, cr,
1386                                     curproc->p_pid);
1387                         } else {
1388                                 mp = allocb_wait(size, BPRI_MED, 0, &error);
1389                         }
1390                         if (mp == NULL) {
1391                                 /* Caught signal while sleeping for memory */
1392                                 eprintline(ENOBUFS);
1393                                 return (NULL);
1394                         }
1395                         break;
1396                 case _ALLOC_NOSLEEP:
1397                 default:
1398                         eprintline(ENOBUFS);
1399                         return (NULL);
1400                 }
1401         }
1402         DB_TYPE(mp) = M_PROTO;
1403         return (mp);
1404 }

1406 /*
1407  * Allocate an M_PROTO message with a single component.
1408  * len is the length of buf. size is the amount to allocate.
1409  *
1410  * buf can be NULL with a non-zero len.
1411  * This results in a bzero'ed chunk being placed the message.
1412  */
1413 mblk_t *
1414 soallocproto1(const void *buf, ssize_t len, ssize_t size, int sleepflg,
1415     cred_t *cr)
1416 {
1417         mblk_t  *mp;

1419         if (size == 0)
1420                 size = len;

1422         ASSERT(size >= len);
1423         /* Round up size for reuse */
1424         size = MAX(size, 64);
1425         mp = soallocproto(size, sleepflg, cr);
1426         if (mp == NULL)
1427                 return (NULL);
1428         mp->b_datap->db_type = M_PROTO;
1429         if (len != 0) {
1430                 if (buf != NULL)
1431                         bcopy(buf, mp->b_wptr, len);
1432                 else
1433                         bzero(mp->b_wptr, len);
1434                 mp->b_wptr += len;
1435         }
1436         return (mp);
1437 }

1439 /*
1440  * Append buf/len to mp.
1441  * The caller has to ensure that there is enough room in the mblk.
1442  *
1443  * buf can be NULL with a non-zero len.
1444  * This results in a bzero'ed chunk being placed the message.
1445  */
1446 void
1447 soappendmsg(mblk_t *mp, const void *buf, ssize_t len)
1448 {
1449         ASSERT(mp);
```

```
1451         if (len != 0) {
1452                 /* Assert for room left */
1453                 ASSERT(mp->b_datap->db_lim - mp->b_wptr >= len);
1454                 if (buf != NULL)
1455                         bcopy(buf, mp->b_wptr, len);
1456                 else
1457                         bzero(mp->b_wptr, len);
1458         }
1459         mp->b_wptr += len;
1460 }

1462 /*
1463  * Create a message using two kernel buffers.
1464  * If size is set that will determine the allocation size (e.g. for future
1465  * soappendmsg calls). If size is zero it is derived from the buffer
1466  * lengths.
1467  */
1468 mblk_t *
1469 soallocproto2(const void *buf1, ssize_t len1, const void *buf2, ssize_t len2,
1470     ssize_t size, int sleepflg, cred_t *cr)
1471 {
1472         mblk_t *mp;

1474         if (size == 0)
1475                 size = len1 + len2;
1476         ASSERT(size >= len1 + len2);

1478         mp = soallocproto1(buf1, len1, size, sleepflg, cr);
1479         if (mp)
1480                 soappendmsg(mp, buf2, len2);
1481         return (mp);
1482 }

1484 /*
1485  * Create a message using three kernel buffers.
1486  * If size is set that will determine the allocation size (for future
1487  * soappendmsg calls). If size is zero it is derived from the buffer
1488  * lengths.
1489  */
1490 mblk_t *
1491 soallocproto3(const void *buf1, ssize_t len1, const void *buf2, ssize_t len2,
1492     const void *buf3, ssize_t len3, ssize_t size, int sleepflg, cred_t *cr)
1493 {
1494         mblk_t *mp;

1496         if (size == 0)
1497                 size = len1 + len2 +len3;
1498         ASSERT(size >= len1 + len2 + len3);

1500         mp = soallocproto1(buf1, len1, size, sleepflg, cr);
1501         if (mp != NULL) {
1502                 soappendmsg(mp, buf2, len2);
1503                 soappendmsg(mp, buf3, len3);
1504         }
1505         return (mp);
1506 }

1508 #ifdef DEBUG
1509 char *
1510 pr_state(uint_t state, uint_t mode)
1511 {
1512         static char buf[1024];

1514         buf[0] = 0;
1515         if (state & SS_ISCONNECTED)
```



```
1516                 (void) strcat(buf, "ISCONNECTED ");
1517         if (state & SS_ISCONNECTING)
1518                 (void) strcat(buf, "ISCONNECTING ");
1519         if (state & SS_ISDISCONNECTING)
1520                 (void) strcat(buf, "ISDISCONNECTING ");
1521         if (state & SS_CANTSENDMORE)
1522                 (void) strcat(buf, "CANTSENDMORE ");

1524         if (state & SS_CANTRCVMORE)
1525                 (void) strcat(buf, "CANTRCVMORE ");
1526         if (state & SS_ISBOUND)
1527                 (void) strcat(buf, "ISBOUND ");
1528         if (state & SS_NDELAY)
1529                 (void) strcat(buf, "NDELAY ");
1530         if (state & SS_NONBLOCK)
1531                 (void) strcat(buf, "NONBLOCK ");

1533         if (state & SS_ASYNC)
1534                 (void) strcat(buf, "ASYNC ");
1535         if (state & SS_ACCEPTCONN)
1536                 (void) strcat(buf, "ACCEPTCONN ");
1537         if (state & SS_SAVEDEOR)
1538                 (void) strcat(buf, "SAVEDEOR ");

1540         if (state & SS_RCVATMARK)
1541                 (void) strcat(buf, "RCVATMARK ");
1542         if (state & SS_OOBPEND)
1543                 (void) strcat(buf, "OOBPEND ");
1544         if (state & SS_HAVEOOBDATA)
1545                 (void) strcat(buf, "HAVEOOBDATA ");
1546         if (state & SS_HADOOBDATA)
1547                 (void) strcat(buf, "HADOOBDATA ");

1549         if (mode & SM_PRIV)
1550                 (void) strcat(buf, "PRIV ");
1551         if (mode & SM_ATOMIC)
1552                 (void) strcat(buf, "ATOMIC ");
1553         if (mode & SM_ADDR)
1554                 (void) strcat(buf, "ADDR ");
1555         if (mode & SM_CONNREQUIRED)
1556                 (void) strcat(buf, "CONNREQUIRED ");

1558         if (mode & SM_FDPASSING)
1559                 (void) strcat(buf, "FDPASSING ");
1560         if (mode & SM_EXDATA)
1561                 (void) strcat(buf, "EXDATA ");
1562         if (mode & SM_OPTDATA)
1563                 (void) strcat(buf, "OPTDATA ");
1564         if (mode & SM_BYTESTREAM)
1565                 (void) strcat(buf, "BYTESTREAM ");
1566         return (buf);
1567 }

1569 char *
1570 pr_addr(int family, struct sockaddr *addr, t_uscalar_t addrlen)
1571 {
1572         static char buf[1024];

1574         if (addr == NULL || addrlen == 0) {
1575                 (void) sprintf(buf, "(len %d) %p", addrlen, (void *)addr);
1576                 return (buf);
1577         }
1578         switch (family) {
1579         case AF_INET: {
1580                 struct sockaddr_in sin;
```

```
1582                 bcopy(addr, &sin, sizeof (sin));

1584                 (void) sprintf(buf, "(len %d) %x/%d",
1585                     addrlen, ntohl(sin.sin_addr.s_addr), ntohs(sin.sin_port));
1586                 break;
1587         }
1588         case AF_INET6: {
1589                 struct sockaddr_in6 sin6;
1590                 uint16_t *piece = (uint16_t *)&sin6.sin6_addr;

1592                 bcopy((char *)addr, (char *)&sin6, sizeof (sin6));
1593                 (void) sprintf(buf, "(len %d) %x:%x:%x:%x:%x:%x:%x:%x/%d",
1594                     addrlen,
1595                     ntohs(piece[0]), ntohs(piece[1]),
1596                     ntohs(piece[2]), ntohs(piece[3]),
1597                     ntohs(piece[4]), ntohs(piece[5]),
1598                     ntohs(piece[6]), ntohs(piece[7]),
1599                     ntohs(sin6.sin6_port));
1600                 break;
1601         }
1602         case AF_UNIX: {
1603                 struct sockaddr_un *soun = (struct sockaddr_un *)addr;

1605                 (void) sprintf(buf, "(len %d) %s", addrlen,
1606                     (soun == NULL) ? "(none)" : soun->sun_path);
1607                 break;
1608         }
1609         default:
1610                 (void) sprintf(buf, "(unknown af %d)", family);
1611                 break;
1612         }
1613         return (buf);
1614 }

1616 /* The logical equivalence operator (a if-and-only-if b) */
1617 #define EQUIVALENT(a, b)        (((a) && (b)) || (!(a) && (!(b))))

1619 /*
1620  * Verify limitations and invariants on oob state.
1621  * Return 1 if OK, otherwise 0 so that it can be used as
1622  *      ASSERT(verify_oobstate(so));
1623  */
1624 int
1625 so_verify_oobstate(struct sonode *so)
1626 {
1627         boolean_t havemark;

1629         ASSERT(MUTEX_HELD(&so->so_lock));

1631         /*
1632          * The possible state combinations are:
1633          *      0
1634          *      SS_OOBPEND
1635          *      SS_OOBPEND|SS_HAVEOOBDATA
1636          *      SS_OOBPEND|SS_HADOOBDATA
1637          *      SS_HADOOBDATA
1638          */
1639         switch (so->so_state & (SS_OOBPEND|SS_HAVEOOBDATA|SS_HADOOBDATA)) {
1640         case 0:
1641         case SS_OOBPEND:
1642         case SS_OOBPEND|SS_HAVEOOBDATA:
1643         case SS_OOBPEND|SS_HADOOBDATA:
1644         case SS_HADOOBDATA:
1645                 break;
1646         default:
1647                 printf("Bad oob state 1 (%p): state %s\n",
```



```
1648                     (void *)so, pr_state(so->so_state, so->so_mode));
1649                 return (0);
1650         }

1652         /* SS_RCVATMARK should only be set when SS_OOBPEND is set */
1653         if ((so->so_state & (SS_RCVATMARK|SS_OOBPEND)) == SS_RCVATMARK) {
1654                 printf("Bad oob state 2 (%p): state %s\n",
1655                     (void *)so, pr_state(so->so_state, so->so_mode));
1656                 return (0);
1657         }

1659         /*
1660          * (havemark != 0 or SS_RCVATMARK) iff SS_OOBPEND
1661          * For TPI, the presence of a "mark" is indicated by sti_oobsigcnt.
1662          */
1663         havemark = (SOCK_IS_NONSTR(so)) ? so->so_oobmark > 0 :
1664             SOTOTPI(so)->sti_oobsigcnt > 0;

1666         if (!EQUIVALENT(havemark || (so->so_state & SS_RCVATMARK),
1667             so->so_state & SS_OOBPEND)) {
1668                 printf("Bad oob state 3 (%p): state %s\n",
1669                     (void *)so, pr_state(so->so_state, so->so_mode));
1670                 return (0);
1671         }

1673         /*
1674          * Unless SO_OOBINLINE we have so_oobmsg != NULL iff SS_HAVEOOBDATA
1675          */
1676         if (!(so->so_options & SO_OOBINLINE) &&
1677             !EQUIVALENT(so->so_oobmsg != NULL, so->so_state & SS_HAVEOOBDATA)) {
1678                 printf("Bad oob state 4 (%p): state %s\n",
1679                     (void *)so, pr_state(so->so_state, so->so_mode));
1680                 return (0);
1681         }

1683         if (!SOCK_IS_NONSTR(so) &&
1684             SOTOTPI(so)->sti_oobsigcnt < SOTOTPI(so)->sti_oobcnt) {
1685                 printf("Bad oob state 5 (%p): counts %d/%d state %s\n",
1686                     (void *)so, SOTOTPI(so)->sti_oobsigcnt,
1687                     SOTOTPI(so)->sti_oobcnt,
1688                     pr_state(so->so_state, so->so_mode));
1689                 return (0);
1690         }

1692         return (1);
1693 }
1694 #undef  EQUIVALENT
1695 #endif /* DEBUG */

1697 /* initialize sockfs zone specific kstat related items                  */
1698 void *
1699 sock_kstat_init(zoneid_t zoneid)
1700 {
1701         kstat_t *ksp;

1703         ksp = kstat_create_zone("sockfs", 0, "sock_unix_list", "misc",
1704             KSTAT_TYPE_RAW, 0, KSTAT_FLAG_VAR_SIZE|KSTAT_FLAG_VIRTUAL, zoneid);

1706         if (ksp != NULL) {
1707                 ksp->ks_update = sockfs_update;
1708                 ksp->ks_snapshot = sockfs_snapshot;
1709                 ksp->ks_lock = &socklist.sl_lock;
1710                 ksp->ks_private = (void *)(uintptr_t)zoneid;
1711                 kstat_install(ksp);
1712         }
```

```
1714         return (ksp);
1715 }

1717 /* tear down sockfs zone specific kstat related items                   */
1718 /*ARGSUSED*/
1719 void
1720 sock_kstat_fini(zoneid_t zoneid, void *arg)
1721 {
1722         kstat_t *ksp = (kstat_t *)arg;

1724         if (ksp != NULL) {
1725                 ASSERT(zoneid == (zoneid_t)(uintptr_t)ksp->ks_private);
1726                 kstat_delete(ksp);
1727         }
1728 }

1730 /*
1731  * Zones:
1732  * Note that nactive is going to be different for each zone.
1733  * This means we require kstat to call sockfs_update and then sockfs_snapshot
1734  * for the same zone, or sockfs_snapshot will be taken into the wrong size
1735  * buffer. This is safe, but if the buffer is too small, user will not be
1736  * given details of all sockets. However, as this kstat has a ks_lock, kstat
1737  * driver will keep it locked between the update and the snapshot, so no
1738  * other process (zone) can currently get inbetween resulting in a wrong size
1739  * buffer allocation.
1740  */

1742 #endif /* ! codereview */
1743 static int
1744 sockfs_update(kstat_t *ksp, int rw)
1745 {
1746         uint_t  n, nactive = 0;         /* # of active AF_UNIX sockets  */
1747         uint_t  tsze, sze;
 769         uint_t  nactive = 0;            /* # of active AF_UNIX sockets  */
1748         struct sonode   *so;            /* current sonode on socklist   */
1749         zoneid_t        myzoneid = (zoneid_t)(uintptr_t)ksp->ks_private;

1751         tsze = sze = 0;

1753 #endif /* ! codereview */
1754         ASSERT((zoneid_t)(uintptr_t)ksp->ks_private == getzoneid());

1756         if (rw == KSTAT_WRITE) {        /* bounce all writes            */
1757                 return (EACCES);
1758         }

1760         for (so = socklist.sl_list; so != NULL; so = SOTOTPI(so)->sti_next_so) {
1761                 if (so->so_count != 0 && so->so_zoneid == myzoneid) {

1763 #endif /* ! codereview */
1764                         nactive++;

1766                         mutex_enter(&so->so_pid_list_lock);
1767                         n = list_size(&so->so_pid_list);
1768                         mutex_exit(&so->so_pid_list_lock);

1770                         sze = sizeof (struct sockinfo);
1771                         sze += (n > 1)?((n - 1) * sizeof (conn_pid_node_t)):0;
1772                         tsze += sze;

1774 #endif /* ! codereview */
1775                 }
1776         }
1777         ksp->ks_ndata = nactive;
1778         ksp->ks_data_size = tsze;
```



```
 773         ksp->ks_data_size = nactive * sizeof (struct k_sockinfo);

1780         return (0);
1781 }

1783 static int
1784 sockfs_snapshot(kstat_t *ksp, void *buf, int rw)
1785 {
1786         int                     ns;     /* # of sonodes we've copied    */
1787         struct sonode           *so;    /* current sonode on socklist   */
1788         struct sockinfo         *psi;   /* where we put sockinfo data   */
 783         struct k_sockinfo       *pksi;  /* where we put sockinfo data   */
1789         t_uscalar_t             sn_len; /* soa_len                      */
1790         zoneid_t                myzoneid = (zoneid_t)(uintptr_t)ksp->ks_private;
1791         sotpi_info_t            *sti;

1793         uint_t                          sze;
1794         conn_pid_node_list_hdr_t        *cph;

1796 #endif /* ! codereview */
1797         ASSERT((zoneid_t)(uintptr_t)ksp->ks_private == getzoneid());

1799         ksp->ks_snaptime = gethrtime();

1801         if (rw == KSTAT_WRITE) {        /* bounce all writes            */
1802                 return (EACCES);
1803         }

1805         /*
1806          * for each sonode on the socklist, we massage the important
1807          * info into buf, in k_sockinfo format.
1808          */
1809         psi = (struct sockinfo *)buf;
 788         pksi = (struct k_sockinfo *)buf;
1810         ns = 0;
1811         for (so = socklist.sl_list; so != NULL; so = SOTOTPI(so)->sti_next_so) {
1812                 /* only stuff active sonodes and the same zone:         */
1813                 if (so->so_count == 0 || so->so_zoneid != myzoneid) {
1814                         continue;
1815                 }

1817                 /* get the pidnode list associated with this sonode */
1818                 cph = so_get_sock_pid_list((sock_upper_handle_t)so);

1820                 /* calculate the size of this sockinfo structure */
1821                 sze = sizeof (struct sockinfo);
1822                 sze += (cph->cph_pn_cnt > 1)?
1823                         ((cph->cph_pn_cnt - 1) * sizeof (conn_pid_node_t)):0;

1825 #endif /* ! codereview */
1826                 /*
1827                  * If the sonode was activated between the update and the
1828                  * snapshot, we're done - as this is only a snapshot. We need
1829                  * to make sure that we have space for this sockinfo.
 796                  * snapshot, we're done - as this is only a snapshot.
1830                  */
1831                 if (((caddr_t)(psi) + sze) >
1832                         ((caddr_t)buf + ksp->ks_data_size)) {
 798                 if ((caddr_t)(pksi) >= (caddr_t)buf + ksp->ks_data_size) {
1833                         break;
1834                 }

1836                 sti = SOTOTPI(so);
1837                 /* copy important info into buf:                        */
1838                 psi->si_size = sze;
1839                 psi->si_family = so->so_family;
```

```
1840                psi->si_type = so->so_type;
1841                psi->si_flag = so->so_flag;
1842                psi->si_state = so->so_state;
1843                psi->si_serv_type = sti->sti_serv_type;
1844                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 =
1845                    sti->sti_ux_laddr.soua_magic;
1846                psi->si_ux_faddr_sou_magic =
 812                pksi->ks_si.si_ux_faddr_sou_magic =
1847                    sti->sti_ux_faddr.soua_magic;
1848                psi->si_laddr_soa_len = sti->sti_laddr.soa_len;
1849                psi->si_faddr_soa_len = sti->sti_faddr.soa_len;
1850                psi->si_szoneid = so->so_zoneid;
1851                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;

1854                mutex_enter(&so->so_lock);

1856                if (sti->sti_laddr_sa != NULL) {
1857                        ASSERT(sti->sti_laddr_sa->sa_data != NULL);
1858                        sn_len = sti->sti_laddr_len;
1859                        ASSERT(sn_len <= sizeof (short) +
1860                            sizeof (psi->si_laddr_sun_path));
 825                            sizeof (pksi->ks_si.si_laddr_sun_path));

1862                        psi->si_laddr_family =
 827                        pksi->ks_si.si_laddr_family =
1863                            sti->sti_laddr_sa->sa_family;
1864                        if (sn_len != 0) {
1865                                /* AF_UNIX socket names are NULL terminated */
1866                                (void) strncpy(psi->si_laddr_sun_path,
 831                                (void) strncpy(pksi->ks_si.si_laddr_sun_path,
1867                                    sti->sti_laddr_sa->sa_data,
1868                                    sizeof (psi->si_laddr_sun_path));
1869                                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);
1870                        }
1871                        psi->si_laddr_sun_path[sn_len] = 0;
 836                        pksi->ks_si.si_laddr_sun_path[sn_len] = 0;
1872                }

1874                if (sti->sti_faddr_sa != NULL) {
1875                        ASSERT(sti->sti_faddr_sa->sa_data != NULL);
1876                        sn_len = sti->sti_faddr_len;
1877                        ASSERT(sn_len <= sizeof (short) +
1878                            sizeof (psi->si_faddr_sun_path));
 843                            sizeof (pksi->ks_si.si_faddr_sun_path));

1880                        psi->si_faddr_family =
 845                        pksi->ks_si.si_faddr_family =
1881                            sti->sti_faddr_sa->sa_family;
1882                        if (sn_len != 0) {
1883                                (void) strncpy(psi->si_faddr_sun_path,
 848                                (void) strncpy(pksi->ks_si.si_faddr_sun_path,
1884                                    sti->sti_faddr_sa->sa_data,
```



```
1885                                    sizeof (psi->si_faddr_sun_path));
1886                                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);
1887                        }
1888                        psi->si_faddr_sun_path[sn_len] = 0;
 853                        pksi->ks_si.si_faddr_sun_path[sn_len] = 0;
1889                }

1891                mutex_exit(&so->so_lock);

1893                (void) sprintf(psi->si_son_straddr, "%p", (void *)so);
1894                (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",
1895                    (void *)sti->sti_ux_laddr.soua_vp);
1896                (void) sprintf(psi->si_fvn_straddr, "%p",
 861                (void) sprintf(pksi->ks_straddr[2], "%p",
1897                    (void *)sti->sti_ux_faddr.soua_vp);

1899                if ((psi->si_pn_cnt = cph->cph_pn_cnt) > 0)
1900                        (void) memcpy(psi->si_pns, cph->cph_cpns,
1901                                psi->si_pn_cnt * sizeof (conn_pid_node_t));

1903                kmem_free(cph, cph->cph_tot_size);

1905                psi = (struct sockinfo *)((char *)psi + psi->si_size);
1906 #endif /* ! codereview */
1907                ns++;
 864                pksi++;
1908        }

1910        ksp->ks_ndata = ns;
1911        return (0);
1912 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   80790 Sun Aug  9 12:47:50 2015
new/usr/src/uts/common/inet/ip/ipclassifier.c
XXXX adding PID information to netstat output
**********************************************************
______unchanged_portion_omitted_
2724 #endif

2726 conn_pid_node_list_hdr_t *
2727 conn_get_pid_list(conn_t *connp)
2728 {
2729         conn_pid_node_list_hdr_t        *cph;

2731         if (connp->conn_upper_handle != NULL) {
2732                 return (*connp->conn_upcalls->su_get_sock_pid_list)
2733                         (connp->conn_upper_handle);
2734         } else if (!IPCL_IS_NONSTR(connp) && connp->conn_rq != NULL &&
2735                         connp->conn_rq->q_stream != NULL) {
2736                 return (sh_get_pid_list(connp->conn_rq->q_stream));
2737         }

2739         /* return an empty header */
2740         cph = kmem_zalloc(sizeof (conn_pid_node_list_hdr_t), KM_SLEEP);
2741         cph->cph_magic = CONN_PID_NODE_LIST_HDR_MAGIC;
2742         cph->cph_contents = CONN_PID_NODE_LIST_HDR_NON;
2743         cph->cph_pn_cnt = 0;
2744         cph->cph_tot_size = sizeof (conn_pid_node_list_hdr_t);
2745         cph->cph_flags = 0;
2746         cph->cph_optional1 = 0;
2747         cph->cph_optional2 = 0;

2749         return (cph);
2750 }
2751 #endif /* ! codereview */
```

```
**********************************************************
   26493 Sun Aug  9 12:47:53 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 conn_pid_node_list_hdr_t *conn_get_pid_list(conn_t *);

733 #ifdef  __cplusplus
734 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   60183 Sun Aug  9 12:47:56 2015
new/usr/src/uts/common/inet/mib2.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_MIB2_H
#define _INET_MIB2_H

#include <netinet/in.h> /* For in6_addr_t */
#include <sys/tsol/label.h> /* For brange_t */
#include <sys/tsol/label_macro.h> /* For brange_t */
#include <sys/pidnode.h>
#endif /* ! codereview */

#ifdef __cplusplus
extern "C" {
#endif

/*
 * The IPv6 parts of this are derived from:
 *      RFC 2465
 *      RFC 2466
 *      RFC 2452
 *      RFC 2454
 */

/*
 * SNMP set/get via M_PROTO T_OPTMGMT_REQ.  Structure is that used
 * for [gs]etsockopt() calls.  get uses T_CURRENT, set uses T_NEOGTIATE
 * MGMT_flags value.  The following definition of opthdr is taken from
 * socket.h:
 *
 * An option specification consists of an opthdr, followed by the value of
 * the option.  An options buffer contains one or more options.  The len
 * field of opthdr specifies the length of the option value in bytes.  This
 * length must be a multiple of sizeof(long) (use OPTLEN macro).
 *
 * struct opthdr {
 *      long    level;  protocol level affected
 *      long    name;   option to modify
 *      long    len;    length of option value
 * };
```



```
 *
 * #define OPTLEN(x) ((((x) + sizeof(long) - 1) / sizeof(long)) * sizeof(long))
 * #define OPTVAL(opt) ((char *)(opt + 1))
 *
 * For get requests (T_CURRENT), any MIB2_xxx value can be used (only
 * "get all" is supported, so all modules get a copy of the request to
 * return everything it knows.   In general, we use MIB2_IP.  There is
 * one exception: in general, IP will not report information related to
 * ire_testhidden and IRE_IF_CLONE routes (e.g., in the MIB2_IP_ROUTE
 * table). However, using the special value EXPER_IP_AND_ALL_IRES will cause
 * all information to be reported.  This special value should only be
 * used by IPMP-aware low-level utilities (e.g. in.mpathd).
 *
 * IMPORTANT:  some fields are grouped in a different structure than
 * suggested by MIB-II, e.g., checksum error counts.  The original MIB-2
 * field name has been retained.  Field names beginning with "mi" are not
 * defined in the MIB but contain important & useful information maintained
 * by the corresponding module.
 */
#ifndef IPPROTO_MAX
#define IPPROTO_MAX     256
#endif

#define MIB2_SYSTEM             (IPPROTO_MAX+1)
#define MIB2_INTERFACES         (IPPROTO_MAX+2)
#define MIB2_AT                 (IPPROTO_MAX+3)
#define MIB2_IP                 (IPPROTO_MAX+4)
#define MIB2_ICMP               (IPPROTO_MAX+5)
#define MIB2_TCP                (IPPROTO_MAX+6)
#define MIB2_UDP                (IPPROTO_MAX+7)
#define MIB2_EGP                (IPPROTO_MAX+8)
#define MIB2_CMOT               (IPPROTO_MAX+9)
#define MIB2_TRANSMISSION       (IPPROTO_MAX+10)
#define MIB2_SNMP               (IPPROTO_MAX+11)
#define MIB2_IP6                (IPPROTO_MAX+12)
#define MIB2_ICMP6              (IPPROTO_MAX+13)
#define MIB2_TCP6               (IPPROTO_MAX+14)
#define MIB2_UDP6               (IPPROTO_MAX+15)
#define MIB2_SCTP               (IPPROTO_MAX+16)

/*
 * Define range of levels for use with MIB2_*
 */
#define MIB2_RANGE_START        (IPPROTO_MAX+1)
#define MIB2_RANGE_END          (IPPROTO_MAX+16)


#define EXPER                   1024    /* experimental - not part of mib */
#define EXPER_IGMP              (EXPER+1)
#define EXPER_DVMRP             (EXPER+2)
#define EXPER_RAWIP             (EXPER+3)
#define EXPER_IP_AND_ALL_IRES   (EXPER+4)

/*
 * Define range of levels for experimental use
 */
#define EXPER_RANGE_START       (EXPER+1)
#define EXPER_RANGE_END         (EXPER+4)

#define BUMP_MIB(s, x)          {                                       \
        extern void __dtrace_probe___mib_##x(int, void *);      \
        void *stataddr = &((s)->x);                             \
        __dtrace_probe___mib_##x(1, stataddr);                  \
        (s)->x++;                                               \
}
```

```
128 #define UPDATE_MIB(s, x, y)     {                                       \
129         extern void __dtrace_probe___mib_##x(int, void *);      \
130         void *stataddr = &((s)->x);                             \
131         __dtrace_probe___mib_##x(y, stataddr);                  \
132         (s)->x += (y);                                          \
133 }

135 #define SET_MIB(x, y)           x = y
136 #define BUMP_LOCAL(x)           (x)++
137 #define UPDATE_LOCAL(x, y)      (x) += (y)
138 #define SYNC32_MIB(s, m32, m64) SET_MIB((s)->m32, (s)->m64 & 0xffffffff)

140 /*
141  * Each struct that has been extended have a macro (MIB_FIRST_NEW_ELM_type)
142  * that is set to the first new element of the extended struct.
143  * The LEGACY_MIB_SIZE macro can be used to determine the size of MIB
144  * objects that needs to be returned to older applications unaware of
145  * these extensions.
146  */
147 #define MIB_PTRDIFF(s, e)       (caddr_t)e - (caddr_t)s
148 #define LEGACY_MIB_SIZE(s, t)   MIB_PTRDIFF(s, &(s)->MIB_FIRST_NEW_ELM_##t)

150 #define OCTET_LENGTH    32      /* Must be at least LIFNAMSIZ */
151 typedef struct Octet_s {
152         int     o_length;
153         char    o_bytes[OCTET_LENGTH];
154 } Octet_t;

156 typedef uint32_t        Counter;
157 typedef uint32_t        Counter32;
158 typedef uint64_t        Counter64;
159 typedef uint32_t        Gauge;
160 typedef uint32_t        IpAddress;
161 typedef struct in6_addr Ip6Address;
162 typedef Octet_t         DeviceName;
163 typedef Octet_t         PhysAddress;
164 typedef uint32_t        DeviceIndex;    /* Interface index */

166 #define MIB2_UNKNOWN_INTERFACE  0
167 #define MIB2_UNKNOWN_PROCESS    0

169 /*
170  *  IP group
171  */
172 #define MIB2_IP_ADDR            20      /* ipAddrEntry */
173 #define MIB2_IP_ROUTE           21      /* ipRouteEntry */
174 #define MIB2_IP_MEDIA           22      /* ipNetToMediaEntry */
175 #define MIB2_IP6_ROUTE          23      /* ipv6RouteEntry */
176 #define MIB2_IP6_MEDIA          24      /* ipv6NetToMediaEntry */
177 #define MIB2_IP6_ADDR           25      /* ipv6AddrEntry */
178 #define MIB2_IP_TRAFFIC_STATS   31      /* ipIfStatsEntry (IPv4) */
179 #define EXPER_IP_GROUP_MEMBERSHIP       100
180 #define EXPER_IP6_GROUP_MEMBERSHIP      101
181 #define EXPER_IP_GROUP_SOURCES          102
182 #define EXPER_IP6_GROUP_SOURCES         103
183 #define EXPER_IP_RTATTR                 104
184 #define EXPER_IP_DCE                    105

186 /*
187  * There can be one of each of these tables per transport (MIB2_* above).
188  */
189 #define EXPER_XPORT_MLP         105     /* transportMLPEntry */
190 #define EXPER_XPORT_PROC_INFO   106     /* conn_pid_node entry */
191 #endif /* ! codereview */

193 /* Old names retained for compatibility */
```



```
194 #define MIB2_IP_20      MIB2_IP_ADDR
195 #define MIB2_IP_21      MIB2_IP_ROUTE
196 #define MIB2_IP_22      MIB2_IP_MEDIA

198 typedef struct mib2_ip {
199                 /* forwarder?  1 gateway, 2 NOT gateway {ip 1} RW */
200         int     ipForwarding;
201                 /* default Time-to-Live for iph         {ip 2} RW */
202         int     ipDefaultTTL;
203                 /* # of input datagrams                 {ip 3} */
204         Counter ipInReceives;
205                 /* # of dg discards for iph error       {ip 4} */
206         Counter ipInHdrErrors;
207                 /* # of dg discards for bad addr        {ip 5} */
208         Counter ipInAddrErrors;
209                 /* # of dg being forwarded              {ip 6} */
210         Counter ipForwDatagrams;
211                 /* # of dg discards for unk protocol    {ip 7} */
212         Counter ipInUnknownProtos;
213                 /* # of dg discards of good dg's        {ip 8} */
214         Counter ipInDiscards;
215                 /* # of dg sent upstream                {ip 9} */
216         Counter ipInDelivers;
217                 /* # of outdgs recv'd from upstream     {ip 10} */
218         Counter ipOutRequests;
219                 /* # of good outdgs discarded           {ip 11} */
220         Counter ipOutDiscards;
221                 /* # of outdg discards: no route found  {ip 12} */
222         Counter ipOutNoRoutes;
223                 /* sec's recv'd frags held for reass.   {ip 13} */
224         int     ipReasmTimeout;
225                 /* # of ip frags needing reassembly     {ip 14} */
226         Counter ipReasmReqds;
227                 /* # of dg's reassembled                {ip 15} */
228         Counter ipReasmOKs;
229                 /* # of reassembly failures (not dg cnt){ip 16} */
230         Counter ipReasmFails;
231                 /* # of dg's fragged                    {ip 17} */
232         Counter ipFragOKs;
233                 /* # of dg discards for no frag set     {ip 18} */
234         Counter ipFragFails;
235                 /* # of dg frags from fragmentation     {ip 19} */
236         Counter ipFragCreates;
237                 /* {ip 20} */
238         int     ipAddrEntrySize;
239                 /* {ip 21} */
240         int     ipRouteEntrySize;
241                 /* {ip 22} */
242         int     ipNetToMediaEntrySize;
243                 /* # of valid route entries discarded   {ip 23} */
244         Counter ipRoutingDiscards;
245 /*
246  * following defined in MIB-II as part of TCP & UDP groups:
247  */
248                 /* total # of segments recv'd with error        { tcp 14 } */
249         Counter tcpInErrs;
250                 /* # of recv'd dg's not deliverable (no appl.)  { udp 2 } */
251         Counter udpNoPorts;
252 /*
253  * In addition to MIB-II
254  */
255                 /* # of bad IP header checksums */
256         Counter ipInCksumErrs;
257                 /* # of complete duplicates in reassembly */
258         Counter ipReasmDuplicates;
259                 /* # of partial duplicates in reassembly */
```

```
260         Counter ipReasmPartDups;
261                 /* # of packets not forwarded due to adminstrative reasons */
262         Counter ipForwProhibits;
263                 /* # of UDP packets with bad UDP checksums */
264         Counter udpInCksumErrs;
265                 /* # of UDP packets droped due to queue overflow */
266         Counter udpInOverflows;
267                 /*
268                  * # of RAW IP packets (all IP protocols except UDP, TCP
269                  * and ICMP) droped due to queue overflow
270                  */
271         Counter rawipInOverflows;

273         /*
274          * Folowing are private IPSEC MIB.
275          */
276         /* # of incoming packets that succeeded policy checks */
277         Counter ipsecInSucceeded;
278         /* # of incoming packets that failed policy checks */
279         Counter ipsecInFailed;
280 /* Compatible extensions added here */
281         int     ipMemberEntrySize;      /* Size of ip_member_t */
282         int     ipGroupSourceEntrySize; /* Size of ip_grpsrc_t */

284         Counter ipInIPv6; /* # of IPv6 packets received by IPv4 and dropped */
285         Counter ipOutIPv6;              /* No longer used */
286         Counter ipOutSwitchIPv6;        /* No longer used */

288         int     ipRouteAttributeSize;   /* Size of mib2_ipAttributeEntry_t */
289         int     transportMLPSize;       /* Size of mib2_transportMLPEntry_t */
290         int     ipDestEntrySize;        /* Size of dest_cache_entry_t */
291 } mib2_ip_t;

293 /*
294  *      ipv6IfStatsEntry OBJECT-TYPE
295  *              SYNTAX     Ipv6IfStatsEntry
296  *              MAX-ACCESS not-accessible
297  *              STATUS     current
298  *              DESCRIPTION
299  *                      "An interface statistics entry containing objects
300  *                      at a particular IPv6 interface."
301  *              AUGMENTS { ipv6IfEntry }
302  *              ::= { ipv6IfStatsTable 1 }
303  *
304  * Per-interface IPv6 statistics table
305  */

307 typedef struct mib2_ipv6IfStatsEntry {
308         /* Local ifindex to identify the interface */
309         DeviceIndex     ipv6IfIndex;

311                 /* forwarder?  1 gateway, 2 NOT gateway {ipv6MIBObjects 1} RW */
312         int     ipv6Forwarding;
313                 /* default Hoplimit for IPv6            {ipv6MIBObjects 2} RW */
314         int     ipv6DefaultHopLimit;

316         int     ipv6IfStatsEntrySize;
317         int     ipv6AddrEntrySize;
318         int     ipv6RouteEntrySize;
319         int     ipv6NetToMediaEntrySize;
320         int     ipv6MemberEntrySize;            /* Size of ipv6_member_t */
321         int     ipv6GroupSourceEntrySize;       /* Size of ipv6_grpsrc_t */

323         /* # input datagrams (incl errors)      { ipv6IfStatsEntry 1 } */
324         Counter ipv6InReceives;
325         /* # errors in IPv6 headers and options { ipv6IfStatsEntry 2 } */
```



```
326         Counter ipv6InHdrErrors;
327         /* # exceeds outgoing link MTU          { ipv6IfStatsEntry 3 } */
328         Counter ipv6InTooBigErrors;
329         /* # discarded due to no route to dest  { ipv6IfStatsEntry 4 } */
330         Counter ipv6InNoRoutes;
331         /* # invalid or unsupported addresses   { ipv6IfStatsEntry 5 } */
332         Counter ipv6InAddrErrors;
333         /* # unknown next header                { ipv6IfStatsEntry 6 } */
334         Counter ipv6InUnknownProtos;
335         /* # too short packets                  { ipv6IfStatsEntry 7 } */
336         Counter ipv6InTruncatedPkts;
337         /* # discarded e.g. due to no buffers   { ipv6IfStatsEntry 8 } */
338         Counter ipv6InDiscards;
339         /* # delivered to upper layer protocols { ipv6IfStatsEntry 9 } */
340         Counter ipv6InDelivers;
341         /* # forwarded out interface            { ipv6IfStatsEntry 10 } */
342         Counter ipv6OutForwDatagrams;
343         /* # originated out interface           { ipv6IfStatsEntry 11 } */
344         Counter ipv6OutRequests;
345         /* # discarded e.g. due to no buffers   { ipv6IfStatsEntry 12 } */
346         Counter ipv6OutDiscards;
347         /* # sucessfully fragmented packets     { ipv6IfStatsEntry 13 } */
348         Counter ipv6OutFragOKs;
349         /* # fragmentation failed               { ipv6IfStatsEntry 14 } */
350         Counter ipv6OutFragFails;
351         /* # fragments created                  { ipv6IfStatsEntry 15 } */
352         Counter ipv6OutFragCreates;
353         /* # fragments to reassemble            { ipv6IfStatsEntry 16 } */
354         Counter ipv6ReasmReqds;
355         /* # packets after reassembly           { ipv6IfStatsEntry 17 } */
356         Counter ipv6ReasmOKs;
357         /* # reassembly failed                  { ipv6IfStatsEntry 18 } */
358         Counter ipv6ReasmFails;
359         /* # received multicast packets         { ipv6IfStatsEntry 19 } */
360         Counter ipv6InMcastPkts;
361         /* # transmitted multicast packets      { ipv6IfStatsEntry 20 } */
362         Counter ipv6OutMcastPkts;
363 /*
364  * In addition to defined MIBs
365  */
366                 /* # discarded due to no route to dest */
367         Counter ipv6OutNoRoutes;
368                 /* # of complete duplicates in reassembly */
369         Counter ipv6ReasmDuplicates;
370                 /* # of partial duplicates in reassembly */
371         Counter ipv6ReasmPartDups;
372                 /* # of packets not forwarded due to adminstrative reasons */
373         Counter ipv6ForwProhibits;
374                 /* # of UDP packets with bad UDP checksums */
375         Counter udpInCksumErrs;
376                 /* # of UDP packets droped due to queue overflow */
377         Counter udpInOverflows;
378                 /*
379                  * # of RAW IPv6 packets (all IPv6 protocols except UDP, TCP
380                  * and ICMPv6) droped due to queue overflow
381                  */
382         Counter rawipInOverflows;

384                 /* # of IPv4 packets received by IPv6 and dropped */
385         Counter ipv6InIPv4;
386                 /* # of IPv4 packets transmitted by ip_wput_wput */
387         Counter ipv6OutIPv4;
388                 /* # of times ip_wput_v6 has switched to become ip_wput */
389         Counter ipv6OutSwitchIPv4;
390 } mib2_ipv6IfStatsEntry_t;
```

```
392 /*
393  * Per interface IP statistics, both v4 and v6.
394  *
395  * Some applications expect to get mib2_ipv6IfStatsEntry_t structs back when
396  * making a request. To ensure backwards compatability, the first
397  * sizeof(mib2_ipv6IfStatsEntry_t) bytes of the structure is identical to
398  * mib2_ipv6IfStatsEntry_t. This should work as long the application is
399  * written correctly (i.e., using ipv6IfStatsEntrySize to get the size of
400  * the struct)
401  *
402  * RFC4293 introduces several new counters, as well as defining 64-bit
403  * versions of existing counters. For a new counters, if they have both 32-
404  * and 64-bit versions, then we only added the latter. However, for already
405  * existing counters, we have added the 64-bit versions without removing the
406  * old (32-bit) ones. The 64- and 32-bit counters will only be synchronized
407  * when the structure contains IPv6 statistics, which is done to ensure
408  * backwards compatibility.
409  */

411 /* The following are defined in RFC 4001 and are used for ipIfStatsIPVersion */
412 #define MIB2_INETADDRESSTYPE_unknown    0
413 #define MIB2_INETADDRESSTYPE_ipv4       1
414 #define MIB2_INETADDRESSTYPE_ipv6       2

416 /*
417  * On amd64, the alignment requirements for long long's is different for
418  * 32 and 64 bits. If we have a struct containing long long's that is being
419  * passed between a 64-bit kernel to a 32-bit application, then it is very
420  * likely that the size of the struct will differ due to padding. Therefore, we
421  * pack the data to ensure that the struct size is the same for 32- and
422  * 64-bits.
423  */
424 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
425 #pragma pack(4)
426 #endif

428 typedef struct mib2_ipIfStatsEntry {

430         /* Local ifindex to identify the interface */
431         DeviceIndex     ipIfStatsIfIndex;

433         /* forwarder?  1 gateway, 2 NOT gateway { ipv6MIBObjects 1} RW */
434         int     ipIfStatsForwarding;
435         /* default Hoplimit for IPv6            { ipv6MIBObjects 2} RW */
436         int     ipIfStatsDefaultHopLimit;
437 #define ipIfStatsDefaultTTL     ipIfStatsDefaultHopLimit

439         int     ipIfStatsEntrySize;
440         int     ipIfStatsAddrEntrySize;
441         int     ipIfStatsRouteEntrySize;
442         int     ipIfStatsNetToMediaEntrySize;
443         int     ipIfStatsMemberEntrySize;
444         int     ipIfStatsGroupSourceEntrySize;

446         /* # input datagrams (incl errors)      { ipIfStatsEntry 3 } */
447         Counter ipIfStatsInReceives;
448         /* # errors in IP headers and options   { ipIfStatsEntry 7 } */
449         Counter ipIfStatsInHdrErrors;
450         /* # exceeds outgoing link MTU(v6 only) { ipv6IfStatsEntry 3 } */
451         Counter ipIfStatsInTooBigErrors;
452         /* # discarded due to no route to dest  { ipIfStatsEntry 8 } */
453         Counter ipIfStatsInNoRoutes;
454         /* # invalid or unsupported addresses   { ipIfStatsEntry 9 } */
455         Counter ipIfStatsInAddrErrors;
456         /* # unknown next header                { ipIfStatsEntry 10 } */
457         Counter ipIfStatsInUnknownProtos;
```



```
458         /* # too short packets                  { ipIfStatsEntry 11 } */
459         Counter ipIfStatsInTruncatedPkts;
460         /* # discarded e.g. due to no buffers   { ipIfStatsEntry 17 } */
461         Counter ipIfStatsInDiscards;
462         /* # delivered to upper layer protocols { ipIfStatsEntry 18 } */
463         Counter ipIfStatsInDelivers;
464         /* # forwarded out interface            { ipIfStatsEntry 23 } */
465         Counter ipIfStatsOutForwDatagrams;
466         /* # originated out interface           { ipIfStatsEntry 20 } */
467         Counter ipIfStatsOutRequests;
468         /* # discarded e.g. due to no buffers   { ipIfStatsEntry 25 } */
469         Counter ipIfStatsOutDiscards;
470         /* # sucessfully fragmented packets     { ipIfStatsEntry 27 } */
471         Counter ipIfStatsOutFragOKs;
472         /* # fragmentation failed               { ipIfStatsEntry 28 } */
473         Counter ipIfStatsOutFragFails;
474         /* # fragments created                  { ipIfStatsEntry 29 } */
475         Counter ipIfStatsOutFragCreates;
476         /* # fragments to reassemble            { ipIfStatsEntry 14 } */
477         Counter ipIfStatsReasmReqds;
478         /* # packets after reassembly           { ipIfStatsEntry 15 } */
479         Counter ipIfStatsReasmOKs;
480         /* # reassembly failed                  { ipIfStatsEntry 16 } */
481         Counter ipIfStatsReasmFails;
482         /* # received multicast packets         { ipIfStatsEntry 34 } */
483         Counter ipIfStatsInMcastPkts;
484         /* # transmitted multicast packets      { ipIfStatsEntry 38 } */
485         Counter ipIfStatsOutMcastPkts;

487         /*
488          * In addition to defined MIBs
489          */

491         /* # discarded due to no route to dest  { ipSystemStatsEntry 22 } */
492         Counter ipIfStatsOutNoRoutes;
493         /* # of complete duplicates in reassembly */
494         Counter ipIfStatsReasmDuplicates;
495         /* # of partial duplicates in reassembly */
496         Counter ipIfStatsReasmPartDups;
497         /* # of packets not forwarded due to adminstrative reasons */
498         Counter ipIfStatsForwProhibits;
499         /* # of UDP packets with bad UDP checksums */
500         Counter udpInCksumErrs;
501 #define udpIfStatsInCksumErrs   udpInCksumErrs
502         /* # of UDP packets droped due to queue overflow */
503         Counter udpInOverflows;
504 #define udpIfStatsInOverflows   udpInOverflows
505         /*
506          * # of RAW IP packets (all IP protocols except UDP, TCP
507          * and ICMP) droped due to queue overflow
508          */
509         Counter rawipInOverflows;
510 #define rawipIfStatsInOverflows rawipInOverflows

512         /*
513          * # of IP packets received with the wrong version (i.e., not equal
514          * to ipIfStatsIPVersion) and that were dropped.
515          */
516         Counter ipIfStatsInWrongIPVersion;
517         /*
518          * This counter is no longer used
519          */
520         Counter ipIfStatsOutWrongIPVersion;
521         /*
522          * This counter is no longer used
523          */
```

```
524         Counter ipIfStatsOutSwitchIPVersion;

526         /*
527          * Fields defined in RFC 4293
528          */

530         /* ip version                           { ipIfStatsEntry 1 } */
531         int             ipIfStatsIPVersion;
532         /* # input datagrams (incl errors)      { ipIfStatsEntry 4 } */
533         Counter64       ipIfStatsHCInReceives;
534         /* # input octets (incl errors)         { ipIfStatsEntry 6 } */
535         Counter64       ipIfStatsHCInOctets;
536         /*
537          *                                      { ipIfStatsEntry 13 }
538          * # input datagrams for which a forwarding attempt was made
539          */
540         Counter64       ipIfStatsHCInForwDatagrams;
541         /* # delivered to upper layer protocols { ipIfStatsEntry 19 } */
542         Counter64       ipIfStatsHCInDelivers;
543         /* # originated out interface           { ipIfStatsEntry 21 } */
544         Counter64       ipIfStatsHCOutRequests;
545         /* # forwarded out interface            { ipIfStatsEntry 23 } */
546         Counter64       ipIfStatsHCOutForwDatagrams;
547         /* # dg's requiring fragmentation       { ipIfStatsEntry 26 } */
548         Counter         ipIfStatsOutFragReqds;
549         /* # output datagrams                   { ipIfStatsEntry 31 } */
550         Counter64       ipIfStatsHCOutTransmits;
551         /* # output octets                      { ipIfStatsEntry 33 } */
552         Counter64       ipIfStatsHCOutOctets;
553         /* # received multicast datagrams       { ipIfStatsEntry 35 } */
554         Counter64       ipIfStatsHCInMcastPkts;
555         /* # received multicast octets          { ipIfStatsEntry 37 } */
556         Counter64       ipIfStatsHCInMcastOctets;
557         /* # transmitted multicast datagrams    { ipIfStatsEntry 39 } */
558         Counter64       ipIfStatsHCOutMcastPkts;
559         /* # transmitted multicast octets       { ipIfStatsEntry 41 } */
560         Counter64       ipIfStatsHCOutMcastOctets;
561         /* # received broadcast datagrams       { ipIfStatsEntry 43 } */
562         Counter64       ipIfStatsHCInBcastPkts;
563         /* # transmitted broadcast datagrams    { ipIfStatsEntry 45 } */
564         Counter64       ipIfStatsHCOutBcastPkts;

566         /*
567          * Fields defined in mib2_ip_t
568          */

570         /* # of incoming packets that succeeded policy checks */
571         Counter         ipsecInSucceeded;
572 #define ipsecIfStatsInSucceeded ipsecInSucceeded
573         /* # of incoming packets that failed policy checks */
574         Counter         ipsecInFailed;
575 #define ipsecIfStatsInFailed    ipsecInFailed
576         /* # of bad IP header checksums */
577         Counter         ipInCksumErrs;
578 #define ipIfStatsInCksumErrs    ipInCksumErrs
579         /* total # of segments recv'd with error        { tcp 14 } */
580         Counter         tcpInErrs;
581 #define tcpIfStatsInErrs        tcpInErrs
582         /* # of recv'd dg's not deliverable (no appl.)  { udp 2 } */
583         Counter         udpNoPorts;
584 #define udpIfStatsNoPorts       udpNoPorts
585 } mib2_ipIfStatsEntry_t;
586 #define MIB_FIRST_NEW_ELM_mib2_ipIfStatsEntry_t ipIfStatsIPVersion

588 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
589 #pragma pack()
```



```
590 #endif

592 /*
593  * The IP address table contains this entity's IP addressing information.
594  *
595  *      ipAddrTable OBJECT-TYPE
596  *              SYNTAX  SEQUENCE OF IpAddrEntry
597  *              ACCESS  not-accessible
598  *              STATUS  mandatory
599  *              DESCRIPTION
600  *                      "The table of addressing information relevant to
601  *                      this entity's IP addresses."
602  *              ::= { ip 20 }
603  */

605 typedef struct mib2_ipAddrEntry {
606                         /* IP address of this entry     {ipAddrEntry 1} */
607         IpAddress       ipAdEntAddr;
608                         /* Unique interface index       {ipAddrEntry 2} */
609         DeviceName      ipAdEntIfIndex;
610                         /* Subnet mask for this IP addr {ipAddrEntry 3} */
611         IpAddress       ipAdEntNetMask;
612                         /* 2^lsb of IP broadcast addr   {ipAddrEntry 4} */
613         int             ipAdEntBcastAddr;
614                         /* max size for dg reassembly   {ipAddrEntry 5} */
615         int             ipAdEntReasmMaxSize;
616                         /* additional ipif_t fields */
617         struct ipAdEntInfo_s {
618                 Gauge           ae_mtu;
619                                 /* BSD if metric */
620                 int             ae_metric;
621                                 /* ipif broadcast addr.  relation to above?? */
622                 IpAddress       ae_broadcast_addr;
623                                 /* point-point dest addr */
624                 IpAddress       ae_pp_dst_addr;
625                 int             ae_flags;       /* IFF_* flags in if.h */
626                 Counter         ae_ibcnt;       /* Inbound packets */
627                 Counter         ae_obcnt;       /* Outbound packets */
628                 Counter         ae_focnt;       /* Forwarded packets */
629                 IpAddress       ae_subnet;      /* Subnet prefix */
630                 int             ae_subnet_len;  /* Subnet prefix length */
631                 IpAddress       ae_src_addr;    /* Source address */
632         }               ipAdEntInfo;
633         uint32_t        ipAdEntRetransmitTime;  /* ipInterfaceRetransmitTime */
634 } mib2_ipAddrEntry_t;
635 #define MIB_FIRST_NEW_ELM_mib2_ipAddrEntry_t    ipAdEntRetransmitTime

637 /*
638  *      ipv6AddrTable OBJECT-TYPE
639  *              SYNTAX      SEQUENCE OF Ipv6AddrEntry
640  *              MAX-ACCESS  not-accessible
641  *              STATUS      current
642  *              DESCRIPTION
643  *                      "The table of addressing information relevant to
644  *                      this node's interface addresses."
645  *              ::= { ipv6MIBObjects 8 }
646  */

648 typedef struct mib2_ipv6AddrEntry {
649         /* Unique interface index                       { Part of INDEX } */
650         DeviceName      ipv6AddrIfIndex;

652         /* IPv6 address of this entry                   { ipv6AddrEntry 1 } */
653         Ip6Address      ipv6AddrAddress;
654         /* Prefix length                                { ipv6AddrEntry 2 } */
655         uint_t          ipv6AddrPfxLength;
```

```
656         /* Type: stateless(1), stateful(2), unknown(3)  { ipv6AddrEntry 3 } */
657         uint_t          ipv6AddrType;
658         /* Anycast: true(1), false(2)                   { ipv6AddrEntry 4 } */
659         uint_t          ipv6AddrAnycastFlag;
660         /*
661          * Address status: preferred(1), deprecated(2), invalid(3),
662          * inaccessible(4), unknown(5)                  { ipv6AddrEntry 5 }
663          */
664         uint_t          ipv6AddrStatus;
665         struct ipv6AddrInfo_s {
666                 Gauge           ae_mtu;
667                                 /* BSD if metric */
668                 int             ae_metric;
669                                 /* point-point dest addr */
670                 Ip6Address      ae_pp_dst_addr;
671                 int             ae_flags;       /* IFF_* flags in if.h */
672                 Counter         ae_ibcnt;       /* Inbound packets */
673                 Counter         ae_obcnt;       /* Outbound packets */
674                 Counter         ae_focnt;       /* Forwarded packets */
675                 Ip6Address      ae_subnet;      /* Subnet prefix */
676                 int             ae_subnet_len;  /* Subnet prefix length */
677                 Ip6Address      ae_src_addr;    /* Source address */
678         }               ipv6AddrInfo;
679         uint32_t        ipv6AddrReasmMaxSize;   /* InterfaceReasmMaxSize */
680         Ip6Address      ipv6AddrIdentifier;     /* InterfaceIdentifier */
681         uint32_t        ipv6AddrIdentifierLen;
682         uint32_t        ipv6AddrReachableTime;  /* InterfaceReachableTime */
683         uint32_t        ipv6AddrRetransmitTime; /* InterfaceRetransmitTime */
684 } mib2_ipv6AddrEntry_t;
685 #define MIB_FIRST_NEW_ELM_mib2_ipv6AddrEntry_t  ipv6AddrReasmMaxSize

687 /*
688  * The IP routing table contains an entry for each route presently known to
689  * this entity. (for IPv4 routes)
690  *
691  *      ipRouteTable OBJECT-TYPE
692  *              SYNTAX  SEQUENCE OF IpRouteEntry
693  *              ACCESS  not-accessible
694  *              STATUS  mandatory
695  *              DESCRIPTION
696  *                      "This entity's IP Routing table."
697  *              ::= { ip 21 }
698  */

700 typedef struct mib2_ipRouteEntry {
701                 /* dest ip addr for this route          {ipRouteEntry 1 } RW */
702         IpAddress       ipRouteDest;
703                 /* unique interface index for this hop  {ipRouteEntry 2 } RW */
704         DeviceName      ipRouteIfIndex;
705                 /* primary route metric                 {ipRouteEntry 3 } RW */
706         int             ipRouteMetric1;
707                 /* alternate route metric               {ipRouteEntry 4 } RW */
708         int             ipRouteMetric2;
709                 /* alternate route metric               {ipRouteEntry 5 } RW */
710         int             ipRouteMetric3;
711                 /* alternate route metric               {ipRouteEntry 6 } RW */
712         int             ipRouteMetric4;
713                 /* ip addr of next hop on this route    {ipRouteEntry 7 } RW */
714         IpAddress       ipRouteNextHop;
715                 /* other(1), inval(2), dir(3), indir(4) {ipRouteEntry 8 } RW */
716         int             ipRouteType;
717                 /* mechanism by which route was learned {ipRouteEntry 9 } */
718         int             ipRouteProto;
719                 /* sec's since last update of route     {ipRouteEntry 10} RW */
720         int             ipRouteAge;
721                 /*                                      {ipRouteEntry 11} RW */
```



```
722         IpAddress       ipRouteMask;
723                 /* alternate route metric               {ipRouteEntry 12} RW */
724         int             ipRouteMetric5;
725                 /* additional info from ire's           {ipRouteEntry 13 } */
726         struct ipRouteInfo_s {
727                 Gauge           re_max_frag;
728                 Gauge           re_rtt;
729                 Counter         re_ref;
730                 int             re_frag_flag;
731                 IpAddress       re_src_addr;
732                 int             re_ire_type;
733                 Counter         re_obpkt;
734                 Counter         re_ibpkt;
735                 int             re_flags;
736                 /*
737                  * The following two elements (re_in_ill and re_in_src_addr)
738                  * are no longer used but are left here for the benefit of
739                  * old Apps that won't be able to handle the change in the
740                  * size of this struct. These elements will always be
741                  * set to zeroes.
742                  */
743                 DeviceName      re_in_ill;      /* Input interface */
744                 IpAddress       re_in_src_addr; /* Input source address */
745         }               ipRouteInfo;
746 } mib2_ipRouteEntry_t;

748 /*
749  * The IPv6 routing table contains an entry for each route presently known to
750  * this entity.
751  *
752  *      ipv6RouteTable OBJECT-TYPE
753  *              SYNTAX  SEQUENCE OF IpRouteEntry
754  *              ACCESS  not-accessible
755  *              STATUS  current
756  *              DESCRIPTION
757  *                      "IPv6 Routing table. This table contains
758  *                      an entry for each valid IPv6 unicast route
759  *                      that can be used for packet forwarding
760  *                      determination."
761  *              ::= { ipv6MIBObjects 11 }
762  */

764 typedef struct mib2_ipv6RouteEntry {
765                 /* dest ip addr for this route          { ipv6RouteEntry 1 } */
766         Ip6Address      ipv6RouteDest;
767                 /* prefix length                        { ipv6RouteEntry 2 } */
768         int             ipv6RoutePfxLength;
769                 /* unique route index                   { ipv6RouteEntry 3 } */
770         unsigned        ipv6RouteIndex;
771                 /* unique interface index for this hop  { ipv6RouteEntry 4 } */
772         DeviceName      ipv6RouteIfIndex;
773                 /* IPv6 addr of next hop on this route  { ipv6RouteEntry 5 } */
774         Ip6Address      ipv6RouteNextHop;
775                 /* other(1), discard(2), local(3), remote(4) */
776                 /*                                      { ipv6RouteEntry 6 } */
777         int             ipv6RouteType;
778                 /* mechanism by which route was learned { ipv6RouteEntry 7 } */
779                 /*
780                  * other(1), local(2), netmgmt(3), ndisc(4), rip(5), ospf(6),
781                  * bgp(7), idrp(8), igrp(9)
782                  */
783         int             ipv6RouteProtocol;
784                 /* policy hook or traffic class         { ipv6RouteEntry 8 } */
785         unsigned        ipv6RoutePolicy;
786                 /* sec's since last update of route     { ipv6RouteEntry 9} */
787         int             ipv6RouteAge;
```

```
788                 /* Routing domain ID of the next hop    { ipv6RouteEntry 10 } */
789         unsigned        ipv6RouteNextHopRDI;
790                 /* route metric                         { ipv6RouteEntry 11 } */
791         unsigned        ipv6RouteMetric;
792                 /* preference (impl specific)           { ipv6RouteEntry 12 } */
793         unsigned        ipv6RouteWeight;
794                 /* additional info from ire's           { } */
795         struct ipv6RouteInfo_s {
796                 Gauge           re_max_frag;
797                 Gauge           re_rtt;
798                 Counter         re_ref;
799                 int             re_frag_flag;
800                 Ip6Address      re_src_addr;
801                 int             re_ire_type;
802                 Counter         re_obpkt;
803                 Counter         re_ibpkt;
804                 int             re_flags;
805         }               ipv6RouteInfo;
806 } mib2_ipv6RouteEntry_t;

808 /*
809  * The IPv4 and IPv6 routing table entries on a trusted system also have
810  * security attributes in the form of label ranges.  This experimental
811  * interface provides information about these labels.
812  *
813  * Each entry in this table contains a label range and an index that refers
814  * back to the entry in the routing table to which it applies.  There may be 0,
815  * 1, or many label ranges for each routing table entry.
816  *
817  * (opthdr.level is set to MIB2_IP for IPv4 entries and MIB2_IP6 for IPv6.
818  * opthdr.name is set to EXPER_IP_GWATTR.)
819  *
820  *      ipRouteAttributeTable OBJECT-TYPE
821  *              SYNTAX  SEQUENCE OF IpAttributeEntry
822  *              ACCESS  not-accessible
823  *              STATUS  current
824  *              DESCRIPTION
825  *                      "IPv4 routing attributes table.  This table contains
826  *                      an entry for each valid trusted label attached to a
827  *                      route in the system."
828  *              ::= { ip 102 }
829  *
830  *      ipv6RouteAttributeTable OBJECT-TYPE
831  *              SYNTAX  SEQUENCE OF IpAttributeEntry
832  *              ACCESS  not-accessible
833  *              STATUS  current
834  *              DESCRIPTION
835  *                      "IPv6 routing attributes table.  This table contains
836  *                      an entry for each valid trusted label attached to a
837  *                      route in the system."
838  *              ::= { ip6 102 }
839  */

841 typedef struct mib2_ipAttributeEntry {
842         uint_t          iae_routeidx;
843         int             iae_doi;
844         brange_t        iae_slrange;
845 } mib2_ipAttributeEntry_t;

847 /*
848  * The IP address translation table contain the IpAddress to
849  * 'physical' address equivalences.  Some interfaces do not
850  * use translation tables for determining address
851  * equivalences (e.g., DDN-X.25 has an algorithmic method);
852  * if all interfaces are of this type, then the Address
853  * Translation table is empty, i.e., has zero entries.
```



```
854  *
855  *      ipNetToMediaTable OBJECT-TYPE
856  *              SYNTAX  SEQUENCE OF IpNetToMediaEntry
857  *              ACCESS  not-accessible
858  *              STATUS  mandatory
859  *              DESCRIPTION
860  *                      "The IP Address Translation table used for mapping
861  *                      from IP addresses to physical addresses."
862  *              ::= { ip 22 }
863  */

865 typedef struct mib2_ipNetToMediaEntry {
866         /* Unique interface index               { ipNetToMediaEntry 1 } RW */
867         DeviceName      ipNetToMediaIfIndex;
868         /* Media dependent physical addr        { ipNetToMediaEntry 2 } RW */
869         PhysAddress     ipNetToMediaPhysAddress;
870         /* ip addr for this physical addr       { ipNetToMediaEntry 3 } RW */
871         IpAddress       ipNetToMediaNetAddress;
872         /* other(1), inval(2), dyn(3), stat(4)  { ipNetToMediaEntry 4 } RW */
873         int             ipNetToMediaType;
874         struct ipNetToMediaInfo_s {
875                 PhysAddress     ntm_mask;       /* subnet mask for entry */
876                 int             ntm_flags;      /* ACE_F_* flags in arp.h */
877         }               ipNetToMediaInfo;
878 } mib2_ipNetToMediaEntry_t;

880 /*
881  *      ipv6NetToMediaTable OBJECT-TYPE
882  *               SYNTAX      SEQUENCE OF Ipv6NetToMediaEntry
883  *               MAX-ACCESS  not-accessible
884  *               STATUS      current
885  *               DESCRIPTION
886  *                      "The IPv6 Address Translation table used for
887  *                      mapping from IPv6 addresses to physical addresses.
888  *
889  *                      The IPv6 address translation table contain the
890  *                      Ipv6Address to 'physical' address equivalencies.
891  *                      Some interfaces do not use translation tables
892  *                      for determining address equivalencies; if all
893  *                      interfaces are of this type, then the Address
894  *                      Translation table is empty, i.e., has zero
895  *                      entries."
896  *              ::= { ipv6MIBObjects 12 }
897  */

899 typedef struct mib2_ipv6NetToMediaEntry {
900         /* Unique interface index               { Part of INDEX } */
901         DeviceIndex     ipv6NetToMediaIfIndex;

903         /* ip addr for this physical addr       { ipv6NetToMediaEntry 1 } */
904         Ip6Address      ipv6NetToMediaNetAddress;
905         /* Media dependent physical addr        { ipv6NetToMediaEntry 2 } */
906         PhysAddress     ipv6NetToMediaPhysAddress;
907         /*
908          * Type of mapping
909          * other(1), dynamic(2), static(3), local(4)
910          *                                      { ipv6NetToMediaEntry 3 }
911          */
912         int             ipv6NetToMediaType;
913         /*
914          * NUD state
915          * reachable(1), stale(2), delay(3), probe(4), invalid(5), unknown(6)
916          * Note: The kernel returns ND_* states.
917          *                                      { ipv6NetToMediaEntry 4 }
918          */
919         int             ipv6NetToMediaState;
```

```
920         /* sysUpTime last time entry was updated { ipv6NetToMediaEntry 5 } */
921         int             ipv6NetToMediaLastUpdated;
922 } mib2_ipv6NetToMediaEntry_t;


925 /*
926  * List of group members per interface
927  */
928 typedef struct ip_member {
929         /* Interface index */
930         DeviceName      ipGroupMemberIfIndex;
931         /* IP Multicast address */
932         IpAddress       ipGroupMemberAddress;
933         /* Number of member sockets */
934         Counter         ipGroupMemberRefCnt;
935         /* Filter mode: 1 => include, 2 => exclude */
936         int             ipGroupMemberFilterMode;
937 } ip_member_t;


940 /*
941  * List of IPv6 group members per interface
942  */
943 typedef struct ipv6_member {
944         /* Interface index */
945         DeviceIndex     ipv6GroupMemberIfIndex;
946         /* IP Multicast address */
947         Ip6Address      ipv6GroupMemberAddress;
948         /* Number of member sockets */
949         Counter         ipv6GroupMemberRefCnt;
950         /* Filter mode: 1 => include, 2 => exclude */
951         int             ipv6GroupMemberFilterMode;
952 } ipv6_member_t;

954 /*
955  * This is used to mark transport layer entities (e.g., TCP connections) that
956  * are capable of receiving packets from a range of labels.  'level' is set to
957  * the protocol of interest (e.g., MIB2_TCP), and 'name' is set to
958  * EXPER_XPORT_MLP.  The tme_connidx refers back to the entry in MIB2_TCP_CONN,
959  * MIB2_TCP6_CONN, or MIB2_SCTP_CONN.
960  *
961  * It is also used to report connections that receive packets at a single label
962  * that's other than the zone's label.  This is the case when a TCP connection
963  * is accepted from a particular peer using an MLP listener.
964  */
965 typedef struct mib2_transportMLPEntry {
966         uint_t          tme_connidx;
967         uint_t          tme_flags;
968         int             tme_doi;
969         bslabel_t       tme_label;
970 } mib2_transportMLPEntry_t;

972 #define MIB2_TMEF_PRIVATE       0x00000001      /* MLP on private addresses */
973 #define MIB2_TMEF_SHARED        0x00000002      /* MLP on shared addresses */
974 #define MIB2_TMEF_ANONMLP       0x00000004      /* Anonymous MLP port */
975 #define MIB2_TMEF_MACEXEMPT     0x00000008      /* MAC-Exempt port */
976 #define MIB2_TMEF_IS_LABELED    0x00000010      /* tme_doi & tme_label exists */
977 #define MIB2_TMEF_MACIMPLICIT   0x00000020      /* MAC-Implicit */
978 /*
979  * List of IPv4 source addresses being filtered per interface
980  */
981 typedef struct ip_grpsrc {
982         /* Interface index */
983         DeviceName      ipGroupSourceIfIndex;
984         /* IP Multicast address */
985         IpAddress       ipGroupSourceGroup;
```



```
986         /* IP Source address */
987         IpAddress       ipGroupSourceAddress;
988 } ip_grpsrc_t;


991 /*
992  * List of IPv6 source addresses being filtered per interface
993  */
994 typedef struct ipv6_grpsrc {
995         /* Interface index */
996         DeviceIndex     ipv6GroupSourceIfIndex;
997         /* IP Multicast address */
998         Ip6Address      ipv6GroupSourceGroup;
999         /* IP Source address */
1000        Ip6Address      ipv6GroupSourceAddress;
1001 } ipv6_grpsrc_t;


1004 /*
1005  * List of destination cache entries
1006  */
1007 typedef struct dest_cache_entry {
1008         /* IP Multicast address */
1009         IpAddress       DestIpv4Address;
1010         Ip6Address      DestIpv6Address;
1011         uint_t          DestFlags;      /* DCEF_* */
1012         uint32_t        DestPmtu;       /* Path MTU if DCEF_PMTU */
1013         uint32_t        DestIdent;      /* Per destination IP ident. */
1014         DeviceIndex     DestIfindex;    /* For IPv6 link-locals */
1015         uint32_t        DestAge;        /* Age of MTU info in seconds */
1016 } dest_cache_entry_t;


1019 /*
1020  * ICMP Group
1021  */
1022 typedef struct mib2_icmp {
1023         /* total # of recv'd ICMP msgs                  { icmp 1 } */
1024         Counter icmpInMsgs;
1025         /* recv'd ICMP msgs with errors                 { icmp 2 } */
1026         Counter icmpInErrors;
1027         /* recv'd "dest unreachable" msg's              { icmp 3 } */
1028         Counter icmpInDestUnreachs;
1029         /* recv'd "time exceeded" msg's                 { icmp 4 } */
1030         Counter icmpInTimeExcds;
1031         /* recv'd "parameter problem" msg's             { icmp 5 } */
1032         Counter icmpInParmProbs;
1033         /* recv'd "source quench" msg's                 { icmp 6 } */
1034         Counter icmpInSrcQuenchs;
1035         /* recv'd "ICMP redirect" msg's                 { icmp 7 } */
1036         Counter icmpInRedirects;
1037         /* recv'd "echo request" msg's                  { icmp 8 } */
1038         Counter icmpInEchos;
1039         /* recv'd "echo reply" msg's                    { icmp 9 } */
1040         Counter icmpInEchoReps;
1041         /* recv'd "timestamp" msg's                     { icmp 10 } */
1042         Counter icmpInTimestamps;
1043         /* recv'd "timestamp reply" msg's               { icmp 11 } */
1044         Counter icmpInTimestampReps;
1045         /* recv'd "address mask request" msg's          { icmp 12 } */
1046         Counter icmpInAddrMasks;
1047         /* recv'd "address mask reply" msg's            { icmp 13 } */
1048         Counter icmpInAddrMaskReps;
1049         /* total # of sent ICMP msg's                   { icmp 14 } */
1050         Counter icmpOutMsgs;
1051         /* # of msg's not sent for internal icmp errors { icmp 15 } */
```

```
1052         Counter icmpOutErrors;
1053         /* # of "dest unreachable" msg's sent           { icmp 16 } */
1054         Counter icmpOutDestUnreachs;
1055         /* # of "time exceeded" msg's sent              { icmp 17 } */
1056         Counter icmpOutTimeExcds;
1057         /* # of "parameter problme" msg's sent          { icmp 18 } */
1058         Counter icmpOutParmProbs;
1059         /* # of "source quench" msg's sent              { icmp 19 } */
1060         Counter icmpOutSrcQuenchs;
1061         /* # of "ICMP redirect" msg's sent              { icmp 20 } */
1062         Counter icmpOutRedirects;
1063         /* # of "Echo request" msg's sent               { icmp 21 } */
1064         Counter icmpOutEchos;
1065         /* # of "Echo reply" msg's sent                 { icmp 22 } */
1066         Counter icmpOutEchoReps;
1067         /* # of "timestamp request" msg's sent          { icmp 23 } */
1068         Counter icmpOutTimestamps;
1069         /* # of "timestamp reply" msg's sent            { icmp 24 } */
1070         Counter icmpOutTimestampReps;
1071         /* # of "address mask request" msg's sent       { icmp 25 } */
1072         Counter icmpOutAddrMasks;
1073         /* # of "address mask reply" msg's sent         { icmp 26 } */
1074         Counter icmpOutAddrMaskReps;
1075 /*
1076  * In addition to MIB-II
1077  */
1078         /* # of received packets with checksum errors */
1079         Counter icmpInCksumErrs;
1080         /* # of received packets with unknow codes */
1081         Counter icmpInUnknowns;
1082         /* # of received unreachables with "fragmentation needed" */
1083         Counter icmpInFragNeeded;
1084         /* # of sent unreachables with "fragmentation needed" */
1085         Counter icmpOutFragNeeded;
1086         /*
1087          * # of msg's not sent since original packet was broadcast/multicast
1088          * or an ICMP error packet
1089          */
1090         Counter icmpOutDrops;
1091         /* # of ICMP packets droped due to queue overflow */
1092         Counter icmpInOverflows;
1093         /* recv'd "ICMP redirect" msg's that are bad thus ignored */
1094         Counter icmpInBadRedirects;
1095 } mib2_icmp_t;


1098 /*
1099  *      ipv6IfIcmpEntry OBJECT-TYPE
1100  *              SYNTAX      Ipv6IfIcmpEntry
1101  *              MAX-ACCESS  not-accessible
1102  *              STATUS      current
1103  *              DESCRIPTION
1104  *                      "An ICMPv6 statistics entry containing
1105  *                      objects at a particular IPv6 interface.
1106  *
1107  *                      Note that a receiving interface is
1108  *                      the interface to which a given ICMPv6 message
1109  *                      is addressed which may not be necessarily
1110  *                      the input interface for the message.
1111  *
1112  *                      Similarly, the sending interface is
1113  *                      the interface that sources a given
1114  *                      ICMP message which is usually but not
1115  *                      necessarily the output interface for the message."
1116  *              AUGMENTS { ipv6IfEntry }
1117  *              ::= { ipv6IfIcmpTable 1 }
```



```
1118  *
1119  * Per-interface ICMPv6 statistics table
1120  */

1122 typedef struct mib2_ipv6IfIcmpEntry {
1123         /* Local ifindex to identify the interface */
1124         DeviceIndex     ipv6IfIcmpIfIndex;

1126         int             ipv6IfIcmpEntrySize;    /* Size of ipv6IfIcmpEntry */

1128         /* The total # ICMP msgs rcvd includes ipv6IfIcmpInErrors */
1129         Counter32       ipv6IfIcmpInMsgs;
1130         /* # ICMP with ICMP-specific errors (bad checkum, length, etc) */
1131         Counter32       ipv6IfIcmpInErrors;
1132         /* # ICMP Destination Unreachable */
1133         Counter32       ipv6IfIcmpInDestUnreachs;
1134         /* # ICMP destination unreachable/communication admin prohibited */
1135         Counter32       ipv6IfIcmpInAdminProhibs;
1136         Counter32       ipv6IfIcmpInTimeExcds;
1137         Counter32       ipv6IfIcmpInParmProblems;
1138         Counter32       ipv6IfIcmpInPktTooBigs;
1139         Counter32       ipv6IfIcmpInEchos;
1140         Counter32       ipv6IfIcmpInEchoReplies;
1141         Counter32       ipv6IfIcmpInRouterSolicits;
1142         Counter32       ipv6IfIcmpInRouterAdvertisements;
1143         Counter32       ipv6IfIcmpInNeighborSolicits;
1144         Counter32       ipv6IfIcmpInNeighborAdvertisements;
1145         Counter32       ipv6IfIcmpInRedirects;
1146         Counter32       ipv6IfIcmpInGroupMembQueries;
1147         Counter32       ipv6IfIcmpInGroupMembResponses;
1148         Counter32       ipv6IfIcmpInGroupMembReductions;
1149         /* Total # ICMP messages attempted to send (includes OutErrors) */
1150         Counter32       ipv6IfIcmpOutMsgs;
1151         /* # ICMP messages not sent due to ICMP problems (e.g. no buffers) */
1152         Counter32       ipv6IfIcmpOutErrors;
1153         Counter32       ipv6IfIcmpOutDestUnreachs;
1154         Counter32       ipv6IfIcmpOutAdminProhibs;
1155         Counter32       ipv6IfIcmpOutTimeExcds;
1156         Counter32       ipv6IfIcmpOutParmProblems;
1157         Counter32       ipv6IfIcmpOutPktTooBigs;
1158         Counter32       ipv6IfIcmpOutEchos;
1159         Counter32       ipv6IfIcmpOutEchoReplies;
1160         Counter32       ipv6IfIcmpOutRouterSolicits;
1161         Counter32       ipv6IfIcmpOutRouterAdvertisements;
1162         Counter32       ipv6IfIcmpOutNeighborSolicits;
1163         Counter32       ipv6IfIcmpOutNeighborAdvertisements;
1164         Counter32       ipv6IfIcmpOutRedirects;
1165         Counter32       ipv6IfIcmpOutGroupMembQueries;
1166         Counter32       ipv6IfIcmpOutGroupMembResponses;
1167         Counter32       ipv6IfIcmpOutGroupMembReductions;
1168 /* Additions beyond the MIB */
1169         Counter32       ipv6IfIcmpInOverflows;
1170         /* recv'd "ICMPv6 redirect" msg's that are bad thus ignored */
1171         Counter32       ipv6IfIcmpBadHoplimit;
1172         Counter32       ipv6IfIcmpInBadNeighborAdvertisements;
1173         Counter32       ipv6IfIcmpInBadNeighborSolicitations;
1174         Counter32       ipv6IfIcmpInBadRedirects;
1175         Counter32       ipv6IfIcmpInGroupMembTotal;
1176         Counter32       ipv6IfIcmpInGroupMembBadQueries;
1177         Counter32       ipv6IfIcmpInGroupMembBadReports;
1178         Counter32       ipv6IfIcmpInGroupMembOurReports;
1179 } mib2_ipv6IfIcmpEntry_t;

1181 /*
1182  * the TCP group
1183  *
```

```
 * Note that instances of object types that represent
 * information about a particular TCP connection are
 * transient; they persist only as long as the connection
 * in question.
 */
#define MIB2_TCP_CONN   13      /* tcpConnEntry */
#define MIB2_TCP6_CONN  14      /* tcp6ConnEntry */

/* Old name retained for compatibility */
#define MIB2_TCP_13     MIB2_TCP_CONN

/* Pack data in mib2_tcp 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_tcp {
                /* algorithm used for transmit timeout value    { tcp 1 } */
        int     tcpRtoAlgorithm;
                /* minimum retransmit timeout (ms)              { tcp 2 } */
        int     tcpRtoMin;
                /* maximum retransmit timeout (ms)              { tcp 3 } */
        int     tcpRtoMax;
                /* maximum # of connections supported           { tcp 4 } */
        int     tcpMaxConn;
                /* # of direct transitions CLOSED -> SYN-SENT   { tcp 5 } */
        Counter tcpActiveOpens;
                /* # of direct transitions LISTEN -> SYN-RCVD   { tcp 6 } */
        Counter tcpPassiveOpens;
                /* # of direct SIN-SENT/RCVD -> CLOSED/LISTEN   { tcp 7 } */
        Counter tcpAttemptFails;
                /* # of direct ESTABLISHED/CLOSE-WAIT -> CLOSED { tcp 8 } */
        Counter tcpEstabResets;
                /* # of connections ESTABLISHED or CLOSE-WAIT   { tcp 9 } */
        Gauge   tcpCurrEstab;
                /* total # of segments recv'd                   { tcp 10 } */
        Counter tcpInSegs;
                /* total # of segments sent                     { tcp 11 } */
        Counter tcpOutSegs;
                /* total # of segments retransmitted            { tcp 12 } */
        Counter tcpRetransSegs;
                /* {tcp 13} */
        int     tcpConnTableSize;       /* Size of tcpConnEntry_t */
        /* in ip                                    {tcp 14} */
                /* # of segments sent with RST flag             { tcp 15 } */
        Counter tcpOutRsts;
/* In addition to MIB-II */
/* Sender */
        /* total # of data segments sent */
        Counter tcpOutDataSegs;
        /* total # of bytes in data segments sent */
        Counter tcpOutDataBytes;
        /* total # of bytes in segments retransmitted */
        Counter tcpRetransBytes;
        /* total # of acks sent */
        Counter tcpOutAck;
        /* total # of delayed acks sent */
        Counter tcpOutAckDelayed;
        /* total # of segments sent with the urg flag on */
        Counter tcpOutUrg;
        /* total # of window updates sent */
        Counter tcpOutWinUpdate;
        /* total # of zero window probes sent */
        Counter tcpOutWinProbe;
        /* total # of control segments sent (syn, fin, rst) */
        Counter tcpOutControl;
        /* total # of segments sent due to "fast retransmit" */
```



```
        Counter tcpOutFastRetrans;
/* Receiver */
        /* total # of ack segments received */
        Counter tcpInAckSegs;
        /* total # of bytes acked */
        Counter tcpInAckBytes;
        /* total # of duplicate acks */
        Counter tcpInDupAck;
        /* total # of acks acking unsent data */
        Counter tcpInAckUnsent;
        /* total # of data segments received in order */
        Counter tcpInDataInorderSegs;
        /* total # of data bytes received in order */
        Counter tcpInDataInorderBytes;
        /* total # of data segments received out of order */
        Counter tcpInDataUnorderSegs;
        /* total # of data bytes received out of order */
        Counter tcpInDataUnorderBytes;
        /* total # of complete duplicate data segments received */
        Counter tcpInDataDupSegs;
        /* total # of bytes in the complete duplicate data segments received */
        Counter tcpInDataDupBytes;
        /* total # of partial duplicate data segments received */
        Counter tcpInDataPartDupSegs;
        /* total # of bytes in the partial duplicate data segments received */
        Counter tcpInDataPartDupBytes;
        /* total # of data segments received past the window */
        Counter tcpInDataPastWinSegs;
        /* total # of data bytes received part the window */
        Counter tcpInDataPastWinBytes;
        /* total # of zero window probes received */
        Counter tcpInWinProbe;
        /* total # of window updates received */
        Counter tcpInWinUpdate;
        /* total # of data segments received after the connection has closed */
        Counter tcpInClosed;
/* Others */
        /* total # of failed attempts to update the rtt estimate */
        Counter tcpRttNoUpdate;
        /* total # of successful attempts to update the rtt estimate */
        Counter tcpRttUpdate;
        /* total # of retransmit timeouts */
        Counter tcpTimRetrans;
        /* total # of retransmit timeouts dropping the connection */
        Counter tcpTimRetransDrop;
        /* total # of keepalive timeouts */
        Counter tcpTimKeepalive;
        /* total # of keepalive timeouts sending a probe */
        Counter tcpTimKeepaliveProbe;
        /* total # of keepalive timeouts dropping the connection */
        Counter tcpTimKeepaliveDrop;
        /* total # of connections refused due to backlog full on listen */
        Counter tcpListenDrop;
        /* total # of connections refused due to half-open queue (q0) full */
        Counter tcpListenDropQ0;
        /* total # of connections dropped from a full half-open queue (q0) */
        Counter tcpHalfOpenDrop;
        /* total # of retransmitted segments by SACK retransmission */
        Counter tcpOutSackRetransSegs;

        int     tcp6ConnTableSize;      /* Size of tcp6ConnEntry_t */

        /*
         * fields from RFC 4022
         */
```

```
1316         /* total # of segments recv'd                          { tcp 17 } */
1317         Counter64       tcpHCInSegs;
1318         /* total # of segments sent                            { tcp 18 } */
1319         Counter64       tcpHCOutSegs;
1320 } mib2_tcp_t;
1321 #define MIB_FIRST_NEW_ELM_mib2_tcp_t    tcpHCInSegs

1323 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1324 #pragma pack()
1325 #endif

1327 /*
1328  * The TCP/IPv4 connection table {tcp 13} contains information about this
1329  * entity's existing TCP connections over IPv4.
1330  */
1331 /* For tcpConnState and tcp6ConnState */
1332 #define MIB2_TCP_closed         1
1333 #define MIB2_TCP_listen         2
1334 #define MIB2_TCP_synSent        3
1335 #define MIB2_TCP_synReceived    4
1336 #define MIB2_TCP_established    5
1337 #define MIB2_TCP_finWait1       6
1338 #define MIB2_TCP_finWait2       7
1339 #define MIB2_TCP_closeWait      8
1340 #define MIB2_TCP_lastAck        9
1341 #define MIB2_TCP_closing        10
1342 #define MIB2_TCP_timeWait       11
1343 #define MIB2_TCP_deleteTCB      12              /* only writeable value */

1345 /* Pack data to make struct size the same for 32- and 64-bits */
1346 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1347 #pragma pack(4)
1348 #endif
1349 typedef struct mib2_tcpConnEntry {
1350                 /* state of tcp connection              { tcpConnEntry 1} RW */
1351         int             tcpConnState;
1352                 /* local ip addr for this connection    { tcpConnEntry 2 } */
1353         IpAddress       tcpConnLocalAddress;
1354                 /* local port for this connection       { tcpConnEntry 3 } */
1355         int             tcpConnLocalPort;       /* In host byte order */
1356                 /* remote ip addr for this connection   { tcpConnEntry 4 } */
1357         IpAddress       tcpConnRemAddress;
1358                 /* remote port for this connection      { tcpConnEntry 5 } */
1359         int             tcpConnRemPort;         /* In host byte order */
1360         struct tcpConnEntryInfo_s {
1361                         /* seq # of next segment to send */
1362                 Gauge           ce_snxt;
1363                                 /* seq # of of last segment unacknowledged */
1364                 Gauge           ce_suna;
1365                                 /* currect send window size */
1366                 Gauge           ce_swnd;
1367                                 /* seq # of next expected segment */
1368                 Gauge           ce_rnxt;
1369                                 /* seq # of last ack'd segment */
1370                 Gauge           ce_rack;
1371                                 /* currenct receive window size */
1372                 Gauge           ce_rwnd;
1373                                         /* current rto (retransmit timeout) */
1374                 Gauge           ce_rto;
1375                                         /* current max segment size */
1376                 Gauge           ce_mss;
1377                                 /* actual internal state */
1378                 int             ce_state;
1379         }               tcpConnEntryInfo;

1381         /* pid of the processes that created this connection */
```



```
1382         uint32_t        tcpConnCreationProcess;
1383         /* system uptime when the connection was created */
1384         uint64_t        tcpConnCreationTime;
1385 } mib2_tcpConnEntry_t;
1386 #define MIB_FIRST_NEW_ELM_mib2_tcpConnEntry_t   tcpConnCreationProcess

1388 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1389 #pragma pack()
1390 #endif


1393 /*
1394  * The TCP/IPv6 connection table {tcp 14} contains information about this
1395  * entity's existing TCP connections over IPv6.
1396  */

1398 /* Pack data to make struct size the same for 32- and 64-bits */
1399 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1400 #pragma pack(4)
1401 #endif
1402 typedef struct mib2_tcp6ConnEntry {
1403         /* local ip addr for this connection    { ipv6TcpConnEntry 1 } */
1404         Ip6Address      tcp6ConnLocalAddress;
1405         /* local port for this connection       { ipv6TcpConnEntry 2 } */
1406         int             tcp6ConnLocalPort;
1407         /* remote ip addr for this connection   { ipv6TcpConnEntry 3 } */
1408         Ip6Address      tcp6ConnRemAddress;
1409         /* remote port for this connection      { ipv6TcpConnEntry 4 } */
1410         int             tcp6ConnRemPort;
1411         /* interface index or zero              { ipv6TcpConnEntry 5 } */
1412         DeviceIndex     tcp6ConnIfIndex;
1413         /* state of tcp6 connection             { ipv6TcpConnEntry 6 } RW */
1414         int             tcp6ConnState;
1415         struct tcp6ConnEntryInfo_s {
1416                         /* seq # of next segment to send */
1417                 Gauge           ce_snxt;
1418                                 /* seq # of of last segment unacknowledged */
1419                 Gauge           ce_suna;
1420                                 /* currect send window size */
1421                 Gauge           ce_swnd;
1422                                 /* seq # of next expected segment */
1423                 Gauge           ce_rnxt;
1424                                 /* seq # of last ack'd segment */
1425                 Gauge           ce_rack;
1426                                 /* currenct receive window size */
1427                 Gauge           ce_rwnd;
1428                                         /* current rto (retransmit timeout) */
1429                 Gauge           ce_rto;
1430                                         /* current max segment size */
1431                 Gauge           ce_mss;
1432                                 /* actual internal state */
1433                 int             ce_state;
1434         }               tcp6ConnEntryInfo;

1436         /* pid of the processes that created this connection */
1437         uint32_t        tcp6ConnCreationProcess;
1438         /* system uptime when the connection was created */
1439         uint64_t        tcp6ConnCreationTime;
1440 } mib2_tcp6ConnEntry_t;
1441 #define MIB_FIRST_NEW_ELM_mib2_tcp6ConnEntry_t  tcp6ConnCreationProcess

1443 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1444 #pragma pack()
1445 #endif

1447 /*
```

```
 * 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;
```

```
1712                /* max # of assocs                      { sctpParams 5 } */
1713        int32_t         sctpMaxAssocs;
1714                /* cookie lifetime in msecs             { sctpParams 6 } */
1715        uint32_t        sctpValCookieLife;
1716                /* max # of retrans in startup          { sctpParams 7 } */
1717        uint32_t        sctpMaxInitRetr;
1718        /* # of conns ESTABLISHED, SHUTDOWN-RECEIVED or SHUTDOWN-PENDING */
1719        Counter32       sctpCurrEstab;          /* { sctpStats 1 } */
1720                /* # of active opens                    { sctpStats 2 } */
1721        Counter32       sctpActiveEstab;
1722                /* # of passive opens                   { sctpStats 3 } */
1723        Counter32       sctpPassiveEstab;
1724                /* # of aborted conns                   { sctpStats 4 } */
1725        Counter32       sctpAborted;
1726                /* # of graceful shutdowns              { sctpStats 5 } */
1727        Counter32       sctpShutdowns;
1728                /* # of OOB packets                     { sctpStats 6 } */
1729        Counter32       sctpOutOfBlue;
1730                /* # of packets discarded due to cksum  { sctpStats 7 } */
1731        Counter32       sctpChecksumError;
1732                /* # of control chunks sent             { sctpStats 8 } */
1733        Counter64       sctpOutCtrlChunks;
1734                /* # of ordered data chunks sent        { sctpStats 9 } */
1735        Counter64       sctpOutOrderChunks;
1736                /* # of unordered data chunks sent      { sctpStats 10 } */
1737        Counter64       sctpOutUnorderChunks;
1738                /* # of retransmitted data chunks */
1739        Counter64       sctpRetransChunks;
1740                /* # of SACK chunks sent */
1741        Counter         sctpOutAck;
1742                /* # of delayed ACK timeouts */
1743        Counter         sctpOutAckDelayed;
1744                /* # of SACK chunks sent to update window */
1745        Counter         sctpOutWinUpdate;
1746                /* # of fast retransmits */
1747        Counter         sctpOutFastRetrans;
1748                /* # of window probes sent */
1749        Counter         sctpOutWinProbe;
1750                /* # of control chunks received         { sctpStats 11 } */
1751        Counter64       sctpInCtrlChunks;
1752                /* # of ordered data chunks rcvd        { sctpStats 12 } */
1753        Counter64       sctpInOrderChunks;
1754                /* # of unord data chunks rcvd          { sctpStats 13 } */
1755        Counter64       sctpInUnorderChunks;
1756                /* # of received SACK chunks */
1757        Counter         sctpInAck;
1758                /* # of received SACK chunks with duplicate TSN */
1759        Counter         sctpInDupAck;
1760                /* # of SACK chunks acking unsent data */
1761        Counter         sctpInAckUnsent;
1762                /* # of Fragmented User Messages        { sctpStats 14 } */
1763        Counter64       sctpFragUsrMsgs;
1764                /* # of Reassembled User Messages       { sctpStats 15 } */
1765        Counter64       sctpReasmUsrMsgs;
1766                /* # of Sent SCTP Packets               { sctpStats 16 } */
1767        Counter64       sctpOutSCTPPkts;
1768                /* # of Received SCTP Packets           { sctpStats 17 } */
1769        Counter64       sctpInSCTPPkts;
1770                /* # of invalid cookies received */
1771        Counter         sctpInInvalidCookie;
1772                /* total # of retransmit timeouts */
1773        Counter         sctpTimRetrans;
1774                /* total # of retransmit timeouts dropping the connection */
1775        Counter         sctpTimRetransDrop;
1776                /* total # of heartbeat probes */
1777        Counter         sctpTimHeartBeatProbe;
```



```
1778                /* total # of heartbeat timeouts dropping the connection */
1779        Counter         sctpTimHeartBeatDrop;
1780                /* total # of conns refused due to backlog full on listen */
1781        Counter         sctpListenDrop;
1782                /* total # of pkts received after the association has closed */
1783        Counter         sctpInClosed;
1784        int             sctpEntrySize;
1785        int             sctpLocalEntrySize;
1786        int             sctpRemoteEntrySize;
1787 } mib2_sctp_t;

1789 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1790 #pragma pack()
1791 #endif


1794 #ifdef  __cplusplus
1795 }
1796 #endif

1798 #endif  /* _INET_MIB2_H */
```

```
**********************************************************
   32379 Sun Aug  9 12:47:57 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;


 561         conn_pid_node_list_hdr_t        *cph;

 563 #endif /* ! codereview */
 564         /*
 565          * Make copies of the original message.
 566          * mpctl will hold SCTP counters,
 567          * mp_conn_ctl will hold list of connections.
 568          */
 569         mp_ret = copymsg(mpctl);
 570         mp_conn_ctl = copymsg(mpctl);
 571         mp_pidnode_ctl = copymsg(mpctl);
 572 #endif /* ! codereview */
 573         mp_local_ctl = copymsg(mpctl);
 574         mp_rem_ctl = copymsg(mpctl);
 575         mp_attr_ctl = copymsg(mpctl);
```



```
 577         mpdata = mpctl->b_cont;

 579         if (mp_conn_ctl == NULL || mp_pidnode_ctl == NULL ||
 580             mp_local_ctl == NULL || mp_rem_ctl == NULL || mp_attr_ctl == NULL ||
 581             mpdata == NULL) {
 531         if (mp_conn_ctl == NULL || mp_local_ctl == NULL ||
 532             mp_rem_ctl == NULL || mp_attr_ctl == NULL || mpdata == NULL) {
 582                 freemsg(mp_attr_ctl);
 583                 freemsg(mp_rem_ctl);
 584                 freemsg(mp_local_ctl);
 585                 freemsg(mp_pidnode_ctl);
 586 #endif /* ! codereview */
 587                 freemsg(mp_conn_ctl);
 588                 freemsg(mp_ret);
 589                 freemsg(mpctl);
 590                 return (NULL);
 591         }
 592         mp_conn_data = mp_conn_ctl->b_cont;
 593         mp_pidnode_data = mp_pidnode_ctl->b_cont;
 594 #endif /* ! codereview */
 595         mp_local_data = mp_local_ctl->b_cont;
 596         mp_rem_data = mp_rem_ctl->b_cont;
 597         mp_attr_data = mp_attr_ctl->b_cont;

 599         bzero(&sctp_mib, sizeof (sctp_mib));

 601         /* hostname address parameters are not supported in Solaris */
 602         sce.sctpAssocRemHostName.o_length = 0;
 603         sce.sctpAssocRemHostName.o_bytes[0] = 0;

 605         /* build table of connections -- need count in fixed part */

 607         idx = 0;
 608         mutex_enter(&sctps->sctps_g_lock);
 609         sctp = list_head(&sctps->sctps_g_list);
 610         while (sctp != NULL) {
 611                 mutex_enter(&sctp->sctp_reflock);
 612                 if (sctp->sctp_condemned) {
 613                         mutex_exit(&sctp->sctp_reflock);
 614                         sctp = list_next(&sctps->sctps_g_list, sctp);
 615                         continue;
 616                 }
 617                 sctp->sctp_refcnt++;
 618                 mutex_exit(&sctp->sctp_reflock);
 619                 mutex_exit(&sctps->sctps_g_lock);
 620                 if (sctp_prev != NULL)
 621                         SCTP_REFRELE(sctp_prev);
 622                 if (sctp->sctp_connp->conn_zoneid != zoneid)
 623                         goto next_sctp;
 624                 if (sctp->sctp_state == SCTPS_ESTABLISHED ||
 625                     sctp->sctp_state == SCTPS_SHUTDOWN_PENDING ||
 626                     sctp->sctp_state == SCTPS_SHUTDOWN_RECEIVED) {
 627                         /*
 628                          * Just bump the local sctp_mib.  The number of
 629                          * existing associations is not kept in kernel.
 630                          */
 631                         BUMP_MIB(&sctp_mib, sctpCurrEstab);
 632                 }
 633                 SCTPS_UPDATE_MIB(sctps, sctpOutSCTPPkts, sctp->sctp_opkts);
 634                 sctp->sctp_opkts = 0;
 635                 SCTPS_UPDATE_MIB(sctps, sctpOutCtrlChunks, sctp->sctp_obchunks);
 636                 UPDATE_LOCAL(sctp->sctp_cum_obchunks,
 637                     sctp->sctp_obchunks);
 638                 sctp->sctp_obchunks = 0;
 639                 SCTPS_UPDATE_MIB(sctps, sctpOutOrderChunks,
 640                     sctp->sctp_odchunks);
```

```
641                 UPDATE_LOCAL(sctp->sctp_cum_odchunks,
642                     sctp->sctp_odchunks);
643                 sctp->sctp_odchunks = 0;
644                 SCTPS_UPDATE_MIB(sctps, sctpOutUnorderChunks,
645                     sctp->sctp_oudchunks);
646                 UPDATE_LOCAL(sctp->sctp_cum_oudchunks,
647                     sctp->sctp_oudchunks);
648                 sctp->sctp_oudchunks = 0;
649                 SCTPS_UPDATE_MIB(sctps, sctpRetransChunks,
650                     sctp->sctp_rxtchunks);
651                 UPDATE_LOCAL(sctp->sctp_cum_rxtchunks,
652                     sctp->sctp_rxtchunks);
653                 sctp->sctp_rxtchunks = 0;
654                 SCTPS_UPDATE_MIB(sctps, sctpInSCTPPkts, sctp->sctp_ipkts);
655                 sctp->sctp_ipkts = 0;
656                 SCTPS_UPDATE_MIB(sctps, sctpInCtrlChunks, sctp->sctp_ibchunks);
657                 UPDATE_LOCAL(sctp->sctp_cum_ibchunks,
658                     sctp->sctp_ibchunks);
659                 sctp->sctp_ibchunks = 0;
660                 SCTPS_UPDATE_MIB(sctps, sctpInOrderChunks, sctp->sctp_idchunks);
661                 UPDATE_LOCAL(sctp->sctp_cum_idchunks,
662                     sctp->sctp_idchunks);
663                 sctp->sctp_idchunks = 0;
664                 SCTPS_UPDATE_MIB(sctps, sctpInUnorderChunks,
665                     sctp->sctp_iudchunks);
666                 UPDATE_LOCAL(sctp->sctp_cum_iudchunks,
667                     sctp->sctp_iudchunks);
668                 sctp->sctp_iudchunks = 0;
669                 SCTPS_UPDATE_MIB(sctps, sctpFragUsrMsgs, sctp->sctp_fragdmsgs);
670                 sctp->sctp_fragdmsgs = 0;
671                 SCTPS_UPDATE_MIB(sctps, sctpReasmUsrMsgs, sctp->sctp_reassmsgs);
672                 sctp->sctp_reassmsgs = 0;

674                 sce.sctpAssocId = ntohl(sctp->sctp_lvtag);
675                 sce.sctpAssocLocalPort = ntohs(sctp->sctp_connp->conn_lport);
676                 sce.sctpAssocRemPort = ntohs(sctp->sctp_connp->conn_fport);

678                 RUN_SCTP(sctp);
679                 if (sctp->sctp_primary != NULL) {
680                         fp = sctp->sctp_primary;

682                         if (IN6_IS_ADDR_V4MAPPED(&fp->sf_faddr)) {
683                                 sce.sctpAssocRemPrimAddrType =
684                                     MIB2_SCTP_ADDR_V4;
685                         } else {
686                                 sce.sctpAssocRemPrimAddrType =
687                                     MIB2_SCTP_ADDR_V6;
688                         }
689                         sce.sctpAssocRemPrimAddr = fp->sf_faddr;
690                         sce.sctpAssocLocPrimAddr = fp->sf_saddr;
691                         sce.sctpAssocHeartBeatInterval = TICK_TO_MSEC(
692                             fp->sf_hb_interval);
693                 } else {
694                         sce.sctpAssocRemPrimAddrType = MIB2_SCTP_ADDR_V4;
695                         bzero(&sce.sctpAssocRemPrimAddr,
696                             sizeof (sce.sctpAssocRemPrimAddr));
697                         bzero(&sce.sctpAssocLocPrimAddr,
698                             sizeof (sce.sctpAssocLocPrimAddr));
699                         sce.sctpAssocHeartBeatInterval =
700                             sctps->sctps_heartbeat_interval;
701                 }

703                 /*
704                  * Table for local addresses
705                  */
706                 scanned = 0;
```



```
707                 for (i = 0; i < SCTP_IPIF_HASH; i++) {
708                         sctp_saddr_ipif_t       *obj;

710                         if (sctp->sctp_saddrs[i].ipif_count == 0)
711                                 continue;
712                         obj = list_head(&sctp->sctp_saddrs[i].sctp_ipif_list);
713                         for (l = 0; l < sctp->sctp_saddrs[i].ipif_count; l++) {
714                                 sctp_ipif_t     *sctp_ipif;
715                                 in6_addr_t      addr;

717                                 sctp_ipif = obj->saddr_ipifp;
718                                 addr = sctp_ipif->sctp_ipif_saddr;
719                                 scanned++;
720                                 scle.sctpAssocId = ntohl(sctp->sctp_lvtag);
721                                 if (IN6_IS_ADDR_V4MAPPED(&addr)) {
722                                         scle.sctpAssocLocalAddrType =
723                                             MIB2_SCTP_ADDR_V4;
724                                 } else {
725                                         scle.sctpAssocLocalAddrType =
726                                             MIB2_SCTP_ADDR_V6;
727                                 }
728                                 scle.sctpAssocLocalAddr = addr;
729                                 (void) snmp_append_data2(mp_local_data,
730                                     &mp_local_tail, (char *)&scle,
731                                     sizeof (scle));
732                                 if (scanned >= sctp->sctp_nsaddrs)
733                                         goto done;
734                                 obj = list_next(&sctp->
735                                     sctp_saddrs[i].sctp_ipif_list, obj);
736                         }
737                 }
738 done:
739                 /*
740                  * Table for remote addresses
741                  */
742                 for (fp = sctp->sctp_faddrs; fp; fp = fp->sf_next) {
743                         scre.sctpAssocId = ntohl(sctp->sctp_lvtag);
744                         if (IN6_IS_ADDR_V4MAPPED(&fp->sf_faddr)) {
745                                 scre.sctpAssocRemAddrType = MIB2_SCTP_ADDR_V4;
746                         } else {
747                                 scre.sctpAssocRemAddrType = MIB2_SCTP_ADDR_V6;
748                         }
749                         scre.sctpAssocRemAddr = fp->sf_faddr;
750                         if (fp->sf_state == SCTP_FADDRS_ALIVE) {
751                                 scre.sctpAssocRemAddrActive =
752                                     scre.sctpAssocRemAddrHBActive =
753                                     MIB2_SCTP_ACTIVE;
754                         } else {
755                                 scre.sctpAssocRemAddrActive =
756                                     scre.sctpAssocRemAddrHBActive =
757                                     MIB2_SCTP_INACTIVE;
758                         }
759                         scre.sctpAssocRemAddrRTO = TICK_TO_MSEC(fp->sf_rto);
760                         scre.sctpAssocRemAddrMaxPathRtx = fp->sf_max_retr;
761                         scre.sctpAssocRemAddrRtx = fp->sf_T3expire;
762                         (void) snmp_append_data2(mp_rem_data, &mp_rem_tail,
763                             (char *)&scre, sizeof (scre));
764                 }
765                 connp = sctp->sctp_connp;
766                 needattr = B_FALSE;
767                 bzero(&mlp, sizeof (mlp));
768                 if (connp->conn_mlp_type != mlptSingle) {
769                         if (connp->conn_mlp_type == mlptShared ||
770                             connp->conn_mlp_type == mlptBoth)
771                                 mlp.tme_flags |= MIB2_TMEF_SHARED;
772                         if (connp->conn_mlp_type == mlptPrivate ||
```

```
773                             connp->conn_mlp_type == mlptBoth)
774                                 mlp.tme_flags |= MIB2_TMEF_PRIVATE;
775                         needattr = B_TRUE;
776                 }
777                 if (connp->conn_anon_mlp) {
778                         mlp.tme_flags |= MIB2_TMEF_ANONMLP;
779                         needattr = B_TRUE;
780                 }
781                 switch (connp->conn_mac_mode) {
782                 case CONN_MAC_DEFAULT:
783                         break;
784                 case CONN_MAC_AWARE:
785                         mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
786                         needattr = B_TRUE;
787                         break;
788                 case CONN_MAC_IMPLICIT:
789                         mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
790                         needattr = B_TRUE;
791                         break;
792                 }
793                 if (sctp->sctp_connp->conn_ixa->ixa_tsl != NULL) {
794                         ts_label_t *tsl;

796                         tsl = sctp->sctp_connp->conn_ixa->ixa_tsl;
797                         mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
798                         mlp.tme_doi = label2doi(tsl);
799                         mlp.tme_label = *label2bslabel(tsl);
800                         needattr = B_TRUE;
801                 }
802                 WAKE_SCTP(sctp);
803                 sce.sctpAssocState = sctp_snmp_state(sctp);
804                 sce.sctpAssocInStreams = sctp->sctp_num_istr;
805                 sce.sctpAssocOutStreams = sctp->sctp_num_ostr;
806                 sce.sctpAssocMaxRetr = sctp->sctp_pa_max_rxt;
807                 /* A 0 here indicates that no primary process is known */
808                 sce.sctpAssocPrimProcess = 0;
809                 sce.sctpAssocT1expired = sctp->sctp_T1expire;
810                 sce.sctpAssocT2expired = sctp->sctp_T2expire;
811                 sce.sctpAssocRtxChunks = sctp->sctp_T3expire;
812                 sce.sctpAssocStartTime = sctp->sctp_assoc_start_time;
813                 sce.sctpConnEntryInfo.ce_sendq = sctp->sctp_unacked +
814                     sctp->sctp_unsent;
815                 sce.sctpConnEntryInfo.ce_recvq = sctp->sctp_rxqueued;
816                 sce.sctpConnEntryInfo.ce_swnd = sctp->sctp_frwnd;
817                 sce.sctpConnEntryInfo.ce_rwnd = sctp->sctp_rwnd;
818                 sce.sctpConnEntryInfo.ce_mss = sctp->sctp_mss;
819                 (void) snmp_append_data2(mp_conn_data, &mp_conn_tail,
820                     (char *)&sce, sizeof (sce));
821                 /* my data */
822                 (void) snmp_append_data2(mp_pidnode_data, &mp_pidnode_tail,
823                     (char *)&sce, sizeof (sce));

825                 cph = conn_get_pid_list(connp);

827                 (void) snmp_append_data2(mp_pidnode_data, &mp_pidnode_tail,
828                     (char *)cph, cph->cph_tot_size);

830                 kmem_free(cph, cph->cph_tot_size);
831                 /* end of my data */
832 #endif /* ! codereview */
833                 mlp.tme_connidx = idx++;
834                 if (needattr)
835                         (void) snmp_append_data2(mp_attr_ctl->b_cont,
836                             &mp_attr_tail, (char *)&mlp, sizeof (mlp));
837 next_sctp:
838                 sctp_prev = sctp;
```



```
839                 mutex_enter(&sctps->sctps_g_lock);
840                 sctp = list_next(&sctps->sctps_g_list, sctp);
841         }
842         mutex_exit(&sctps->sctps_g_lock);
843         if (sctp_prev != NULL)
844                 SCTP_REFRELE(sctp_prev);

846         sctp_sum_mib(sctps, &sctp_mib);

848         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
849         optp->level = MIB2_SCTP;
850         optp->name = 0;
851         (void) snmp_append_data(mpdata, (char *)&sctp_mib, sizeof (sctp_mib));
852         optp->len = msgdsize(mpdata);
853         qreply(q, mpctl);

855         /* table of connections... */
856         optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
857             sizeof (struct T_optmgmt_ack)];
858         optp->level = MIB2_SCTP;
859         optp->name = MIB2_SCTP_CONN;
860         optp->len = msgdsize(mp_conn_data);
861         qreply(q, mp_conn_ctl);

863         /* table of EXPER_XPORT_PROC_INFO */
864         optp = (struct opthdr *)&mp_pidnode_ctl->b_rptr[
865             sizeof (struct T_optmgmt_ack)];
866         optp->level = MIB2_SCTP;
867         optp->name = EXPER_XPORT_PROC_INFO;
868         optp->len = msgdsize(mp_pidnode_data);
869         qreply(q, mp_pidnode_ctl);
870 #endif /* ! codereview */

872         /* assoc local address table */
873         optp = (struct opthdr *)&mp_local_ctl->b_rptr[
874             sizeof (struct T_optmgmt_ack)];
875         optp->level = MIB2_SCTP;
876         optp->name = MIB2_SCTP_CONN_LOCAL;
877         optp->len = msgdsize(mp_local_data);
878         qreply(q, mp_local_ctl);

880         /* assoc remote address table */
881         optp = (struct opthdr *)&mp_rem_ctl->b_rptr[
882             sizeof (struct T_optmgmt_ack)];
883         optp->level = MIB2_SCTP;
884         optp->name = MIB2_SCTP_CONN_REMOTE;
885         optp->len = msgdsize(mp_rem_data);
886         qreply(q, mp_rem_ctl);

888         /* table of MLP attributes */
889         optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
890             sizeof (struct T_optmgmt_ack)];
891         optp->level = MIB2_SCTP;
892         optp->name = EXPER_XPORT_MLP;
893         optp->len = msgdsize(mp_attr_data);
894         if (optp->len == 0)
895                 freemsg(mp_attr_ctl);
896         else
897                 qreply(q, mp_attr_ctl);

899         return (mp_ret);
900 }

902 /* Translate SCTP state to MIB2 SCTP state. */
903 static int
904 sctp_snmp_state(sctp_t *sctp)
```

```
{
        if (sctp == NULL)
                return (0);

        switch (sctp->sctp_state) {
        case SCTPS_IDLE:
        case SCTPS_BOUND:
                return (MIB2_SCTP_closed);
        case SCTPS_LISTEN:
                return (MIB2_SCTP_listen);
        case SCTPS_COOKIE_WAIT:
                return (MIB2_SCTP_cookieWait);
        case SCTPS_COOKIE_ECHOED:
                return (MIB2_SCTP_cookieEchoed);
        case SCTPS_ESTABLISHED:
                return (MIB2_SCTP_established);
        case SCTPS_SHUTDOWN_PENDING:
                return (MIB2_SCTP_shutdownPending);
        case SCTPS_SHUTDOWN_SENT:
                return (MIB2_SCTP_shutdownSent);
        case SCTPS_SHUTDOWN_RECEIVED:
                return (MIB2_SCTP_shutdownReceived);
        case SCTPS_SHUTDOWN_ACK_SENT:
                return (MIB2_SCTP_shutdownAckSent);
        default:
                return (0);
        }
}

/*
 * To sum up all MIB2 stats for a sctp_stack_t from all per CPU stats.  The
 * caller should initialize the target mib2_sctp_t properly as this function
 * just adds up all the per CPU stats.
 */
static void
sctp_sum_mib(sctp_stack_t *sctps, mib2_sctp_t *sctp_mib)
{
        int i;
        int cnt;

        /* Static componets of mib2_sctp_t. */
        SET_MIB(sctp_mib->sctpRtoAlgorithm, MIB2_SCTP_RTOALGO_VANJ);
        SET_MIB(sctp_mib->sctpRtoMin, sctps->sctps_rto_ming);
        SET_MIB(sctp_mib->sctpRtoMax, sctps->sctps_rto_maxg);
        SET_MIB(sctp_mib->sctpRtoInitial, sctps->sctps_rto_initialg);
        SET_MIB(sctp_mib->sctpMaxAssocs, -1);
        SET_MIB(sctp_mib->sctpValCookieLife, sctps->sctps_cookie_life);
        SET_MIB(sctp_mib->sctpMaxInitRetr, sctps->sctps_max_init_retr);

        /* fixed length structure for IPv4 and IPv6 counters */
        SET_MIB(sctp_mib->sctpEntrySize, sizeof (mib2_sctpConnEntry_t));
        SET_MIB(sctp_mib->sctpLocalEntrySize,
            sizeof (mib2_sctpConnLocalEntry_t));
        SET_MIB(sctp_mib->sctpRemoteEntrySize,
            sizeof (mib2_sctpConnRemoteEntry_t));

        /*
         * sctps_sc_cnt may change in the middle of the loop.  It is better
         * to get its value first.
         */
        cnt = sctps->sctps_sc_cnt;
        for (i = 0; i < cnt; i++)
                sctp_add_mib(&sctps->sctps_sc[i]->sctp_sc_mib, sctp_mib);
}

static void
```



```
sctp_add_mib(mib2_sctp_t *from, mib2_sctp_t *to)
{
        to->sctpActiveEstab += from->sctpActiveEstab;
        to->sctpPassiveEstab += from->sctpPassiveEstab;
        to->sctpAborted += from->sctpAborted;
        to->sctpShutdowns += from->sctpShutdowns;
        to->sctpOutOfBlue += from->sctpOutOfBlue;
        to->sctpChecksumError += from->sctpChecksumError;
        to->sctpOutCtrlChunks += from->sctpOutCtrlChunks;
        to->sctpOutOrderChunks += from->sctpOutOrderChunks;
        to->sctpOutUnorderChunks += from->sctpOutUnorderChunks;
        to->sctpRetransChunks += from->sctpRetransChunks;
        to->sctpOutAck += from->sctpOutAck;
        to->sctpOutAckDelayed += from->sctpOutAckDelayed;
        to->sctpOutWinUpdate += from->sctpOutWinUpdate;
        to->sctpOutFastRetrans += from->sctpOutFastRetrans;
        to->sctpOutWinProbe += from->sctpOutWinProbe;
        to->sctpInCtrlChunks += from->sctpInCtrlChunks;
        to->sctpInOrderChunks += from->sctpInOrderChunks;
        to->sctpInUnorderChunks += from->sctpInUnorderChunks;
        to->sctpInAck += from->sctpInAck;
        to->sctpInDupAck += from->sctpInDupAck;
        to->sctpInAckUnsent += from->sctpInAckUnsent;
        to->sctpFragUsrMsgs += from->sctpFragUsrMsgs;
        to->sctpReasmUsrMsgs += from->sctpReasmUsrMsgs;
        to->sctpOutSCTPPkts += from->sctpOutSCTPPkts;
        to->sctpInSCTPPkts += from->sctpInSCTPPkts;
        to->sctpInInvalidCookie += from->sctpInInvalidCookie;
        to->sctpTimRetrans += from->sctpTimRetrans;
        to->sctpTimRetransDrop += from->sctpTimRetransDrop;
        to->sctpTimHeartBeatProbe += from->sctpTimHeartBeatProbe;
        to->sctpTimHeartBeatDrop += from->sctpTimHeartBeatDrop;
        to->sctpListenDrop += from->sctpListenDrop;
        to->sctpInClosed += from->sctpInClosed;
}
```

```
**********************************************************
   54247 Sun Aug  9 12:47:58 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 conn_pid_node_list_hdr_t *
    sctp_get_sock_pid_list(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 */
```

```
128         sosctp_setsockopt,              /* sop_setsockopt */
129         sosctp_ioctl,                   /* sop_ioctl    */
130         so_poll,                        /* sop_poll     */
131         sosctp_close,                   /* sop_close    */
132 };

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 };

151 /* All the upcalls expect the upper handle to be sonode. */
152 sock_upcalls_t sosctp_sock_upcalls = {
153         so_newconn,
154         so_connected,
155         so_disconnected,
156         so_opctl,
157         so_queue_msg,
158         so_set_prop,
159         so_txq_full,
160         NULL,                   /* su_signal_oob */
161 };

162 /* All the upcalls expect the upper handle to be sctp_sonode/sctp_soassoc. */
163 sock_upcalls_t sosctp_assoc_upcalls = {
164         sctp_assoc_newconn,
165         sctp_assoc_connected,
166         sctp_assoc_disconnected,
167         sctp_assoc_disconnecting,
168         sctp_assoc_recv,
169         sctp_assoc_properties,
170         sctp_assoc_xmitted,
171         NULL,                   /* su_recv_space */
172         NULL,                   /* su_signal_oob */
173         NULL,                   /* su_set_error */
174         NULL,                   /* su_closed */
175         sctp_get_sock_pid_list
176 #endif /* ! codereview */
177 };

178 /* ARGSUSED */
179 static int
180 sosctp_init(struct sonode *so, struct sonode *pso, struct cred *cr, int flags)
181 {
182         struct sctp_sonode *ss;
183         struct sctp_sonode *pss;
184         sctp_sockbuf_limits_t sbl;
185         int err;

186         ss = SOTOSSO(so);

187         if (pso != NULL) {
```



```
188                 /*
189                  * Passive open, just inherit settings from parent. We should
190                  * not end up here for SOCK_SEQPACKET type sockets, since no
191                  * new sonode is created in that case.
192                  */
193                 ASSERT(so->so_type == SOCK_STREAM);
194                 pss = SOTOSSO(pso);

195                 mutex_enter(&pso->so_lock);
196                 so->so_state |= (SS_ISBOUND | SS_ISCONNECTED |
197                     (pso->so_state & SS_ASYNC));
198                 sosctp_so_inherit(pss, ss);
199                 so->so_proto_props = pso->so_proto_props;
200                 so->so_mode = pso->so_mode;
201                 mutex_exit(&pso->so_lock);

202                 return (0);
203         }

204         if ((err = secpolicy_basic_net_access(cr)) != 0)
205                 return (err);

206         if (so->so_type == SOCK_STREAM) {
207                 so->so_proto_handle = (sock_lower_handle_t)sctp_create(so,
208                     NULL, so->so_family, so->so_type, SCTP_CAN_BLOCK,
209                     &sosctp_sock_upcalls, &sbl, cr);
210                 so->so_mode = SM_CONNREQUIRED;
211         } else {
212                 ASSERT(so->so_type == SOCK_SEQPACKET);
213                 so->so_proto_handle = (sock_lower_handle_t)sctp_create(ss,
214                     NULL, so->so_family, so->so_type, SCTP_CAN_BLOCK,
215                     &sosctp_assoc_upcalls, &sbl, cr);
216         }

217         if (so->so_proto_handle == NULL)
218                 return (ENOMEM);

219         so->so_rcvbuf = sbl.sbl_rxbuf;
220         so->so_rcvlowat = sbl.sbl_rxlowat;
221         so->so_sndbuf = sbl.sbl_txbuf;
222         so->so_sndlowat = sbl.sbl_txlowat;

223         return (0);
224 }

225 /*
226  * Accept incoming connection.
227  */
228 /*ARGSUSED*/
229 static int
230 sosctp_accept(struct sonode *so, int fflag, struct cred *cr,
231     struct sonode **nsop)
232 {
233         int error = 0;

234         if ((so->so_state & SS_ACCEPTCONN) == 0)
235                 return (EINVAL);

236         error = so_acceptq_dequeue(so, (fflag & (FNONBLOCK|FNDELAY)), nsop);

237         return (error);
238 }

239 /*
240  * Bind local endpoint.
241  */
```

```
/*ARGSUSED*/
static int
sosctp_bind(struct sonode *so, struct sockaddr *name, socklen_t namelen,
    int flags, struct cred *cr)
{
        int error;

        if (!(flags & _SOBIND_LOCK_HELD)) {
                mutex_enter(&so->so_lock);
                so_lock_single(so);     /* Set SOLOCKED */
        } else {
                ASSERT(MUTEX_HELD(&so->so_lock));
        }

        /*
         * X/Open requires this check
         */
        if (so->so_state & SS_CANTSENDMORE) {
                error = EINVAL;
                goto done;
        }


        /*
         * Protocol module does address family checks.
         */
        mutex_exit(&so->so_lock);

        error = sctp_bind((struct sctp_s *)so->so_proto_handle, name, namelen);

        mutex_enter(&so->so_lock);
        if (error == 0) {
                so->so_state |= SS_ISBOUND;
        } else {
                eprintsoline(so, error);
        }
done:
        if (!(flags & _SOBIND_LOCK_HELD)) {
                so_unlock_single(so, SOLOCKED);
                mutex_exit(&so->so_lock);
        } else {
                /* If the caller held the lock don't release it here */
                ASSERT(MUTEX_HELD(&so->so_lock));
                ASSERT(so->so_flag & SOLOCKED);
        }

        return (error);
}

/*
 * Turn socket into a listen socket.
 */
/* ARGSUSED */
static int
sosctp_listen(struct sonode *so, int backlog, struct cred *cr)
{
        int error = 0;

        mutex_enter(&so->so_lock);
        so_lock_single(so);

        /*
         * If this socket is trying to do connect, or if it has
         * been connected, disallow.
         */
        if (so->so_state & (SS_ISCONNECTING | SS_ISCONNECTED |
```



```
            SS_ISDISCONNECTING | SS_CANTRCVMORE | SS_CANTSENDMORE)) {
                error = EINVAL;
                eprintsoline(so, error);
                goto done;
        }

        if (backlog < 0) {
                backlog = 0;
        }

        /*
         * If listen() is only called to change backlog, we don't
         * need to notify protocol module.
         */
        if (so->so_state & SS_ACCEPTCONN) {
                so->so_backlog = backlog;
                goto done;
        }

        mutex_exit(&so->so_lock);
        error = sctp_listen((struct sctp_s *)so->so_proto_handle);
        mutex_enter(&so->so_lock);
        if (error == 0) {
                so->so_state |= (SS_ACCEPTCONN|SS_ISBOUND);
                so->so_backlog = backlog;
        } else {
                eprintsoline(so, error);
        }
done:
        so_unlock_single(so, SOLOCKED);
        mutex_exit(&so->so_lock);

        return (error);
}

/*
 * Active open.
 */
/*ARGSUSED*/
static int
sosctp_connect(struct sonode *so, struct sockaddr *name,
    socklen_t namelen, int fflag, int flags, struct cred *cr)
{
        int error = 0;
        pid_t pid = curproc->p_pid;

        ASSERT(so->so_type == SOCK_STREAM);

        mutex_enter(&so->so_lock);
        so_lock_single(so);

        /*
         * Can't connect() after listen(), or if the socket is already
         * connected.
         */
        if (so->so_state & (SS_ACCEPTCONN|SS_ISCONNECTED|SS_ISCONNECTING)) {
                if (so->so_state & SS_ISCONNECTED) {
                        error = EISCONN;
                } else if (so->so_state & SS_ISCONNECTING) {
                        error = EALREADY;
                } else {
                        error = EOPNOTSUPP;
                }
                eprintsoline(so, error);
                goto done;
        }
```

```
393         /*
394          * Check for failure of an earlier call
395          */
396         if (so->so_error != 0) {
397                 error = sogeterr(so, B_TRUE);
398                 eprintsoline(so, error);
399                 goto done;
400         }

402         /*
403          * Connection is closing, or closed, don't allow reconnect.
404          * TCP allows this to proceed, but the socket remains unwriteable.
405          * BSD returns EINVAL.
406          */
407         if (so->so_state & (SS_ISDISCONNECTING|SS_CANTRCVMORE|
408             SS_CANTSENDMORE)) {
409                 error = EINVAL;
410                 eprintsoline(so, error);
411                 goto done;
412         }

414         if (name == NULL || namelen == 0) {
415                 mutex_exit(&so->so_lock);
416                 error = EINVAL;
417                 eprintsoline(so, error);
418                 goto done;
419         }

421         soisconnecting(so);
422         mutex_exit(&so->so_lock);

424         error = sctp_connect((struct sctp_s *)so->so_proto_handle,
425             name, namelen, cr, pid);

427         mutex_enter(&so->so_lock);
428         if (error == 0) {
429                 /*
430                  * Allow other threads to access the socket
431                  */
432                 error = sowaitconnected(so, fflag, 0);
433         }
434 done:
435         so_unlock_single(so, SOLOCKED);
436         mutex_exit(&so->so_lock);
437         return (error);
438 }

440 /*
441  * Active open for 1-N sockets, create a new association and
442  * call connect on that.
443  * If there parent hasn't been bound yet (this is the first association),
444  * make it so.
445  */
446 static int
447 sosctp_seq_connect(struct sonode *so, struct sockaddr *name,
448     socklen_t namelen, int fflag, int flags, struct cred *cr)
449 {
450         struct sctp_soassoc *ssa;
451         struct sctp_sonode *ss;
452         int error;

454         ASSERT(so->so_type == SOCK_SEQPACKET);

456         mutex_enter(&so->so_lock);
457         so_lock_single(so);
```



```
459         if (name == NULL || namelen == 0) {
460                 error = EINVAL;
461                 eprintsoline(so, error);
462                 goto done;
463         }

465         ss = SOTOSSO(so);

467         error = sosctp_assoc_createconn(ss, name, namelen, NULL, 0, fflag,
468             cr, &ssa);
469         if (error != 0) {
470                 if ((error == EHOSTUNREACH) && (flags & _SOCONNECT_XPG4_2)) {
471                         error = ENETUNREACH;
472                 }
473         }
474         if (ssa != NULL) {
475                 SSA_REFRELE(ss, ssa);
476         }

478 done:
479         so_unlock_single(so, SOLOCKED);
480         mutex_exit(&so->so_lock);
481         return (error);
482 }

484 /*
485  * Receive data.
486  */
487 /* ARGSUSED */
488 static int
489 sosctp_recvmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
490     struct cred *cr)
491 {
492         struct sctp_sonode *ss = SOTOSSO(so);
493         struct sctp_soassoc *ssa = NULL;
494         int flags, error = 0;
495         struct T_unitdata_ind *tind;
496         ssize_t orig_resid = uiop->uio_resid;
497         int len, count, readcnt = 0;
498         socklen_t controllen, namelen;
499         void *opt;
500         mblk_t *mp;
501         rval_t  rval;

503         controllen = msg->msg_controllen;
504         namelen = msg->msg_namelen;
505         flags = msg->msg_flags;
506         msg->msg_flags = 0;
507         msg->msg_controllen = 0;
508         msg->msg_namelen = 0;

510         if (so->so_type == SOCK_STREAM) {
511                 if (!(so->so_state & (SS_ISCONNECTED|SS_ISCONNECTING|
512                     SS_CANTRCVMORE))) {
513                         return (ENOTCONN);
514                 }
515         } else {
516                 /* NOTE: Will come here from vop_read() as well */
517                 /* For 1-N socket, recv() cannot be used. */
518                 if (namelen == 0)
519                         return (EOPNOTSUPP);
520                 /*
521                  * If there are no associations, and no new connections are
522                  * coming in, there's not going to be new messages coming
523                  * in either.
```

```
524                  */
525                 if (so->so_rcv_q_head == NULL && so->so_rcv_head == NULL &&
526                     ss->ss_assoccnt == 0 && !(so->so_state & SS_ACCEPTCONN)) {
527                         return (ENOTCONN);
528                 }
529         }
531         /*
532          * out-of-band data not supported.
533          */
534         if (flags & MSG_OOB) {
535                 return (EOPNOTSUPP);
536         }
538         /*
539          * flag possibilities:
540          *
541          * MSG_PEEK     Don't consume data
542          * MSG_WAITALL  Wait for full quantity of data (ignored if MSG_PEEK)
543          * MSG_DONTWAIT Non-blocking (same as FNDELAY | FNONBLOCK)
544          *
545          * MSG_WAITALL can return less than the full buffer if either
546          *
547          * 1. we would block and we are non-blocking
548          * 2. a full message cannot be delivered
549          *
550          * Given that we always get a full message from proto below,
551          * MSG_WAITALL is not meaningful.
552          */
554         mutex_enter(&so->so_lock);
556         /*
557          * Allow just one reader at a time.
558          */
559         error = so_lock_read_intr(so,
560             uiop->uio_fmode | ((flags & MSG_DONTWAIT) ? FNONBLOCK : 0));
561         if (error) {
562                 mutex_exit(&so->so_lock);
563                 return (error);
564         }
565         mutex_exit(&so->so_lock);
566 again:
567         error = so_dequeue_msg(so, &mp, uiop, &rval, flags | MSG_DUPCTRL);
568         if (mp != NULL) {
569                 if (so->so_type == SOCK_SEQPACKET) {
570                         ssa = *(struct sctp_soassoc **)DB_BASE(mp);
571                 }
573                 tind = (struct T_unitdata_ind *)mp->b_rptr;
575                 len = tind->SRC_length;
577                 if (namelen > 0 && len > 0) {
579                         opt = sogetoff(mp, tind->SRC_offset, len, 1);
581                         ASSERT(opt != NULL);
583                         msg->msg_name = kmem_alloc(len, KM_SLEEP);
584                         msg->msg_namelen = len;
586                         bcopy(opt, msg->msg_name, len);
587                 }
589                 len = tind->OPT_length;
```



```
590                 if (controllen == 0) {
591                         if (len > 0) {
592                                 msg->msg_flags |= MSG_CTRUNC;
593                         }
594                 } else if (len > 0) {
595                         opt = sogetoff(mp, tind->OPT_offset, len,
596                             __TPI_ALIGN_SIZE);
598                         ASSERT(opt != NULL);
599                         sosctp_pack_cmsg(opt, msg, len);
600                 }
602                 if (mp->b_flag & SCTP_NOTIFICATION) {
603                         msg->msg_flags |= MSG_NOTIFICATION;
604                 }
606                 if (!(mp->b_flag & SCTP_PARTIAL_DATA) &&
607                     !(rval.r_val1 & MOREDATA)) {
608                         msg->msg_flags |= MSG_EOR;
609                 }
610                 freemsg(mp);
611         }
612 done:
613         if (!(flags & MSG_PEEK))
614                 readcnt = orig_resid - uiop->uio_resid;
615         /*
616          * Determine if we need to update SCTP about the buffer
617          * space.  For performance reason, we cannot update SCTP
618          * every time a message is read.  The socket buffer low
619          * watermark is used as the threshold.
620          */
621         if (ssa == NULL) {
622                 mutex_enter(&so->so_lock);
623                 count = so->so_rcvbuf - so->so_rcv_queued;
625                 ASSERT(so->so_rcv_q_head != NULL ||
626                     so->so_rcv_head != NULL ||
627                     so->so_rcv_queued == 0);
629                 so_unlock_read(so);
631                 /*
632                  * so_dequeue_msg() sets r_val2 to true if flow control was
633                  * cleared and we need to update SCTP.  so_flowctrld was
634                  * cleared in so_dequeue_msg() via so_check_flow_control().
635                  */
636                 if (rval.r_val2) {
637                         mutex_exit(&so->so_lock);
638                         sctp_recvd((struct sctp_s *)so->so_proto_handle, count);
639                 } else {
640                         mutex_exit(&so->so_lock);
641                 }
642         } else {
643                 /*
644                  * Each association keeps track of how much data it has
645                  * queued; we need to update the value here. Note that this
646                  * is slightly different from SOCK_STREAM type sockets, which
647                  * does not need to update the byte count, as it is already
648                  * done in so_dequeue_msg().
649                  */
650                 mutex_enter(&so->so_lock);
651                 ssa->ssa_rcv_queued -= readcnt;
652                 count = so->so_rcvbuf - ssa->ssa_rcv_queued;
654                 so_unlock_read(so);
```

```
656                 if (readcnt > 0 && ssa->ssa_flowctrld &&
657                     ssa->ssa_rcv_queued < so->so_rcvlowat) {
658                         /*
659                          * Need to clear ssa_flowctrld, different from 1-1
660                          * style.
661                          */
662                         ssa->ssa_flowctrld = B_FALSE;
663                         mutex_exit(&so->so_lock);
664                         sctp_recvd(ssa->ssa_conn, count);
665                         mutex_enter(&so->so_lock);
666                 }

668                 /*
669                  * MOREDATA flag is set if all data could not be copied
670                  */
671                 if (!(flags & MSG_PEEK) && !(rval.r_val1 & MOREDATA)) {
672                         SSA_REFRELE(ss, ssa);
673                 }
674                 mutex_exit(&so->so_lock);
675         }

677         return (error);
678 }

680 int
681 sosctp_uiomove(mblk_t *hdr_mp, ssize_t count, ssize_t blk_size, int wroff,
682     struct uio *uiop, int flags)
683 {
684         ssize_t size;
685         int error;
686         mblk_t *mp;
687         dblk_t *dp;

689         if (blk_size == INFPSZ)
690                 blk_size = count;

692         /*
693          * Loop until we have all data copied into mblk's.
694          */
695         while (count > 0) {
696                 size = MIN(count, blk_size);

698                 /*
699                  * As a message can be splitted up and sent in different
700                  * packets, each mblk will have the extra space before
701                  * data to accommodate what SCTP wants to put in there.
702                  */
703                 while ((mp = allocb(size + wroff, BPRI_MED)) == NULL) {
704                         if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
705                             (flags & MSG_DONTWAIT)) {
706                                 return (EAGAIN);
707                         }
708                         if ((error = strwaitbuf(size + wroff, BPRI_MED))) {
709                                 return (error);
710                         }
711                 }

713                 dp = mp->b_datap;
714                 dp->db_cpid = curproc->p_pid;
715                 ASSERT(wroff <= dp->db_lim - mp->b_wptr);
716                 mp->b_rptr += wroff;
717                 error = uiomove(mp->b_rptr, size, UIO_WRITE, uiop);
718                 if (error != 0) {
719                         freeb(mp);
720                         return (error);
721                 }
```



```
722                 mp->b_wptr = mp->b_rptr + size;
723                 count -= size;
724                 hdr_mp->b_cont = mp;
725                 hdr_mp = mp;
726         }
727         return (0);
728 }

730 /*
731  * Send message.
732  */
733 static int
734 sosctp_sendmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
735     struct cred *cr)
736 {
737         mblk_t *mctl;
738         struct cmsghdr *cmsg;
739         struct sctp_sndrcvinfo *sinfo;
740         int optlen, flags, fflag;
741         ssize_t count, msglen;
742         int error;

744         ASSERT(so->so_type == SOCK_STREAM);

746         flags = msg->msg_flags;
747         if (flags & MSG_OOB) {
748                 /*
749                  * No out-of-band data support.
750                  */
751                 return (EOPNOTSUPP);
752         }

754         if (msg->msg_controllen != 0) {
755                 optlen = msg->msg_controllen;
756                 cmsg = sosctp_find_cmsg(msg->msg_control, optlen, SCTP_SNDRCV);
757                 if (cmsg != NULL) {
758                         if (cmsg->cmsg_len <
759                             (sizeof (*sinfo) + sizeof (*cmsg))) {
760                                 eprintsoline(so, EINVAL);
761                                 return (EINVAL);
762                         }
763                         sinfo = (struct sctp_sndrcvinfo *)(cmsg + 1);

765                         /* Both flags should not be set together. */
766                         if ((sinfo->sinfo_flags & MSG_EOF) &&
767                             (sinfo->sinfo_flags & MSG_ABORT)) {
768                                 eprintsoline(so, EINVAL);
769                                 return (EINVAL);
770                         }

772                         /* Initiate a graceful shutdown. */
773                         if (sinfo->sinfo_flags & MSG_EOF) {
774                                 /* Can't include data in MSG_EOF message. */
775                                 if (uiop->uio_resid != 0) {
776                                         eprintsoline(so, EINVAL);
777                                         return (EINVAL);
778                                 }

780                                 /*
781                                  * This is the same sequence as done in
782                                  * shutdown(SHUT_WR).
783                                  */
784                                 mutex_enter(&so->so_lock);
785                                 so_lock_single(so);
786                                 socantsendmore(so);
787                                 cv_broadcast(&so->so_snd_cv);
```

```
788                                 so->so_state |= SS_ISDISCONNECTING;
789                                 mutex_exit(&so->so_lock);
791                                 pollwakeup(&so->so_poll_list, POLLOUT);
792                                 sctp_recvd((struct sctp_s *)so->so_proto_handle,
793                                     so->so_rcvbuf);
794                                 error = sctp_disconnect(
795                                     (struct sctp_s *)so->so_proto_handle);
797                                 mutex_enter(&so->so_lock);
798                                 so_unlock_single(so, SOLOCKED);
799                                 mutex_exit(&so->so_lock);
800                                 return (error);
801                         }
802                 }
803         } else {
804                 optlen = 0;
805         }
807         mutex_enter(&so->so_lock);
808         for (;;) {
809                 if (so->so_state & SS_CANTSENDMORE) {
810                         mutex_exit(&so->so_lock);
811                         return (EPIPE);
812                 }
814                 if (so->so_error != 0) {
815                         error = sogeterr(so, B_TRUE);
816                         mutex_exit(&so->so_lock);
817                         return (error);
818                 }
820                 if (!so->so_snd_qfull)
821                         break;
823                 if (so->so_state & SS_CLOSING) {
824                         mutex_exit(&so->so_lock);
825                         return (EINTR);
826                 }
827                 /*
828                  * Xmit window full in a blocking socket.
829                  */
830                 if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
831                     (flags & MSG_DONTWAIT)) {
832                         mutex_exit(&so->so_lock);
833                         return (EAGAIN);
834                 } else {
835                         /*
836                          * Wait for space to become available and try again.
837                          */
838                         error = cv_wait_sig(&so->so_snd_cv, &so->so_lock);
839                         if (!error) { /* signal */
840                                 mutex_exit(&so->so_lock);
841                                 return (EINTR);
842                         }
843                 }
844         }
845         msglen = count = uiop->uio_resid;
847         /* Don't allow sending a message larger than the send buffer size. */
848         /* XXX Transport module need to enforce this */
849         if (msglen > so->so_sndbuf) {
850                 mutex_exit(&so->so_lock);
851                 return (EMSGSIZE);
852         }
```



```
854         /*
855          * Allow piggybacking data on handshake messages (SS_ISCONNECTING).
856          */
857         if (!(so->so_state & (SS_ISCONNECTING | SS_ISCONNECTED))) {
858                 /*
859                  * We need to check here for listener so that the
860                  * same error will be returned as with a TCP socket.
861                  * In this case, sosctp_connect() returns EOPNOTSUPP
862                  * while a TCP socket returns ENOTCONN instead.  Catch it
863                  * here to have the same behavior as a TCP socket.
864                  *
865                  * We also need to make sure that the peer address is
866                  * provided before we attempt to do the connect.
867                  */
868                 if ((so->so_state & SS_ACCEPTCONN) ||
869                     msg->msg_name == NULL) {
870                         mutex_exit(&so->so_lock);
871                         error = ENOTCONN;
872                         goto error_nofree;
873                 }
874                 mutex_exit(&so->so_lock);
875                 fflag = uiop->uio_fmode;
876                 if (flags & MSG_DONTWAIT) {
877                         fflag |= FNDELAY;
878                 }
879                 error = sosctp_connect(so, msg->msg_name, msg->msg_namelen,
880                     fflag, (so->so_version == SOV_XPG4_2) * _SOCONNECT_XPG4_2,
881                     cr);
882                 if (error) {
883                         /*
884                          * Check for non-fatal errors, socket connected
885                          * while the lock had been lifted.
886                          */
887                         if (error != EISCONN && error != EALREADY) {
888                                 goto error_nofree;
889                         }
890                         error = 0;
891                 }
892         } else {
893                 mutex_exit(&so->so_lock);
894         }
896         mctl = sctp_alloc_hdr(msg->msg_name, msg->msg_namelen,
897             msg->msg_control, optlen, SCTP_CAN_BLOCK);
898         if (mctl == NULL) {
899                 error = EINTR;
900                 goto error_nofree;
901         }
903         /* Copy in the message. */
904         if ((error = sosctp_uiomove(mctl, count, so->so_proto_props.sopp_maxblk,
905             so->so_proto_props.sopp_wroff, uiop, flags)) != 0) {
906                 goto error_ret;
907         }
908         error = sctp_sendmsg((struct sctp_s *)so->so_proto_handle, mctl, 0);
909         if (error == 0)
910                 return (0);
912 error_ret:
913         freemsg(mctl);
914 error_nofree:
915         mutex_enter(&so->so_lock);
916         if ((error == EPIPE) && (so->so_state & SS_CANTSENDMORE)) {
917                 /*
918                  * We received shutdown between the time lock was
919                  * lifted and call to sctp_sendmsg().
```

```
920                  */
921                 mutex_exit(&so->so_lock);
922                 return (EPIPE);
923         }
924         mutex_exit(&so->so_lock);
925         return (error);
926 }

928 /*
929  * Send message on 1-N socket. Connects automatically if there is
930  * no association.
931  */
932 static int
933 sosctp_seq_sendmsg(struct sonode *so, struct nmsghdr *msg, struct uio *uiop,
934     struct cred *cr)
935 {
936         struct sctp_sonode *ss;
937         struct sctp_soassoc *ssa;
938         struct cmsghdr *cmsg;
939         struct sctp_sndrcvinfo *sinfo;
940         int aid = 0;
941         mblk_t *mctl;
942         int namelen, optlen, flags;
943         ssize_t count, msglen;
944         int error;
945         uint16_t s_flags = 0;

947         ASSERT(so->so_type == SOCK_SEQPACKET);

949         /*
950          * There shouldn't be problems with alignment, as the memory for
951          * msg_control was alloced with kmem_alloc.
952          */
953         cmsg = sosctp_find_cmsg(msg->msg_control, msg->msg_controllen,
954             SCTP_SNDRCV);
955         if (cmsg != NULL) {
956                 if (cmsg->cmsg_len < (sizeof (*sinfo) + sizeof (*cmsg))) {
957                         eprintsoline(so, EINVAL);
958                         return (EINVAL);
959                 }
960                 sinfo = (struct sctp_sndrcvinfo *)(cmsg + 1);
961                 s_flags = sinfo->sinfo_flags;
962                 aid = sinfo->sinfo_assoc_id;
963         }

965         ss = SOTOSSO(so);
966         namelen = msg->msg_namelen;

968         if (msg->msg_controllen > 0) {
969                 optlen = msg->msg_controllen;
970         } else {
971                 optlen = 0;
972         }

974         mutex_enter(&so->so_lock);

976         /*
977          * If there is no association id, connect to address specified
978          * in msg_name.  Otherwise look up the association using the id.
979          */
980         if (aid == 0) {
981                 /*
982                  * Connect and shutdown cannot be done together, so check for
983                  * MSG_EOF.
984                  */
985                 if (msg->msg_name == NULL || namelen == 0 ||
```



```
986                     (s_flags & MSG_EOF)) {
987                         error = EINVAL;
988                         eprintsoline(so, error);
989                         goto done;
990                 }
991                 flags = uiop->uio_fmode;
992                 if (msg->msg_flags & MSG_DONTWAIT) {
993                         flags |= FNDELAY;
994                 }
995                 so_lock_single(so);
996                 error = sosctp_assoc_createconn(ss, msg->msg_name, namelen,
997                     msg->msg_control, optlen, flags, cr, &ssa);
998                 if (error) {
999                         if ((so->so_version == SOV_XPG4_2) &&
1000                            (error == EHOSTUNREACH)) {
1001                                error = ENETUNREACH;
1002                        }
1003                        if (ssa == NULL) {
1004                                /*
1005                                 * Fatal error during connect(). Bail out.
1006                                 * If ssa exists, it means that the handshake
1007                                 * is in progress.
1008                                 */
1009                                eprintsoline(so, error);
1010                                so_unlock_single(so, SOLOCKED);
1011                                goto done;
1012                        }
1013                        /*
1014                         * All the errors are non-fatal ones, don't return
1015                         * e.g. EINPROGRESS from sendmsg().
1016                         */
1017                        error = 0;
1018                }
1019                so_unlock_single(so, SOLOCKED);
1020        } else {
1021                if ((error = sosctp_assoc(ss, aid, &ssa)) != 0) {
1022                        eprintsoline(so, error);
1023                        goto done;
1024                }
1025        }

1027        /*
1028         * Now we have an association.
1029         */
1030        flags = msg->msg_flags;

1032        /*
1033         * MSG_EOF initiates graceful shutdown.
1034         */
1035        if (s_flags & MSG_EOF) {
1036                if (uiop->uio_resid) {
1037                        /*
1038                         * Can't include data in MSG_EOF message.
1039                         */
1040                        error = EINVAL;
1041                } else {
1042                        mutex_exit(&so->so_lock);
1043                        ssa->ssa_state |= SS_ISDISCONNECTING;
1044                        sctp_recvd(ssa->ssa_conn, so->so_rcvbuf);
1045                        error = sctp_disconnect(ssa->ssa_conn);
1046                        mutex_enter(&so->so_lock);
1047                }
1048                goto refrele;
1049        }

1051        for (;;) {
```

```
1052                 if (ssa->ssa_state & SS_CANTSENDMORE) {
1053                         SSA_REFRELE(ss, ssa);
1054                         mutex_exit(&so->so_lock);
1055                         return (EPIPE);
1056                 }
1057                 if (ssa->ssa_error != 0) {
1058                         error = ssa->ssa_error;
1059                         ssa->ssa_error = 0;
1060                         goto refrele;
1061                 }

1063                 if (!ssa->ssa_snd_qfull)
1064                         break;

1066                 if (so->so_state & SS_CLOSING) {
1067                         error = EINTR;
1068                         goto refrele;
1069                 }
1070                 if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
1071                     (flags & MSG_DONTWAIT)) {
1072                         error = EAGAIN;
1073                         goto refrele;
1074                 } else {
1075                         /*
1076                          * Wait for space to become available and try again.
1077                          */
1078                         error = cv_wait_sig(&so->so_snd_cv, &so->so_lock);
1079                         if (!error) { /* signal */
1080                                 error = EINTR;
1081                                 goto refrele;
1082                         }
1083                 }
1084         }

1086         msglen = count = uiop->uio_resid;

1088         /* Don't allow sending a message larger than the send buffer size. */
1089         if (msglen > so->so_sndbuf) {
1090                 error = EMSGSIZE;
1091                 goto refrele;
1092         }

1094         /*
1095          * Update TX buffer usage here so that we can lift the socket lock.
1096          */
1097         mutex_exit(&so->so_lock);

1099         mctl = sctp_alloc_hdr(msg->msg_name, namelen, msg->msg_control,
1100             optlen, SCTP_CAN_BLOCK);
1101         if (mctl == NULL) {
1102                 error = EINTR;
1103                 goto lock_rele;
1104         }

1106         /* Copy in the message. */
1107         if ((error = sosctp_uiomove(mctl, count, ssa->ssa_wrsize,
1108             ssa->ssa_wroff, uiop, flags)) != 0) {
1109                 goto lock_rele;
1110         }
1111         error = sctp_sendmsg((struct sctp_s *)ssa->ssa_conn, mctl, 0);
1112 lock_rele:
1113         mutex_enter(&so->so_lock);
1114         if (error != 0) {
1115                 freemsg(mctl);
1116                 if ((error == EPIPE) && (ssa->ssa_state & SS_CANTSENDMORE)) {
1117                         /*
```



```
1118                          * We received shutdown between the time lock was
1119                          * lifted and call to sctp_sendmsg().
1120                          */
1121                         SSA_REFRELE(ss, ssa);
1122                         mutex_exit(&so->so_lock);
1123                         return (EPIPE);
1124                 }
1125         }

1127 refrele:
1128         SSA_REFRELE(ss, ssa);
1129 done:
1130         mutex_exit(&so->so_lock);
1131         return (error);
1132 }

1134 /*
1135  * Get address of remote node.
1136  */
1137 /* ARGSUSED */
1138 static int
1139 sosctp_getpeername(struct sonode *so, struct sockaddr *addr, socklen_t *addrlen,
1140     boolean_t accept, struct cred *cr)
1141 {
1142         return (sctp_getpeername((struct sctp_s *)so->so_proto_handle, addr,
1143             addrlen));
1144 }

1146 /*
1147  * Get local address.
1148  */
1149 /* ARGSUSED */
1150 static int
1151 sosctp_getsockname(struct sonode *so, struct sockaddr *addr, socklen_t *addrlen,
1152     struct cred *cr)
1153 {
1154         return (sctp_getsockname((struct sctp_s *)so->so_proto_handle, addr,
1155             addrlen));
1156 }

1158 /*
1159  * Called from shutdown().
1160  */
1161 /* ARGSUSED */
1162 static int
1163 sosctp_shutdown(struct sonode *so, int how, struct cred *cr)
1164 {
1165         uint_t state_change;
1166         int wakesig = 0;
1167         int error = 0;

1169         mutex_enter(&so->so_lock);
1170         /*
1171          * Record the current state and then perform any state changes.
1172          * Then use the difference between the old and new states to
1173          * determine which needs to be done.
1174          */
1175         state_change = so->so_state;

1177         switch (how) {
1178         case SHUT_RD:
1179                 socantrcvmore(so);
1180                 break;
1181         case SHUT_WR:
1182                 socantsendmore(so);
1183                 break;
```

```
1184         case SHUT_RDWR:
1185                 socantsendmore(so);
1186                 socantrcvmore(so);
1187                 break;
1188         default:
1189                 mutex_exit(&so->so_lock);
1190                 return (EINVAL);
1191         }

1193         state_change = so->so_state & ~state_change;

1195         if (state_change & SS_CANTRCVMORE) {
1196                 if (so->so_rcv_q_head == NULL) {
1197                         cv_signal(&so->so_rcv_cv);
1198                 }
1199                 wakesig = POLLIN|POLLRDNORM;

1201                 socket_sendsig(so, SOCKETSIG_READ);
1202         }
1203         if (state_change & SS_CANTSENDMORE) {
1204                 cv_broadcast(&so->so_snd_cv);
1205                 wakesig |= POLLOUT;

1207                 so->so_state |= SS_ISDISCONNECTING;
1208         }
1209         mutex_exit(&so->so_lock);

1211         pollwakeup(&so->so_poll_list, wakesig);

1213         if (state_change & SS_CANTSENDMORE) {
1214                 sctp_recvd((struct sctp_s *)so->so_proto_handle, so->so_rcvbuf);
1215                 error = sctp_disconnect((struct sctp_s *)so->so_proto_handle);
1216         }

1218         /*
1219          * HACK: sctp_disconnect() may return EWOULDBLOCK.  But this error is
1220          * not documented in standard socket API.  Catch it here.
1221          */
1222         if (error == EWOULDBLOCK)
1223                 error = 0;
1224         return (error);
1225 }

1227 /*
1228  * Get socket options.
1229  */
1230 /*ARGSUSED5*/
1231 static int
1232 sosctp_getsockopt(struct sonode *so, int level, int option_name,
1233     void *optval, socklen_t *optlenp, int flags, struct cred *cr)
1234 {
1235         socklen_t maxlen = *optlenp;
1236         socklen_t len;
1237         socklen_t optlen;
1238         uint8_t buffer[4];
1239         void    *optbuf = &buffer;
1240         int     error = 0;

1242         if (level == SOL_SOCKET) {
1243                 switch (option_name) {
1244                 /* Not supported options */
1245                 case SO_SNDTIMEO:
1246                 case SO_RCVTIMEO:
1247                 case SO_EXCLBIND:
1248                         eprintsoline(so, ENOPROTOOPT);
1249                         return (ENOPROTOOPT);
```



```
1250                 default:
1251                         error = socket_getopt_common(so, level, option_name,
1252                             optval, optlenp, flags);
1253                         if (error >= 0)
1254                                 return (error);
1255                         /* Pass the request to the protocol */
1256                         break;
1257                 }
1258         }

1260         if (level == IPPROTO_SCTP) {
1261                 /*
1262                  * Should go through ioctl().
1263                  */
1264                 return (EINVAL);
1265         }

1267         if (maxlen > sizeof (buffer)) {
1268                 optbuf = kmem_alloc(maxlen, KM_SLEEP);
1269         }
1270         optlen = maxlen;

1272         /*
1273          * If the resulting optlen is greater than the provided maxlen, then
1274          * we sliently trucate.
1275          */
1276         error = sctp_get_opt((struct sctp_s *)so->so_proto_handle, level,
1277             option_name, optbuf, &optlen);

1279         if (error != 0) {
1280                 eprintsoline(so, error);
1281                 goto free;
1282         }
1283         len = optlen;

1285 copyout:

1287         len = MIN(len, maxlen);
1288         bcopy(optbuf, optval, len);
1289         *optlenp = optlen;
1290 free:
1291         if (optbuf != &buffer) {
1292                 kmem_free(optbuf, maxlen);
1293         }

1295         return (error);
1296 }

1298 /*
1299  * Set socket options
1300  */
1301 /* ARGSUSED */
1302 static int
1303 sosctp_setsockopt(struct sonode *so, int level, int option_name,
1304     const void *optval, t_uscalar_t optlen, struct cred *cr)
1305 {
1306         struct sctp_sonode *ss = SOTOSSO(so);
1307         struct sctp_soassoc *ssa = NULL;
1308         sctp_assoc_t id;
1309         int error, rc;
1310         void *conn = NULL;

1312         mutex_enter(&so->so_lock);

1314         /*
1315          * For some SCTP level options, one can select the association this
```

```
1316          * applies to.
1317          */
1318         if (so->so_type == SOCK_STREAM) {
1319                 conn = so->so_proto_handle;
1320         } else {
1321                 /*
1322                  * SOCK_SEQPACKET only
1323                  */
1324                 id = 0;
1325                 if (level == IPPROTO_SCTP) {
1326                         switch (option_name) {
1327                         case SCTP_RTOINFO:
1328                         case SCTP_ASSOCINFO:
1329                         case SCTP_SET_PEER_PRIMARY_ADDR:
1330                         case SCTP_PRIMARY_ADDR:
1331                         case SCTP_PEER_ADDR_PARAMS:
1332                                 /*
1333                                  * Association ID is the first element
1334                                  * params struct
1335                                  */
1336                                 if (optlen < sizeof (sctp_assoc_t)) {
1337                                         error = EINVAL;
1338                                         eprintsoline(so, error);
1339                                         goto done;
1340                                 }
1341                                 id = *(sctp_assoc_t *)optval;
1342                                 break;
1343                         case SCTP_DEFAULT_SEND_PARAM:
1344                                 if (optlen != sizeof (struct sctp_sndrcvinfo)) {
1345                                         error = EINVAL;
1346                                         eprintsoline(so, error);
1347                                         goto done;
1348                                 }
1349                                 id = ((struct sctp_sndrcvinfo *)
1350                                     optval)->sinfo_assoc_id;
1351                                 break;
1352                         case SCTP_INITMSG:
1353                                 /*
1354                                  * Only applies to future associations
1355                                  */
1356                                 conn = so->so_proto_handle;
1357                                 break;
1358                         default:
1359                                 break;
1360                         }
1361                 } else if (level == SOL_SOCKET) {
1362                         if (option_name == SO_LINGER) {
1363                                 error = EOPNOTSUPP;
1364                                 eprintsoline(so, error);
1365                                 goto done;
1366                         }
1367                         /*
1368                          * These 2 options are applied to all associations.
1369                          * The other socket level options are only applied
1370                          * to the socket (not associations).
1371                          */
1372                         if ((option_name != SO_RCVBUF) &&
1373                             (option_name != SO_SNDBUF)) {
1374                                 conn = so->so_proto_handle;
1375                         }
1376                 } else {
1377                         conn = NULL;
1378                 }

1380                 /*
1381                  * If association ID was specified, do op on that assoc.
```



```
1382                  * Otherwise set the default setting of a socket.
1383                  */
1384                 if (id != 0) {
1385                         if ((error = sosctp_assoc(ss, id, &ssa)) != 0) {
1386                                 eprintsoline(so, error);
1387                                 goto done;
1388                         }
1389                         conn = ssa->ssa_conn;
1390                 }
1391         }
1392         dprint(2, ("sosctp_setsockopt %p (%d) - conn %p %d %d id:%d\n",
1393             (void *)ss, so->so_type, (void *)conn, level, option_name, id));

1395         ASSERT(ssa == NULL || (ssa != NULL && conn != NULL));
1396         if (conn != NULL) {
1397                 mutex_exit(&so->so_lock);
1398                 error = sctp_set_opt((struct sctp_s *)conn, level, option_name,
1399                     optval, optlen);
1400                 mutex_enter(&so->so_lock);
1401                 if (ssa != NULL)
1402                         SSA_REFRELE(ss, ssa);
1403         } else {
1404                 /*
1405                  * 1-N socket, and we have to apply the operation to ALL
1406                  * associations. Like with anything of this sort, the
1407                  * problem is what to do if the operation fails.
1408                  * Just try to apply the setting to everyone, but store
1409                  * error number if someone returns such.  And since we are
1410                  * looping through all possible aids, some of them can be
1411                  * invalid.  We just ignore this kind (sosctp_assoc()) of
1412                  * errors.
1413                  */
1414                 sctp_assoc_t aid;

1416                 mutex_exit(&so->so_lock);
1417                 error = sctp_set_opt((struct sctp_s *)so->so_proto_handle,
1418                     level, option_name, optval, optlen);
1419                 mutex_enter(&so->so_lock);
1420                 for (aid = 1; aid < ss->ss_maxassoc; aid++) {
1421                         if (sosctp_assoc(ss, aid, &ssa) != 0)
1422                                 continue;
1423                         mutex_exit(&so->so_lock);
1424                         rc = sctp_set_opt((struct sctp_s *)ssa->ssa_conn, level,
1425                             option_name, optval, optlen);
1426                         mutex_enter(&so->so_lock);
1427                         SSA_REFRELE(ss, ssa);
1428                         if (error == 0) {
1429                                 error = rc;
1430                         }
1431                 }
1432         }
1433 done:
1434         mutex_exit(&so->so_lock);
1435         return (error);
1436 }

1438 /*ARGSUSED*/
1439 static int
1440 sosctp_ioctl(struct sonode *so, int cmd, intptr_t arg, int mode,
1441     struct cred *cr, int32_t *rvalp)
1442 {
1443         struct sctp_sonode      *ss;
1444         int32_t                 value;
1445         int                     error;
1446         int                     intval;
1447         pid_t                   pid;
```

```
1448        struct sctp_soassoc     *ssa;
1449        void                    *conn;
1450        void                    *buf;
1451        STRUCT_DECL(sctpopt, opt);
1452        uint32_t                optlen;
1453        int                     buflen;

1455        ss = SOTOSSO(so);

1457        /* handle socket specific ioctls */
1458        switch (cmd) {
1459        case FIONBIO:
1460                if (so_copyin((void *)arg, &value, sizeof (int32_t),
1461                    (mode & (int)FKIOCTL))) {
1462                        return (EFAULT);
1463                }
1464                mutex_enter(&so->so_lock);
1465                if (value) {
1466                        so->so_state |= SS_NDELAY;
1467                } else {
1468                        so->so_state &= ~SS_NDELAY;
1469                }
1470                mutex_exit(&so->so_lock);
1471                return (0);

1473        case FIOASYNC:
1474                if (so_copyin((void *)arg, &value, sizeof (int32_t),
1475                    (mode & (int)FKIOCTL))) {
1476                        return (EFAULT);
1477                }
1478                mutex_enter(&so->so_lock);

1480                if (value) {
1481                        /* Turn on SIGIO */
1482                        so->so_state |= SS_ASYNC;
1483                } else {
1484                        /* Turn off SIGIO */
1485                        so->so_state &= ~SS_ASYNC;
1486                }
1487                mutex_exit(&so->so_lock);
1488                return (0);

1490        case SIOCSPGRP:
1491        case FIOSETOWN:
1492                if (so_copyin((void *)arg, &pid, sizeof (pid_t),
1493                    (mode & (int)FKIOCTL))) {
1494                        return (EFAULT);
1495                }
1496                mutex_enter(&so->so_lock);

1498                error = (pid != so->so_pgrp) ? socket_chgpgrp(so, pid) : 0;
1499                mutex_exit(&so->so_lock);
1500                return (error);

1502        case SIOCGPGRP:
1503        case FIOGETOWN:
1504                if (so_copyout(&so->so_pgrp, (void *)arg,
1505                    sizeof (pid_t), (mode & (int)FKIOCTL)))
1506                        return (EFAULT);
1507                return (0);

1509        case FIONREAD:
1510                /* XXX: Cannot be used unless standard buffer is used */
1511                /*
1512                 * Return number of bytes of data in all data messages
1513                 * in queue in "arg".
```



```
1514                 * For stream socket, amount of available data.
1515                 * For sock_dgram, # of available bytes + addresses.
1516                 */
1517                intval = (so->so_state & SS_ACCEPTCONN) ? 0 :
1518                    MIN(so->so_rcv_queued, INT_MAX);
1519                if (so_copyout(&intval, (void *)arg, sizeof (intval),
1520                    (mode & (int)FKIOCTL)))
1521                        return (EFAULT);
1522                return (0);
1523        case SIOCATMARK:
1524                /*
1525                 * No support for urgent data.
1526                 */
1527                intval = 0;

1529                if (so_copyout(&intval, (void *)arg, sizeof (int),
1530                    (mode & (int)FKIOCTL)))
1531                        return (EFAULT);
1532                return (0);
1533        case _I_GETPEERCRED: {
1534                int error = 0;

1536                if ((mode & FKIOCTL) == 0)
1537                        return (EINVAL);

1539                mutex_enter(&so->so_lock);
1540                if ((so->so_mode & SM_CONNREQUIRED) == 0) {
1541                        error = ENOTSUP;
1542                } else if ((so->so_state & SS_ISCONNECTED) == 0) {
1543                        error = ENOTCONN;
1544                } else if (so->so_peercred != NULL) {
1545                        k_peercred_t *kp = (k_peercred_t *)arg;
1546                        kp->pc_cr = so->so_peercred;
1547                        kp->pc_cpid = so->so_cpid;
1548                        crhold(so->so_peercred);
1549                } else {
1550                        error = EINVAL;
1551                }
1552                mutex_exit(&so->so_lock);
1553                return (error);
1554        }
1555        case SIOCSCTPGOPT:
1556                STRUCT_INIT(opt, mode);

1558                if (so_copyin((void *)arg, STRUCT_BUF(opt), STRUCT_SIZE(opt),
1559                    (mode & (int)FKIOCTL))) {
1560                        return (EFAULT);
1561                }
1562                if ((optlen = STRUCT_FGET(opt, sopt_len)) > SO_MAXARGSIZE)
1563                        return (EINVAL);

1565                /*
1566                 * Find the correct sctp_t based on whether it is 1-N socket
1567                 * or not.
1568                 */
1569                intval = STRUCT_FGET(opt, sopt_aid);
1570                mutex_enter(&so->so_lock);
1571                if ((so->so_type == SOCK_SEQPACKET) && intval) {
1572                        if ((error = sosctp_assoc(ss, intval, &ssa)) != 0) {
1573                                mutex_exit(&so->so_lock);
1574                                return (error);
1575                        }
1576                        conn = ssa->ssa_conn;
1577                        ASSERT(conn != NULL);
1578                } else {
1579                        conn = so->so_proto_handle;
```

```
1580                        ssa = NULL;
1581                }
1582                mutex_exit(&so->so_lock);

1584                /* Copyin the option buffer and then call sctp_get_opt(). */
1585                buflen = optlen;
1586                /* Let's allocate a buffer enough to hold an int */
1587                if (buflen < sizeof (uint32_t))
1588                        buflen = sizeof (uint32_t);
1589                buf = kmem_alloc(buflen, KM_SLEEP);
1590                if (so_copyin(STRUCT_FGETP(opt, sopt_val), buf, optlen,
1591                    (mode & (int)FKIOCTL))) {
1592                        if (ssa != NULL) {
1593                                mutex_enter(&so->so_lock);
1594                                SSA_REFRELE(ss, ssa);
1595                                mutex_exit(&so->so_lock);
1596                        }
1597                        kmem_free(buf, buflen);
1598                        return (EFAULT);
1599                }
1600                /* The option level has to be IPPROTO_SCTP */
1601                error = sctp_get_opt((struct sctp_s *)conn, IPPROTO_SCTP,
1602                    STRUCT_FGET(opt, sopt_name), buf, &optlen);
1603                if (ssa != NULL) {
1604                        mutex_enter(&so->so_lock);
1605                        SSA_REFRELE(ss, ssa);
1606                        mutex_exit(&so->so_lock);
1607                }
1608                optlen = MIN(buflen, optlen);
1609                /* No error, copyout the result with the correct buf len. */
1610                if (error == 0) {
1611                        STRUCT_FSET(opt, sopt_len, optlen);
1612                        if (so_copyout(STRUCT_BUF(opt), (void *)arg,
1613                            STRUCT_SIZE(opt), (mode & (int)FKIOCTL))) {
1614                                error = EFAULT;
1615                        } else if (so_copyout(buf, STRUCT_FGETP(opt, sopt_val),
1616                            optlen, (mode & (int)FKIOCTL))) {
1617                                error = EFAULT;
1618                        }
1619                }
1620                kmem_free(buf, buflen);
1621                return (error);

1623        case SIOCSCTPSOPT:
1624                STRUCT_INIT(opt, mode);

1626                if (so_copyin((void *)arg, STRUCT_BUF(opt), STRUCT_SIZE(opt),
1627                    (mode & (int)FKIOCTL))) {
1628                        return (EFAULT);
1629                }
1630                if ((optlen = STRUCT_FGET(opt, sopt_len)) > SO_MAXARGSIZE)
1631                        return (EINVAL);

1633                /*
1634                 * Find the correct sctp_t based on whether it is 1-N socket
1635                 * or not.
1636                 */
1637                intval = STRUCT_FGET(opt, sopt_aid);
1638                mutex_enter(&so->so_lock);
1639                if (intval != 0) {
1640                        if ((error = sosctp_assoc(ss, intval, &ssa)) != 0) {
1641                                mutex_exit(&so->so_lock);
1642                                return (error);
1643                        }
1644                        conn = ssa->ssa_conn;
1645                        ASSERT(conn != NULL);
```



```
1646                } else {
1647                        conn = so->so_proto_handle;
1648                        ssa = NULL;
1649                }
1650                mutex_exit(&so->so_lock);

1652                /* Copyin the option buffer and then call sctp_set_opt(). */
1653                buf = kmem_alloc(optlen, KM_SLEEP);
1654                if (so_copyin(STRUCT_FGETP(opt, sopt_val), buf, optlen,
1655                    (mode & (int)FKIOCTL))) {
1656                        if (ssa != NULL) {
1657                                mutex_enter(&so->so_lock);
1658                                SSA_REFRELE(ss, ssa);
1659                                mutex_exit(&so->so_lock);
1660                        }
1661                        kmem_free(buf, intval);
1662                        return (EFAULT);
1663                }
1664                /* The option level has to be IPPROTO_SCTP */
1665                error = sctp_set_opt((struct sctp_s *)conn, IPPROTO_SCTP,
1666                    STRUCT_FGET(opt, sopt_name), buf, optlen);
1667                if (ssa) {
1668                        mutex_enter(&so->so_lock);
1669                        SSA_REFRELE(ss, ssa);
1670                        mutex_exit(&so->so_lock);
1671                }
1672                kmem_free(buf, optlen);
1673                return (error);

1675        case SIOCSCTPPEELOFF: {
1676                struct sonode *nso;
1677                struct sctp_uc_swap us;
1678                int nfd;
1679                struct file *nfp;
1680                struct vnode *nvp = NULL;
1681                struct sockparams *sp;

1683                dprint(2, ("sctppeeloff %p\n", (void *)ss));

1685                if (so->so_type != SOCK_SEQPACKET) {
1686                        return (EOPNOTSUPP);
1687                }
1688                if (so_copyin((void *)arg, &intval, sizeof (intval),
1689                    (mode & (int)FKIOCTL))) {
1690                        return (EFAULT);
1691                }
1692                if (intval == 0) {
1693                        return (EINVAL);
1694                }

1696                /*
1697                 * Find sockparams. This is different from parent's entry,
1698                 * as the socket type is different.
1699                 */
1700                error = solookup(so->so_family, SOCK_STREAM, so->so_protocol,
1701                    &sp);
1702                if (error != 0)
1703                        return (error);

1705                /*
1706                 * Allocate the user fd.
1707                 */
1708                if ((nfd = ufalloc(0)) == -1) {
1709                        eprintsoline(so, EMFILE);
1710                        SOCKPARAMS_DEC_REF(sp);
1711                        return (EMFILE);
```

```
1712                }

1714                /*
1715                 * Copy the fd out.
1716                 */
1717                if (so_copyout(&nfd, (void *)arg, sizeof (nfd),
1718                    (mode & (int)FKIOCTL))) {
1719                        error = EFAULT;
1720                        goto err;
1721                }
1722                mutex_enter(&so->so_lock);

1724                /*
1725                 * Don't use sosctp_assoc() in order to peel off disconnected
1726                 * associations.
1727                 */
1728                ssa = ((uint32_t)intval >= ss->ss_maxassoc) ? NULL :
1729                    ss->ss_assocs[intval].ssi_assoc;
1730                if (ssa == NULL) {
1731                        mutex_exit(&so->so_lock);
1732                        error = EINVAL;
1733                        goto err;
1734                }
1735                SSA_REFHOLD(ssa);

1737                nso = socksctp_create(sp, so->so_family, SOCK_STREAM,
1738                    so->so_protocol, so->so_version, SOCKET_NOSLEEP,
1739                    &error, cr);
1740                if (nso == NULL) {
1741                        SSA_REFRELE(ss, ssa);
1742                        mutex_exit(&so->so_lock);
1743                        goto err;
1744                }
1745                nvp = SOTOV(nso);
1746                so_lock_single(so);
1747                mutex_exit(&so->so_lock);

1749                /* cannot fail, only inheriting properties */
1750                (void) sosctp_init(nso, so, CRED(), 0);

1752                /*
1753                 * We have a single ref on the new socket. This is normally
1754                 * handled by socket_{create,newconn}, but since they are not
1755                 * used we have to do it here.
1756                 */
1757                nso->so_count = 1;

1759                us.sus_handle = nso;
1760                us.sus_upcalls = &sosctp_sock_upcalls;

1762                /*
1763                 * Upcalls to new socket are blocked for the duration of
1764                 * downcall.
1765                 */
1766                mutex_enter(&nso->so_lock);

1768                error = sctp_set_opt((struct sctp_s *)ssa->ssa_conn,
1769                    IPPROTO_SCTP, SCTP_UC_SWAP, &us, sizeof (us));
1770                if (error) {
1771                        goto peelerr;
1772                }
1773                error = falloc(nvp, FWRITE|FREAD, &nfp, NULL);
1774                if (error) {
1775                        goto peelerr;
1776                }
```



```
1778                /*
1779                 * fill in the entries that falloc reserved
1780                 */
1781                nfp->f_vnode = nvp;
1782                mutex_exit(&nfp->f_tlock);
1783                setf(nfd, nfp);

1785                /* add curproc to the pid list associated with that file */
1786                if (nfp->f_vnode != NULL)
1787                        (void) VOP_IOCTL(nfp->f_vnode, F_FORKED,
1788                            (intptr_t)curproc, FKIOCTL, kcred, NULL, NULL);

1790 #endif /* ! codereview */
1791                mutex_enter(&so->so_lock);

1793                sosctp_assoc_move(ss, SOTOSSO(nso), ssa);

1795                mutex_exit(&nso->so_lock);

1797                ssa->ssa_conn = NULL;
1798                sosctp_assoc_free(ss, ssa);

1800                so_unlock_single(so, SOLOCKED);
1801                mutex_exit(&so->so_lock);

1803                return (0);

1805 err:
1806                SOCKPARAMS_DEC_REF(sp);
1807                setf(nfd, NULL);
1808                eprintsoline(so, error);
1809                return (error);

1811 peelerr:
1812                mutex_exit(&nso->so_lock);
1813                mutex_enter(&so->so_lock);
1814                ASSERT(nso->so_count == 1);
1815                nso->so_count = 0;
1816                so_unlock_single(so, SOLOCKED);
1817                SSA_REFRELE(ss, ssa);
1818                mutex_exit(&so->so_lock);

1820                setf(nfd, NULL);
1821                ASSERT(nvp->v_count == 1);
1822                socket_destroy(nso);
1823                eprintsoline(so, error);
1824                return (error);
1825        }
1826        default:
1827                return (EINVAL);
1828        }
1829 }

1831 /*ARGSUSED*/
1832 static int
1833 sosctp_close(struct sonode *so, int flag, struct cred *cr)
1834 {
1835        struct sctp_sonode *ss;
1836        struct sctp_sa_id *ssi;
1837        struct sctp_soassoc *ssa;
1838        int32_t i;

1840        ss = SOTOSSO(so);

1842        /*
1843         * Initiate connection shutdown.  Tell SCTP if there is any data
```

```
1844          * left unread.
1845          */
1846         sctp_recvd((struct sctp_s *)so->so_proto_handle,
1847             so->so_rcvbuf - so->so_rcv_queued);
1848         (void) sctp_disconnect((struct sctp_s *)so->so_proto_handle);

1850         /*
1851          * New associations can't come in, but old ones might get
1852          * closed in upcall. Protect against that by taking a reference
1853          * on the association.
1854          */
1855         mutex_enter(&so->so_lock);
1856         ssi = ss->ss_assocs;
1857         for (i = 0; i < ss->ss_maxassoc; i++, ssi++) {
1858                 if ((ssa = ssi->ssi_assoc) != NULL) {
1859                         SSA_REFHOLD(ssa);
1860                         sosctp_assoc_isdisconnected(ssa, 0);
1861                         mutex_exit(&so->so_lock);

1863                         sctp_recvd(ssa->ssa_conn, so->so_rcvbuf -
1864                             ssa->ssa_rcv_queued);
1865                         (void) sctp_disconnect(ssa->ssa_conn);

1867                         mutex_enter(&so->so_lock);
1868                         SSA_REFRELE(ss, ssa);
1869                 }
1870         }
1871         mutex_exit(&so->so_lock);

1873         return (0);
1874 }

1876 /*
1877  * Closes incoming connections which were never accepted, frees
1878  * resources.
1879  */
1880 /* ARGSUSED */
1881 void
1882 sosctp_fini(struct sonode *so, struct cred *cr)
1883 {
1884         struct sctp_sonode *ss;
1885         struct sctp_sa_id *ssi;
1886         struct sctp_soassoc *ssa;
1887         int32_t i;

1889         ss = SOTOSSO(so);

1891         ASSERT(so->so_ops == &sosctp_sonodeops ||
1892             so->so_ops == &sosctp_seq_sonodeops);

1894         /* We are the sole owner of so now */
1895         mutex_enter(&so->so_lock);

1897         /* Free all pending connections */
1898         so_acceptq_flush(so, B_TRUE);

1900         ssi = ss->ss_assocs;
1901         for (i = 0; i < ss->ss_maxassoc; i++, ssi++) {
1902                 if ((ssa = ssi->ssi_assoc) != NULL) {
1903                         SSA_REFHOLD(ssa);
1904                         mutex_exit(&so->so_lock);

1906                         sctp_close((struct sctp_s *)ssa->ssa_conn);

1908                         mutex_enter(&so->so_lock);
1909                         ssa->ssa_conn = NULL;
```



```
1910                         sosctp_assoc_free(ss, ssa);
1911                 }
1912         }
1913         if (ss->ss_assocs != NULL) {
1914                 ASSERT(ss->ss_assoccnt == 0);
1915                 kmem_free(ss->ss_assocs,
1916                     ss->ss_maxassoc * sizeof (struct sctp_sa_id));
1917         }
1918         mutex_exit(&so->so_lock);

1920         if (so->so_proto_handle)
1921                 sctp_close((struct sctp_s *)so->so_proto_handle);
1922         so->so_proto_handle = NULL;

1924         /*
1925          * Note until sctp_close() is called, SCTP can still send up
1926          * messages, such as event notifications.  So we should flush
1927          * the recevie buffer after calling sctp_close().
1928          */
1929         mutex_enter(&so->so_lock);
1930         so_rcv_flush(so);
1931         mutex_exit(&so->so_lock);

1933         sonode_fini(so);
1934 }

1936 /*
1937  * Upcalls from SCTP
1938  */

1940 /*
1941  * This is the upcall function for 1-N (SOCK_SEQPACKET) socket when a new
1942  * association is created.  Note that the first argument (handle) is of type
1943  * sctp_sonode *, which is the one changed to a listener for new
1944  * associations.  All the other upcalls for 1-N socket take sctp_soassoc *
1945  * as handle.  The only exception is the su_properties upcall, which
1946  * can take both types as handle.
1947  */
1948 /* ARGSUSED */
1949 sock_upper_handle_t
1950 sctp_assoc_newconn(sock_upper_handle_t parenthandle,
1951     sock_lower_handle_t connind, sock_downcalls_t *dc,
1952     struct cred *peer_cred, pid_t peer_cpid, sock_upcalls_t **ucp)
1953 {
1954         struct sctp_sonode *lss = (struct sctp_sonode *)parenthandle;
1955         struct sonode *lso = &lss->ss_so;
1956         struct sctp_soassoc *ssa;
1957         sctp_assoc_t id;

1959         ASSERT(lss->ss_type == SOSCTP_SOCKET);
1960         ASSERT(lso->so_state & SS_ACCEPTCONN);
1961         ASSERT(lso->so_proto_handle != NULL); /* closed conn */
1962         ASSERT(lso->so_type == SOCK_SEQPACKET);

1964         mutex_enter(&lso->so_lock);

1966         if ((id = sosctp_aid_get(lss)) == -1) {
1967                 /*
1968                  * Array not large enough; increase size.
1969                  */
1970                 if (sosctp_aid_grow(lss, lss->ss_maxassoc, KM_NOSLEEP) < 0) {
1971                         mutex_exit(&lso->so_lock);
1972                         return (NULL);
1973                 }
1974                 id = sosctp_aid_get(lss);
1975                 ASSERT(id != -1);
```

```
1976         }

1978         /*
1979          * Create soassoc for this connection
1980          */
1981         ssa = sosctp_assoc_create(lss, KM_NOSLEEP);
1982         if (ssa == NULL) {
1983                 mutex_exit(&lso->so_lock);
1984                 return (NULL);
1985         }
1986         sosctp_aid_reserve(lss, id, 1);
1987         lss->ss_assocs[id].ssi_assoc = ssa;
1988         ++lss->ss_assoccnt;
1989         ssa->ssa_id = id;
1990         ssa->ssa_conn = (struct sctp_s *)connind;
1991         ssa->ssa_state = (SS_ISBOUND | SS_ISCONNECTED);
1992         ssa->ssa_wroff = lss->ss_wroff;
1993         ssa->ssa_wrsize = lss->ss_wrsize;

1995         mutex_exit(&lso->so_lock);

1997         *ucp = &sosctp_assoc_upcalls;

1999         return ((sock_upper_handle_t)ssa);
2000 }

2002 /* ARGSUSED */
2003 static void
2004 sctp_assoc_connected(sock_upper_handle_t handle, sock_connid_t id,
2005     struct cred *peer_cred, pid_t peer_cpid)
2006 {
2007         struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2008         struct sonode *so = &ssa->ssa_sonode->ss_so;

2010         ASSERT(so->so_type == SOCK_SEQPACKET);
2011         ASSERT(ssa->ssa_conn);

2013         mutex_enter(&so->so_lock);
2014         sosctp_assoc_isconnected(ssa);
2015         mutex_exit(&so->so_lock);
2016 }

2018 /* ARGSUSED */
2019 static int
2020 sctp_assoc_disconnected(sock_upper_handle_t handle, sock_connid_t id, int error)
2021 {
2022         struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2023         struct sonode *so = &ssa->ssa_sonode->ss_so;
2024         int ret;

2026         ASSERT(so->so_type == SOCK_SEQPACKET);
2027         ASSERT(ssa->ssa_conn != NULL);

2029         mutex_enter(&so->so_lock);
2030         sosctp_assoc_isdisconnected(ssa, error);
2031         if (ssa->ssa_refcnt == 1) {
2032                 ret = 1;
2033                 ssa->ssa_conn = NULL;
2034         } else {
2035                 ret = 0;
2036         }
2037         SSA_REFRELE(SOTOSSO(so), ssa);

2039         cv_broadcast(&so->so_snd_cv);

2041         mutex_exit(&so->so_lock);
```



```
2043         return (ret);
2044 }

2046 /* ARGSUSED */
2047 static void
2048 sctp_assoc_disconnecting(sock_upper_handle_t handle, sock_opctl_action_t action,
2049     uintptr_t arg)
2050 {
2051         struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2052         struct sonode *so = &ssa->ssa_sonode->ss_so;

2054         ASSERT(so->so_type == SOCK_SEQPACKET);
2055         ASSERT(ssa->ssa_conn != NULL);
2056         ASSERT(action == SOCK_OPCTL_SHUT_SEND);

2058         mutex_enter(&so->so_lock);
2059         sosctp_assoc_isdisconnecting(ssa);
2060         mutex_exit(&so->so_lock);
2061 }

2063 /* ARGSUSED */
2064 static ssize_t
2065 sctp_assoc_recv(sock_upper_handle_t handle, mblk_t *mp, size_t len, int flags,
2066     int *errorp, boolean_t *forcepush)
2067 {
2068         struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2069         struct sctp_sonode *ss = ssa->ssa_sonode;
2070         struct sonode *so = &ss->ss_so;
2071         struct T_unitdata_ind *tind;
2072         mblk_t *mp2;
2073         union sctp_notification *sn;
2074         struct sctp_sndrcvinfo *sinfo;
2075         ssize_t space_available;

2077         ASSERT(ssa->ssa_type == SOSCTP_ASSOC);
2078         ASSERT(so->so_type == SOCK_SEQPACKET);
2079         ASSERT(ssa->ssa_conn != NULL); /* closed conn */
2080         ASSERT(mp != NULL);

2082         ASSERT(errorp != NULL);
2083         *errorp = 0;

2085         /*
2086          * Should be getting T_unitdata_req's only.
2087          * Must have address as part of packet.
2088          */
2089         tind = (struct T_unitdata_ind *)mp->b_rptr;
2090         ASSERT((DB_TYPE(mp) == M_PROTO) &&
2091             (tind->PRIM_type == T_UNITDATA_IND));
2092         ASSERT(tind->SRC_length);

2094         mutex_enter(&so->so_lock);

2096         /*
2097          * For notify messages, need to fill in association id.
2098          * For data messages, sndrcvinfo could be in ancillary data.
2099          */
2100         if (mp->b_flag & SCTP_NOTIFICATION) {
2101                 mp2 = mp->b_cont;
2102                 sn = (union sctp_notification *)mp2->b_rptr;
2103                 switch (sn->sn_header.sn_type) {
2104                 case SCTP_ASSOC_CHANGE:
2105                         sn->sn_assoc_change.sac_assoc_id = ssa->ssa_id;
2106                         break;
2107                 case SCTP_PEER_ADDR_CHANGE:
```

```
2108                        sn->sn_paddr_change.spc_assoc_id = ssa->ssa_id;
2109                        break;
2110                case SCTP_REMOTE_ERROR:
2111                        sn->sn_remote_error.sre_assoc_id = ssa->ssa_id;
2112                        break;
2113                case SCTP_SEND_FAILED:
2114                        sn->sn_send_failed.ssf_assoc_id = ssa->ssa_id;
2115                        break;
2116                case SCTP_SHUTDOWN_EVENT:
2117                        sn->sn_shutdown_event.sse_assoc_id = ssa->ssa_id;
2118                        break;
2119                case SCTP_ADAPTATION_INDICATION:
2120                        sn->sn_adaptation_event.sai_assoc_id = ssa->ssa_id;
2121                        break;
2122                case SCTP_PARTIAL_DELIVERY_EVENT:
2123                        sn->sn_pdapi_event.pdapi_assoc_id = ssa->ssa_id;
2124                        break;
2125                default:
2126                        ASSERT(0);
2127                        break;
2128                }
2129        } else {
2130                if (tind->OPT_length > 0) {
2131                        struct cmsghdr  *cmsg;
2132                        char            *cend;
2133
2134                        cmsg = (struct cmsghdr *)
2135                            ((uchar_t *)mp->b_rptr + tind->OPT_offset);
2136                        cend = (char *)cmsg + tind->OPT_length;
2137                        for (;;) {
2138                                if ((char *)(cmsg + 1) > cend ||
2139                                    ((char *)cmsg + cmsg->cmsg_len) > cend) {
2140                                        break;
2141                                }
2142                                if ((cmsg->cmsg_level == IPPROTO_SCTP) &&
2143                                    (cmsg->cmsg_type == SCTP_SNDRCV)) {
2144                                        sinfo = (struct sctp_sndrcvinfo *)
2145                                            (cmsg + 1);
2146                                        sinfo->sinfo_assoc_id = ssa->ssa_id;
2147                                        break;
2148                                }
2149                                if (cmsg->cmsg_len > 0) {
2150                                        cmsg = (struct cmsghdr *)
2151                                            ((uchar_t *)cmsg + cmsg->cmsg_len);
2152                                } else {
2153                                        break;
2154                                }
2155                        }
2156                }
2157        }
2158
2159        /*
2160         * SCTP has reserved space in the header for storing a pointer.
2161         * Put the pointer to assocation there, and queue the data.
2162         */
2163        SSA_REFHOLD(ssa);
2164        ASSERT((mp->b_rptr - DB_BASE(mp)) >= sizeof (ssa));
2165        *(struct sctp_soassoc **)DB_BASE(mp) = ssa;
2166
2167        ssa->ssa_rcv_queued += len;
2168        space_available = so->so_rcvbuf - ssa->ssa_rcv_queued;
2169        if (space_available <= 0)
2170                ssa->ssa_flowctrld = B_TRUE;
2171
2172        so_enqueue_msg(so, mp, len);
```



```
2174        /* so_notify_data drops so_lock */
2175        so_notify_data(so, len);
2176
2177        return (space_available);
2178 }
2179
2180 static void
2181 sctp_assoc_xmitted(sock_upper_handle_t handle, boolean_t qfull)
2182 {
2183        struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2184        struct sctp_sonode *ss = ssa->ssa_sonode;
2185
2186        ASSERT(ssa->ssa_type == SOSCTP_ASSOC);
2187        ASSERT(ss->ss_so.so_type == SOCK_SEQPACKET);
2188        ASSERT(ssa->ssa_conn != NULL);
2189
2190        mutex_enter(&ss->ss_so.so_lock);
2191
2192        ssa->ssa_snd_qfull = qfull;
2193
2194        /*
2195         * Wake blocked writers.
2196         */
2197        cv_broadcast(&ss->ss_so.so_snd_cv);
2198
2199        mutex_exit(&ss->ss_so.so_lock);
2200 }
2201
2202 static void
2203 sctp_assoc_properties(sock_upper_handle_t handle,
2204     struct sock_proto_props *soppp)
2205 {
2206        struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2207        struct sonode *so;
2208
2209        if (ssa->ssa_type == SOSCTP_ASSOC) {
2210                so = &ssa->ssa_sonode->ss_so;
2211
2212                mutex_enter(&so->so_lock);
2213
2214                /* Per assoc_id properties. */
2215                if (soppp->sopp_flags & SOCKOPT_WROFF)
2216                        ssa->ssa_wroff = soppp->sopp_wroff;
2217                if (soppp->sopp_flags & SOCKOPT_MAXBLK)
2218                        ssa->ssa_wrsize = soppp->sopp_maxblk;
2219        } else {
2220                so = &((struct sctp_sonode *)handle)->ss_so;
2221                mutex_enter(&so->so_lock);
2222
2223                if (soppp->sopp_flags & SOCKOPT_WROFF)
2224                        so->so_proto_props.sopp_wroff = soppp->sopp_wroff;
2225                if (soppp->sopp_flags & SOCKOPT_MAXBLK)
2226                        so->so_proto_props.sopp_maxblk = soppp->sopp_maxblk;
2227                if (soppp->sopp_flags & SOCKOPT_RCVHIWAT) {
2228                        ssize_t lowat;
2229
2230                        so->so_rcvbuf = soppp->sopp_rxhiwat;
2231                        /*
2232                         * The low water mark should be adjusted properly
2233                         * if the high water mark is changed.  It should
2234                         * not be bigger than 1/4 of high water mark.
2235                         */
2236                        lowat = soppp->sopp_rxhiwat >> 2;
2237                        if (so->so_rcvlowat > lowat) {
2238                                /* Sanity check... */
2239                                if (lowat == 0)
```

```
2240                                        so->so_rcvlowat = soppp->sopp_rxhiwat;
2241                                } else
2242                                        so->so_rcvlowat = lowat;
2243                        }
2244                }
2245        }
2246        mutex_exit(&so->so_lock);
2247 }

2249 static conn_pid_node_list_hdr_t *
2250 sctp_get_sock_pid_list(sock_upper_handle_t handle)
2251 {
2252        struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2253        struct sonode *so;

2255        if (ssa->ssa_type == SOSCTP_ASSOC)
2256                so = &ssa->ssa_sonode->ss_so;
2257        else
2258                so = &((struct sctp_sonode *)handle)->ss_so;

2260        return (so_get_sock_pid_list((sock_upper_handle_t)so));
2261 #endif /* ! codereview */
2262 }
```

```
**********************************************************
   35594 Sun Aug  9 12:47:59 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         conn_pid_node_list_hdr_t        *cph;

137 #endif /* ! codereview */
138         /*
139          * make a copy of the original message
140          */
141         mp2ctl = copymsg(mpctl);

143         if (mpctl == NULL ||
144             (mpdata = mpctl->b_cont) == NULL ||
145             (mp_conn_ctl = copymsg(mpctl)) == NULL ||
146             (mp_attr_ctl = copymsg(mpctl)) == NULL ||
147             (mp_pidnode_ctl = copymsg(mpctl)) == NULL ||
148 #endif /* ! codereview */
149             (mp6_conn_ctl = copymsg(mpctl)) == NULL ||
150             (mp6_attr_ctl = copymsg(mpctl)) == NULL ||
151             (mp6_pidnode_ctl = copymsg(mpctl)) == NULL) {
 33             (mp6_attr_ctl = copymsg(mpctl)) == NULL) {
152                 freemsg(mp_conn_ctl);
153                 freemsg(mp_attr_ctl);
154                 freemsg(mp_pidnode_ctl);
155 #endif /* ! codereview */
156                 freemsg(mp6_conn_ctl);
157                 freemsg(mp6_attr_ctl);
158                 freemsg(mp6_pidnode_ctl);
159 #endif /* ! codereview */
160                 freemsg(mpctl);
161                 freemsg(mp2ctl);
162                 return (NULL);
163         }

165         ipst = connp->conn_netstack->netstack_ip;
166         tcps = connp->conn_netstack->netstack_tcp;

168         if (legacy_req) {
169                 tcp_mib_size = LEGACY_MIB_SIZE(&tcp_mib, mib2_tcp_t);
170                 tce_size = LEGACY_MIB_SIZE(&tce, mib2_tcpConnEntry_t);
171                 tce6_size = LEGACY_MIB_SIZE(&tce6, mib2_tcp6ConnEntry_t);
172         } else {
173                 tcp_mib_size = sizeof (mib2_tcp_t);
174                 tce_size = sizeof (mib2_tcpConnEntry_t);
175                 tce6_size = sizeof (mib2_tcp6ConnEntry_t);
176         }

178         bzero(&tcp_mib, sizeof (tcp_mib));

180         /* build table of connections -- need count in fixed part */
181         SET_MIB(tcp_mib.tcpRtoAlgorithm, 4);   /* vanj */
182         SET_MIB(tcp_mib.tcpRtoMin, tcps->tcps_rexmit_interval_min);
183         SET_MIB(tcp_mib.tcpRtoMax, tcps->tcps_rexmit_interval_max);
184         SET_MIB(tcp_mib.tcpMaxConn, -1);
185         SET_MIB(tcp_mib.tcpCurrEstab, 0);

187         ispriv =
188             secpolicy_ip_config((Q_TO_CONN(q))->conn_cred, B_TRUE) == 0;
189         zoneid = Q_TO_CONN(q)->conn_zoneid;

191         v4_conn_idx = v6_conn_idx = 0;
192         mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;
```



```
193         mp_pidnode_tail = mp6_pidnode_tail = NULL;
194 #endif /* ! codereview */

196         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
197                 ipst = tcps->tcps_netstack->netstack_ip;

199                 connfp = &ipst->ips_ipcl_globalhash_fanout[i];

201                 connp = NULL;

203                 while ((connp =
204                     ipcl_get_next_conn(connfp, connp, IPCL_TCPCONN)) != NULL) {
205                         tcp_t *tcp;
206                         boolean_t needattr;

208                         if (connp->conn_zoneid != zoneid)
209                                 continue;       /* not in this zone */

211                         tcp = connp->conn_tcp;
212                         TCPS_UPDATE_MIB(tcps, tcpHCInSegs, tcp->tcp_ibsegs);
213                         tcp->tcp_ibsegs = 0;
214                         TCPS_UPDATE_MIB(tcps, tcpHCOutSegs, tcp->tcp_obsegs);
215                         tcp->tcp_obsegs = 0;

217                         tce6.tcp6ConnState = tce.tcpConnState =
218                             tcp_snmp_state(tcp);
219                         if (tce.tcpConnState == MIB2_TCP_established ||
220                             tce.tcpConnState == MIB2_TCP_closeWait)
221                                 BUMP_MIB(&tcp_mib, tcpCurrEstab);

223                         needattr = B_FALSE;
224                         bzero(&mlp, sizeof (mlp));
225                         if (connp->conn_mlp_type != mlptSingle) {
226                                 if (connp->conn_mlp_type == mlptShared ||
227                                     connp->conn_mlp_type == mlptBoth)
228                                         mlp.tme_flags |= MIB2_TMEF_SHARED;
229                                 if (connp->conn_mlp_type == mlptPrivate ||
230                                     connp->conn_mlp_type == mlptBoth)
231                                         mlp.tme_flags |= MIB2_TMEF_PRIVATE;
232                                 needattr = B_TRUE;
233                         }
234                         if (connp->conn_anon_mlp) {
235                                 mlp.tme_flags |= MIB2_TMEF_ANONMLP;
236                                 needattr = B_TRUE;
237                         }
238                         switch (connp->conn_mac_mode) {
239                         case CONN_MAC_DEFAULT:
240                                 break;
241                         case CONN_MAC_AWARE:
242                                 mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
243                                 needattr = B_TRUE;
244                                 break;
245                         case CONN_MAC_IMPLICIT:
246                                 mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
247                                 needattr = B_TRUE;
248                                 break;
249                         }
250                         if (connp->conn_ixa->ixa_tsl != NULL) {
251                                 ts_label_t *tsl;

253                                 tsl = connp->conn_ixa->ixa_tsl;
254                                 mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
255                                 mlp.tme_doi = label2doi(tsl);
256                                 mlp.tme_label = *label2bslabel(tsl);
257                                 needattr = B_TRUE;
258                         }
```

```
260                         /* Create a message to report on IPv6 entries */
261                         if (connp->conn_ipversion == IPV6_VERSION) {
262                         tce6.tcp6ConnLocalAddress = connp->conn_laddr_v6;
263                         tce6.tcp6ConnRemAddress = connp->conn_faddr_v6;
264                         tce6.tcp6ConnLocalPort = ntohs(connp->conn_lport);
265                         tce6.tcp6ConnRemPort = ntohs(connp->conn_fport);
266                         if (connp->conn_ixa->ixa_flags & IXAF_SCOPEID_SET) {
267                                 tce6.tcp6ConnIfIndex =
268                                     connp->conn_ixa->ixa_scopeid;
269                         } else {
270                                 tce6.tcp6ConnIfIndex = connp->conn_bound_if;
271                         }
272                         /* Don't want just anybody seeing these... */
273                         if (ispriv) {
274                                 tce6.tcp6ConnEntryInfo.ce_snxt =
275                                     tcp->tcp_snxt;
276                                 tce6.tcp6ConnEntryInfo.ce_suna =
277                                     tcp->tcp_suna;
278                                 tce6.tcp6ConnEntryInfo.ce_rnxt =
279                                     tcp->tcp_rnxt;
280                                 tce6.tcp6ConnEntryInfo.ce_rack =
281                                     tcp->tcp_rack;
282                         } else {
283                                 /*
284                                  * Netstat, unfortunately, uses this to
285                                  * get send/receive queue sizes.  How to fix?
286                                  * Why not compute the difference only?
287                                  */
288                                 tce6.tcp6ConnEntryInfo.ce_snxt =
289                                     tcp->tcp_snxt - tcp->tcp_suna;
290                                 tce6.tcp6ConnEntryInfo.ce_suna = 0;
291                                 tce6.tcp6ConnEntryInfo.ce_rnxt =
292                                     tcp->tcp_rnxt - tcp->tcp_rack;
293                                 tce6.tcp6ConnEntryInfo.ce_rack = 0;
294                         }

296                         tce6.tcp6ConnEntryInfo.ce_swnd = tcp->tcp_swnd;
297                         tce6.tcp6ConnEntryInfo.ce_rwnd = tcp->tcp_rwnd;
298                         tce6.tcp6ConnEntryInfo.ce_rto =  tcp->tcp_rto;
299                         tce6.tcp6ConnEntryInfo.ce_mss =  tcp->tcp_mss;
300                         tce6.tcp6ConnEntryInfo.ce_state = tcp->tcp_state;

302                         tce6.tcp6ConnCreationProcess =
303                             (connp->conn_cpid < 0) ? MIB2_UNKNOWN_PROCESS :
304                             connp->conn_cpid;
305                         tce6.tcp6ConnCreationTime = connp->conn_open_time;

307                         (void) snmp_append_data2(mp6_conn_ctl->b_cont,
308                             &mp6_conn_tail, (char *)&tce6, tce6_size);

310                         /* my data */
311                         /* push connt_t */
312                         (void) snmp_append_data2(mp6_pidnode_ctl->b_cont,
313                                 &mp6_pidnode_tail, (char *)&tce6, tce6_size);

315                         cph = conn_get_pid_list(connp);

317                         /* push the header + conn pid nodes */
318                         (void) snmp_append_data2(mp6_pidnode_ctl->b_cont,
319                             &mp6_pidnode_tail,
320                             (char *)cph, cph->cph_tot_size);

322                         kmem_free(cph, cph->cph_tot_size);
323                         /* end of my  data */
```



```
325 #endif /* ! codereview */
326                         mlp.tme_connidx = v6_conn_idx++;
327                         if (needattr)
328                                 (void) snmp_append_data2(mp6_attr_ctl->b_cont,
329                                     &mp6_attr_tail, (char *)&mlp, sizeof (mlp));
330                         }
331                         /*
332                          * Create an IPv4 table entry for IPv4 entries and also
333                          * for IPv6 entries which are bound to in6addr_any
334                          * but don't have IPV6_V6ONLY set.
335                          * (i.e. anything an IPv4 peer could connect to)
336                          */
337                         if (connp->conn_ipversion == IPV4_VERSION ||
338                             (tcp->tcp_state <= TCPS_LISTEN &&
339                             !connp->conn_ipv6_v6only &&
340                             IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6))) {
341                                 if (connp->conn_ipversion == IPV6_VERSION) {
342                                         tce.tcpConnRemAddress = INADDR_ANY;
343                                         tce.tcpConnLocalAddress = INADDR_ANY;
344                                 } else {
345                                         tce.tcpConnRemAddress =
346                                             connp->conn_faddr_v4;
347                                         tce.tcpConnLocalAddress =
348                                             connp->conn_laddr_v4;
349                                 }
350                                 tce.tcpConnLocalPort = ntohs(connp->conn_lport);
351                                 tce.tcpConnRemPort = ntohs(connp->conn_fport);
352                                 /* Don't want just anybody seeing these... */
353                                 if (ispriv) {
354                                         tce.tcpConnEntryInfo.ce_snxt =
355                                             tcp->tcp_snxt;
356                                         tce.tcpConnEntryInfo.ce_suna =
357                                             tcp->tcp_suna;
358                                         tce.tcpConnEntryInfo.ce_rnxt =
359                                             tcp->tcp_rnxt;
360                                         tce.tcpConnEntryInfo.ce_rack =
361                                             tcp->tcp_rack;
362                                 } else {
363                                         /*
364                                          * Netstat, unfortunately, uses this to
365                                          * get send/receive queue sizes.  How
366                                          * to fix?
367                                          * Why not compute the difference only?
368                                          */
369                                         tce.tcpConnEntryInfo.ce_snxt =
370                                             tcp->tcp_snxt - tcp->tcp_suna;
371                                         tce.tcpConnEntryInfo.ce_suna = 0;
372                                         tce.tcpConnEntryInfo.ce_rnxt =
373                                             tcp->tcp_rnxt - tcp->tcp_rack;
374                                         tce.tcpConnEntryInfo.ce_rack = 0;
375                                 }

377                                 tce.tcpConnEntryInfo.ce_swnd = tcp->tcp_swnd;
378                                 tce.tcpConnEntryInfo.ce_rwnd = tcp->tcp_rwnd;
379                                 tce.tcpConnEntryInfo.ce_rto =  tcp->tcp_rto;
380                                 tce.tcpConnEntryInfo.ce_mss =  tcp->tcp_mss;
381                                 tce.tcpConnEntryInfo.ce_state =
382                                     tcp->tcp_state;

384                                 tce.tcpConnCreationProcess =
385                                     (connp->conn_cpid < 0) ?
386                                     MIB2_UNKNOWN_PROCESS :
387                                     connp->conn_cpid;
388                                 tce.tcpConnCreationTime = connp->conn_open_time;

390                                 (void) snmp_append_data2(mp_conn_ctl->b_cont,
```

```
391                                     &mp_conn_tail, (char *)&tce, tce_size);

393                                 /* my data */
394                                 /* push connt_t */
395                                 (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
396                                     &mp_pidnode_tail, (char *)&tce, tce_size);

398                                 cph = conn_get_pid_list(connp);

400                                 /* push the header + conn pid nodes */
401                                 (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
402                                        &mp_pidnode_tail, (char *)cph,
403                                        cph->cph_tot_size);

405                                 kmem_free(cph, cph->cph_tot_size);
406                                 /* end of my code */

408 #endif /* ! codereview */
409                                 mlp.tme_connidx = v4_conn_idx++;
410                                 if (needattr)
411                                         (void) snmp_append_data2(
412                                             mp_attr_ctl->b_cont,
413                                             &mp_attr_tail, (char *)&mlp,
414                                             sizeof (mlp));
415                         }
416                 }
417         }

419         tcp_sum_mib(tcps, &tcp_mib);

421         /* Fixed length structure for IPv4 and IPv6 counters */
422         SET_MIB(tcp_mib.tcpConnTableSize, tce_size);
423         SET_MIB(tcp_mib.tcp6ConnTableSize, tce6_size);

425         /*
426          * Synchronize 32- and 64-bit counters.  Note that tcpInSegs and
427          * tcpOutSegs are not updated anywhere in TCP.  The new 64 bits
428          * counters are used.  Hence the old counters' values in tcp_sc_mib
429          * are always 0.
430          */
431         SYNC32_MIB(&tcp_mib, tcpInSegs, tcpHCInSegs);
432         SYNC32_MIB(&tcp_mib, tcpOutSegs, tcpHCOutSegs);

434         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
435         optp->level = MIB2_TCP;
436         optp->name = 0;
437         (void) snmp_append_data(mpdata, (char *)&tcp_mib, tcp_mib_size);
438         optp->len = msgdsize(mpdata);
439         qreply(q, mpctl);

441         /* table of connections... */
442         optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
443             sizeof (struct T_optmgmt_ack)];
444         optp->level = MIB2_TCP;
445         optp->name = MIB2_TCP_CONN;
446         optp->len = msgdsize(mp_conn_ctl->b_cont);
447         qreply(q, mp_conn_ctl);

449         /* table of MLP attributes... */
450         optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
451             sizeof (struct T_optmgmt_ack)];
452         optp->level = MIB2_TCP;
453         optp->name = EXPER_XPORT_MLP;
454         optp->len = msgdsize(mp_attr_ctl->b_cont);
455         if (optp->len == 0)
456                 freemsg(mp_attr_ctl);
```



```
457         else
458                 qreply(q, mp_attr_ctl);

460         /* table of IPv6 connections... */
461         optp = (struct opthdr *)&mp6_conn_ctl->b_rptr[
462             sizeof (struct T_optmgmt_ack)];
463         optp->level = MIB2_TCP6;
464         optp->name = MIB2_TCP6_CONN;
465         optp->len = msgdsize(mp6_conn_ctl->b_cont);
466         qreply(q, mp6_conn_ctl);

468         /* table of IPv6 MLP attributes... */
469         optp = (struct opthdr *)&mp6_attr_ctl->b_rptr[
470             sizeof (struct T_optmgmt_ack)];
471         optp->level = MIB2_TCP6;
472         optp->name = EXPER_XPORT_MLP;
473         optp->len = msgdsize(mp6_attr_ctl->b_cont);
474         if (optp->len == 0)
475                 freemsg(mp6_attr_ctl);
476         else
477                 qreply(q, mp6_attr_ctl);


480         /* table of EXPER_XPORT_PROC_INFO  ipv4 */
481         optp = (struct opthdr *)&mp_pidnode_ctl->b_rptr[
482                 sizeof (struct T_optmgmt_ack)];
483         optp->level = MIB2_TCP;
484         optp->name = EXPER_XPORT_PROC_INFO;
485         optp->len = msgdsize(mp_pidnode_ctl->b_cont);
486         if (optp->len == 0)
487                 freemsg(mp_pidnode_ctl);
488         else
489                 qreply(q, mp_pidnode_ctl);

491         /* table of EXPER_XPORT_PROC_INFO  ipv6 */
492         optp = (struct opthdr *)&mp6_pidnode_ctl->b_rptr[
493                 sizeof (struct T_optmgmt_ack)];
494         optp->level = MIB2_TCP6;
495         optp->name = EXPER_XPORT_PROC_INFO;
496         optp->len = msgdsize(mp6_pidnode_ctl->b_cont);
497         if (optp->len == 0)
498                 freemsg(mp6_pidnode_ctl);
499         else
500                 qreply(q, mp6_pidnode_ctl);

502 #endif /* ! codereview */
503         return (mp2ctl);
504 }

506 /* Return 0 if invalid set request, 1 otherwise, including non-tcp requests  */
507 /* ARGSUSED */
508 int
509 tcp_snmp_set(queue_t *q, int level, int name, uchar_t *ptr, int len)
510 {
511         mib2_tcpConnEntry_t     *tce = (mib2_tcpConnEntry_t *)ptr;

513         switch (level) {
514         case MIB2_TCP:
515                 switch (name) {
516                 case 13:
517                         if (tce->tcpConnState != MIB2_TCP_deleteTCB)
518                                 return (0);
519                         /* TODO: delete entry defined by tce */
520                         return (1);
521                 default:
522                         return (0);
```

```
523                 }
524         default:
525                 return (1);
526         }
527 }

529 /*
530  * TCP Kstats implementation
531  */
532 void *
533 tcp_kstat_init(netstackid_t stackid)
534 {
535         kstat_t *ksp;

537         tcp_named_kstat_t template = {
538                 { "rtoAlgorithm",       KSTAT_DATA_INT32, 0 },
539                 { "rtoMin",             KSTAT_DATA_INT32, 0 },
540                 { "rtoMax",             KSTAT_DATA_INT32, 0 },
541                 { "maxConn",            KSTAT_DATA_INT32, 0 },
542                 { "activeOpens",        KSTAT_DATA_UINT32, 0 },
543                 { "passiveOpens",       KSTAT_DATA_UINT32, 0 },
544                 { "attemptFails",       KSTAT_DATA_UINT32, 0 },
545                 { "estabResets",        KSTAT_DATA_UINT32, 0 },
546                 { "currEstab",          KSTAT_DATA_UINT32, 0 },
547                 { "inSegs",             KSTAT_DATA_UINT64, 0 },
548                 { "outSegs",            KSTAT_DATA_UINT64, 0 },
549                 { "retransSegs",        KSTAT_DATA_UINT32, 0 },
550                 { "connTableSize",      KSTAT_DATA_INT32, 0 },
551                 { "outRsts",            KSTAT_DATA_UINT32, 0 },
552                 { "outDataSegs",        KSTAT_DATA_UINT32, 0 },
553                 { "outDataBytes",       KSTAT_DATA_UINT32, 0 },
554                 { "retransBytes",       KSTAT_DATA_UINT32, 0 },
555                 { "outAck",             KSTAT_DATA_UINT32, 0 },
556                 { "outAckDelayed",      KSTAT_DATA_UINT32, 0 },
557                 { "outUrg",             KSTAT_DATA_UINT32, 0 },
558                 { "outWinUpdate",       KSTAT_DATA_UINT32, 0 },
559                 { "outWinProbe",        KSTAT_DATA_UINT32, 0 },
560                 { "outControl",         KSTAT_DATA_UINT32, 0 },
561                 { "outFastRetrans",     KSTAT_DATA_UINT32, 0 },
562                 { "inAckSegs",          KSTAT_DATA_UINT32, 0 },
563                 { "inAckBytes",         KSTAT_DATA_UINT32, 0 },
564                 { "inDupAck",           KSTAT_DATA_UINT32, 0 },
565                 { "inAckUnsent",        KSTAT_DATA_UINT32, 0 },
566                 { "inDataInorderSegs",  KSTAT_DATA_UINT32, 0 },
567                 { "inDataInorderBytes", KSTAT_DATA_UINT32, 0 },
568                 { "inDataUnorderSegs",  KSTAT_DATA_UINT32, 0 },
569                 { "inDataUnorderBytes", KSTAT_DATA_UINT32, 0 },
570                 { "inDataDupSegs",      KSTAT_DATA_UINT32, 0 },
571                 { "inDataDupBytes",     KSTAT_DATA_UINT32, 0 },
572                 { "inDataPartDupSegs",  KSTAT_DATA_UINT32, 0 },
573                 { "inDataPartDupBytes", KSTAT_DATA_UINT32, 0 },
574                 { "inDataPastWinSegs",  KSTAT_DATA_UINT32, 0 },
575                 { "inDataPastWinBytes", KSTAT_DATA_UINT32, 0 },
576                 { "inWinProbe",         KSTAT_DATA_UINT32, 0 },
577                 { "inWinUpdate",        KSTAT_DATA_UINT32, 0 },
578                 { "inClosed",           KSTAT_DATA_UINT32, 0 },
579                 { "rttUpdate",          KSTAT_DATA_UINT32, 0 },
580                 { "rttNoUpdate",        KSTAT_DATA_UINT32, 0 },
581                 { "timRetrans",         KSTAT_DATA_UINT32, 0 },
582                 { "timRetransDrop",     KSTAT_DATA_UINT32, 0 },
583                 { "timKeepalive",       KSTAT_DATA_UINT32, 0 },
584                 { "timKeepaliveProbe",  KSTAT_DATA_UINT32, 0 },
585                 { "timKeepaliveDrop",   KSTAT_DATA_UINT32, 0 },
586                 { "listenDrop",         KSTAT_DATA_UINT32, 0 },
587                 { "listenDropQ0",       KSTAT_DATA_UINT32, 0 },
588                 { "halfOpenDrop",       KSTAT_DATA_UINT32, 0 },
```



```
589                 { "outSackRetransSegs", KSTAT_DATA_UINT32, 0 },
590                 { "connTableSize6",     KSTAT_DATA_INT32, 0 }
591         };

593         ksp = kstat_create_netstack(TCP_MOD_NAME, stackid, TCP_MOD_NAME, "mib2",
594             KSTAT_TYPE_NAMED, NUM_OF_FIELDS(tcp_named_kstat_t), 0, stackid);

596         if (ksp == NULL)
597                 return (NULL);

599         template.rtoAlgorithm.value.ui32 = 4;
600         template.maxConn.value.i32 = -1;

602         bcopy(&template, ksp->ks_data, sizeof (template));
603         ksp->ks_update = tcp_kstat_update;
604         ksp->ks_private = (void *)(uintptr_t)stackid;

606         /*
607          * If this is an exclusive netstack for a local zone, the global zone
608          * should still be able to read the kstat.
609          */
610         if (stackid != GLOBAL_NETSTACKID)
611                 kstat_zone_add(ksp, GLOBAL_ZONEID);

613         kstat_install(ksp);
614         return (ksp);
615 }

617 void
618 tcp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
619 {
620         if (ksp != NULL) {
621                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
622                 kstat_delete_netstack(ksp, stackid);
623         }
624 }

626 static int
627 tcp_kstat_update(kstat_t *kp, int rw)
628 {
629         tcp_named_kstat_t *tcpkp;
630         tcp_t             *tcp;
631         connf_t           *connfp;
632         conn_t            *connp;
633         int               i;
634         netstackid_t      stackid = (netstackid_t)(uintptr_t)kp->ks_private;
635         netstack_t        *ns;
636         tcp_stack_t       *tcps;
637         ip_stack_t        *ipst;
638         mib2_tcp_t        tcp_mib;

640         if (rw == KSTAT_WRITE)
641                 return (EACCES);

643         ns = netstack_find_by_stackid(stackid);
644         if (ns == NULL)
645                 return (-1);
646         tcps = ns->netstack_tcp;
647         if (tcps == NULL) {
648                 netstack_rele(ns);
649                 return (-1);
650         }

652         tcpkp = (tcp_named_kstat_t *)kp->ks_data;

654         tcpkp->currEstab.value.ui32 = 0;
```

```
655         tcpkp->rtoMin.value.ui32 = tcps->tcps_rexmit_interval_min;
656         tcpkp->rtoMax.value.ui32 = tcps->tcps_rexmit_interval_max;

658         ipst = ns->netstack_ip;

660         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
661                 connfp = &ipst->ips_ipcl_globalhash_fanout[i];
662                 connp = NULL;
663                 while ((connp =
664                     ipcl_get_next_conn(connfp, connp, IPCL_TCPCONN)) != NULL) {
665                         tcp = connp->conn_tcp;
666                         switch (tcp_snmp_state(tcp)) {
667                         case MIB2_TCP_established:
668                         case MIB2_TCP_closeWait:
669                                 tcpkp->currEstab.value.ui32++;
670                                 break;
671                         }
672                 }
673         }
674         bzero(&tcp_mib, sizeof (tcp_mib));
675         tcp_sum_mib(tcps, &tcp_mib);

677         /* Fixed length structure for IPv4 and IPv6 counters */
678         SET_MIB(tcp_mib.tcpConnTableSize, sizeof (mib2_tcpConnEntry_t));
679         SET_MIB(tcp_mib.tcp6ConnTableSize, sizeof (mib2_tcp6ConnEntry_t));

681         tcpkp->activeOpens.value.ui32 = tcp_mib.tcpActiveOpens;
682         tcpkp->passiveOpens.value.ui32 = tcp_mib.tcpPassiveOpens;
683         tcpkp->attemptFails.value.ui32 = tcp_mib.tcpAttemptFails;
684         tcpkp->estabResets.value.ui32 = tcp_mib.tcpEstabResets;
685         tcpkp->inSegs.value.ui64 = tcp_mib.tcpHCInSegs;
686         tcpkp->outSegs.value.ui64 = tcp_mib.tcpHCOutSegs;
687         tcpkp->retransSegs.value.ui32 = tcp_mib.tcpRetransSegs;
688         tcpkp->connTableSize.value.i32 = tcp_mib.tcpConnTableSize;
689         tcpkp->outRsts.value.ui32 = tcp_mib.tcpOutRsts;
690         tcpkp->outDataSegs.value.ui32 = tcp_mib.tcpOutDataSegs;
691         tcpkp->outDataBytes.value.ui32 = tcp_mib.tcpOutDataBytes;
692         tcpkp->retransBytes.value.ui32 = tcp_mib.tcpRetransBytes;
693         tcpkp->outAck.value.ui32 = tcp_mib.tcpOutAck;
694         tcpkp->outAckDelayed.value.ui32 = tcp_mib.tcpOutAckDelayed;
695         tcpkp->outUrg.value.ui32 = tcp_mib.tcpOutUrg;
696         tcpkp->outWinUpdate.value.ui32 = tcp_mib.tcpOutWinUpdate;
697         tcpkp->outWinProbe.value.ui32 = tcp_mib.tcpOutWinProbe;
698         tcpkp->outControl.value.ui32 = tcp_mib.tcpOutControl;
699         tcpkp->outFastRetrans.value.ui32 = tcp_mib.tcpOutFastRetrans;
700         tcpkp->inAckSegs.value.ui32 = tcp_mib.tcpInAckSegs;
701         tcpkp->inAckBytes.value.ui32 = tcp_mib.tcpInAckBytes;
702         tcpkp->inDupAck.value.ui32 = tcp_mib.tcpInDupAck;
703         tcpkp->inAckUnsent.value.ui32 = tcp_mib.tcpInAckUnsent;
704         tcpkp->inDataInorderSegs.value.ui32 = tcp_mib.tcpInDataInorderSegs;
705         tcpkp->inDataInorderBytes.value.ui32 = tcp_mib.tcpInDataInorderBytes;
706         tcpkp->inDataUnorderSegs.value.ui32 = tcp_mib.tcpInDataUnorderSegs;
707         tcpkp->inDataUnorderBytes.value.ui32 = tcp_mib.tcpInDataUnorderBytes;
708         tcpkp->inDataDupSegs.value.ui32 = tcp_mib.tcpInDataDupSegs;
709         tcpkp->inDataDupBytes.value.ui32 = tcp_mib.tcpInDataDupBytes;
710         tcpkp->inDataPartDupSegs.value.ui32 = tcp_mib.tcpInDataPartDupSegs;
711         tcpkp->inDataPartDupBytes.value.ui32 = tcp_mib.tcpInDataPartDupBytes;
712         tcpkp->inDataPastWinSegs.value.ui32 = tcp_mib.tcpInDataPastWinSegs;
713         tcpkp->inDataPastWinBytes.value.ui32 = tcp_mib.tcpInDataPastWinBytes;
714         tcpkp->inWinProbe.value.ui32 = tcp_mib.tcpInWinProbe;
715         tcpkp->inWinUpdate.value.ui32 = tcp_mib.tcpInWinUpdate;
716         tcpkp->inClosed.value.ui32 = tcp_mib.tcpInClosed;
717         tcpkp->rttNoUpdate.value.ui32 = tcp_mib.tcpRttNoUpdate;
718         tcpkp->rttUpdate.value.ui32 = tcp_mib.tcpRttUpdate;
719         tcpkp->timRetrans.value.ui32 = tcp_mib.tcpTimRetrans;
720         tcpkp->timRetransDrop.value.ui32 = tcp_mib.tcpTimRetransDrop;
```



```
721         tcpkp->timKeepalive.value.ui32 = tcp_mib.tcpTimKeepalive;
722         tcpkp->timKeepaliveProbe.value.ui32 = tcp_mib.tcpTimKeepaliveProbe;
723         tcpkp->timKeepaliveDrop.value.ui32 = tcp_mib.tcpTimKeepaliveDrop;
724         tcpkp->listenDrop.value.ui32 = tcp_mib.tcpListenDrop;
725         tcpkp->listenDropQ0.value.ui32 = tcp_mib.tcpListenDropQ0;
726         tcpkp->halfOpenDrop.value.ui32 = tcp_mib.tcpHalfOpenDrop;
727         tcpkp->outSackRetransSegs.value.ui32 = tcp_mib.tcpOutSackRetransSegs;
728         tcpkp->connTableSize6.value.i32 = tcp_mib.tcp6ConnTableSize;

730         netstack_rele(ns);
731         return (0);
732 }

734 /*
735  * kstats related to squeues i.e. not per IP instance
736  */
737 void *
738 tcp_g_kstat_init(tcp_g_stat_t *tcp_g_statp)
739 {
740         kstat_t *ksp;

742         tcp_g_stat_t template = {
743                 { "tcp_timermp_alloced",        KSTAT_DATA_UINT64 },
744                 { "tcp_timermp_allocfail",      KSTAT_DATA_UINT64 },
745                 { "tcp_timermp_allocdblfail",   KSTAT_DATA_UINT64 },
746                 { "tcp_freelist_cleanup",       KSTAT_DATA_UINT64 },
747         };

749         ksp = kstat_create(TCP_MOD_NAME, 0, "tcpstat_g", "net",
750             KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t),
751             KSTAT_FLAG_VIRTUAL);

753         if (ksp == NULL)
754                 return (NULL);

756         bcopy(&template, tcp_g_statp, sizeof (template));
757         ksp->ks_data = (void *)tcp_g_statp;

759         kstat_install(ksp);
760         return (ksp);
761 }

763 void
764 tcp_g_kstat_fini(kstat_t *ksp)
765 {
766         if (ksp != NULL) {
767                 kstat_delete(ksp);
768         }
769 }

771 void *
772 tcp_kstat2_init(netstackid_t stackid)
773 {
774         kstat_t *ksp;

776         tcp_stat_t template = {
777                 { "tcp_time_wait_syn_success",  KSTAT_DATA_UINT64, 0 },
778                 { "tcp_clean_death_nondetached",        KSTAT_DATA_UINT64, 0 },
779                 { "tcp_eager_blowoff_q",        KSTAT_DATA_UINT64, 0 },
780                 { "tcp_eager_blowoff_q0",       KSTAT_DATA_UINT64, 0 },
781                 { "tcp_no_listener",            KSTAT_DATA_UINT64, 0 },
782                 { "tcp_listendrop",             KSTAT_DATA_UINT64, 0 },
783                 { "tcp_listendropq0",           KSTAT_DATA_UINT64, 0 },
784                 { "tcp_wsrv_called",            KSTAT_DATA_UINT64, 0 },
785                 { "tcp_flwctl_on",              KSTAT_DATA_UINT64, 0 },
786                 { "tcp_timer_fire_early",       KSTAT_DATA_UINT64, 0 },
```

```
787                 { "tcp_timer_fire_miss",        KSTAT_DATA_UINT64, 0 },
788                 { "tcp_zcopy_on",               KSTAT_DATA_UINT64, 0 },
789                 { "tcp_zcopy_off",              KSTAT_DATA_UINT64, 0 },
790                 { "tcp_zcopy_backoff",          KSTAT_DATA_UINT64, 0 },
791                 { "tcp_fusion_flowctl",         KSTAT_DATA_UINT64, 0 },
792                 { "tcp_fusion_backenabled",     KSTAT_DATA_UINT64, 0 },
793                 { "tcp_fusion_urg",             KSTAT_DATA_UINT64, 0 },
794                 { "tcp_fusion_putnext",         KSTAT_DATA_UINT64, 0 },
795                 { "tcp_fusion_unfusable",       KSTAT_DATA_UINT64, 0 },
796                 { "tcp_fusion_aborted",         KSTAT_DATA_UINT64, 0 },
797                 { "tcp_fusion_unqualified",     KSTAT_DATA_UINT64, 0 },
798                 { "tcp_fusion_rrw_busy",        KSTAT_DATA_UINT64, 0 },
799                 { "tcp_fusion_rrw_msgcnt",      KSTAT_DATA_UINT64, 0 },
800                 { "tcp_fusion_rrw_plugged",     KSTAT_DATA_UINT64, 0 },
801                 { "tcp_in_ack_unsent_drop",     KSTAT_DATA_UINT64, 0 },
802                 { "tcp_sock_fallback",          KSTAT_DATA_UINT64, 0 },
803                 { "tcp_lso_enabled",            KSTAT_DATA_UINT64, 0 },
804                 { "tcp_lso_disabled",           KSTAT_DATA_UINT64, 0 },
805                 { "tcp_lso_times",              KSTAT_DATA_UINT64, 0 },
806                 { "tcp_lso_pkt_out",            KSTAT_DATA_UINT64, 0 },
807                 { "tcp_listen_cnt_drop",        KSTAT_DATA_UINT64, 0 },
808                 { "tcp_listen_mem_drop",        KSTAT_DATA_UINT64, 0 },
809                 { "tcp_zwin_mem_drop",          KSTAT_DATA_UINT64, 0 },
810                 { "tcp_zwin_ack_syn",           KSTAT_DATA_UINT64, 0 },
811                 { "tcp_rst_unsent",             KSTAT_DATA_UINT64, 0 },
812                 { "tcp_reclaim_cnt",            KSTAT_DATA_UINT64, 0 },
813                 { "tcp_reass_timeout",          KSTAT_DATA_UINT64, 0 },
814 #ifdef TCP_DEBUG_COUNTER
815                 { "tcp_time_wait",              KSTAT_DATA_UINT64, 0 },
816                 { "tcp_rput_time_wait",         KSTAT_DATA_UINT64, 0 },
817                 { "tcp_detach_time_wait",       KSTAT_DATA_UINT64, 0 },
818                 { "tcp_timeout_calls",          KSTAT_DATA_UINT64, 0 },
819                 { "tcp_timeout_cached_alloc",   KSTAT_DATA_UINT64, 0 },
820                 { "tcp_timeout_cancel_reqs",    KSTAT_DATA_UINT64, 0 },
821                 { "tcp_timeout_canceled",       KSTAT_DATA_UINT64, 0 },
822                 { "tcp_timermp_freed",          KSTAT_DATA_UINT64, 0 },
823                 { "tcp_push_timer_cnt",         KSTAT_DATA_UINT64, 0 },
824                 { "tcp_ack_timer_cnt",          KSTAT_DATA_UINT64, 0 },
825 #endif
826         };

828         ksp = kstat_create_netstack(TCP_MOD_NAME, stackid, "tcpstat", "net",
829             KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t), 0,
830             stackid);

832         if (ksp == NULL)
833                 return (NULL);

835         bcopy(&template, ksp->ks_data, sizeof (template));
836         ksp->ks_private = (void *)(uintptr_t)stackid;
837         ksp->ks_update = tcp_kstat2_update;

839         /*
840          * If this is an exclusive netstack for a local zone, the global zone
841          * should still be able to read the kstat.
842          */
843         if (stackid != GLOBAL_NETSTACKID)
844                 kstat_zone_add(ksp, GLOBAL_ZONEID);

846         kstat_install(ksp);
847         return (ksp);
848 }

850 void
851 tcp_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
852 {
```



```
853         if (ksp != NULL) {
854                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
855                 kstat_delete_netstack(ksp, stackid);
856         }
857 }

859 /*
860  * Sum up all per CPU tcp_stat_t kstat counters.
861  */
862 static int
863 tcp_kstat2_update(kstat_t *kp, int rw)
864 {
865         netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
866         netstack_t      *ns;
867         tcp_stack_t     *tcps;
868         tcp_stat_t      *stats;
869         int             i;
870         int             cnt;

872         if (rw == KSTAT_WRITE)
873                 return (EACCES);

875         ns = netstack_find_by_stackid(stackid);
876         if (ns == NULL)
877                 return (-1);
878         tcps = ns->netstack_tcp;
879         if (tcps == NULL) {
880                 netstack_rele(ns);
881                 return (-1);
882         }

884         stats = (tcp_stat_t *)kp->ks_data;
885         tcp_clr_stats(stats);

887         /*
888          * tcps_sc_cnt may change in the middle of the loop.  It is better
889          * to get its value first.
890          */
891         cnt = tcps->tcps_sc_cnt;
892         for (i = 0; i < cnt; i++)
893                 tcp_add_stats(&tcps->tcps_sc[i]->tcp_sc_stats, stats);

895         netstack_rele(ns);
896         return (0);
897 }

899 /*
900  * To add stats from one mib2_tcp_t to another.  Static fields are not added.
901  * The caller should set them up propertly.
902  */
903 static void
904 tcp_add_mib(mib2_tcp_t *from, mib2_tcp_t *to)
905 {
906         to->tcpActiveOpens += from->tcpActiveOpens;
907         to->tcpPassiveOpens += from->tcpPassiveOpens;
908         to->tcpAttemptFails += from->tcpAttemptFails;
909         to->tcpEstabResets += from->tcpEstabResets;
910         to->tcpInSegs += from->tcpInSegs;
911         to->tcpOutSegs += from->tcpOutSegs;
912         to->tcpRetransSegs += from->tcpRetransSegs;
913         to->tcpOutRsts += from->tcpOutRsts;

915         to->tcpOutDataSegs += from->tcpOutDataSegs;
916         to->tcpOutDataBytes += from->tcpOutDataBytes;
917         to->tcpRetransBytes += from->tcpRetransBytes;
918         to->tcpOutAck += from->tcpOutAck;
```

```
919         to->tcpOutAckDelayed += from->tcpOutAckDelayed;
920         to->tcpOutUrg += from->tcpOutUrg;
921         to->tcpOutWinUpdate += from->tcpOutWinUpdate;
922         to->tcpOutWinProbe += from->tcpOutWinProbe;
923         to->tcpOutControl += from->tcpOutControl;
924         to->tcpOutFastRetrans += from->tcpOutFastRetrans;

926         to->tcpInAckBytes += from->tcpInAckBytes;
927         to->tcpInDupAck += from->tcpInDupAck;
928         to->tcpInAckUnsent += from->tcpInAckUnsent;
929         to->tcpInDataInorderSegs += from->tcpInDataInorderSegs;
930         to->tcpInDataInorderBytes += from->tcpInDataInorderBytes;
931         to->tcpInDataUnorderSegs += from->tcpInDataUnorderSegs;
932         to->tcpInDataUnorderBytes += from->tcpInDataUnorderBytes;
933         to->tcpInDataDupSegs += from->tcpInDataDupSegs;
934         to->tcpInDataDupBytes += from->tcpInDataDupBytes;
935         to->tcpInDataPartDupSegs += from->tcpInDataPartDupSegs;
936         to->tcpInDataPartDupBytes += from->tcpInDataPartDupBytes;
937         to->tcpInDataPastWinSegs += from->tcpInDataPastWinSegs;
938         to->tcpInDataPastWinBytes += from->tcpInDataPastWinBytes;
939         to->tcpInWinProbe += from->tcpInWinProbe;
940         to->tcpInWinUpdate += from->tcpInWinUpdate;
941         to->tcpInClosed += from->tcpInClosed;

943         to->tcpRttNoUpdate += from->tcpRttNoUpdate;
944         to->tcpRttUpdate += from->tcpRttUpdate;
945         to->tcpTimRetrans += from->tcpTimRetrans;
946         to->tcpTimRetransDrop += from->tcpTimRetransDrop;
947         to->tcpTimKeepalive += from->tcpTimKeepalive;
948         to->tcpTimKeepaliveProbe += from->tcpTimKeepaliveProbe;
949         to->tcpTimKeepaliveDrop += from->tcpTimKeepaliveDrop;
950         to->tcpListenDrop += from->tcpListenDrop;
951         to->tcpListenDropQ0 += from->tcpListenDropQ0;
952         to->tcpHalfOpenDrop += from->tcpHalfOpenDrop;
953         to->tcpOutSackRetransSegs += from->tcpOutSackRetransSegs;
954         to->tcpHCInSegs += from->tcpHCInSegs;
955         to->tcpHCOutSegs += from->tcpHCOutSegs;
956 }

958 /*
959  * To sum up all MIB2 stats for a tcp_stack_t from all per CPU stats.  The
960  * caller should initialize the target mib2_tcp_t properly as this function
961  * just adds up all the per CPU stats.
962  */
963 static void
964 tcp_sum_mib(tcp_stack_t *tcps, mib2_tcp_t *tcp_mib)
965 {
966         int i;
967         int cnt;

969         /*
970          * tcps_sc_cnt may change in the middle of the loop.  It is better
971          * to get its value first.
972          */
973         cnt = tcps->tcps_sc_cnt;
974         for (i = 0; i < cnt; i++)
975                 tcp_add_mib(&tcps->tcps_sc[i]->tcp_sc_mib, tcp_mib);
976 }

978 /*
979  * To set all tcp_stat_t counters to 0.
980  */
981 static void
982 tcp_clr_stats(tcp_stat_t *stats)
983 {
984         stats->tcp_time_wait_syn_success.value.ui64 = 0;
```



```
985         stats->tcp_clean_death_nondetached.value.ui64 = 0;
986         stats->tcp_eager_blowoff_q.value.ui64 = 0;
987         stats->tcp_eager_blowoff_q0.value.ui64 = 0;
988         stats->tcp_no_listener.value.ui64 = 0;
989         stats->tcp_listendrop.value.ui64 = 0;
990         stats->tcp_listendropq0.value.ui64 = 0;
991         stats->tcp_wsrv_called.value.ui64 = 0;
992         stats->tcp_flwctl_on.value.ui64 = 0;
993         stats->tcp_timer_fire_early.value.ui64 = 0;
994         stats->tcp_timer_fire_miss.value.ui64 = 0;
995         stats->tcp_zcopy_on.value.ui64 = 0;
996         stats->tcp_zcopy_off.value.ui64 = 0;
997         stats->tcp_zcopy_backoff.value.ui64 = 0;
998         stats->tcp_fusion_flowctl.value.ui64 = 0;
999         stats->tcp_fusion_backenabled.value.ui64 = 0;
1000        stats->tcp_fusion_urg.value.ui64 = 0;
1001        stats->tcp_fusion_putnext.value.ui64 = 0;
1002        stats->tcp_fusion_unfusable.value.ui64 = 0;
1003        stats->tcp_fusion_aborted.value.ui64 = 0;
1004        stats->tcp_fusion_unqualified.value.ui64 = 0;
1005        stats->tcp_fusion_rrw_busy.value.ui64 = 0;
1006        stats->tcp_fusion_rrw_msgcnt.value.ui64 = 0;
1007        stats->tcp_fusion_rrw_plugged.value.ui64 = 0;
1008        stats->tcp_in_ack_unsent_drop.value.ui64 = 0;
1009        stats->tcp_sock_fallback.value.ui64 = 0;
1010        stats->tcp_lso_enabled.value.ui64 = 0;
1011        stats->tcp_lso_disabled.value.ui64 = 0;
1012        stats->tcp_lso_times.value.ui64 = 0;
1013        stats->tcp_lso_pkt_out.value.ui64 = 0;
1014        stats->tcp_listen_cnt_drop.value.ui64 = 0;
1015        stats->tcp_listen_mem_drop.value.ui64 = 0;
1016        stats->tcp_zwin_mem_drop.value.ui64 = 0;
1017        stats->tcp_zwin_ack_syn.value.ui64 = 0;
1018        stats->tcp_rst_unsent.value.ui64 = 0;
1019        stats->tcp_reclaim_cnt.value.ui64 = 0;
1020        stats->tcp_reass_timeout.value.ui64 = 0;

1022 #ifdef TCP_DEBUG_COUNTER
1023        stats->tcp_time_wait.value.ui64 = 0;
1024        stats->tcp_rput_time_wait.value.ui64 = 0;
1025        stats->tcp_detach_time_wait.value.ui64 = 0;
1026        stats->tcp_timeout_calls.value.ui64 = 0;
1027        stats->tcp_timeout_cached_alloc.value.ui64 = 0;
1028        stats->tcp_timeout_cancel_reqs.value.ui64 = 0;
1029        stats->tcp_timeout_canceled.value.ui64 = 0;
1030        stats->tcp_timermp_freed.value.ui64 = 0;
1031        stats->tcp_push_timer_cnt.value.ui64 = 0;
1032        stats->tcp_ack_timer_cnt.value.ui64 = 0;
1033 #endif
1034 }

1036 /*
1037  * To add counters from the per CPU tcp_stat_counter_t to the stack
1038  * tcp_stat_t.
1039  */
1040 static void
1041 tcp_add_stats(tcp_stat_counter_t *from, tcp_stat_t *to)
1042 {
1043        to->tcp_time_wait_syn_success.value.ui64 +=
1044            from->tcp_time_wait_syn_success;
1045        to->tcp_clean_death_nondetached.value.ui64 +=
1046            from->tcp_clean_death_nondetached;
1047        to->tcp_eager_blowoff_q.value.ui64 +=
1048            from->tcp_eager_blowoff_q;
1049        to->tcp_eager_blowoff_q0.value.ui64 +=
1050            from->tcp_eager_blowoff_q0;
```

```
1051         to->tcp_no_listener.value.ui64 +=
1052             from->tcp_no_listener;
1053         to->tcp_listendrop.value.ui64 +=
1054             from->tcp_listendrop;
1055         to->tcp_listendropq0.value.ui64 +=
1056             from->tcp_listendropq0;
1057         to->tcp_wsrv_called.value.ui64 +=
1058             from->tcp_wsrv_called;
1059         to->tcp_flwctl_on.value.ui64 +=
1060             from->tcp_flwctl_on;
1061         to->tcp_timer_fire_early.value.ui64 +=
1062             from->tcp_timer_fire_early;
1063         to->tcp_timer_fire_miss.value.ui64 +=
1064             from->tcp_timer_fire_miss;
1065         to->tcp_zcopy_on.value.ui64 +=
1066             from->tcp_zcopy_on;
1067         to->tcp_zcopy_off.value.ui64 +=
1068             from->tcp_zcopy_off;
1069         to->tcp_zcopy_backoff.value.ui64 +=
1070             from->tcp_zcopy_backoff;
1071         to->tcp_fusion_flowctl.value.ui64 +=
1072             from->tcp_fusion_flowctl;
1073         to->tcp_fusion_backenabled.value.ui64 +=
1074             from->tcp_fusion_backenabled;
1075         to->tcp_fusion_urg.value.ui64 +=
1076             from->tcp_fusion_urg;
1077         to->tcp_fusion_putnext.value.ui64 +=
1078             from->tcp_fusion_putnext;
1079         to->tcp_fusion_unfusable.value.ui64 +=
1080             from->tcp_fusion_unfusable;
1081         to->tcp_fusion_aborted.value.ui64 +=
1082             from->tcp_fusion_aborted;
1083         to->tcp_fusion_unqualified.value.ui64 +=
1084             from->tcp_fusion_unqualified;
1085         to->tcp_fusion_rrw_busy.value.ui64 +=
1086             from->tcp_fusion_rrw_busy;
1087         to->tcp_fusion_rrw_msgcnt.value.ui64 +=
1088             from->tcp_fusion_rrw_msgcnt;
1089         to->tcp_fusion_rrw_plugged.value.ui64 +=
1090             from->tcp_fusion_rrw_plugged;
1091         to->tcp_in_ack_unsent_drop.value.ui64 +=
1092             from->tcp_in_ack_unsent_drop;
1093         to->tcp_sock_fallback.value.ui64 +=
1094             from->tcp_sock_fallback;
1095         to->tcp_lso_enabled.value.ui64 +=
1096             from->tcp_lso_enabled;
1097         to->tcp_lso_disabled.value.ui64 +=
1098             from->tcp_lso_disabled;
1099         to->tcp_lso_times.value.ui64 +=
1100             from->tcp_lso_times;
1101         to->tcp_lso_pkt_out.value.ui64 +=
1102             from->tcp_lso_pkt_out;
1103         to->tcp_listen_cnt_drop.value.ui64 +=
1104             from->tcp_listen_cnt_drop;
1105         to->tcp_listen_mem_drop.value.ui64 +=
1106             from->tcp_listen_mem_drop;
1107         to->tcp_zwin_mem_drop.value.ui64 +=
1108             from->tcp_zwin_mem_drop;
1109         to->tcp_zwin_ack_syn.value.ui64 +=
1110             from->tcp_zwin_ack_syn;
1111         to->tcp_rst_unsent.value.ui64 +=
1112             from->tcp_rst_unsent;
1113         to->tcp_reclaim_cnt.value.ui64 +=
1114             from->tcp_reclaim_cnt;
1115         to->tcp_reass_timeout.value.ui64 +=
1116             from->tcp_reass_timeout;
```



```
1118 #ifdef TCP_DEBUG_COUNTER
1119         to->tcp_time_wait.value.ui64 +=
1120             from->tcp_time_wait;
1121         to->tcp_rput_time_wait.value.ui64 +=
1122             from->tcp_rput_time_wait;
1123         to->tcp_detach_time_wait.value.ui64 +=
1124             from->tcp_detach_time_wait;
1125         to->tcp_timeout_calls.value.ui64 +=
1126             from->tcp_timeout_calls;
1127         to->tcp_timeout_cached_alloc.value.ui64 +=
1128             from->tcp_timeout_cached_alloc;
1129         to->tcp_timeout_cancel_reqs.value.ui64 +=
1130             from->tcp_timeout_cancel_reqs;
1131         to->tcp_timeout_canceled.value.ui64 +=
1132             from->tcp_timeout_canceled;
1133         to->tcp_timermp_freed.value.ui64 +=
1134             from->tcp_timermp_freed;
1135         to->tcp_push_timer_cnt.value.ui64 +=
1136             from->tcp_push_timer_cnt;
1137         to->tcp_ack_timer_cnt.value.ui64 +=
1138             from->tcp_ack_timer_cnt;
1139 #endif
1140 }
```

```
**********************************************************
   18076 Sun Aug  9 12:48:00 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;

        conn_pid_node_list_hdr_t        *cph;

#endif /* ! codereview */

        /*
         * 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) {
            (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);
```

```
127         } else {
128                 udp_mib_size = sizeof (mib2_udp_t);
129                 ude_size = sizeof (mib2_udpEntry_t);
130                 ude6_size = sizeof (mib2_udp6Entry_t);
131         }

133         bzero(&udp_mib, sizeof (udp_mib));
134         /* fixed length structure for IPv4 and IPv6 counters */
135         SET_MIB(udp_mib.udpEntrySize, ude_size);
136         SET_MIB(udp_mib.udp6EntrySize, ude6_size);

138         udp_sum_mib(us, &udp_mib);

140         /*
141          * Synchronize 32- and 64-bit counters.  Note that udpInDatagrams and
142          * udpOutDatagrams are not updated anywhere in UDP.  The new 64 bits
143          * counters are used.  Hence the old counters' values in us_sc_mib
144          * are always 0.
145          */
146         SYNC32_MIB(&udp_mib, udpInDatagrams, udpHCInDatagrams);
147         SYNC32_MIB(&udp_mib, udpOutDatagrams, udpHCOutDatagrams);

149         optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
150         optp->level = MIB2_UDP;
151         optp->name = 0;
152         (void) snmp_append_data(mpdata, (char *)&udp_mib, udp_mib_size);
153         optp->len = msgdsize(mpdata);
154         qreply(q, mpctl);

156         mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;
157         mp_pidnode_tail = mp6_pidnode_tail = NULL;
158 #endif /* ! codereview */
159         v4_conn_idx = v6_conn_idx = 0;

161         for (i = 0; i < CONN_G_HASH_SIZE; i++) {
162                 connfp = &ipst->ips_ipcl_globalhash_fanout[i];
163                 connp = NULL;

165                 while ((connp = ipcl_get_next_conn(connfp, connp,
166                     IPCL_UDPCONN))) {
167                         udp = connp->conn_udp;
168                         if (zoneid != connp->conn_zoneid)
169                                 continue;

171                         /*
172                          * Note that the port numbers are sent in
173                          * host byte order
174                          */

176                         if (udp->udp_state == TS_UNBND)
177                                 state = MIB2_UDP_unbound;
178                         else if (udp->udp_state == TS_IDLE)
179                                 state = MIB2_UDP_idle;
180                         else if (udp->udp_state == TS_DATA_XFER)
181                                 state = MIB2_UDP_connected;
182                         else
183                                 state = MIB2_UDP_unknown;

185                         needattr = B_FALSE;
186                         bzero(&mlp, sizeof (mlp));
187                         if (connp->conn_mlp_type != mlptSingle) {
188                                 if (connp->conn_mlp_type == mlptShared ||
189                                     connp->conn_mlp_type == mlptBoth)
190                                         mlp.tme_flags |= MIB2_TMEF_SHARED;
191                                 if (connp->conn_mlp_type == mlptPrivate ||
192                                     connp->conn_mlp_type == mlptBoth)
```



```
193                                         mlp.tme_flags |= MIB2_TMEF_PRIVATE;
194                                 needattr = B_TRUE;
195                         }
196                         if (connp->conn_anon_mlp) {
197                                 mlp.tme_flags |= MIB2_TMEF_ANONMLP;
198                                 needattr = B_TRUE;
199                         }
200                         switch (connp->conn_mac_mode) {
201                         case CONN_MAC_DEFAULT:
202                                 break;
203                         case CONN_MAC_AWARE:
204                                 mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
205                                 needattr = B_TRUE;
206                                 break;
207                         case CONN_MAC_IMPLICIT:
208                                 mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
209                                 needattr = B_TRUE;
210                                 break;
211                         }
212                         mutex_enter(&connp->conn_lock);
213                         if (udp->udp_state == TS_DATA_XFER &&
214                             connp->conn_ixa->ixa_tsl != NULL) {
215                                 ts_label_t *tsl;

217                                 tsl = connp->conn_ixa->ixa_tsl;
218                                 mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
219                                 mlp.tme_doi = label2doi(tsl);
220                                 mlp.tme_label = *label2bslabel(tsl);
221                                 needattr = B_TRUE;
222                         }
223                         mutex_exit(&connp->conn_lock);

225                         /*
226                          * Create an IPv4 table entry for IPv4 entries and also
227                          * any IPv6 entries which are bound to in6addr_any
228                          * (i.e. anything a IPv4 peer could connect/send to).
229                          */
230                         if (connp->conn_ipversion == IPV4_VERSION ||
231                             (udp->udp_state <= TS_IDLE &&
232                             IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6))) {
233                                 ude.udpEntryInfo.ue_state = state;
234                                 /*
235                                  * If in6addr_any this will set it to
236                                  * INADDR_ANY
237                                  */
238                                 ude.udpLocalAddress = connp->conn_laddr_v4;
239                                 ude.udpLocalPort = ntohs(connp->conn_lport);
240                                 if (udp->udp_state == TS_DATA_XFER) {
241                                         /*
242                                          * Can potentially get here for
243                                          * v6 socket if another process
244                                          * (say, ping) has just done a
245                                          * sendto(), changing the state
246                                          * from the TS_IDLE above to
247                                          * TS_DATA_XFER by the time we hit
248                                          * this part of the code.
249                                          */
250                                         ude.udpEntryInfo.ue_RemoteAddress =
251                                             connp->conn_faddr_v4;
252                                         ude.udpEntryInfo.ue_RemotePort =
253                                             ntohs(connp->conn_fport);
254                                 } else {
255                                         ude.udpEntryInfo.ue_RemoteAddress = 0;
256                                         ude.udpEntryInfo.ue_RemotePort = 0;
257                                 }
```

```
259                                 /*
260                                  * We make the assumption that all udp_t
261                                  * structs will be created within an address
262                                  * region no larger than 32-bits.
263                                  */
264                                 ude.udpInstance = (uint32_t)(uintptr_t)udp;
265                                 ude.udpCreationProcess =
266                                     (connp->conn_cpid < 0) ?
267                                     MIB2_UNKNOWN_PROCESS :
268                                     connp->conn_cpid;
269                                 ude.udpCreationTime = connp->conn_open_time;

271                                 (void) snmp_append_data2(mp_conn_ctl->b_cont,
272                                     &mp_conn_tail, (char *)&ude, ude_size);
273                                 /* my data */
274                                 (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
275                                     &mp_pidnode_tail, (char *)&ude, ude_size);

277                                 cph = conn_get_pid_list(connp);
278                                 (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
279                                     &mp_pidnode_tail, (char *)cph,
280                                     cph->cph_tot_size);

282                                 kmem_free(cph, cph->cph_tot_size);
283                                 /* end of my data */

285 #endif /* ! codereview */
286                                 mlp.tme_connidx = v4_conn_idx++;
287                                 if (needattr)
288                                         (void) snmp_append_data2(
289                                             mp_attr_ctl->b_cont, &mp_attr_tail,
290                                             (char *)&mlp, sizeof (mlp));
291                         }
292                         if (connp->conn_ipversion == IPV6_VERSION) {
293                                 ude6.udp6EntryInfo.ue_state  = state;
294                                 ude6.udp6LocalAddress = connp->conn_laddr_v6;
295                                 ude6.udp6LocalPort = ntohs(connp->conn_lport);
296                                 mutex_enter(&connp->conn_lock);
297                                 if (connp->conn_ixa->ixa_flags &
298                                     IXAF_SCOPEID_SET) {
299                                         ude6.udp6IfIndex =
300                                             connp->conn_ixa->ixa_scopeid;
301                                 } else {
302                                         ude6.udp6IfIndex = connp->conn_bound_if;
303                                 }
304                                 mutex_exit(&connp->conn_lock);
305                                 if (udp->udp_state == TS_DATA_XFER) {
306                                         ude6.udp6EntryInfo.ue_RemoteAddress =
307                                             connp->conn_faddr_v6;
308                                         ude6.udp6EntryInfo.ue_RemotePort =
309                                             ntohs(connp->conn_fport);
310                                 } else {
311                                         ude6.udp6EntryInfo.ue_RemoteAddress =
312                                             sin6_null.sin6_addr;
313                                         ude6.udp6EntryInfo.ue_RemotePort = 0;
314                                 }
315                                 /*
316                                  * We make the assumption that all udp_t
317                                  * structs will be created within an address
318                                  * region no larger than 32-bits.
319                                  */
320                                 ude6.udp6Instance = (uint32_t)(uintptr_t)udp;
321                                 ude6.udp6CreationProcess =
322                                     (connp->conn_cpid < 0) ?
323                                     MIB2_UNKNOWN_PROCESS :
324                                     connp->conn_cpid;
```



```
325                                 ude6.udp6CreationTime = connp->conn_open_time;

327                                 (void) snmp_append_data2(mp6_conn_ctl->b_cont,
328                                     &mp6_conn_tail, (char *)&ude6, ude6_size);
329                                 /* my dat */
330                                 (void) snmp_append_data2(
331                                     mp6_pidnode_ctl->b_cont, &mp6_pidnode_tail,
332                                     (char *)&ude6, ude6_size);

334                                 cph = conn_get_pid_list(connp);
335                                 (void) snmp_append_data2(
336                                     mp6_pidnode_ctl->b_cont, &mp6_pidnode_tail,
337                                     (char *)cph, cph->cph_tot_size);

339                                 kmem_free(cph, cph->cph_tot_size);
340                                 /* end of my data */
341 #endif /* ! codereview */
342                                 mlp.tme_connidx = v6_conn_idx++;
343                                 if (needattr)
344                                         (void) snmp_append_data2(
345                                             mp6_attr_ctl->b_cont,
346                                             &mp6_attr_tail, (char *)&mlp,
347                                             sizeof (mlp));
348                         }
349                 }
350         }

352         /* IPv4 UDP endpoints */
353         optp = (struct opthdr *)&mp_conn_ctl->b_rptr[
354             sizeof (struct T_optmgmt_ack)];
355         optp->level = MIB2_UDP;
356         optp->name = MIB2_UDP_ENTRY;
357         optp->len = msgdsize(mp_conn_ctl->b_cont);
358         qreply(q, mp_conn_ctl);

360         /* table of MLP attributes... */
361         optp = (struct opthdr *)&mp_attr_ctl->b_rptr[
362             sizeof (struct T_optmgmt_ack)];
363         optp->level = MIB2_UDP;
364         optp->name = EXPER_XPORT_MLP;
365         optp->len = msgdsize(mp_attr_ctl->b_cont);
366         if (optp->len == 0)
367                 freemsg(mp_attr_ctl);
368         else
369                 qreply(q, mp_attr_ctl);

371         /* table of EXPER_XPORT_PROC_INFO  ipv4 */
372         optp = (struct opthdr *)&mp_pidnode_ctl->b_rptr[
373             sizeof (struct T_optmgmt_ack)];
374         optp->level = MIB2_UDP;
375         optp->name = EXPER_XPORT_PROC_INFO;
376         optp->len = msgdsize(mp_pidnode_ctl->b_cont);
377         if (optp->len == 0)
378                 freemsg(mp_pidnode_ctl);
379         else
380                 qreply(q, mp_pidnode_ctl);

382 #endif /* ! codereview */
383         /* IPv6 UDP endpoints */
384         optp = (struct opthdr *)&mp6_conn_ctl->b_rptr[
385             sizeof (struct T_optmgmt_ack)];
386         optp->level = MIB2_UDP6;
387         optp->name = MIB2_UDP6_ENTRY;
388         optp->len = msgdsize(mp6_conn_ctl->b_cont);
389         qreply(q, mp6_conn_ctl);
```

```
391         /* table of MLP attributes... */
392         optp = (struct opthdr *)&mp6_attr_ctl->b_rptr[
393             sizeof (struct T_optmgmt_ack)];
394         optp->level = MIB2_UDP6;
395         optp->name = EXPER_XPORT_MLP;
396         optp->len = msgdsize(mp6_attr_ctl->b_cont);
397         if (optp->len == 0)
398                 freemsg(mp6_attr_ctl);
399         else
400                 qreply(q, mp6_attr_ctl);

402         /* table of EXPER_XPORT_PROC_INFO  ipv6 */
403         optp = (struct opthdr *)&mp6_pidnode_ctl->b_rptr[
404             sizeof (struct T_optmgmt_ack)];
405         optp->level = MIB2_UDP6;
406         optp->name = EXPER_XPORT_PROC_INFO;
407         optp->len = msgdsize(mp6_pidnode_ctl->b_cont);
408         if (optp->len == 0)
409                 freemsg(mp6_pidnode_ctl);
410         else
411                 qreply(q, mp6_pidnode_ctl);
412 #endif /* ! codereview */

414         return (mp2ctl);
415 }

417 /*
418  * Return 0 if invalid set request, 1 otherwise, including non-udp requests.
419  * NOTE: Per MIB-II, UDP has no writable data.
420  * TODO:  If this ever actually tries to set anything, it needs to be
421  * to do the appropriate locking.
422  */
423 /* ARGSUSED */
424 int
425 udp_snmp_set(queue_t *q, t_scalar_t level, t_scalar_t name,
426     uchar_t *ptr, int len)
427 {
428         switch (level) {
429         case MIB2_UDP:
430                 return (0);
431         default:
432                 return (1);
433         }
434 }

436 void
437 udp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
438 {
439         if (ksp != NULL) {
440                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
441                 kstat_delete_netstack(ksp, stackid);
442         }
443 }

445 /*
446  * To add stats from one mib2_udp_t to another.  Static fields are not added.
447  * The caller should set them up propertly.
448  */
449 static void
450 udp_add_mib(mib2_udp_t *from, mib2_udp_t *to)
451 {
452         to->udpHCInDatagrams += from->udpHCInDatagrams;
453         to->udpInErrors += from->udpInErrors;
454         to->udpHCOutDatagrams += from->udpHCOutDatagrams;
455         to->udpOutErrors += from->udpOutErrors;
456 }
```



```
459 void *
460 udp_kstat2_init(netstackid_t stackid)
461 {
462         kstat_t *ksp;

464         udp_stat_t template = {
465                 { "udp_sock_fallback",          KSTAT_DATA_UINT64 },
466                 { "udp_out_opt",                KSTAT_DATA_UINT64 },
467                 { "udp_out_err_notconn",        KSTAT_DATA_UINT64 },
468                 { "udp_out_err_output",         KSTAT_DATA_UINT64 },
469                 { "udp_out_err_tudr",           KSTAT_DATA_UINT64 },
470 #ifdef DEBUG
471                 { "udp_data_conn",              KSTAT_DATA_UINT64 },
472                 { "udp_data_notconn",           KSTAT_DATA_UINT64 },
473                 { "udp_out_lastdst",            KSTAT_DATA_UINT64 },
474                 { "udp_out_diffdst",            KSTAT_DATA_UINT64 },
475                 { "udp_out_ipv6",               KSTAT_DATA_UINT64 },
476                 { "udp_out_mapped",             KSTAT_DATA_UINT64 },
477                 { "udp_out_ipv4",               KSTAT_DATA_UINT64 },
478 #endif
479         };

481         ksp = kstat_create_netstack(UDP_MOD_NAME, 0, "udpstat", "net",
482             KSTAT_TYPE_NAMED, sizeof (template) / sizeof (kstat_named_t),
483             0, stackid);

485         if (ksp == NULL)
486                 return (NULL);

488         bcopy(&template, ksp->ks_data, sizeof (template));
489         ksp->ks_update = udp_kstat2_update;
490         ksp->ks_private = (void *)(uintptr_t)stackid;

492         kstat_install(ksp);
493         return (ksp);
494 }

496 void
497 udp_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
498 {
499         if (ksp != NULL) {
500                 ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
501                 kstat_delete_netstack(ksp, stackid);
502         }
503 }

505 /*
506  * To copy counters from the per CPU udpp_stat_counter_t to the stack
507  * udp_stat_t.
508  */
509 static void
510 udp_add_stats(udp_stat_counter_t *from, udp_stat_t *to)
511 {
512         to->udp_sock_fallback.value.ui64 += from->udp_sock_fallback;
513         to->udp_out_opt.value.ui64 += from->udp_out_opt;
514         to->udp_out_err_notconn.value.ui64 += from->udp_out_err_notconn;
515         to->udp_out_err_output.value.ui64 += from->udp_out_err_output;
516         to->udp_out_err_tudr.value.ui64 += from->udp_out_err_tudr;
517 #ifdef DEBUG
518         to->udp_data_conn.value.ui64 += from->udp_data_conn;
519         to->udp_data_notconn.value.ui64 += from->udp_data_notconn;
520         to->udp_out_lastdst.value.ui64 += from->udp_out_lastdst;
521         to->udp_out_diffdst.value.ui64 += from->udp_out_diffdst;
522         to->udp_out_ipv6.value.ui64 += from->udp_out_ipv6;
```

```
523         to->udp_out_mapped.value.ui64 += from->udp_out_mapped;
524         to->udp_out_ipv4.value.ui64 += from->udp_out_ipv4;
525 #endif
526 }

528 /*
529  * To set all udp_stat_t counters to 0.
530  */
531 static void
532 udp_clr_stats(udp_stat_t *stats)
533 {
534         stats->udp_sock_fallback.value.ui64 = 0;
535         stats->udp_out_opt.value.ui64 = 0;
536         stats->udp_out_err_notconn.value.ui64 = 0;
537         stats->udp_out_err_output.value.ui64 = 0;
538         stats->udp_out_err_tudr.value.ui64 = 0;
539 #ifdef DEBUG
540         stats->udp_data_conn.value.ui64 = 0;
541         stats->udp_data_notconn.value.ui64 = 0;
542         stats->udp_out_lastdst.value.ui64 = 0;
543         stats->udp_out_diffdst.value.ui64 = 0;
544         stats->udp_out_ipv6.value.ui64 = 0;
545         stats->udp_out_mapped.value.ui64 = 0;
546         stats->udp_out_ipv4.value.ui64 = 0;
547 #endif
548 }

550 int
551 udp_kstat2_update(kstat_t *kp, int rw)
552 {
553         udp_stat_t      *stats;
554         netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
555         netstack_t      *ns;
556         udp_stack_t     *us;
557         int             i;
558         int             cnt;

560         if (rw == KSTAT_WRITE)
561                 return (EACCES);

563         ns = netstack_find_by_stackid(stackid);
564         if (ns == NULL)
565                 return (-1);
566         us = ns->netstack_udp;
567         if (us == NULL) {
568                 netstack_rele(ns);
569                 return (-1);
570         }
571         stats = (udp_stat_t *)kp->ks_data;
572         udp_clr_stats(stats);

574         cnt = us->us_sc_cnt;
575         for (i = 0; i < cnt; i++)
576                 udp_add_stats(&us->us_sc[i]->udp_sc_stats, stats);

578         netstack_rele(ns);
579         return (0);
580 }

582 void *
583 udp_kstat_init(netstackid_t stackid)
584 {
585         kstat_t *ksp;

587         udp_named_kstat_t template = {
588                 { "inDatagrams",        KSTAT_DATA_UINT64, 0 },
```



```
589                 { "inErrors",           KSTAT_DATA_UINT32, 0 },
590                 { "outDatagrams",       KSTAT_DATA_UINT64, 0 },
591                 { "entrySize",          KSTAT_DATA_INT32, 0 },
592                 { "entry6Size",         KSTAT_DATA_INT32, 0 },
593                 { "outErrors",          KSTAT_DATA_UINT32, 0 },
594         };

596         ksp = kstat_create_netstack(UDP_MOD_NAME, 0, UDP_MOD_NAME, "mib2",
597             KSTAT_TYPE_NAMED, NUM_OF_FIELDS(udp_named_kstat_t), 0, stackid);

599         if (ksp == NULL)
600                 return (NULL);

602         template.entrySize.value.ui32 = sizeof (mib2_udpEntry_t);
603         template.entry6Size.value.ui32 = sizeof (mib2_udp6Entry_t);

605         bcopy(&template, ksp->ks_data, sizeof (template));
606         ksp->ks_update = udp_kstat_update;
607         ksp->ks_private = (void *)(uintptr_t)stackid;

609         kstat_install(ksp);
610         return (ksp);
611 }

613 /*
614  * To sum up all MIB2 stats for a udp_stack_t from all per CPU stats.  The
615  * caller should initialize the target mib2_udp_t properly as this function
616  * just adds up all the per CPU stats.
617  */
618 static void
619 udp_sum_mib(udp_stack_t *us, mib2_udp_t *udp_mib)
620 {
621         int i;
622         int cnt;

624         cnt = us->us_sc_cnt;
625         for (i = 0; i < cnt; i++)
626                 udp_add_mib(&us->us_sc[i]->udp_sc_mib, udp_mib);
627 }

629 static int
630 udp_kstat_update(kstat_t *kp, int rw)
631 {
632         udp_named_kstat_t *udpkp;
633         netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
634         netstack_t      *ns;
635         udp_stack_t     *us;
636         mib2_udp_t      udp_mib;

638         if (rw == KSTAT_WRITE)
639                 return (EACCES);

641         ns = netstack_find_by_stackid(stackid);
642         if (ns == NULL)
643                 return (-1);
644         us = ns->netstack_udp;
645         if (us == NULL) {
646                 netstack_rele(ns);
647                 return (-1);
648         }
649         udpkp = (udp_named_kstat_t *)kp->ks_data;

651         bzero(&udp_mib, sizeof (udp_mib));
652         udp_sum_mib(us, &udp_mib);

654         udpkp->inDatagrams.value.ui64 = udp_mib.udpHCInDatagrams;
```

```
655         udpkp->inErrors.value.ui32 =    udp_mib.udpInErrors;
656         udpkp->outDatagrams.value.ui64 = udp_mib.udpHCOutDatagrams;
657         udpkp->outErrors.value.ui32 =   udp_mib.udpOutErrors;
658         netstack_rele(ns);
659         return (0);
660 }
```

```
**********************************************************
   47074 Sun Aug  9 12:48:00 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                         VOP_IOCTL(cufp->uf_file->f_vnode, F_FORKED,
 869                                 (intptr_t)cp,
 870                                 FKIOCTL,
 871                                 kcred, NULL, NULL);
 872                 }

 874 #endif /* ! codereview */
 875                 if (pufp->uf_file == NULL) {
 876                         ASSERT(pufp->uf_flag == 0);
 877                         if (pufp->uf_busy) {
 878                                 /*
 879                                  * Grab locks to appease ASSERTs in fd_reserve
 880                                  */
 881                                 mutex_enter(&cfip->fi_lock);
 882                                 mutex_enter(&cufp->uf_lock);
 883                                 fd_reserve(cfip, fd, -1);
 884                                 mutex_exit(&cufp->uf_lock);
 885                                 mutex_exit(&cfip->fi_lock);
 886                         }
 887                 }
 888         }
 889 }

 891 /*
 892  * Close all open file descriptors for the current process.
 893  * This is only called from exit(), which is single-threaded,
```



```
 894  * so we don't need any locking.
 895  */
 896 void
 897 closeall(uf_info_t *fip)
 898 {
 899         int fd;
 900         file_t *fp;
 901         uf_entry_t *ufp;

 903         ufp = fip->fi_list;
 904         for (fd = 0; fd < fip->fi_nfiles; fd++, ufp++) {
 905                 if ((fp = ufp->uf_file) != NULL) {
 906                         ufp->uf_file = NULL;
 907                         if (ufp->uf_portfd != NULL) {
 908                                 portfd_t *pfd;
 909                                 /* remove event port association */
 910                                 pfd = ufp->uf_portfd;
 911                                 ufp->uf_portfd = NULL;
 912                                 port_close_fd(pfd);
 913                         }
 914                         ASSERT(ufp->uf_fpollinfo == NULL);
 915                         (void) closef(fp);
 916                 }
 917         }

 919         kmem_free(fip->fi_list, fip->fi_nfiles * sizeof (uf_entry_t));
 920         fip->fi_list = NULL;
 921         fip->fi_nfiles = 0;
 922         while (fip->fi_rlist != NULL) {
 923                 uf_rlist_t *urp = fip->fi_rlist;
 924                 fip->fi_rlist = urp->ur_next;
 925                 kmem_free(urp->ur_list, urp->ur_nfiles * sizeof (uf_entry_t));
 926                 kmem_free(urp, sizeof (uf_rlist_t));
 927         }
 928 }

 930 /*
 931  * Internal form of close.  Decrement reference count on file
 932  * structure.  Decrement reference count on the vnode following
 933  * removal of the referencing file structure.
 934  */
 935 int
 936 closef(file_t *fp)
 937 {
 938         vnode_t *vp;
 939         int error;
 940         int count;
 941         int flag;
 942         offset_t offset;

 944         /*
 945          * audit close of file (may be exit)
 946          */
 947         if (AU_AUDITING())
 948                 audit_closef(fp);
 949         ASSERT(MUTEX_NOT_HELD(&P_FINFO(curproc)->fi_lock));

 951         mutex_enter(&fp->f_tlock);

 953         ASSERT(fp->f_count > 0);

 955         count = fp->f_count--;
 956         flag = fp->f_flag;
 957         offset = fp->f_offset;

 959         vp = fp->f_vnode;
```

```
960         if (vp != NULL)
961                 VOP_IOCTL(vp, F_CLOSED, (intptr_t)ttoproc(curthread),
962                     FKIOCTL, kcred, NULL, NULL);
963 #endif /* ! codereview */
964
965         error = VOP_CLOSE(vp, flag, count, offset, fp->f_cred, NULL);
966
967         if (count > 1) {
968                 mutex_exit(&fp->f_tlock);
969                 return (error);
970         }
971         ASSERT(fp->f_count == 0);
972         mutex_exit(&fp->f_tlock);
973
974         /*
975          * If DTrace has getf() subroutines active, it will set dtrace_closef
976          * to point to code that implements a barrier with respect to probe
977          * context.  This must be called before the file_t is freed (and the
978          * vnode that it refers to is released) -- but it must be after the
979          * file_t has been removed from the uf_entry_t.  That is, there must
980          * be no way for a racing getf() in probe context to yield the fp that
981          * we're operating upon.
982          */
983         if (dtrace_closef != NULL)
984                 (*dtrace_closef)();
985
986         VN_RELE(vp);
987         /*
988          * deallocate resources to audit_data
989          */
990         if (audit_active)
991                 audit_unfalloc(fp);
992         crfree(fp->f_cred);
993         kmem_cache_free(file_cache, fp);
994         return (error);
995 }
996
997 /*
998  * This is a combination of ufalloc() and setf().
999  */
1000 int
1001 ufalloc_file(int start, file_t *fp)
1002 {
1003         proc_t *p = curproc;
1004         uf_info_t *fip = P_FINFO(p);
1005         int filelimit;
1006         uf_entry_t *ufp;
1007         int nfiles;
1008         int fd;
1009
1010         /*
1011          * Assertion is to convince the correctness of the following
1012          * assignment for filelimit after casting to int.
1013          */
1014         ASSERT(p->p_fno_ctl <= INT_MAX);
1015         filelimit = (int)p->p_fno_ctl;
1016
1017         for (;;) {
1018                 mutex_enter(&fip->fi_lock);
1019                 fd = fd_find(fip, start);
1020                 if (fd >= 0 && fd == fip->fi_badfd) {
1021                         start = fd + 1;
1022                         mutex_exit(&fip->fi_lock);
1023                         continue;
1024                 }
1025                 if ((uint_t)fd < filelimit)
```



```
1026                         break;
1027                 if (fd >= filelimit) {
1028                         mutex_exit(&fip->fi_lock);
1029                         mutex_enter(&p->p_lock);
1030                         (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1031                             p->p_rctls, p, RCA_SAFE);
1032                         mutex_exit(&p->p_lock);
1033                         return (-1);
1034                 }
1035                 /* fd_find() returned -1 */
1036                 nfiles = fip->fi_nfiles;
1037                 mutex_exit(&fip->fi_lock);
1038                 flist_grow(MAX(start, nfiles));
1039         }
1040
1041         UF_ENTER(ufp, fip, fd);
1042         fd_reserve(fip, fd, 1);
1043         ASSERT(ufp->uf_file == NULL);
1044         ufp->uf_file = fp;
1045         UF_EXIT(ufp);
1046         mutex_exit(&fip->fi_lock);
1047         return (fd);
1048 }
1049
1050 /*
1051  * Allocate a user file descriptor greater than or equal to "start".
1052  */
1053 int
1054 ufalloc(int start)
1055 {
1056         return (ufalloc_file(start, NULL));
1057 }
1058
1059 /*
1060  * Check that a future allocation of count fds on proc p has a good
1061  * chance of succeeding.  If not, do rctl processing as if we'd failed
1062  * the allocation.
1063  *
1064  * Our caller must guarantee that p cannot disappear underneath us.
1065  */
1066 int
1067 ufcanalloc(proc_t *p, uint_t count)
1068 {
1069         uf_info_t *fip = P_FINFO(p);
1070         int filelimit;
1071         int current;
1072
1073         if (count == 0)
1074                 return (1);
1075
1076         ASSERT(p->p_fno_ctl <= INT_MAX);
1077         filelimit = (int)p->p_fno_ctl;
1078
1079         mutex_enter(&fip->fi_lock);
1080         current = flist_nalloc(fip);            /* # of in-use descriptors */
1081         mutex_exit(&fip->fi_lock);
1082
1083         /*
1084          * If count is a positive integer, the worst that can happen is
1085          * an overflow to a negative value, which is caught by the >= 0 check.
1086          */
1087         current += count;
1088         if (count <= INT_MAX && current >= 0 && current <= filelimit)
1089                 return (1);
1090
1091         mutex_enter(&p->p_lock);
```

```
1092         (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1093             p->p_rctls, p, RCA_SAFE);
1094         mutex_exit(&p->p_lock);
1095         return (0);
1096 }

1098 /*
1099  * Allocate a user file descriptor and a file structure.
1100  * Initialize the descriptor to point at the file structure.
1101  * If fdp is NULL, the user file descriptor will not be allocated.
1102  */
1103 int
1104 falloc(vnode_t *vp, int flag, file_t **fpp, int *fdp)
1105 {
1106         file_t *fp;
1107         int fd;

1109         if (fdp) {
1110                 if ((fd = ufalloc(0)) == -1)
1111                         return (EMFILE);
1112         }
1113         fp = kmem_cache_alloc(file_cache, KM_SLEEP);
1114         /*
1115          * Note: falloc returns the fp locked
1116          */
1117         mutex_enter(&fp->f_tlock);
1118         fp->f_count = 1;
1119         fp->f_flag = (ushort_t)flag;
1120         fp->f_flag2 = (flag & (FSEARCH|FEXEC)) >> 16;
1121         fp->f_vnode = vp;
1122         fp->f_offset = 0;
1123         fp->f_audit_data = 0;
1124         crhold(fp->f_cred = CRED());
1125         /*
1126          * allocate resources to audit_data
1127          */
1128         if (audit_active)
1129                 audit_falloc(fp);
1130         *fpp = fp;
1131         if (fdp)
1132                 *fdp = fd;
1133         return (0);
1134 }

1136 /*ARGSUSED*/
1137 static int
1138 file_cache_constructor(void *buf, void *cdrarg, int kmflags)
1139 {
1140         file_t *fp = buf;

1142         mutex_init(&fp->f_tlock, NULL, MUTEX_DEFAULT, NULL);
1143         return (0);
1144 }

1146 /*ARGSUSED*/
1147 static void
1148 file_cache_destructor(void *buf, void *cdrarg)
1149 {
1150         file_t *fp = buf;

1152         mutex_destroy(&fp->f_tlock);
1153 }

1155 void
1156 finit()
1157 {
```



```
1158         file_cache = kmem_cache_create("file_cache", sizeof (file_t), 0,
1159             file_cache_constructor, file_cache_destructor, NULL, NULL, NULL, 0);
1160 }

1162 void
1163 unfalloc(file_t *fp)
1164 {
1165         ASSERT(MUTEX_HELD(&fp->f_tlock));
1166         if (--fp->f_count <= 0) {
1167                 /*
1168                  * deallocate resources to audit_data
1169                  */
1170                 if (audit_active)
1171                         audit_unfalloc(fp);
1172                 crfree(fp->f_cred);
1173                 mutex_exit(&fp->f_tlock);
1174                 kmem_cache_free(file_cache, fp);
1175         } else
1176                 mutex_exit(&fp->f_tlock);
1177 }

1179 /*
1180  * Given a file descriptor, set the user's
1181  * file pointer to the given parameter.
1182  */
1183 void
1184 setf(int fd, file_t *fp)
1185 {
1186         uf_info_t *fip = P_FINFO(curproc);
1187         uf_entry_t *ufp;

1189         if (AU_AUDITING())
1190                 audit_setf(fp, fd);

1192         if (fp == NULL) {
1193                 mutex_enter(&fip->fi_lock);
1194                 UF_ENTER(ufp, fip, fd);
1195                 fd_reserve(fip, fd, -1);
1196                 mutex_exit(&fip->fi_lock);
1197         } else {
1198                 UF_ENTER(ufp, fip, fd);
1199                 ASSERT(ufp->uf_busy);
1200         }
1201         ASSERT(ufp->uf_fpollinfo == NULL);
1202         ASSERT(ufp->uf_flag == 0);
1203         ufp->uf_file = fp;
1204         cv_broadcast(&ufp->uf_wanted_cv);
1205         UF_EXIT(ufp);
1206 }

1208 /*
1209  * Given a file descriptor, return the file table flags, plus,
1210  * if this is a socket in asynchronous mode, the FASYNC flag.
1211  * getf() may or may not have been called before calling f_getfl().
1212  */
1213 int
1214 f_getfl(int fd, int *flagp)
1215 {
1216         uf_info_t *fip = P_FINFO(curproc);
1217         uf_entry_t *ufp;
1218         file_t *fp;
1219         int error;

1221         if ((uint_t)fd >= fip->fi_nfiles)
1222                 error = EBADF;
1223         else {
```

```
1224                UF_ENTER(ufp, fip, fd);
1225                if ((fp = ufp->uf_file) == NULL)
1226                        error = EBADF;
1227                else {
1228                        vnode_t *vp = fp->f_vnode;
1229                        int flag = fp->f_flag | (fp->f_flag2 << 16);

1231                        /*
1232                         * BSD fcntl() FASYNC compatibility.
1233                         */
1234                        if (vp->v_type == VSOCK)
1235                                flag |= sock_getfasync(vp);
1236                        *flagp = flag;
1237                        error = 0;
1238                }
1239                UF_EXIT(ufp);
1240        }

1242        return (error);
1243 }

1245 /*
1246  * Given a file descriptor, return the user's file flags.
1247  * Force the FD_CLOEXEC flag for writable self-open /proc files.
1248  * getf() may or may not have been called before calling f_getfd_error().
1249  */
1250 int
1251 f_getfd_error(int fd, int *flagp)
1252 {
1253        uf_info_t *fip = P_FINFO(curproc);
1254        uf_entry_t *ufp;
1255        file_t *fp;
1256        int flag;
1257        int error;

1259        if ((uint_t)fd >= fip->fi_nfiles)
1260                error = EBADF;
1261        else {
1262                UF_ENTER(ufp, fip, fd);
1263                if ((fp = ufp->uf_file) == NULL)
1264                        error = EBADF;
1265                else {
1266                        flag = ufp->uf_flag;
1267                        if ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode))
1268                                flag |= FD_CLOEXEC;
1269                        *flagp = flag;
1270                        error = 0;
1271                }
1272                UF_EXIT(ufp);
1273        }

1275        return (error);
1276 }

1278 /*
1279  * getf() must have been called before calling f_getfd().
1280  */
1281 char
1282 f_getfd(int fd)
1283 {
1284        int flag = 0;
1285        (void) f_getfd_error(fd, &flag);
1286        return ((char)flag);
1287 }

1289 /*
```



```
1290  * Given a file descriptor and file flags, set the user's file flags.
1291  * At present, the only valid flag is FD_CLOEXEC.
1292  * getf() may or may not have been called before calling f_setfd_error().
1293  */
1294 int
1295 f_setfd_error(int fd, int flags)
1296 {
1297        uf_info_t *fip = P_FINFO(curproc);
1298        uf_entry_t *ufp;
1299        int error;

1301        if ((uint_t)fd >= fip->fi_nfiles)
1302                error = EBADF;
1303        else {
1304                UF_ENTER(ufp, fip, fd);
1305                if (ufp->uf_file == NULL)
1306                        error = EBADF;
1307                else {
1308                        ufp->uf_flag = flags & FD_CLOEXEC;
1309                        error = 0;
1310                }
1311                UF_EXIT(ufp);
1312        }
1313        return (error);
1314 }

1316 void
1317 f_setfd(int fd, char flags)
1318 {
1319        (void) f_setfd_error(fd, flags);
1320 }

1322 #define BADFD_MIN       3
1323 #define BADFD_MAX       255

1325 /*
1326  * Attempt to allocate a file descriptor which is bad and which
1327  * is "poison" to the application.  It cannot be closed (except
1328  * on exec), allocated for a different use, etc.
1329  */
1330 int
1331 f_badfd(int start, int *fdp, int action)
1332 {
1333        int fdr;
1334        int badfd;
1335        uf_info_t *fip = P_FINFO(curproc);

1337 #ifdef _LP64
1338        /* No restrictions on 64 bit _file */
1339        if (get_udatamodel() != DATAMODEL_ILP32)
1340                return (EINVAL);
1341 #endif

1343        if (start > BADFD_MAX || start < BADFD_MIN)
1344                return (EINVAL);

1346        if (action >= NSIG || action < 0)
1347                return (EINVAL);

1349        mutex_enter(&fip->fi_lock);
1350        badfd = fip->fi_badfd;
1351        mutex_exit(&fip->fi_lock);

1353        if (badfd != -1)
1354                return (EAGAIN);
```

```
1356         fdr = ufalloc(start);

1358         if (fdr > BADFD_MAX) {
1359                 setf(fdr, NULL);
1360                 return (EMFILE);
1361         }
1362         if (fdr < 0)
1363                 return (EMFILE);

1365         mutex_enter(&fip->fi_lock);
1366         if (fip->fi_badfd != -1) {
1367                 /* Lost race */
1368                 mutex_exit(&fip->fi_lock);
1369                 setf(fdr, NULL);
1370                 return (EAGAIN);
1371         }
1372         fip->fi_action = action;
1373         fip->fi_badfd = fdr;
1374         mutex_exit(&fip->fi_lock);
1375         setf(fdr, NULL);

1377         *fdp = fdr;

1379         return (0);
1380 }

1382 /*
1383  * Allocate a file descriptor and assign it to the vnode "*vpp",
1384  * performing the usual open protocol upon it and returning the
1385  * file descriptor allocated.  It is the responsibility of the
1386  * caller to dispose of "*vpp" if any error occurs.
1387  */
1388 int
1389 fassign(vnode_t **vpp, int mode, int *fdp)
1390 {
1391         file_t *fp;
1392         int error;
1393         int fd;

1395         if (error = falloc((vnode_t *)NULL, mode, &fp, &fd))
1396                 return (error);
1397         if (error = VOP_OPEN(vpp, mode, fp->f_cred, NULL)) {
1398                 setf(fd, NULL);
1399                 unfalloc(fp);
1400                 return (error);
1401         }
1402         fp->f_vnode = *vpp;
1403         mutex_exit(&fp->f_tlock);
1404         /*
1405          * Fill in the slot falloc reserved.
1406          */
1407         setf(fd, fp);
1408         *fdp = fd;
1409         return (0);
1410 }

1412 /*
1413  * When a process forks it must increment the f_count of all file pointers
1414  * since there is a new process pointing at them.  fcnt_add(fip, 1) does this.
1415  * Since we are called when there is only 1 active lwp we don't need to
1416  * hold fi_lock or any uf_lock.  If the fork fails, fork_fail() calls
1417  * fcnt_add(fip, -1) to restore the counts.
1418  */
1419 void
1420 fcnt_add(uf_info_t *fip, int incr)
1421 {
```



```
1422         int i;
1423         uf_entry_t *ufp;
1424         file_t *fp;

1426         ufp = fip->fi_list;
1427         for (i = 0; i < fip->fi_nfiles; i++, ufp++) {
1428                 if ((fp = ufp->uf_file) != NULL) {
1429                         mutex_enter(&fp->f_tlock);
1430                         ASSERT((incr == 1 && fp->f_count >= 1) ||
1431                             (incr == -1 && fp->f_count >= 2));
1432                         fp->f_count += incr;
1433                         mutex_exit(&fp->f_tlock);
1434                 }
1435         }
1436 }

1438 /*
1439  * This is called from exec to close all fd's that have the FD_CLOEXEC flag
1440  * set and also to close all self-open for write /proc file descriptors.
1441  */
1442 void
1443 close_exec(uf_info_t *fip)
1444 {
1445         int fd;
1446         file_t *fp;
1447         fpollinfo_t *fpip;
1448         uf_entry_t *ufp;
1449         portfd_t *pfd;

1451         ufp = fip->fi_list;
1452         for (fd = 0; fd < fip->fi_nfiles; fd++, ufp++) {
1453                 if ((fp = ufp->uf_file) != NULL &&
1454                     ((ufp->uf_flag & FD_CLOEXEC) ||
1455                     ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode)))) {
1456                         fpip = ufp->uf_fpollinfo;
1457                         mutex_enter(&fip->fi_lock);
1458                         mutex_enter(&ufp->uf_lock);
1459                         fd_reserve(fip, fd, -1);
1460                         mutex_exit(&fip->fi_lock);
1461                         ufp->uf_file = NULL;
1462                         ufp->uf_fpollinfo = NULL;
1463                         ufp->uf_flag = 0;
1464                         /*
1465                          * We may need to cleanup some cached poll states
1466                          * in t_pollstate before the fd can be reused. It
1467                          * is important that we don't access a stale thread
1468                          * structure. We will do the cleanup in two
1469                          * phases to avoid deadlock and holding uf_lock for
1470                          * too long. In phase 1, hold the uf_lock and call
1471                          * pollblockexit() to set state in t_pollstate struct
1472                          * so that a thread does not exit on us. In phase 2,
1473                          * we drop the uf_lock and call pollcacheclean().
1474                          */
1475                         pfd = ufp->uf_portfd;
1476                         ufp->uf_portfd = NULL;
1477                         if (fpip != NULL)
1478                                 pollblockexit(fpip);
1479                         mutex_exit(&ufp->uf_lock);
1480                         if (fpip != NULL)
1481                                 pollcacheclean(fpip, fd);
1482                         if (pfd)
1483                                 port_close_fd(pfd);
1484                         (void) closef(fp);
1485                 }
1486         }
```

```
1488         /* Reset bad fd */
1489         fip->fi_badfd = -1;
1490         fip->fi_action = -1;
1491 }

1493 /*
1494  * Utility function called by most of the *at() system call interfaces.
1495  *
1496  * Generate a starting vnode pointer for an (fd, path) pair where 'fd'
1497  * is an open file descriptor for a directory to be used as the starting
1498  * point for the lookup of the relative pathname 'path' (or, if path is
1499  * NULL, generate a vnode pointer for the direct target of the operation).
1500  *
1501  * If we successfully return a non-NULL startvp, it has been the target
1502  * of VN_HOLD() and the caller must call VN_RELE() on it.
1503  */
1504 int
1505 fgetstartvp(int fd, char *path, vnode_t **startvpp)
1506 {
1507         vnode_t         *startvp;
1508         file_t          *startfp;
1509         char            startchar;

1511         if (fd == AT_FDCWD && path == NULL)
1512                 return (EFAULT);

1514         if (fd == AT_FDCWD) {
1515                 /*
1516                  * Start from the current working directory.
1517                  */
1518                 startvp = NULL;
1519         } else {
1520                 if (path == NULL)
1521                         startchar = '\0';
1522                 else if (copyin(path, &startchar, sizeof (char)))
1523                         return (EFAULT);

1525                 if (startchar == '/') {
1526                         /*
1527                          * 'path' is an absolute pathname.
1528                          */
1529                         startvp = NULL;
1530                 } else {
1531                         /*
1532                          * 'path' is a relative pathname or we will
1533                          * be applying the operation to 'fd' itself.
1534                          */
1535                         if ((startfp = getf(fd)) == NULL)
1536                                 return (EBADF);
1537                         startvp = startfp->f_vnode;
1538                         VN_HOLD(startvp);
1539                         releasef(fd);
1540                 }
1541         }
1542         *startvpp = startvp;
1543         return (0);
1544 }

1546 /*
1547  * Called from fchownat() and fchmodat() to set ownership and mode.
1548  * The contents of *vap must be set before calling here.
1549  */
1550 int
1551 fsetattrat(int fd, char *path, int flags, struct vattr *vap)
1552 {
1553         vnode_t         *startvp;
```



```
1554         vnode_t         *vp;
1555         int             error;

1557         /*
1558          * Since we are never called to set the size of a file, we don't
1559          * need to check for non-blocking locks (via nbl_need_check(vp)).
1560          */
1561         ASSERT(!(vap->va_mask & AT_SIZE));

1563         if ((error = fgetstartvp(fd, path, &startvp)) != 0)
1564                 return (error);
1565         if (AU_AUDITING() && startvp != NULL)
1566                 audit_setfsat_path(1);

1568         /*
1569          * Do lookup for fchownat/fchmodat when path not NULL
1570          */
1571         if (path != NULL) {
1572                 if (error = lookupnameat(path, UIO_USERSPACE,
1573                     (flags == AT_SYMLINK_NOFOLLOW) ?
1574                     NO_FOLLOW : FOLLOW,
1575                     NULLVPP, &vp, startvp)) {
1576                         if (startvp != NULL)
1577                                 VN_RELE(startvp);
1578                         return (error);
1579                 }
1580         } else {
1581                 vp = startvp;
1582                 ASSERT(vp);
1583                 VN_HOLD(vp);
1584         }

1586         if (vn_is_readonly(vp)) {
1587                 error = EROFS;
1588         } else {
1589                 error = VOP_SETATTR(vp, vap, 0, CRED(), NULL);
1590         }

1592         if (startvp != NULL)
1593                 VN_RELE(startvp);
1594         VN_RELE(vp);

1596         return (error);
1597 }

1599 /*
1600  * Return true if the given vnode is referenced by any
1601  * entry in the current process's file descriptor table.
1602  */
1603 int
1604 fisopen(vnode_t *vp)
1605 {
1606         int fd;
1607         file_t *fp;
1608         vnode_t *ovp;
1609         uf_info_t *fip = P_FINFO(curproc);
1610         uf_entry_t *ufp;

1612         mutex_enter(&fip->fi_lock);
1613         for (fd = 0; fd < fip->fi_nfiles; fd++) {
1614                 UF_ENTER(ufp, fip, fd);
1615                 if ((fp = ufp->uf_file) != NULL &&
1616                     (ovp = fp->f_vnode) != NULL && VN_CMP(vp, ovp)) {
1617                         UF_EXIT(ufp);
1618                         mutex_exit(&fip->fi_lock);
1619                         return (1);
```

```
1620                 }
1621                 UF_EXIT(ufp);
1622         }
1623         mutex_exit(&fip->fi_lock);
1624         return (0);
1625 }

1627 /*
1628  * Return zero if at least one file currently open (by curproc) shouldn't be
1629  * allowed to change zones.
1630  */
1631 int
1632 files_can_change_zones(void)
1633 {
1634         int fd;
1635         file_t *fp;
1636         uf_info_t *fip = P_FINFO(curproc);
1637         uf_entry_t *ufp;

1639         mutex_enter(&fip->fi_lock);
1640         for (fd = 0; fd < fip->fi_nfiles; fd++) {
1641                 UF_ENTER(ufp, fip, fd);
1642                 if ((fp = ufp->uf_file) != NULL &&
1643                     !vn_can_change_zones(fp->f_vnode)) {
1644                         UF_EXIT(ufp);
1645                         mutex_exit(&fip->fi_lock);
1646                         return (0);
1647                 }
1648                 UF_EXIT(ufp);
1649         }
1650         mutex_exit(&fip->fi_lock);
1651         return (1);
1652 }

1654 #ifdef DEBUG

1656 /*
1657  * The following functions are only used in ASSERT()s elsewhere.
1658  * They do not modify the state of the system.
1659  */

1661 /*
1662  * Return true (1) if the current thread is in the fpollinfo
1663  * list for this file descriptor, else false (0).
1664  */
1665 static int
1666 curthread_in_plist(uf_entry_t *ufp)
1667 {
1668         fpollinfo_t *fpip;

1670         ASSERT(MUTEX_HELD(&ufp->uf_lock));
1671         for (fpip = ufp->uf_fpollinfo; fpip; fpip = fpip->fp_next)
1672                 if (fpip->fp_thread == curthread)
1673                         return (1);
1674         return (0);
1675 }

1677 /*
1678  * Sanity check to make sure that after lwp_exit(),
1679  * curthread does not appear on any fd's fpollinfo list.
1680  */
1681 void
1682 checkfpollinfo(void)
1683 {
1684         int fd;
1685         uf_info_t *fip = P_FINFO(curproc);
```



```
1686         uf_entry_t *ufp;

1688         mutex_enter(&fip->fi_lock);
1689         for (fd = 0; fd < fip->fi_nfiles; fd++) {
1690                 UF_ENTER(ufp, fip, fd);
1691                 ASSERT(!curthread_in_plist(ufp));
1692                 UF_EXIT(ufp);
1693         }
1694         mutex_exit(&fip->fi_lock);
1695 }

1697 /*
1698  * Return true (1) if the current thread is in the fpollinfo
1699  * list for this file descriptor, else false (0).
1700  * This is the same as curthread_in_plist(),
1701  * but is called w/o holding uf_lock.
1702  */
1703 int
1704 infpollinfo(int fd)
1705 {
1706         uf_info_t *fip = P_FINFO(curproc);
1707         uf_entry_t *ufp;
1708         int rc;

1710         UF_ENTER(ufp, fip, fd);
1711         rc = curthread_in_plist(ufp);
1712         UF_EXIT(ufp);
1713         return (rc);
1714 }

1716 #endif  /* DEBUG */

1718 /*
1719  * Add the curthread to fpollinfo list, meaning this fd is currently in the
1720  * thread's poll cache. Each lwp polling this file descriptor should call
1721  * this routine once.
1722  */
1723 void
1724 addfpollinfo(int fd)
1725 {
1726         struct uf_entry *ufp;
1727         fpollinfo_t *fpip;
1728         uf_info_t *fip = P_FINFO(curproc);

1730         fpip = kmem_zalloc(sizeof (fpollinfo_t), KM_SLEEP);
1731         fpip->fp_thread = curthread;
1732         UF_ENTER(ufp, fip, fd);
1733         /*
1734          * Assert we are not already on the list, that is, that
1735          * this lwp did not call addfpollinfo twice for the same fd.
1736          */
1737         ASSERT(!curthread_in_plist(ufp));
1738         /*
1739          * addfpollinfo is always done inside the getf/releasef pair.
1740          */
1741         ASSERT(ufp->uf_refcnt >= 1);
1742         fpip->fp_next = ufp->uf_fpollinfo;
1743         ufp->uf_fpollinfo = fpip;
1744         UF_EXIT(ufp);
1745 }

1747 /*
1748  * Delete curthread from fpollinfo list if it is there.
1749  */
1750 void
1751 delfpollinfo(int fd)
```

```
1752 {
1753         struct uf_entry *ufp;
1754         struct fpollinfo *fpip;
1755         struct fpollinfo **fpipp;
1756         uf_info_t *fip = P_FINFO(curproc);

1758         UF_ENTER(ufp, fip, fd);
1759         for (fpipp = &ufp->uf_fpollinfo;
1760             (fpip = *fpipp) != NULL;
1761             fpipp = &fpip->fp_next) {
1762                 if (fpip->fp_thread == curthread) {
1763                         *fpipp = fpip->fp_next;
1764                         kmem_free(fpip, sizeof (fpollinfo_t));
1765                         break;
1766                 }
1767         }
1768         /*
1769          * Assert that we are not still on the list, that is, that
1770          * this lwp did not call addfpollinfo twice for the same fd.
1771          */
1772         ASSERT(!curthread_in_plist(ufp));
1773         UF_EXIT(ufp);
1774 }

1776 /*
1777  * fd is associated with a port. pfd is a pointer to the fd entry in the
1778  * cache of the port.
1779  */

1781 void
1782 addfd_port(int fd, portfd_t *pfd)
1783 {
1784         struct uf_entry *ufp;
1785         uf_info_t *fip = P_FINFO(curproc);

1787         UF_ENTER(ufp, fip, fd);
1788         /*
1789          * addfd_port is always done inside the getf/releasef pair.
1790          */
1791         ASSERT(ufp->uf_refcnt >= 1);
1792         if (ufp->uf_portfd == NULL) {
1793                 /* first entry */
1794                 ufp->uf_portfd = pfd;
1795                 pfd->pfd_next = NULL;
1796         } else {
1797                 pfd->pfd_next = ufp->uf_portfd;
1798                 ufp->uf_portfd = pfd;
1799                 pfd->pfd_next->pfd_prev = pfd;
1800         }
1801         UF_EXIT(ufp);
1802 }

1804 void
1805 delfd_port(int fd, portfd_t *pfd)
1806 {
1807         struct uf_entry *ufp;
1808         uf_info_t *fip = P_FINFO(curproc);

1810         UF_ENTER(ufp, fip, fd);
1811         /*
1812          * delfd_port is always done inside the getf/releasef pair.
1813          */
1814         ASSERT(ufp->uf_refcnt >= 1);
1815         if (ufp->uf_portfd == pfd) {
1816                 /* remove first entry */
1817                 ufp->uf_portfd = pfd->pfd_next;
```



```
1818         } else {
1819                 pfd->pfd_prev->pfd_next = pfd->pfd_next;
1820                 if (pfd->pfd_next != NULL)
1821                         pfd->pfd_next->pfd_prev = pfd->pfd_prev;
1822         }
1823         UF_EXIT(ufp);
1824 }

1826 static void
1827 port_close_fd(portfd_t *pfd)
1828 {
1829         portfd_t        *pfdn;

1831         /*
1832          * At this point, no other thread should access
1833          * the portfd_t list for this fd. The uf_file, uf_portfd
1834          * pointers in the uf_entry_t struct for this fd would
1835          * be set to NULL.
1836          */
1837         for (; pfd != NULL; pfd = pfdn) {
1838                 pfdn = pfd->pfd_next;
1839                 port_close_pfd(pfd);
1840         }
1841 }
```

```
**********************************************************
   37125 Sun Aug  9 12:48:01 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_
```

```
**********************************************************
  217469 Sun Aug  9 12:48:02 2015
new/usr/src/uts/common/os/streamio.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) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


/*
 * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/signal.h>
#include <sys/stat.h>
#include <sys/proc.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/stream.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/tihdr.h>
#include <sys/var.h>
#include <sys/poll.h>
#include <sys/termio.h>
#include <sys/ttold.h>
#include <sys/systm.h>
#include <sys/uio.h>
#include <sys/cmn_err.h>
#include <sys/sad.h>
#include <sys/netstack.h>
#include <sys/priocntl.h>
#include <sys/jioctl.h>
#include <sys/procset.h>
#include <sys/session.h>
#include <sys/kmem.h>
#include <sys/filio.h>
#include <sys/vtrace.h>
#include <sys/debug.h>
#include <sys/strredir.h>
```



```
#include <sys/fs/fifonode.h>
#include <sys/fs/snode.h>
#include <sys/strlog.h>
#include <sys/strsun.h>
#include <sys/project.h>
#include <sys/kbio.h>
#include <sys/msio.h>
#include <sys/tty.h>
#include <sys/ptyvar.h>
#include <sys/vuid_event.h>
#include <sys/modctl.h>
#include <sys/sunddi.h>
#include <sys/sunldi_impl.h>
#include <sys/autoconf.h>
#include <sys/policy.h>
#include <sys/dld.h>
#include <sys/zone.h>
#include <c2/audit.h>
#include <sys/fcntl.h>
#endif /* ! codereview */

/*
 * This define helps improve the readability of streams code while
 * still maintaining a very old streams performance enhancement.  The
 * performance enhancement basically involved having all callers
 * of straccess() perform the first check that straccess() will do
 * locally before actually calling straccess().  (There by reducing
 * the number of unnecessary calls to straccess().)
 */
#define i_straccess(x, y)       ((stp->sd_sidp == NULL) ? 0 : \
                                    (stp->sd_vnode->v_type == VFIFO) ? 0 : \
                                    straccess((x), (y)))

/*
 * what is mblk_pull_len?
 *
 * If a streams message consists of many short messages,
 * a performance degradation occurs from copyout overhead.
 * To decrease the per mblk overhead, messages that are
 * likely to consist of many small mblks are pulled up into
 * one continuous chunk of memory.
 *
 * To avoid the processing overhead of examining every
 * mblk, a quick heuristic is used. If the first mblk in
 * the message is shorter than mblk_pull_len, it is likely
 * that the rest of the mblk will be short.
 *
 * This heuristic was decided upon after performance tests
 * indicated that anything more complex slowed down the main
 * code path.
 */
#define MBLK_PULL_LEN 64
uint32_t mblk_pull_len = MBLK_PULL_LEN;

/*
 * The sgttyb_handling flag controls the handling of the old BSD
 * TIOCGETP, TIOCSETP, and TIOCSETN ioctls as follows:
 *
 * 0 - Emit no warnings at all and retain old, broken behavior.
 * 1 - Emit no warnings and silently handle new semantics.
 * 2 - Send cmn_err(CE_NOTE) when either TIOCSETP or TIOCSETN is used
 *     (once per system invocation).  Handle with new semantics.
 * 3 - Send SIGSYS when any TIOCGETP, TIOCSETP, or TIOCSETN call is
 *     made (so that offenders drop core and are easy to debug).
 *
 * 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 *);

/*
 * 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, 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
         * driver in the stream.  Otherwise create the stream.
         */
```

```
260         TRACE_1(TR_FAC_STREAMS_FR, TR_STROPEN, "stropen:%p", vp);
261 retry:
262         mutex_enter(&vp->v_lock);
263         if ((stp = vp->v_stream) != NULL) {
265                 /*
266                  * Waiting for stream to be created to device
267                  * due to another open.
268                  */
269                 mutex_exit(&vp->v_lock);
271                 if (STRMATED(stp)) {
272                         struct stdata *strmatep = stp->sd_mate;
274                         STRLOCKMATES(stp);
275                         if (strmatep->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
276                                 if (flag & (FNDELAY|FNONBLOCK)) {
277                                         error = EAGAIN;
278                                         mutex_exit(&strmatep->sd_lock);
279                                         goto ckreturn;
280                                 }
281                                 mutex_exit(&stp->sd_lock);
282                                 if (!cv_wait_sig(&strmatep->sd_monitor,
283                                     &strmatep->sd_lock)) {
284                                         error = EINTR;
285                                         mutex_exit(&strmatep->sd_lock);
286                                         mutex_enter(&stp->sd_lock);
287                                         goto ckreturn;
288                                 }
289                                 mutex_exit(&strmatep->sd_lock);
290                                 goto retry;
291                         }
292                         if (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
293                                 if (flag & (FNDELAY|FNONBLOCK)) {
294                                         error = EAGAIN;
295                                         mutex_exit(&strmatep->sd_lock);
296                                         goto ckreturn;
297                                 }
298                                 mutex_exit(&strmatep->sd_lock);
299                                 if (!cv_wait_sig(&stp->sd_monitor,
300                                     &stp->sd_lock)) {
301                                         error = EINTR;
302                                         goto ckreturn;
303                                 }
304                                 mutex_exit(&stp->sd_lock);
305                                 goto retry;
306                         }
308                         if (stp->sd_flag & (STRDERR|STWRERR)) {
309                                 error = EIO;
310                                 mutex_exit(&strmatep->sd_lock);
311                                 goto ckreturn;
312                         }
314                         stp->sd_flag |= STWOPEN;
315                         STRUNLOCKMATES(stp);
316                 } else {
317                         mutex_enter(&stp->sd_lock);
318                         if (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
319                                 if (flag & (FNDELAY|FNONBLOCK)) {
320                                         error = EAGAIN;
321                                         goto ckreturn;
322                                 }
323                                 if (!cv_wait_sig(&stp->sd_monitor,
324                                     &stp->sd_lock)) {
325                                         error = EINTR;
```



```
326                                         goto ckreturn;
327                                 }
328                                 mutex_exit(&stp->sd_lock);
329                                 goto retry;  /* could be clone! */
330                         }
332                         if (stp->sd_flag & (STRDERR|STWRERR)) {
333                                 error = EIO;
334                                 goto ckreturn;
335                         }
337                         stp->sd_flag |= STWOPEN;
338                         mutex_exit(&stp->sd_lock);
339                 }
341                 /*
342                  * Open all modules and devices down stream to notify
343                  * that another user is streaming.  For modules, set the
344                  * last argument to MODOPEN and do not pass any open flags.
345                  * Ignore dummydev since this is not the first open.
346                  */
347                 claimstr(stp->sd_wrq);
348                 qp = stp->sd_wrq;
349                 while (_SAMESTR(qp)) {
350                         qp = qp->q_next;
351                         if ((error = qreopen(_RD(qp), devp, flag, crp)) != 0)
352                                 break;
353                 }
354                 releasestr(stp->sd_wrq);
355                 mutex_enter(&stp->sd_lock);
356                 stp->sd_flag &= ~(STRHUP|STWOPEN|STRDERR|STWRERR);
357                 stp->sd_rerror = 0;
358                 stp->sd_werror = 0;
359 ckreturn:
360                 cv_broadcast(&stp->sd_monitor);
361                 mutex_exit(&stp->sd_lock);
362                 return (error);
363         }
365         /*
366          * This vnode isn't streaming.  SPECFS already
367          * checked for multiple vnodes pointing to the
368          * same stream, so create a stream to the driver.
369          */
370         qp = allocq();
371         stp = shalloc(qp);
373         /*
374          * Initialize stream head.  shalloc() has given us
375          * exclusive access, and we have the vnode locked;
376          * we can do whatever we want with stp.
377          */
378         stp->sd_flag = STWOPEN;
379         stp->sd_siglist = NULL;
380         stp->sd_pollist.ph_list = NULL;
381         stp->sd_sigflags = 0;
382         stp->sd_mark = NULL;
383         stp->sd_closetime = STRTIMOUT;
384         stp->sd_sidp = NULL;
385         stp->sd_pgidp = NULL;
386         stp->sd_vnode = vp;
387         stp->sd_rerror = 0;
388         stp->sd_werror = 0;
389         stp->sd_wroff = 0;
390         stp->sd_tail = 0;
391         stp->sd_iocblk = NULL;
```

```
392         stp->sd_cmdblk = NULL;
393         stp->sd_pushcnt = 0;
394         stp->sd_qn_minpsz = 0;
395         stp->sd_qn_maxpsz = INFPSZ - 1; /* used to check for initialization */
396         stp->sd_maxblk = INFPSZ;
397         qp->q_ptr = _WR(qp)->q_ptr = stp;
398         STREAM(qp) = STREAM(_WR(qp)) = stp;
399         vp->v_stream = stp;
400         mutex_exit(&vp->v_lock);
401         if (vp->v_type == VFIFO) {
402                 stp->sd_flag |= OLDNDELAY;
403                 /*
404                  * This means, both for pipes and fifos
405                  * strwrite will send SIGPIPE if the other
406                  * end is closed. For putmsg it depends
407                  * on whether it is a XPG4_2 application
408                  * or not
409                  */
410                 stp->sd_wput_opt = SW_SIGPIPE;

412                 /* setq might sleep in kmem_alloc - avoid holding locks. */
413                 setq(qp, &fifo_strdata, &fifo_stwdata, NULL, QMTSAFE,
414                     SQ_CI|SQ_CO, B_FALSE);

416                 set_qend(qp);
417                 stp->sd_strtab = fifo_getinfo();
418                 _WR(qp)->q_nfsrv = _WR(qp);
419                 qp->q_nfsrv = qp;
420                 /*
421                  * Wake up others that are waiting for stream to be created.
422                  */
423                 mutex_enter(&stp->sd_lock);
424                 /*
425                  * nothing is be pushed on stream yet, so
426                  * optimized stream head packetsizes are just that
427                  * of the read queue
428                  */
429                 stp->sd_qn_minpsz = qp->q_minpsz;
430                 stp->sd_qn_maxpsz = qp->q_maxpsz;
431                 stp->sd_flag &= ~STWOPEN;
432                 goto fifo_opendone;
433         }
434         /* setq might sleep in kmem_alloc - avoid holding locks. */
435         setq(qp, &strdata, &stwdata, NULL, QMTSAFE, SQ_CI|SQ_CO, B_FALSE);

437         set_qend(qp);

439         /*
440          * Open driver and create stream to it (via qattach).
441          */
442         savedev = *devp;
443         cloneopen = (getmajor(*devp) == clone_major);
444         if ((error = qattach(qp, devp, flag, crp, NULL, B_FALSE)) != 0) {
445                 mutex_enter(&vp->v_lock);
446                 vp->v_stream = NULL;
447                 mutex_exit(&vp->v_lock);
448                 mutex_enter(&stp->sd_lock);
449                 cv_broadcast(&stp->sd_monitor);
450                 mutex_exit(&stp->sd_lock);
451                 freeq(_RD(qp));
452                 shfree(stp);
453                 return (error);
454         }
455         /*
456          * Set sd_strtab after open in order to handle clonable drivers
457          */
```



```
458         stp->sd_strtab = STREAMSTAB(getmajor(*devp));

460         /*
461          * Historical note: dummydev used to be be prior to the initial
462          * open (via qattach above), which made the value seen
463          * inconsistent between an I_PUSH and an autopush of a module.
464          */
465         dummydev = *devp;

467         /*
468          * For clone open of old style (Q not associated) network driver,
469          * push DRMODNAME module to handle DL_ATTACH/DL_DETACH
470          */
471         brq = _RD(_WR(qp)->q_next);
472         major = getmajor(*devp);
473         if (push_drcompat && cloneopen && NETWORK_DRV(major) &&
474             ((brq->q_flag & _QASSOCIATED) == 0)) {
475                 if (push_mod(qp, &dummydev, stp, DRMODNAME, 0, crp, 0) != 0)
476                         cmn_err(CE_WARN, "cannot push " DRMODNAME
477                             " streams module");
478         }

480         if (!NETWORK_DRV(major)) {
481                 savedev = *devp;
482         } else {
483                 /*
484                  * For network devices, process differently based on the
485                  * return value from dld_autopush():
486                  *
487                  *   0: the passed-in device points to a GLDv3 datalink with
488                  *   per-link autopush configuration; use that configuration
489                  *   and ignore any per-driver autopush configuration.
490                  *
491                  *   1: the passed-in device points to a physical GLDv3
492                  *   datalink without per-link autopush configuration.  The
493                  *   passed in device was changed to refer to the actual
494                  *   physical device (if it's not already); we use that new
495                  *   device to look up any per-driver autopush configuration.
496                  *
497                  *   -1: neither of the above cases applied; use the initial
498                  *   device to look up any per-driver autopush configuration.
499                  */
500                 switch (dld_autopush(&savedev, &dlap)) {
501                 case 0:
502                         zoneid = crgetzoneid(crp);
503                         for (s = 0; s < dlap.dap_npush; s++) {
504                                 error = push_mod(qp, &dummydev, stp,
505                                     dlap.dap_aplist[s], dlap.dap_anchor, crp,
506                                     zoneid);
507                                 if (error != 0)
508                                         break;
509                         }
510                         goto opendone;
511                 case 1:
512                         break;
513                 case -1:
514                         savedev = *devp;
515                         break;
516                 }
517         }
518         /*
519          * Find the autopush configuration based on "savedev". Start with the
520          * global zone. If not found check in the local zone.
521          */
522         zoneid = GLOBAL_ZONEID;
523 retryap:
```

```
524         ss = netstack_find_by_stackid(zoneid_to_netstackid(zoneid))->
525             netstack_str;
526         if ((ap = sad_ap_find_by_dev(savedev, ss)) == NULL) {
527                 netstack_rele(ss->ss_netstack);
528                 if (zoneid == GLOBAL_ZONEID) {
529                         /*
530                          * None found. Also look in the zone's autopush table.
531                          */
532                         zoneid = crgetzoneid(crp);
533                         if (zoneid != GLOBAL_ZONEID)
534                                 goto retryap;
535                 }
536                 goto opendone;
537         }
538         anchor = ap->ap_anchor;
539         zoneid = crgetzoneid(crp);
540         for (s = 0; s < ap->ap_npush; s++) {
541                 error = push_mod(qp, &dummydev, stp, ap->ap_list[s],
542                     anchor, crp, zoneid);
543                 if (error != 0)
544                         break;
545         }
546         sad_ap_rele(ap, ss);
547         netstack_rele(ss->ss_netstack);

549 opendone:

551         /*
552          * let specfs know that open failed part way through
553          */
554         if (error) {
555                 mutex_enter(&stp->sd_lock);
556                 stp->sd_flag |= STREOPENFAIL;
557                 mutex_exit(&stp->sd_lock);
558         }

560         /*
561          * Wake up others that are waiting for stream to be created.
562          */
563         mutex_enter(&stp->sd_lock);
564         stp->sd_flag &= ~STWOPEN;

566         /*
567          * As a performance concern we are caching the values of
568          * q_minpsz and q_maxpsz of the module below the stream
569          * head in the stream head.
570          */
571         mutex_enter(QLOCK(stp->sd_wrq->q_next));
572         rmin = stp->sd_wrq->q_next->q_minpsz;
573         rmax = stp->sd_wrq->q_next->q_maxpsz;
574         mutex_exit(QLOCK(stp->sd_wrq->q_next));

576         /* do this processing here as a performance concern */
577         if (strmsgsz != 0) {
578                 if (rmax == INFPSZ)
579                         rmax = strmsgsz;
580                 else
581                         rmax = MIN(strmsgsz, rmax);
582         }

584         mutex_enter(QLOCK(stp->sd_wrq));
585         stp->sd_qn_minpsz = rmin;
586         stp->sd_qn_maxpsz = rmax;
587         mutex_exit(QLOCK(stp->sd_wrq));

589 fifo_opendone:
```



```
590         cv_broadcast(&stp->sd_monitor);
591         mutex_exit(&stp->sd_lock);
592         return (error);
593 }

595 static int strsink(queue_t *, mblk_t *);
596 static struct qinit deadrend = {
597         strsink, NULL, NULL, NULL, NULL, &strm_info, NULL
598 };
599 static struct qinit deadwend = {
600         NULL, NULL, NULL, NULL, NULL, &stwm_info, NULL
601 };

603 /*
604  * Close a stream.
605  * This is called from closef() on the last close of an open stream.
606  * Strclean() will already have removed the siglist and pollist
607  * information, so all that remains is to remove all multiplexor links
608  * for the stream, pop all the modules (and the driver), and free the
609  * stream structure.
610  */

612 int
613 strclose(struct vnode *vp, int flag, cred_t *crp)
614 {
615         struct stdata *stp;
616         queue_t *qp;
617         int rval;
618         int freestp = 1;
619         queue_t *rmq;

621         TRACE_1(TR_FAC_STREAMS_FR,
622             TR_STRCLOSE, "strclose:%p", vp);
623         ASSERT(vp->v_stream);

625         stp = vp->v_stream;
626         ASSERT(!(stp->sd_flag & STPLEX));
627         qp = stp->sd_wrq;

629         /*
630          * Needed so that strpoll will return non-zero for this fd.
631          * Note that with POLLNOERR STRHUP does still cause POLLHUP.
632          */
633         mutex_enter(&stp->sd_lock);
634         stp->sd_flag |= STRHUP;
635         mutex_exit(&stp->sd_lock);

637         /*
638          * If the registered process or process group did not have an
639          * open instance of this stream then strclean would not be
640          * called. Thus at the time of closing all remaining siglist entries
641          * are removed.
642          */
643         if (stp->sd_siglist != NULL)
644                 strcleanall(vp);

646         ASSERT(stp->sd_siglist == NULL);
647         ASSERT(stp->sd_sigflags == 0);

649         if (STRMATED(stp)) {
650                 struct stdata *strmatep = stp->sd_mate;
651                 int waited = 1;

653                 STRLOCKMATES(stp);
654                 while (waited) {
655                         waited = 0;
```

```
656                         while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
657                                 mutex_exit(&strmatep->sd_lock);
658                                 cv_wait(&stp->sd_monitor, &stp->sd_lock);
659                                 mutex_exit(&stp->sd_lock);
660                                 STRLOCKMATES(stp);
661                                 waited = 1;
662                         }
663                         while (strmatep->sd_flag &
664                             (STWOPEN|STRCLOSE|STRPLUMB)) {
665                                 mutex_exit(&stp->sd_lock);
666                                 cv_wait(&strmatep->sd_monitor,
667                                     &strmatep->sd_lock);
668                                 mutex_exit(&strmatep->sd_lock);
669                                 STRLOCKMATES(stp);
670                                 waited = 1;
671                         }
672                 }
673                 stp->sd_flag |= STRCLOSE;
674                 STRUNLOCKMATES(stp);
675         } else {
676                 mutex_enter(&stp->sd_lock);
677                 stp->sd_flag |= STRCLOSE;
678                 mutex_exit(&stp->sd_lock);
679         }

681         ASSERT(qp->q_first == NULL);    /* No more delayed write */

683         /* Check if an I_LINK was ever done on this stream */
684         if (stp->sd_flag & STRHASLINKS) {
685                 netstack_t *ns;
686                 str_stack_t *ss;

688                 ns = netstack_find_by_cred(crp);
689                 ASSERT(ns != NULL);
690                 ss = ns->netstack_str;
691                 ASSERT(ss != NULL);

693                 (void) munlinkall(stp, LINKCLOSE|LINKNORMAL, crp, &rval, ss);
694                 netstack_rele(ss->ss_netstack);
695         }

697         while (_SAMESTR(qp)) {
698                 /*
699                  * Holding sd_lock prevents q_next from changing in
700                  * this stream.
701                  */
702                 mutex_enter(&stp->sd_lock);
703                 if (!(flag & (FNDELAY|FNONBLOCK)) && (stp->sd_closetime > 0)) {

705                         /*
706                          * sleep until awakened by strwsrv() or timeout
707                          */
708                         for (;;) {
709                                 mutex_enter(QLOCK(qp->q_next));
710                                 if (!(qp->q_next->q_mblkcnt)) {
711                                         mutex_exit(QLOCK(qp->q_next));
712                                         break;
713                                 }
714                                 stp->sd_flag |= WSLEEP;

716                                 /* ensure strwsrv gets enabled */
717                                 qp->q_next->q_flag |= QWANTW;
718                                 mutex_exit(QLOCK(qp->q_next));
719                                 /* get out if we timed out or recv'd a signal */
720                                 if (str_cv_wait(&qp->q_wait, &stp->sd_lock,
721                                     stp->sd_closetime, 0) <= 0) {
```



```
722                                         break;
723                                 }
724                         }
725                         stp->sd_flag &= ~WSLEEP;
726                 }
727                 mutex_exit(&stp->sd_lock);

729                 rmq = qp->q_next;
730                 if (rmq->q_flag & QISDRV) {
731                         ASSERT(!_SAMESTR(rmq));
732                         wait_sq_svc(_RD(qp)->q_syncq);
733                 }

735                 qdetach(_RD(rmq), 1, flag, crp, B_FALSE);
736         }

738         /*
739          * Since we call pollwakeup in close() now, the poll list should
740          * be empty in most cases. The only exception is the layered devices
741          * (e.g. the console drivers with redirection modules pushed on top
742          * of it).  We have to do this after calling qdetach() because
743          * the redirection module won't have torn down the console
744          * redirection until after qdetach() has been invoked.
745          */
746         if (stp->sd_pollist.ph_list != NULL) {
747                 pollwakeup(&stp->sd_pollist, POLLERR);
748                 pollhead_clean(&stp->sd_pollist);
749         }
750         ASSERT(stp->sd_pollist.ph_list == NULL);
751         ASSERT(stp->sd_sidp == NULL);
752         ASSERT(stp->sd_pgidp == NULL);

754         /* Prevent qenable from re-enabling the stream head queue */
755         disable_svc(_RD(qp));

757         /*
758          * Wait until service procedure of each queue is
759          * run, if QINSERVICE is set.
760          */
761         wait_svc(_RD(qp));

763         /*
764          * Now, flush both queues.
765          */
766         flushq(_RD(qp), FLUSHALL);
767         flushq(qp, FLUSHALL);

769         /*
770          * If the write queue of the stream head is pointing to a
771          * read queue, we have a twisted stream.  If the read queue
772          * is alive, convert the stream head queues into a dead end.
773          * If the read queue is dead, free the dead pair.
774          */
775         if (qp->q_next && !_SAMESTR(qp)) {
776                 if (qp->q_next->q_qinfo == &deadrend) { /* half-closed pipe */
777                         flushq(qp->q_next, FLUSHALL); /* ensure no message */
778                         shfree(qp->q_next->q_stream);
779                         freeq(qp->q_next);
780                         freeq(_RD(qp));
781                 } else if (qp->q_next == _RD(qp)) {     /* fifo */
782                         freeq(_RD(qp));
783                 } else {                                /* pipe */
784                         freestp = 0;
785                         /*
786                          * The q_info pointers are never accessed when
787                          * SQLOCK is held.
```

```
788                          */
789                         ASSERT(qp->q_syncq == _RD(qp)->q_syncq);
790                         mutex_enter(SQLOCK(qp->q_syncq));
791                         qp->q_qinfo = &deadwend;
792                         _RD(qp)->q_qinfo = &deadrend;
793                         mutex_exit(SQLOCK(qp->q_syncq));
794                 }
795         } else {
796                 freeq(_RD(qp)); /* free stream head queue pair */
797         }
798
799         mutex_enter(&vp->v_lock);
800         if (stp->sd_iocblk) {
801                 if (stp->sd_iocblk != (mblk_t *)-1) {
802                         freemsg(stp->sd_iocblk);
803                 }
804                 stp->sd_iocblk = NULL;
805         }
806         stp->sd_vnode = NULL;
807         vp->v_stream = NULL;
808         mutex_exit(&vp->v_lock);
809         mutex_enter(&stp->sd_lock);
810         freemsg(stp->sd_cmdblk);
811         stp->sd_cmdblk = NULL;
812         stp->sd_flag &= ~STRCLOSE;
813         cv_broadcast(&stp->sd_monitor);
814         mutex_exit(&stp->sd_lock);
815
816         if (freestp)
817                 shfree(stp);
818         return (0);
819 }
820
821 static int
822 strsink(queue_t *q, mblk_t *bp)
823 {
824         struct copyresp *resp;
825
826         switch (bp->b_datap->db_type) {
827         case M_FLUSH:
828                 if ((*bp->b_rptr & FLUSHW) && !(bp->b_flag & MSGNOLOOP)) {
829                         *bp->b_rptr &= ~FLUSHR;
830                         bp->b_flag |= MSGNOLOOP;
831                         /*
832                          * Protect against the driver passing up
833                          * messages after it has done a qprocsoff.
834                          */
835                         if (_OTHERQ(q)->q_next == NULL)
836                                 freemsg(bp);
837                         else
838                                 qreply(q, bp);
839                 } else {
840                         freemsg(bp);
841                 }
842                 break;
843
844         case M_COPYIN:
845         case M_COPYOUT:
846                 if (bp->b_cont) {
847                         freemsg(bp->b_cont);
848                         bp->b_cont = NULL;
849                 }
850                 bp->b_datap->db_type = M_IOCDATA;
851                 bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
852                 resp = (struct copyresp *)bp->b_rptr;
853                 resp->cp_rval = (caddr_t)1;     /* failure */
```



```
854                 /*
855                  * Protect against the driver passing up
856                  * messages after it has done a qprocsoff.
857                  */
858                 if (_OTHERQ(q)->q_next == NULL)
859                         freemsg(bp);
860                 else
861                         qreply(q, bp);
862                 break;
863
864         case M_IOCTL:
865                 if (bp->b_cont) {
866                         freemsg(bp->b_cont);
867                         bp->b_cont = NULL;
868                 }
869                 bp->b_datap->db_type = M_IOCNAK;
870                 /*
871                  * Protect against the driver passing up
872                  * messages after it has done a qprocsoff.
873                  */
874                 if (_OTHERQ(q)->q_next == NULL)
875                         freemsg(bp);
876                 else
877                         qreply(q, bp);
878                 break;
879
880         default:
881                 freemsg(bp);
882                 break;
883         }
884
885         return (0);
886 }
887
888 /*
889  * Clean up after a process when it closes a stream.  This is called
890  * from closef for all closes, whereas strclose is called only for the
891  * last close on a stream.  The siglist is scanned for entries for the
892  * current process, and these are removed.
893  */
894 void
895 strclean(struct vnode *vp)
896 {
897         strsig_t *ssp, *pssp, *tssp;
898         stdata_t *stp;
899         int update = 0;
900
901         TRACE_1(TR_FAC_STREAMS_FR,
902             TR_STRCLEAN, "strclean:%p", vp);
903         stp = vp->v_stream;
904         pssp = NULL;
905         mutex_enter(&stp->sd_lock);
906         ssp = stp->sd_siglist;
907         while (ssp) {
908                 if (ssp->ss_pidp == curproc->p_pidp) {
909                         tssp = ssp->ss_next;
910                         if (pssp)
911                                 pssp->ss_next = tssp;
912                         else
913                                 stp->sd_siglist = tssp;
914                         mutex_enter(&pidlock);
915                         PID_RELE(ssp->ss_pidp);
916                         mutex_exit(&pidlock);
917                         kmem_free(ssp, sizeof (strsig_t));
918                         update = 1;
919                         ssp = tssp;
```

```
920                 } else {
921                         pssp = ssp;
922                         ssp = ssp->ss_next;
923                 }
924         }
925         if (update) {
926                 stp->sd_sigflags = 0;
927                 for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
928                         stp->sd_sigflags |= ssp->ss_events;
929         }
930         mutex_exit(&stp->sd_lock);
931 }

933 /*
934  * Used on the last close to remove any remaining items on the siglist.
935  * These could be present on the siglist due to I_ESETSIG calls that
936  * use process groups or processed that do not have an open file descriptor
937  * for this stream (Such entries would not be removed by strclean).
938  */
939 static void
940 strcleanall(struct vnode *vp)
941 {
942         strsig_t *ssp, *nssp;
943         stdata_t *stp;

945         stp = vp->v_stream;
946         mutex_enter(&stp->sd_lock);
947         ssp = stp->sd_siglist;
948         stp->sd_siglist = NULL;
949         while (ssp) {
950                 nssp = ssp->ss_next;
951                 mutex_enter(&pidlock);
952                 PID_RELE(ssp->ss_pidp);
953                 mutex_exit(&pidlock);
954                 kmem_free(ssp, sizeof (strsig_t));
955                 ssp = nssp;
956         }
957         stp->sd_sigflags = 0;
958         mutex_exit(&stp->sd_lock);
959 }

961 /*
962  * Retrieve the next message from the logical stream head read queue
963  * using either rwnext (if sync stream) or getq_noenab.
964  * It is the callers responsibility to call qbackenable after
965  * it is finished with the message. The caller should not call
966  * qbackenable until after any putback calls to avoid spurious backenabling.
967  */
968 mblk_t *
969 strget(struct stdata *stp, queue_t *q, struct uio *uiop, int first,
970     int *errorp)
971 {
972         mblk_t *bp;
973         int error;
974         ssize_t rbytes = 0;

976         /* Holding sd_lock prevents the read queue from changing  */
977         ASSERT(MUTEX_HELD(&stp->sd_lock));

979         if (uiop != NULL && stp->sd_struiordq != NULL &&
980             q->q_first == NULL &&
981             (!first || (stp->sd_wakeq & RSLEEP))) {
982                 /*
983                  * Stream supports rwnext() for the read side.
984                  * If this is the first time we're called by e.g. strread
985                  * only do the downcall if there is a deferred wakeup
```



```
986                  * (registered in sd_wakeq).
987                  */
988                 struiod_t uiod;

990                 if (first)
991                         stp->sd_wakeq &= ~RSLEEP;

993                 (void) uiodup(uiop, &uiod.d_uio, uiod.d_iov,
994                     sizeof (uiod.d_iov) / sizeof (*uiod.d_iov));
995                 uiod.d_mp = 0;
996                 /*
997                  * Mark that a thread is in rwnext on the read side
998                  * to prevent strrput from nacking ioctls immediately.
999                  * When the last concurrent rwnext returns
1000                 * the ioctls are nack'ed.
1001                 */
1002                ASSERT(MUTEX_HELD(&stp->sd_lock));
1003                stp->sd_struiodnak++;
1004                /*
1005                 * Note: rwnext will drop sd_lock.
1006                 */
1007                error = rwnext(q, &uiod);
1008                ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));
1009                mutex_enter(&stp->sd_lock);
1010                stp->sd_struiodnak--;
1011                while (stp->sd_struiodnak == 0 &&
1012                    ((bp = stp->sd_struionak) != NULL)) {
1013                        stp->sd_struionak = bp->b_next;
1014                        bp->b_next = NULL;
1015                        bp->b_datap->db_type = M_IOCNAK;
1016                        /*
1017                         * Protect against the driver passing up
1018                         * messages after it has done a qprocsoff.
1019                         */
1020                        if (_OTHERQ(q)->q_next == NULL)
1021                                freemsg(bp);
1022                        else {
1023                                mutex_exit(&stp->sd_lock);
1024                                qreply(q, bp);
1025                                mutex_enter(&stp->sd_lock);
1026                        }
1027                }
1028                ASSERT(MUTEX_HELD(&stp->sd_lock));
1029                if (error == 0 || error == EWOULDBLOCK) {
1030                        if ((bp = uiod.d_mp) != NULL) {
1031                                *errorp = 0;
1032                                ASSERT(MUTEX_HELD(&stp->sd_lock));
1033                                return (bp);
1034                        }
1035                        error = 0;
1036                } else if (error == EINVAL) {
1037                        /*
1038                         * The stream plumbing must have
1039                         * changed while we were away, so
1040                         * just turn off rwnext()s.
1041                         */
1042                        error = 0;
1043                } else if (error == EBUSY) {
1044                        /*
1045                         * The module might have data in transit using putnext
1046                         * Fall back on waiting + getq.
1047                         */
1048                        error = 0;
1049                } else {
1050                        *errorp = error;
1051                        ASSERT(MUTEX_HELD(&stp->sd_lock));
```

```
1052                        return (NULL);
1053                }
1054                /*
1055                 * Try a getq in case a rwnext() generated mblk
1056                 * has bubbled up via strrput().
1057                 */
1058        }
1059        *errorp = 0;
1060        ASSERT(MUTEX_HELD(&stp->sd_lock));

1062        /*
1063         * If we have a valid uio, try and use this as a guide for how
1064         * many bytes to retrieve from the queue via getq_noenab().
1065         * Doing this can avoid unneccesary counting of overlong
1066         * messages in putback(). We currently only do this for sockets
1067         * and only if there is no sd_rputdatafunc hook.
1068         *
1069         * The sd_rputdatafunc hook transforms the entire message
1070         * before any bytes in it can be given to a client. So, rbytes
1071         * must be 0 if there is a hook.
1072         */
1073        if ((uiop != NULL) && (stp->sd_vnode->v_type == VSOCK) &&
1074            (stp->sd_rputdatafunc == NULL))
1075                rbytes = uiop->uio_resid;

1077        return (getq_noenab(q, rbytes));
1078 }

1080 /*
1081  * Copy out the message pointed to by 'bp' into the uio pointed to by 'uiop'.
1082  * If the message does not fit in the uio the remainder of it is returned;
1083  * otherwise NULL is returned.  Any embedded zero-length mblk_t's are
1084  * consumed, even if uio_resid reaches zero.  On error, '*errorp' is set to
1085  * the error code, the message is consumed, and NULL is returned.
1086  */
1087 static mblk_t *
1088 struiocopyout(mblk_t *bp, struct uio *uiop, int *errorp)
1089 {
1090        int error;
1091        ptrdiff_t n;
1092        mblk_t *nbp;

1094        ASSERT(bp->b_wptr >= bp->b_rptr);

1096        do {
1097                if ((n = MIN(uiop->uio_resid, MBLKL(bp))) != 0) {
1098                        ASSERT(n > 0);

1100                        error = uiomove(bp->b_rptr, n, UIO_READ, uiop);
1101                        if (error != 0) {
1102                                freemsg(bp);
1103                                *errorp = error;
1104                                return (NULL);
1105                        }
1106                }

1108                bp->b_rptr += n;
1109                while (bp != NULL && (bp->b_rptr >= bp->b_wptr)) {
1110                        nbp = bp;
1111                        bp = bp->b_cont;
1112                        freeb(nbp);
1113                }
1114        } while (bp != NULL && uiop->uio_resid > 0);

1116        *errorp = 0;
1117        return (bp);
```



```
1118 }

1120 /*
1121  * Read a stream according to the mode flags in sd_flag:
1122  *
1123  * (default mode)               - Byte stream, msg boundaries are ignored
1124  * RD_MSGDIS (msg discard)      - Read on msg boundaries and throw away
1125  *                              any data remaining in msg
1126  * RD_MSGNODIS (msg non-discard) - Read on msg boundaries and put back
1127  *                              any remaining data on head of read queue
1128  *
1129  * Consume readable messages on the front of the queue until
1130  * ttolwp(curthread)->lwp_count
1131  * is satisfied, the readable messages are exhausted, or a message
1132  * boundary is reached in a message mode.  If no data was read and
1133  * the stream was not opened with the NDELAY flag, block until data arrives.
1134  * Otherwise return the data read and update the count.
1135  *
1136  * In default mode a 0 length message signifies end-of-file and terminates
1137  * a read in progress.  The 0 length message is removed from the queue
1138  * only if it is the only message read (no data is read).
1139  *
1140  * An attempt to read an M_PROTO or M_PCPROTO message results in an
1141  * EBADMSG error return, unless either RD_PROTDAT or RD_PROTDIS are set.
1142  * If RD_PROTDAT is set, M_PROTO and M_PCPROTO messages are read as data.
1143  * If RD_PROTDIS is set, the M_PROTO and M_PCPROTO parts of the message
1144  * are unlinked from and M_DATA blocks in the message, the protos are
1145  * thrown away, and the data is read.
1146  */
1147 /* ARGSUSED */
1148 int
1149 strread(struct vnode *vp, struct uio *uiop, cred_t *crp)
1150 {
1151        struct stdata *stp;
1152        mblk_t *bp, *nbp;
1153        queue_t *q;
1154        int error = 0;
1155        uint_t old_sd_flag;
1156        int first;
1157        char rflg;
1158        uint_t mark;            /* Contains MSG*MARK and _LASTMARK */
1159 #define _LASTMARK       0x8000  /* Distinct from MSG*MARK */
1160        short delim;
1161        unsigned char pri = 0;
1162        char waitflag;
1163        unsigned char type;

1165        TRACE_1(TR_FAC_STREAMS_FR,
1166            TR_STRREAD_ENTER, "strread:%p", vp);
1167        ASSERT(vp->v_stream);
1168        stp = vp->v_stream;

1170        mutex_enter(&stp->sd_lock);

1172        if ((error = i_straccess(stp, JCREAD)) != 0) {
1173                mutex_exit(&stp->sd_lock);
1174                return (error);
1175        }

1177        if (stp->sd_flag & (STRDERR|STPLEX)) {
1178                error = strgeterr(stp, STRDERR|STPLEX, 0);
1179                if (error != 0) {
1180                        mutex_exit(&stp->sd_lock);
1181                        return (error);
1182                }
1183        }
```

```
1185         /*
1186          * Loop terminates when uiop->uio_resid == 0.
1187          */
1188         rflg = 0;
1189         waitflag = READWAIT;
1190         q = _RD(stp->sd_wrq);
1191         for (;;) {
1192                 ASSERT(MUTEX_HELD(&stp->sd_lock));
1193                 old_sd_flag = stp->sd_flag;
1194                 mark = 0;
1195                 delim = 0;
1196                 first = 1;
1197                 while ((bp = strget(stp, q, uiop, first, &error)) == NULL) {
1198                         int done = 0;

1200                         ASSERT(MUTEX_HELD(&stp->sd_lock));

1202                         if (error != 0)
1203                                 goto oops;

1205                         if (stp->sd_flag & (STRHUP|STREOF)) {
1206                                 goto oops;
1207                         }
1208                         if (rflg && !(stp->sd_flag & STRDELIM)) {
1209                                 goto oops;
1210                         }
1211                         /*
1212                          * If a read(fd,buf,0) has been done, there is no
1213                          * need to sleep. We always have zero bytes to
1214                          * return.
1215                          */
1216                         if (uiop->uio_resid == 0) {
1217                                 goto oops;
1218                         }

1220                         qbackenable(q, 0);

1222                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRREAD_WAIT,
1223                             "strread calls strwaitq:%p, %p, %p",
1224                             vp, uiop, crp);
1225                         if ((error = strwaitq(stp, waitflag, uiop->uio_resid,
1226                             uiop->uio_fmode, -1, &done)) != 0 || done) {
1227                                 TRACE_3(TR_FAC_STREAMS_FR, TR_STRREAD_DONE,
1228                                     "strread error or done:%p, %p, %p",
1229                                     vp, uiop, crp);
1230                                 if ((uiop->uio_fmode & FNDELAY) &&
1231                                     (stp->sd_flag & OLDNDELAY) &&
1232                                     (error == EAGAIN))
1233                                         error = 0;
1234                                 goto oops;
1235                         }
1236                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRREAD_AWAKE,
1237                             "strread awakes:%p, %p, %p", vp, uiop, crp);
1238                         if ((error = i_straccess(stp, JCREAD)) != 0) {
1239                                 goto oops;
1240                         }
1241                         first = 0;
1242                 }

1244                 ASSERT(MUTEX_HELD(&stp->sd_lock));
1245                 ASSERT(bp);
1246                 pri = bp->b_band;
1247                 /*
1248                  * Extract any mark information. If the message is not
1249                  * completely consumed this information will be put in the mblk
```



```
1250                  * that is putback.
1251                  * If MSGMARKNEXT is set and the message is completely consumed
1252                  * the STRATMARK flag will be set below. Likewise, if
1253                  * MSGNOTMARKNEXT is set and the message is
1254                  * completely consumed STRNOTATMARK will be set.
1255                  *
1256                  * For some unknown reason strread only breaks the read at the
1257                  * last mark.
1258                  */
1259                 mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
1260                 ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
1261                     (MSGMARKNEXT|MSGNOTMARKNEXT));
1262                 if (mark != 0 && bp == stp->sd_mark) {
1263                         if (rflg) {
1264                                 putback(stp, q, bp, pri);
1265                                 goto oops;
1266                         }
1267                         mark |= _LASTMARK;
1268                         stp->sd_mark = NULL;
1269                 }
1270                 if ((stp->sd_flag & STRDELIM) && (bp->b_flag & MSGDELIM))
1271                         delim = 1;
1272                 mutex_exit(&stp->sd_lock);

1274                 if (STREAM_NEEDSERVICE(stp))
1275                         stream_runservice(stp);

1277                 type = bp->b_datap->db_type;

1279                 switch (type) {

1281                 case M_DATA:
1282         ismdata:
1283                         if (msgnodata(bp)) {
1284                                 if (mark || delim) {
1285                                         freemsg(bp);
1286                                 } else if (rflg) {

1288                                         /*
1289                                          * If already read data put zero
1290                                          * length message back on queue else
1291                                          * free msg and return 0.
1292                                          */
1293                                         bp->b_band = pri;
1294                                         mutex_enter(&stp->sd_lock);
1295                                         putback(stp, q, bp, pri);
1296                                         mutex_exit(&stp->sd_lock);
1297                                 } else {
1298                                         freemsg(bp);
1299                                 }
1300                                 error =  0;
1301                                 goto oops1;
1302                         }

1304                         rflg = 1;
1305                         waitflag |= NOINTR;
1306                         bp = struiocopyout(bp, uiop, &error);
1307                         if (error != 0)
1308                                 goto oops1;

1310                         mutex_enter(&stp->sd_lock);
1311                         if (bp) {
1312                                 /*
1313                                  * Have remaining data in message.
1314                                  * Free msg if in discard mode.
1315                                  */
```

```
1316                                    if (stp->sd_read_opt & RD_MSGDIS) {
1317                                            freemsg(bp);
1318                                    } else {
1319                                            bp->b_band = pri;
1320                                            if ((mark & _LASTMARK) &&
1321                                                (stp->sd_mark == NULL))
1322                                                    stp->sd_mark = bp;
1323                                            bp->b_flag |= mark & ~_LASTMARK;
1324                                            if (delim)
1325                                                    bp->b_flag |= MSGDELIM;
1326                                            if (msgnodata(bp))
1327                                                    freemsg(bp);
1328                                            else
1329                                                    putback(stp, q, bp, pri);
1330                                    }
1331                            } else {
1332                                    /*
1333                                     * Consumed the complete message.
1334                                     * Move the MSG*MARKNEXT information
1335                                     * to the stream head just in case
1336                                     * the read queue becomes empty.
1337                                     *
1338                                     * If the stream head was at the mark
1339                                     * (STRATMARK) before we dropped sd_lock above
1340                                     * and some data was consumed then we have
1341                                     * moved past the mark thus STRATMARK is
1342                                     * cleared. However, if a message arrived in
1343                                     * strrput during the copyout above causing
1344                                     * STRATMARK to be set we can not clear that
1345                                     * flag.
1346                                     */
1347                                    if (mark &
1348                                        (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
1349                                            if (mark & MSGMARKNEXT) {
1350                                                    stp->sd_flag &= ~STRNOTATMARK;
1351                                                    stp->sd_flag |= STRATMARK;
1352                                            } else if (mark & MSGNOTMARKNEXT) {
1353                                                    stp->sd_flag &= ~STRATMARK;
1354                                                    stp->sd_flag |= STRNOTATMARK;
1355                                            } else {
1356                                                    stp->sd_flag &=
1357                                                        ~(STRATMARK|STRNOTATMARK);
1358                                            }
1359                                    } else if (rflg && (old_sd_flag & STRATMARK)) {
1360                                            stp->sd_flag &= ~STRATMARK;
1361                                    }
1362                            }

1364                            /*
1365                             * Check for signal messages at the front of the read
1366                             * queue and generate the signal(s) if appropriate.
1367                             * The only signal that can be on queue is M_SIG at
1368                             * this point.
1369                             */
1370                            while ((((bp = q->q_first)) != NULL) &&
1371                                (bp->b_datap->db_type == M_SIG)) {
1372                                    bp = getq_noenab(q, 0);
1373                                    /*
1374                                     * sd_lock is held so the content of the
1375                                     * read queue can not change.
1376                                     */
1377                                    ASSERT(bp != NULL && DB_TYPE(bp) == M_SIG);
1378                                    strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
1379                                    mutex_exit(&stp->sd_lock);
1380                                    freemsg(bp);
1381                                    if (STREAM_NEEDSERVICE(stp))
```



```
1382                                            stream_runservice(stp);
1383                                    mutex_enter(&stp->sd_lock);
1384                            }

1386                            if ((uiop->uio_resid == 0) || (mark & _LASTMARK) ||
1387                                delim ||
1388                                (stp->sd_read_opt & (RD_MSGDIS|RD_MSGNODIS))) {
1389                                    goto oops;
1390                            }
1391                            continue;

1393                    case M_SIG:
1394                            strsignal(stp, *bp->b_rptr, (int32_t)bp->b_band);
1395                            freemsg(bp);
1396                            mutex_enter(&stp->sd_lock);
1397                            continue;

1399                    case M_PROTO:
1400                    case M_PCPROTO:
1401                            /*
1402                             * Only data messages are readable.
1403                             * Any others generate an error, unless
1404                             * RD_PROTDIS or RD_PROTDAT is set.
1405                             */
1406                            if (stp->sd_read_opt & RD_PROTDAT) {
1407                                    for (nbp = bp; nbp; nbp = nbp->b_next) {
1408                                            if ((nbp->b_datap->db_type ==
1409                                                M_PROTO) ||
1410                                                (nbp->b_datap->db_type ==
1411                                                M_PCPROTO)) {
1412                                                    nbp->b_datap->db_type = M_DATA;
1413                                            } else {
1414                                                    break;
1415                                            }
1416                                    }
1417                                    /*
1418                                     * clear stream head hi pri flag based on
1419                                     * first message
1420                                     */
1421                                    if (type == M_PCPROTO) {
1422                                            mutex_enter(&stp->sd_lock);
1423                                            stp->sd_flag &= ~STRPRI;
1424                                            mutex_exit(&stp->sd_lock);
1425                                    }
1426                                    goto ismdata;
1427                            } else if (stp->sd_read_opt & RD_PROTDIS) {
1428                                    /*
1429                                     * discard non-data messages
1430                                     */
1431                                    while (bp &&
1432                                        ((bp->b_datap->db_type == M_PROTO) ||
1433                                        (bp->b_datap->db_type == M_PCPROTO))) {
1434                                            nbp = unlinkb(bp);
1435                                            freeb(bp);
1436                                            bp = nbp;
1437                                    }
1438                                    /*
1439                                     * clear stream head hi pri flag based on
1440                                     * first message
1441                                     */
1442                                    if (type == M_PCPROTO) {
1443                                            mutex_enter(&stp->sd_lock);
1444                                            stp->sd_flag &= ~STRPRI;
1445                                            mutex_exit(&stp->sd_lock);
1446                                    }
1447                                    if (bp) {
```

```
1448                                    bp->b_band = pri;
1449                                    goto ismdata;
1450                            } else {
1451                                    break;
1452                            }
1453                    }
1454                    /* FALLTHRU */
1455            case M_PASSFP:
1456                    if ((bp->b_datap->db_type == M_PASSFP) &&
1457                        (stp->sd_read_opt & RD_PROTDIS)) {
1458                            freemsg(bp);
1459                            break;
1460                    }
1461                    mutex_enter(&stp->sd_lock);
1462                    putback(stp, q, bp, pri);
1463                    mutex_exit(&stp->sd_lock);
1464                    if (rflg == 0)
1465                            error = EBADMSG;
1466                    goto oops1;

1468            default:
1469                    /*
1470                     * Garbage on stream head read queue.
1471                     */
1472                    cmn_err(CE_WARN, "bad %x found at stream head\n",
1473                        bp->b_datap->db_type);
1474                    freemsg(bp);
1475                    goto oops1;
1476            }
1477            mutex_enter(&stp->sd_lock);
1478    }
1479 oops:
1480    mutex_exit(&stp->sd_lock);
1481 oops1:
1482    qbackenable(q, pri);
1483    return (error);
1484 #undef  _LASTMARK
1485 }

1487 /*
1488  * Default processing of M_PROTO/M_PCPROTO messages.
1489  * Determine which wakeups and signals are needed.
1490  * This can be replaced by a user-specified procedure for kernel users
1491  * of STREAMS.
1492  */
1493 /* ARGSUSED */
1494 mblk_t *
1495 strrput_proto(vnode_t *vp, mblk_t *mp,
1496     strwakeup_t *wakeups, strsigset_t *firstmsgsigs,
1497     strsigset_t *allmsgsigs, strpollset_t *pollwakeups)
1498 {
1499    *wakeups = RSLEEP;
1500    *allmsgsigs = 0;

1502    switch (mp->b_datap->db_type) {
1503    case M_PROTO:
1504            if (mp->b_band == 0) {
1505                    *firstmsgsigs = S_INPUT | S_RDNORM;
1506                    *pollwakeups = POLLIN | POLLRDNORM;
1507            } else {
1508                    *firstmsgsigs = S_INPUT | S_RDBAND;
1509                    *pollwakeups = POLLIN | POLLRDBAND;
1510            }
1511            break;
1512    case M_PCPROTO:
1513            *firstmsgsigs = S_HIPRI;
```



```
1514            *pollwakeups = POLLPRI;
1515            break;
1516    }
1517    return (mp);
1518 }

1520 /*
1521  * Default processing of everything but M_DATA, M_PROTO, M_PCPROTO and
1522  * M_PASSFP messages.
1523  * Determine which wakeups and signals are needed.
1524  * This can be replaced by a user-specified procedure for kernel users
1525  * of STREAMS.
1526  */
1527 /* ARGSUSED */
1528 mblk_t *
1529 strrput_misc(vnode_t *vp, mblk_t *mp,
1530     strwakeup_t *wakeups, strsigset_t *firstmsgsigs,
1531     strsigset_t *allmsgsigs, strpollset_t *pollwakeups)
1532 {
1533    *wakeups = 0;
1534    *firstmsgsigs = 0;
1535    *allmsgsigs = 0;
1536    *pollwakeups = 0;
1537    return (mp);
1538 }

1540 /*
1541  * Stream read put procedure.  Called from downstream driver/module
1542  * with messages for the stream head.  Data, protocol, and in-stream
1543  * signal messages are placed on the queue, others are handled directly.
1544  */
1545 int
1546 strrput(queue_t *q, mblk_t *bp)
1547 {
1548    struct stdata   *stp;
1549    ulong_t         rput_opt;
1550    strwakeup_t     wakeups;
1551    strsigset_t     firstmsgsigs;   /* Signals if first message on queue */
1552    strsigset_t     allmsgsigs;     /* Signals for all messages */
1553    strsigset_t     signals;        /* Signals events to generate */
1554    strpollset_t    pollwakeups;
1555    mblk_t          *nextbp;
1556    uchar_t         band = 0;
1557    int             hipri_sig;

1559    stp = (struct stdata *)q->q_ptr;
1560    /*
1561     * Use rput_opt for optimized access to the SR_ flags except
1562     * SR_POLLIN. That flag has to be checked under sd_lock since it
1563     * is modified by strpoll().
1564     */
1565    rput_opt = stp->sd_rput_opt;

1567    ASSERT(qclaimed(q));
1568    TRACE_2(TR_FAC_STREAMS_FR, TR_STRRPUT_ENTER,
1569        "strrput called with message type:q %p bp %p", q, bp);

1571    /*
1572     * Perform initial processing and pass to the parameterized functions.
1573     */
1574    ASSERT(bp->b_next == NULL);

1576    switch (bp->b_datap->db_type) {
1577    case M_DATA:
1578            /*
1579             * sockfs is the only consumer of STREOF and when it is set,
```

```
1580                 * it implies that the receiver is not interested in receiving
1581                 * any more data, hence the mblk is freed to prevent unnecessary
1582                 * message queueing at the stream head.
1583                 */
1584                if (stp->sd_flag == STREOF) {
1585                        freemsg(bp);
1586                        return (0);
1587                }
1588                if ((rput_opt & SR_IGN_ZEROLEN) &&
1589                    bp->b_rptr == bp->b_wptr && msgnodata(bp)) {
1590                        /*
1591                         * Ignore zero-length M_DATA messages. These might be
1592                         * generated by some transports.
1593                         * The zero-length M_DATA messages, even if they
1594                         * are ignored, should effect the atmark tracking and
1595                         * should wake up a thread sleeping in strwaitmark.
1596                         */
1597                        mutex_enter(&stp->sd_lock);
1598                        if (bp->b_flag & MSGMARKNEXT) {
1599                                /*
1600                                 * Record the position of the mark either
1601                                 * in q_last or in STRATMARK.
1602                                 */
1603                                if (q->q_last != NULL) {
1604                                        q->q_last->b_flag &= ~MSGNOTMARKNEXT;
1605                                        q->q_last->b_flag |= MSGMARKNEXT;
1606                                } else {
1607                                        stp->sd_flag &= ~STRNOTATMARK;
1608                                        stp->sd_flag |= STRATMARK;
1609                                }
1610                        } else if (bp->b_flag & MSGNOTMARKNEXT) {
1611                                /*
1612                                 * Record that this is not the position of
1613                                 * the mark either in q_last or in
1614                                 * STRNOTATMARK.
1615                                 */
1616                                if (q->q_last != NULL) {
1617                                        q->q_last->b_flag &= ~MSGMARKNEXT;
1618                                        q->q_last->b_flag |= MSGNOTMARKNEXT;
1619                                } else {
1620                                        stp->sd_flag &= ~STRATMARK;
1621                                        stp->sd_flag |= STRNOTATMARK;
1622                                }
1623                        }
1624                        if (stp->sd_flag & RSLEEP) {
1625                                stp->sd_flag &= ~RSLEEP;
1626                                cv_broadcast(&q->q_wait);
1627                        }
1628                        mutex_exit(&stp->sd_lock);
1629                        freemsg(bp);
1630                        return (0);
1631                }
1632                wakeups = RSLEEP;
1633                if (bp->b_band == 0) {
1634                        firstmsgsigs = S_INPUT | S_RDNORM;
1635                        pollwakeups = POLLIN | POLLRDNORM;
1636                } else {
1637                        firstmsgsigs = S_INPUT | S_RDBAND;
1638                        pollwakeups = POLLIN | POLLRDBAND;
1639                }
1640                if (rput_opt & SR_SIGALLDATA)
1641                        allmsgsigs = firstmsgsigs;
1642                else
1643                        allmsgsigs = 0;

1645                mutex_enter(&stp->sd_lock);
```



```
1646                if ((rput_opt & SR_CONSOL_DATA) &&
1647                    (q->q_last != NULL) &&
1648                    (bp->b_flag & (MSGMARK|MSGDELIM)) == 0) {
1649                        /*
1650                         * Consolidate an M_DATA message onto an M_DATA,
1651                         * M_PROTO, or M_PCPROTO by merging it with q_last.
1652                         * The consolidation does not take place if
1653                         * the old message is marked with either of the
1654                         * marks or the delim flag or if the new
1655                         * message is marked with MSGMARK. The MSGMARK
1656                         * check is needed to handle the odd semantics of
1657                         * MSGMARK where essentially the whole message
1658                         * is to be treated as marked.
1659                         * Carry any MSGMARKNEXT  and MSGNOTMARKNEXT from the
1660                         * new message to the front of the b_cont chain.
1661                         */
1662                        mblk_t *lbp = q->q_last;
1663                        unsigned char db_type = lbp->b_datap->db_type;

1665                        if ((db_type == M_DATA || db_type == M_PROTO ||
1666                            db_type == M_PCPROTO) &&
1667                            !(lbp->b_flag & (MSGDELIM|MSGMARK|MSGMARKNEXT))) {
1668                                rmvq_noenab(q, lbp);
1669                                /*
1670                                 * The first message in the b_cont list
1671                                 * tracks MSGMARKNEXT and MSGNOTMARKNEXT.
1672                                 * We need to handle the case where we
1673                                 * are appending:
1674                                 *
1675                                 * 1) a MSGMARKNEXT to a MSGNOTMARKNEXT.
1676                                 * 2) a MSGMARKNEXT to a plain message.
1677                                 * 3) a MSGNOTMARKNEXT to a plain message
1678                                 * 4) a MSGNOTMARKNEXT to a MSGNOTMARKNEXT
1679                                 *    message.
1680                                 *
1681                                 * Thus we never append a MSGMARKNEXT or
1682                                 * MSGNOTMARKNEXT to a MSGMARKNEXT message.
1683                                 */
1684                                if (bp->b_flag & MSGMARKNEXT) {
1685                                        lbp->b_flag |= MSGMARKNEXT;
1686                                        lbp->b_flag &= ~MSGNOTMARKNEXT;
1687                                        bp->b_flag &= ~MSGMARKNEXT;
1688                                } else if (bp->b_flag & MSGNOTMARKNEXT) {
1689                                        lbp->b_flag |= MSGNOTMARKNEXT;
1690                                        bp->b_flag &= ~MSGNOTMARKNEXT;
1691                                }

1693                                linkb(lbp, bp);
1694                                bp = lbp;
1695                                /*
1696                                 * The new message logically isn't the first
1697                                 * even though the q_first check below thinks
1698                                 * it is. Clear the firstmsgsigs to make it
1699                                 * not appear to be first.
1700                                 */
1701                                firstmsgsigs = 0;
1702                        }
1703                }
1704                break;

1706        case M_PASSFP:
1707                wakeups = RSLEEP;
1708                allmsgsigs = 0;
1709                if (bp->b_band == 0) {
1710                        firstmsgsigs = S_INPUT | S_RDNORM;
1711                        pollwakeups = POLLIN | POLLRDNORM;
```

```
1712                 } else {
1713                         firstmsgsigs = S_INPUT | S_RDBAND;
1714                         pollwakeups = POLLIN | POLLRDBAND;
1715                 }
1716                 mutex_enter(&stp->sd_lock);
1717                 break;

1719         case M_PROTO:
1720         case M_PCPROTO:
1721                 ASSERT(stp->sd_rprotofunc != NULL);
1722                 bp = (stp->sd_rprotofunc)(stp->sd_vnode, bp,
1723                     &wakeups, &firstmsgsigs, &allmsgsigs, &pollwakeups);
1724 #define ALLSIG  (S_INPUT|S_HIPRI|S_OUTPUT|S_MSG|S_ERROR|S_HANGUP|S_RDNORM|\
1725                 S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)
1726 #define ALLPOLL (POLLIN|POLLPRI|POLLOUT|POLLRDNORM|POLLWRNORM|POLLRDBAND|\
1727                 POLLWRBAND)

1729                 ASSERT((wakeups & ~(RSLEEP|WSLEEP)) == 0);
1730                 ASSERT((firstmsgsigs & ~ALLSIG) == 0);
1731                 ASSERT((allmsgsigs & ~ALLSIG) == 0);
1732                 ASSERT((pollwakeups & ~ALLPOLL) == 0);

1734                 mutex_enter(&stp->sd_lock);
1735                 break;

1737         default:
1738                 ASSERT(stp->sd_rmiscfunc != NULL);
1739                 bp = (stp->sd_rmiscfunc)(stp->sd_vnode, bp,
1740                     &wakeups, &firstmsgsigs, &allmsgsigs, &pollwakeups);
1741                 ASSERT((wakeups & ~(RSLEEP|WSLEEP)) == 0);
1742                 ASSERT((firstmsgsigs & ~ALLSIG) == 0);
1743                 ASSERT((allmsgsigs & ~ALLSIG) == 0);
1744                 ASSERT((pollwakeups & ~ALLPOLL) == 0);
1745 #undef  ALLSIG
1746 #undef  ALLPOLL
1747                 mutex_enter(&stp->sd_lock);
1748                 break;
1749         }
1750         ASSERT(MUTEX_HELD(&stp->sd_lock));

1752         /* By default generate superset of signals */
1753         signals = (firstmsgsigs | allmsgsigs);

1755         /*
1756          * The  proto and misc functions can return multiple messages
1757          * as a b_next chain. Such messages are processed separately.
1758          */
1759 one_more:
1760         hipri_sig = 0;
1761         if (bp == NULL) {
1762                 nextbp = NULL;
1763         } else {
1764                 nextbp = bp->b_next;
1765                 bp->b_next = NULL;

1767                 switch (bp->b_datap->db_type) {
1768                 case M_PCPROTO:
1769                         /*
1770                          * Only one priority protocol message is allowed at the
1771                          * stream head at a time.
1772                          */
1773                         if (stp->sd_flag & STRPRI) {
1774                                 TRACE_0(TR_FAC_STREAMS_FR, TR_STRRPUT_PROTERR,
1775                                     "M_PCPROTO already at head");
1776                                 freemsg(bp);
1777                                 mutex_exit(&stp->sd_lock);
```



```
1778                                 goto done;
1779                         }
1780                         stp->sd_flag |= STRPRI;
1781                         hipri_sig = 1;
1782                         /* FALLTHRU */
1783                 case M_DATA:
1784                 case M_PROTO:
1785                 case M_PASSFP:
1786                         band = bp->b_band;
1787                         /*
1788                          * Marking doesn't work well when messages
1789                          * are marked in more than one band.  We only
1790                          * remember the last message received, even if
1791                          * it is placed on the queue ahead of other
1792                          * marked messages.
1793                          */
1794                         if (bp->b_flag & MSGMARK)
1795                                 stp->sd_mark = bp;
1796                         (void) putq(q, bp);

1798                         /*
1799                          * If message is a PCPROTO message, always use
1800                          * firstmsgsigs to determine if a signal should be
1801                          * sent as strrput is the only place to send
1802                          * signals for PCPROTO. Other messages are based on
1803                          * the STRGETINPROG flag. The flag determines if
1804                          * strrput or (k)strgetmsg will be responsible for
1805                          * sending the signals, in the firstmsgsigs case.
1806                          */
1807                         if ((hipri_sig == 1) ||
1808                             (((stp->sd_flag & STRGETINPROG) == 0) &&
1809                             (q->q_first == bp)))
1810                                 signals = (firstmsgsigs | allmsgsigs);
1811                         else
1812                                 signals = allmsgsigs;
1813                         break;

1815                 default:
1816                         mutex_exit(&stp->sd_lock);
1817                         (void) strrput_nondata(q, bp);
1818                         mutex_enter(&stp->sd_lock);
1819                         break;
1820                 }
1821         }
1822         ASSERT(MUTEX_HELD(&stp->sd_lock));
1823         /*
1824          * Wake sleeping read/getmsg and cancel deferred wakeup
1825          */
1826         if (wakeups & RSLEEP)
1827                 stp->sd_wakeq &= ~RSLEEP;

1829         wakeups &= stp->sd_flag;
1830         if (wakeups & RSLEEP) {
1831                 stp->sd_flag &= ~RSLEEP;
1832                 cv_broadcast(&q->q_wait);
1833         }
1834         if (wakeups & WSLEEP) {
1835                 stp->sd_flag &= ~WSLEEP;
1836                 cv_broadcast(&_WR(q)->q_wait);
1837         }

1839         if (pollwakeups != 0) {
1840                 if (pollwakeups == (POLLIN | POLLRDNORM)) {
1841                         /*
1842                          * Can't use rput_opt since it was not
1843                          * read when sd_lock was held and SR_POLLIN is changed
```

```
1844                          * by strpoll() under sd_lock.
1845                          */
1846                         if (!(stp->sd_rput_opt & SR_POLLIN))
1847                                 goto no_pollwake;
1848                         stp->sd_rput_opt &= ~SR_POLLIN;
1849                 }
1850                 mutex_exit(&stp->sd_lock);
1851                 pollwakeup(&stp->sd_pollist, pollwakeups);
1852                 mutex_enter(&stp->sd_lock);
1853         }
1854 no_pollwake:

1856         /*
1857          * strsendsig can handle multiple signals with a
1858          * single call.
1859          */
1860         if (stp->sd_sigflags & signals)
1861                 strsendsig(stp->sd_siglist, signals, band, 0);
1862         mutex_exit(&stp->sd_lock);


1865 done:
1866         if (nextbp == NULL)
1867                 return (0);

1869         /*
1870          * Any signals were handled the first time.
1871          * Wakeups and pollwakeups are redone to avoid any race
1872          * conditions - all the messages are not queued until the
1873          * last message has been processed by strrput.
1874          */
1875         bp = nextbp;
1876         signals = firstmsgsigs = allmsgsigs = 0;
1877         mutex_enter(&stp->sd_lock);
1878         goto one_more;
1879 }

1881 static void
1882 log_dupioc(queue_t *rq, mblk_t *bp)
1883 {
1884         queue_t *wq, *qp;
1885         char *modnames, *mnp, *dname;
1886         size_t maxmodstr;
1887         boolean_t islast;

1889         /*
1890          * Allocate a buffer large enough to hold the names of nstrpush modules
1891          * and one driver, with spaces between and NUL terminator.  If we can't
1892          * get memory, then we'll just log the driver name.
1893          */
1894         maxmodstr = nstrpush * (FMNAMESZ + 1);
1895         mnp = modnames = kmem_alloc(maxmodstr, KM_NOSLEEP);

1897         /* march down write side to print log message down to the driver */
1898         wq = WR(rq);

1900         /* make sure q_next doesn't shift around while we're grabbing data */
1901         claimstr(wq);
1902         qp = wq->q_next;
1903         do {
1904                 dname = Q2NAME(qp);
1905                 islast = !SAMESTR(qp) || qp->q_next == NULL;
1906                 if (modnames == NULL) {
1907                         /*
1908                          * If we don't have memory, then get the driver name in
1909                          * the log where we can see it.  Note that memory
```



```
1910                          * pressure is a possible cause of these sorts of bugs.
1911                          */
1912                         if (islast) {
1913                                 modnames = dname;
1914                                 maxmodstr = 0;
1915                         }
1916                 } else {
1917                         mnp += snprintf(mnp, FMNAMESZ + 1, "%s", dname);
1918                         if (!islast)
1919                                 *mnp++ = ' ';
1920                 }
1921                 qp = qp->q_next;
1922         } while (!islast);
1923         releasestr(wq);
1924         /* Cannot happen unless stream head is corrupt. */
1925         ASSERT(modnames != NULL);
1926         (void) strlog(rq->q_qinfo->qi_minfo->mi_idnum, 0, 1,
1927             SL_CONSOLE|SL_TRACE|SL_ERROR,
1928             "Warning: stream %p received duplicate %X M_IOC%s; module list: %s",
1929             rq->q_ptr, ((struct iocblk *)bp->b_rptr)->ioc_cmd,
1930             (DB_TYPE(bp) == M_IOCACK ? "ACK" : "NAK"), modnames);
1931         if (maxmodstr != 0)
1932                 kmem_free(modnames, maxmodstr);
1933 }

1935 int
1936 strrput_nondata(queue_t *q, mblk_t *bp)
1937 {
1938         struct stdata *stp;
1939         struct iocblk *iocbp;
1940         struct stroptions *sop;
1941         struct copyreq *reqp;
1942         struct copyresp *resp;
1943         unsigned char bpri;
1944         unsigned char  flushed_already = 0;

1946         stp = (struct stdata *)q->q_ptr;

1948         ASSERT(!(stp->sd_flag & STPLEX));
1949         ASSERT(qclaimed(q));

1951         switch (bp->b_datap->db_type) {
1952         case M_ERROR:
1953                 /*
1954                  * An error has occurred downstream, the errno is in the first
1955                  * bytes of the message.
1956                  */
1957                 if ((bp->b_wptr - bp->b_rptr) == 2) {   /* New flavor */
1958                         unsigned char rw = 0;

1960                         mutex_enter(&stp->sd_lock);
1961                         if (*bp->b_rptr != NOERROR) {   /* read error */
1962                                 if (*bp->b_rptr != 0) {
1963                                         if (stp->sd_flag & STRDERR)
1964                                                 flushed_already |= FLUSHR;
1965                                         stp->sd_flag |= STRDERR;
1966                                         rw |= FLUSHR;
1967                                 } else {
1968                                         stp->sd_flag &= ~STRDERR;
1969                                 }
1970                                 stp->sd_rerror = *bp->b_rptr;
1971                         }
1972                         bp->b_rptr++;
1973                         if (*bp->b_rptr != NOERROR) {   /* write error */
1974                                 if (*bp->b_rptr != 0) {
1975                                         if (stp->sd_flag & STWRERR)
```

```
1976                                        flushed_already |= FLUSHW;
1977                                stp->sd_flag |= STWRERR;
1978                                rw |= FLUSHW;
1979                        } else {
1980                                stp->sd_flag &= ~STWRERR;
1981                        }
1982                        stp->sd_werror = *bp->b_rptr;
1983                }
1984                if (rw) {
1985                        TRACE_2(TR_FAC_STREAMS_FR, TR_STRRPUT_WAKE,
1986                            "strrput cv_broadcast:q %p, bp %p",
1987                            q, bp);
1988                        cv_broadcast(&q->q_wait); /* readers */
1989                        cv_broadcast(&_WR(q)->q_wait); /* writers */
1990                        cv_broadcast(&stp->sd_monitor); /* ioctllers */

1992                        mutex_exit(&stp->sd_lock);
1993                        pollwakeup(&stp->sd_pollist, POLLERR);
1994                        mutex_enter(&stp->sd_lock);

1996                        if (stp->sd_sigflags & S_ERROR)
1997                                strsendsig(stp->sd_siglist, S_ERROR, 0,
1998                                    ((rw & FLUSHR) ? stp->sd_rerror :
1999                                    stp->sd_werror));
2000                        mutex_exit(&stp->sd_lock);
2001                        /*
2002                         * Send the M_FLUSH only
2003                         * for the first M_ERROR
2004                         * message on the stream
2005                         */
2006                        if (flushed_already == rw) {
2007                                freemsg(bp);
2008                                return (0);
2009                        }

2011                        bp->b_datap->db_type = M_FLUSH;
2012                        *bp->b_rptr = rw;
2013                        bp->b_wptr = bp->b_rptr + 1;
2014                        /*
2015                         * Protect against the driver
2016                         * passing up messages after
2017                         * it has done a qprocsoff
2018                         */
2019                        if (_OTHERQ(q)->q_next == NULL)
2020                                freemsg(bp);
2021                        else
2022                                qreply(q, bp);
2023                        return (0);
2024                } else
2025                        mutex_exit(&stp->sd_lock);
2026        } else if (*bp->b_rptr != 0) {          /* Old flavor */
2027                        if (stp->sd_flag & (STRDERR|STWRERR))
2028                                flushed_already = FLUSHRW;
2029                        mutex_enter(&stp->sd_lock);
2030                        stp->sd_flag |= (STRDERR|STWRERR);
2031                        stp->sd_rerror = *bp->b_rptr;
2032                        stp->sd_werror = *bp->b_rptr;
2033                        TRACE_2(TR_FAC_STREAMS_FR,
2034                            TR_STRRPUT_WAKE2,
2035                            "strrput wakeup #2:q %p, bp %p", q, bp);
2036                        cv_broadcast(&q->q_wait); /* the readers */
2037                        cv_broadcast(&_WR(q)->q_wait); /* the writers */
2038                        cv_broadcast(&stp->sd_monitor); /* ioctllers */

2040                        mutex_exit(&stp->sd_lock);
2041                        pollwakeup(&stp->sd_pollist, POLLERR);
```



```
2042                                mutex_enter(&stp->sd_lock);

2044                                if (stp->sd_sigflags & S_ERROR)
2045                                        strsendsig(stp->sd_siglist, S_ERROR, 0,
2046                                            (stp->sd_werror ? stp->sd_werror :
2047                                            stp->sd_rerror));
2048                                mutex_exit(&stp->sd_lock);

2050                                /*
2051                                 * Send the M_FLUSH only
2052                                 * for the first M_ERROR
2053                                 * message on the stream
2054                                 */
2055                                if (flushed_already != FLUSHRW) {
2056                                        bp->b_datap->db_type = M_FLUSH;
2057                                        *bp->b_rptr = FLUSHRW;
2058                                        /*
2059                                         * Protect against the driver passing up
2060                                         * messages after it has done a
2061                                         * qprocsoff.
2062                                         */
2063                                if (_OTHERQ(q)->q_next == NULL)
2064                                        freemsg(bp);
2065                                else
2066                                        qreply(q, bp);
2067                                return (0);
2068                                }
2069                        }
2070                        freemsg(bp);
2071                        return (0);

2073                case M_HANGUP:

2075                        freemsg(bp);
2076                        mutex_enter(&stp->sd_lock);
2077                        stp->sd_werror = ENXIO;
2078                        stp->sd_flag |= STRHUP;
2079                        stp->sd_flag &= ~(WSLEEP|RSLEEP);

2081                        /*
2082                         * send signal if controlling tty
2083                         */

2085                        if (stp->sd_sidp) {
2086                                prsignal(stp->sd_sidp, SIGHUP);
2087                                if (stp->sd_sidp != stp->sd_pgidp)
2088                                        pgsignal(stp->sd_pgidp, SIGTSTP);
2089                        }

2091                        /*
2092                         * wake up read, write, and exception pollers and
2093                         * reset wakeup mechanism.
2094                         */
2095                        cv_broadcast(&q->q_wait);       /* the readers */
2096                        cv_broadcast(&_WR(q)->q_wait);  /* the writers */
2097                        cv_broadcast(&stp->sd_monitor); /* the ioctllers */
2098                        strhup(stp);
2099                        mutex_exit(&stp->sd_lock);
2100                        return (0);

2102                case M_UNHANGUP:
2103                        freemsg(bp);
2104                        mutex_enter(&stp->sd_lock);
2105                        stp->sd_werror = 0;
2106                        stp->sd_flag &= ~STRHUP;
2107                        mutex_exit(&stp->sd_lock);
```

```
2108                return (0);
2109
2110        case M_SIG:
2111                /*
2112                 * Someone downstream wants to post a signal.  The
2113                 * signal to post is contained in the first byte of the
2114                 * message.  If the message would go on the front of
2115                 * the queue, send a signal to the process group
2116                 * (if not SIGPOLL) or to the siglist processes
2117                 * (SIGPOLL).  If something is already on the queue,
2118                 * OR if we are delivering a delayed suspend (*sigh*
2119                 * another "tty" hack) and there's no one sleeping already,
2120                 * just enqueue the message.
2121                 */
2122                mutex_enter(&stp->sd_lock);
2123                if (q->q_first || (*bp->b_rptr == SIGTSTP &&
2124                    !(stp->sd_flag & RSLEEP))) {
2125                        (void) putq(q, bp);
2126                        mutex_exit(&stp->sd_lock);
2127                        return (0);
2128                }
2129                mutex_exit(&stp->sd_lock);
2130                /* FALLTHRU */
2131
2132        case M_PCSIG:
2133                /*
2134                 * Don't enqueue, just post the signal.
2135                 */
2136                strsignal(stp, *bp->b_rptr, 0L);
2137                freemsg(bp);
2138                return (0);
2139
2140        case M_CMD:
2141                if (MBLKL(bp) != sizeof (cmdblk_t)) {
2142                        freemsg(bp);
2143                        return (0);
2144                }
2145
2146                mutex_enter(&stp->sd_lock);
2147                if (stp->sd_flag & STRCMDWAIT) {
2148                        ASSERT(stp->sd_cmdblk == NULL);
2149                        stp->sd_cmdblk = bp;
2150                        cv_broadcast(&stp->sd_monitor);
2151                        mutex_exit(&stp->sd_lock);
2152                } else {
2153                        mutex_exit(&stp->sd_lock);
2154                        freemsg(bp);
2155                }
2156                return (0);
2157
2158        case M_FLUSH:
2159                /*
2160                 * Flush queues.  The indication of which queues to flush
2161                 * is in the first byte of the message.  If the read queue
2162                 * is specified, then flush it.  If FLUSHBAND is set, just
2163                 * flush the band specified by the second byte of the message.
2164                 *
2165                 * If a module has issued a M_SETOPT to not flush hi
2166                 * priority messages off of the stream head, then pass this
2167                 * flag into the flushq code to preserve such messages.
2168                 */
2169
2170                if (*bp->b_rptr & FLUSHR) {
2171                        mutex_enter(&stp->sd_lock);
2172                        if (*bp->b_rptr & FLUSHBAND) {
2173                                ASSERT((bp->b_wptr - bp->b_rptr) >= 2);
```



```
2174                                flushband(q, *(bp->b_rptr + 1), FLUSHALL);
2175                        } else
2176                                flushq_common(q, FLUSHALL,
2177                                    stp->sd_read_opt & RFLUSHPCPROT);
2178                        if ((q->q_first == NULL) ||
2179                            (q->q_first->b_datap->db_type < QPCTL))
2180                                stp->sd_flag &= ~STRPRI;
2181                        else {
2182                                ASSERT(stp->sd_flag & STRPRI);
2183                        }
2184                        mutex_exit(&stp->sd_lock);
2185                }
2186                if ((*bp->b_rptr & FLUSHW) && !(bp->b_flag & MSGNOLOOP)) {
2187                        *bp->b_rptr &= ~FLUSHR;
2188                        bp->b_flag |= MSGNOLOOP;
2189                        /*
2190                         * Protect against the driver passing up
2191                         * messages after it has done a qprocsoff.
2192                         */
2193                        if (_OTHERQ(q)->q_next == NULL)
2194                                freemsg(bp);
2195                        else
2196                                qreply(q, bp);
2197                        return (0);
2198                }
2199                freemsg(bp);
2200                return (0);
2201
2202        case M_IOCACK:
2203        case M_IOCNAK:
2204                iocbp = (struct iocblk *)bp->b_rptr;
2205                /*
2206                 * If not waiting for ACK or NAK then just free msg.
2207                 * If incorrect id sequence number then just free msg.
2208                 * If already have ACK or NAK for user then this is a
2209                 *    duplicate, display a warning and free the msg.
2210                 */
2211                mutex_enter(&stp->sd_lock);
2212                if ((stp->sd_flag & IOCWAIT) == 0 || stp->sd_iocblk ||
2213                    (stp->sd_iocid != iocbp->ioc_id)) {
2214                        /*
2215                         * If the ACK/NAK is a dup, display a message
2216                         * Dup is when sd_iocid == ioc_id, and
2217                         * sd_iocblk == <valid ptr> or -1 (the former
2218                         * is when an ioctl has been put on the stream
2219                         * head, but has not yet been consumed, the
2220                         * later is when it has been consumed).
2221                         */
2222                        if ((stp->sd_iocid == iocbp->ioc_id) &&
2223                            (stp->sd_iocblk != NULL)) {
2224                                log_dupioc(q, bp);
2225                        }
2226                        freemsg(bp);
2227                        mutex_exit(&stp->sd_lock);
2228                        return (0);
2229                }
2230
2231                /*
2232                 * Assign ACK or NAK to user and wake up.
2233                 */
2234                stp->sd_iocblk = bp;
2235                cv_broadcast(&stp->sd_monitor);
2236                mutex_exit(&stp->sd_lock);
2237                return (0);
2238
2239        case M_COPYIN:
```

```
2240         case M_COPYOUT:
2241                 reqp = (struct copyreq *)bp->b_rptr;

2243                 /*
2244                  * If not waiting for ACK or NAK then just fail request.
2245                  * If already have ACK, NAK, or copy request, then just
2246                  * fail request.
2247                  * If incorrect id sequence number then just fail request.
2248                  */
2249                 mutex_enter(&stp->sd_lock);
2250                 if ((stp->sd_flag & IOCWAIT) == 0 || stp->sd_iocblk ||
2251                     (stp->sd_iocid != reqp->cq_id)) {
2252                         if (bp->b_cont) {
2253                                 freemsg(bp->b_cont);
2254                                 bp->b_cont = NULL;
2255                         }
2256                         bp->b_datap->db_type = M_IOCDATA;
2257                         bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
2258                         resp = (struct copyresp *)bp->b_rptr;
2259                         resp->cp_rval = (caddr_t)1;     /* failure */
2260                         mutex_exit(&stp->sd_lock);
2261                         putnext(stp->sd_wrq, bp);
2262                         return (0);
2263                 }

2265                 /*
2266                  * Assign copy request to user and wake up.
2267                  */
2268                 stp->sd_iocblk = bp;
2269                 cv_broadcast(&stp->sd_monitor);
2270                 mutex_exit(&stp->sd_lock);
2271                 return (0);

2273         case M_SETOPTS:
2274                 /*
2275                  * Set stream head options (read option, write offset,
2276                  * min/max packet size, and/or high/low water marks for
2277                  * the read side only).
2278                  */

2280                 bpri = 0;
2281                 sop = (struct stroptions *)bp->b_rptr;
2282                 mutex_enter(&stp->sd_lock);
2283                 if (sop->so_flags & SO_READOPT) {
2284                         switch (sop->so_readopt & RMODEMASK) {
2285                         case RNORM:
2286                                 stp->sd_read_opt &= ~(RD_MSGDIS | RD_MSGNODIS);
2287                                 break;

2289                         case RMSGD:
2290                                 stp->sd_read_opt =
2291                                     ((stp->sd_read_opt & ~RD_MSGNODIS) |
2292                                     RD_MSGDIS);
2293                                 break;

2295                         case RMSGN:
2296                                 stp->sd_read_opt =
2297                                     ((stp->sd_read_opt & ~RD_MSGDIS) |
2298                                     RD_MSGNODIS);
2299                                 break;
2300                         }
2301                         switch (sop->so_readopt & RPROTMASK) {
2302                         case RPROTNORM:
2303                                 stp->sd_read_opt &= ~(RD_PROTDAT | RD_PROTDIS);
2304                                 break;
```



```
2306                         case RPROTDAT:
2307                                 stp->sd_read_opt =
2308                                     ((stp->sd_read_opt & ~RD_PROTDIS) |
2309                                     RD_PROTDAT);
2310                                 break;

2312                         case RPROTDIS:
2313                                 stp->sd_read_opt =
2314                                     ((stp->sd_read_opt & ~RD_PROTDAT) |
2315                                     RD_PROTDIS);
2316                                 break;
2317                         }
2318                         switch (sop->so_readopt & RFLUSHMASK) {
2319                         case RFLUSHPCPROT:
2320                                 /*
2321                                  * This sets the stream head to NOT flush
2322                                  * M_PCPROTO messages.
2323                                  */
2324                                 stp->sd_read_opt |= RFLUSHPCPROT;
2325                                 break;
2326                         }
2327                 }
2328                 if (sop->so_flags & SO_ERROPT) {
2329                         switch (sop->so_erropt & RERRMASK) {
2330                         case RERRNORM:
2331                                 stp->sd_flag &= ~STRDERRNONPERSIST;
2332                                 break;
2333                         case RERRNONPERSIST:
2334                                 stp->sd_flag |= STRDERRNONPERSIST;
2335                                 break;
2336                         }
2337                         switch (sop->so_erropt & WERRMASK) {
2338                         case WERRNORM:
2339                                 stp->sd_flag &= ~STWRERRNONPERSIST;
2340                                 break;
2341                         case WERRNONPERSIST:
2342                                 stp->sd_flag |= STWRERRNONPERSIST;
2343                                 break;
2344                         }
2345                 }
2346                 if (sop->so_flags & SO_COPYOPT) {
2347                         if (sop->so_copyopt & ZCVMSAFE) {
2348                                 stp->sd_copyflag |= STZCVMSAFE;
2349                                 stp->sd_copyflag &= ~STZCVMUNSAFE;
2350                         } else if (sop->so_copyopt & ZCVMUNSAFE) {
2351                                 stp->sd_copyflag |= STZCVMUNSAFE;
2352                                 stp->sd_copyflag &= ~STZCVMSAFE;
2353                         }

2355                         if (sop->so_copyopt & COPYCACHED) {
2356                                 stp->sd_copyflag |= STRCOPYCACHED;
2357                         }
2358                 }
2359                 if (sop->so_flags & SO_WROFF)
2360                         stp->sd_wroff = sop->so_wroff;
2361                 if (sop->so_flags & SO_TAIL)
2362                         stp->sd_tail = sop->so_tail;
2363                 if (sop->so_flags & SO_MINPSZ)
2364                         q->q_minpsz = sop->so_minpsz;
2365                 if (sop->so_flags & SO_MAXPSZ)
2366                         q->q_maxpsz = sop->so_maxpsz;
2367                 if (sop->so_flags & SO_MAXBLK)
2368                         stp->sd_maxblk = sop->so_maxblk;
2369                 if (sop->so_flags & SO_HIWAT) {
2370                         if (sop->so_flags & SO_BAND) {
2371                                 if (strqset(q, QHIWAT,
```

```
2372                                    sop->so_band, sop->so_hiwat)) {
2373                                        cmn_err(CE_WARN, "strrput: could not "
2374                                            "allocate qband\n");
2375                                } else {
2376                                        bpri = sop->so_band;
2377                                }
2378                        } else {
2379                                q->q_hiwat = sop->so_hiwat;
2380                        }
2381                }
2382                if (sop->so_flags & SO_LOWAT) {
2383                        if (sop->so_flags & SO_BAND) {
2384                                if (strqset(q, QLOWAT,
2385                                    sop->so_band, sop->so_lowat)) {
2386                                        cmn_err(CE_WARN, "strrput: could not "
2387                                            "allocate qband\n");
2388                                } else {
2389                                        bpri = sop->so_band;
2390                                }
2391                        } else {
2392                                q->q_lowat = sop->so_lowat;
2393                        }
2394                }
2395                if (sop->so_flags & SO_MREADON)
2396                        stp->sd_flag |= SNDMREAD;
2397                if (sop->so_flags & SO_MREADOFF)
2398                        stp->sd_flag &= ~SNDMREAD;
2399                if (sop->so_flags & SO_NDELON)
2400                        stp->sd_flag |= OLDNDELAY;
2401                if (sop->so_flags & SO_NDELOFF)
2402                        stp->sd_flag &= ~OLDNDELAY;
2403                if (sop->so_flags & SO_ISTTY)
2404                        stp->sd_flag |= STRISTTY;
2405                if (sop->so_flags & SO_ISNTTY)
2406                        stp->sd_flag &= ~STRISTTY;
2407                if (sop->so_flags & SO_TOSTOP)
2408                        stp->sd_flag |= STRTOSTOP;
2409                if (sop->so_flags & SO_TONSTOP)
2410                        stp->sd_flag &= ~STRTOSTOP;
2411                if (sop->so_flags & SO_DELIM)
2412                        stp->sd_flag |= STRDELIM;
2413                if (sop->so_flags & SO_NODELIM)
2414                        stp->sd_flag &= ~STRDELIM;

2416                mutex_exit(&stp->sd_lock);
2417                freemsg(bp);

2419                /* Check backenable in case the water marks changed */
2420                qbackenable(q, bpri);
2421                return (0);

2423        /*
2424         * The following set of cases deal with situations where two stream
2425         * heads are connected to each other (twisted streams).  These messages
2426         * have no meaning at the stream head.
2427         */
2428        case M_BREAK:
2429        case M_CTL:
2430        case M_DELAY:
2431        case M_START:
2432        case M_STOP:
2433        case M_IOCDATA:
2434        case M_STARTI:
2435        case M_STOPI:
2436                freemsg(bp);
2437                return (0);
```



```
2439        case M_IOCTL:
2440                /*
2441                 * Always NAK this condition
2442                 * (makes no sense)
2443                 * If there is one or more threads in the read side
2444                 * rwnext we have to defer the nacking until that thread
2445                 * returns (in strget).
2446                 */
2447                mutex_enter(&stp->sd_lock);
2448                if (stp->sd_struiodnak != 0) {
2449                        /*
2450                         * Defer NAK to the streamhead. Queue at the end
2451                         * the list.
2452                         */
2453                        mblk_t *mp = stp->sd_struionak;

2455                        while (mp && mp->b_next)
2456                                mp = mp->b_next;
2457                        if (mp)
2458                                mp->b_next = bp;
2459                        else
2460                                stp->sd_struionak = bp;
2461                        bp->b_next = NULL;
2462                        mutex_exit(&stp->sd_lock);
2463                        return (0);
2464                }
2465                mutex_exit(&stp->sd_lock);

2467                bp->b_datap->db_type = M_IOCNAK;
2468                /*
2469                 * Protect against the driver passing up
2470                 * messages after it has done a qprocsoff.
2471                 */
2472                if (_OTHERQ(q)->q_next == NULL)
2473                        freemsg(bp);
2474                else
2475                        qreply(q, bp);
2476                return (0);

2478        default:
2479 #ifdef DEBUG
2480                cmn_err(CE_WARN,
2481                    "bad message type %x received at stream head\n",
2482                    bp->b_datap->db_type);
2483 #endif
2484                freemsg(bp);
2485                return (0);
2486        }

2488        /* NOTREACHED */
2489 }

2491 /*
2492  * Check if the stream pointed to by 'stp' can be written to, and return an
2493  * error code if not.  If 'eiohup' is set, then return EIO if STRHUP is set.
2494  * If 'sigpipeok' is set and the SW_SIGPIPE option is enabled on the stream,
2495  * then always return EPIPE and send a SIGPIPE to the invoking thread.
2496  */
2497 static int
2498 strwriteable(struct stdata *stp, boolean_t eiohup, boolean_t sigpipeok)
2499 {
2500        int error;

2502        ASSERT(MUTEX_HELD(&stp->sd_lock));
```

```
2504         /*
2505          * For modem support, POSIX states that on writes, EIO should
2506          * be returned if the stream has been hung up.
2507          */
2508         if (eiohup && (stp->sd_flag & (STPLEX|STRHUP)) == STRHUP)
2509                 error = EIO;
2510         else
2511                 error = strgeterr(stp, STRHUP|STPLEX|STWRERR, 0);

2513         if (error != 0) {
2514                 if (!(stp->sd_flag & STPLEX) &&
2515                     (stp->sd_wput_opt & SW_SIGPIPE) && sigpipeok) {
2516                         tsignal(curthread, SIGPIPE);
2517                         error = EPIPE;
2518                 }
2519         }

2521         return (error);
2522 }

2524 /*
2525  * Copyin and send data down a stream.
2526  * The caller will allocate and copyin any control part that precedes the
2527  * message and pass that in as mctl.
2528  *
2529  * Caller should *not* hold sd_lock.
2530  * When EWOULDBLOCK is returned the caller has to redo the canputnext
2531  * under sd_lock in order to avoid missing a backenabling wakeup.
2532  *
2533  * Use iosize = -1 to not send any M_DATA. iosize = 0 sends zero-length M_DATA.
2534  *
2535  * Set MSG_IGNFLOW in flags to ignore flow control for hipri messages.
2536  * For sync streams we can only ignore flow control by reverting to using
2537  * putnext.
2538  *
2539  * If sd_maxblk is less than *iosize this routine might return without
2540  * transferring all of *iosize. In all cases, on return *iosize will contain
2541  * the amount of data that was transferred.
2542  */
2543 static int
2544 strput(struct stdata *stp, mblk_t *mctl, struct uio *uiop, ssize_t *iosize,
2545     int b_flag, int pri, int flags)
2546 {
2547         struiod_t uiod;
2548         mblk_t *mp;
2549         queue_t *wqp = stp->sd_wrq;
2550         int error = 0;
2551         ssize_t count = *iosize;

2553         ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));

2555         if (uiop != NULL && count >= 0)
2556                 flags |= stp->sd_struiowrq ? STRUIO_POSTPONE : 0;

2558         if (!(flags & STRUIO_POSTPONE)) {
2559                 /*
2560                  * Use regular canputnext, strmakedata, putnext sequence.
2561                  */
2562                 if (pri == 0) {
2563                         if (!canputnext(wqp) && !(flags & MSG_IGNFLOW)) {
2564                                 freemsg(mctl);
2565                                 return (EWOULDBLOCK);
2566                         }
2567                 } else {
2568                         if (!(flags & MSG_IGNFLOW) && !bcanputnext(wqp, pri)) {
2569                                 freemsg(mctl);
```



```
2570                                 return (EWOULDBLOCK);
2571                         }
2572                 }

2574                 if ((error = strmakedata(iosize, uiop, stp, flags,
2575                     &mp)) != 0) {
2576                         freemsg(mctl);
2577                         /*
2578                          * need to change return code to ENOMEM
2579                          * so that this is not confused with
2580                          * flow control, EAGAIN.
2581                          */

2583                         if (error == EAGAIN)
2584                                 return (ENOMEM);
2585                         else
2586                                 return (error);
2587                 }
2588                 if (mctl != NULL) {
2589                         if (mctl->b_cont == NULL)
2590                                 mctl->b_cont = mp;
2591                         else if (mp != NULL)
2592                                 linkb(mctl, mp);
2593                         mp = mctl;
2594                 } else if (mp == NULL)
2595                         return (0);

2597                 mp->b_flag |= b_flag;
2598                 mp->b_band = (uchar_t)pri;

2600                 if (flags & MSG_IGNFLOW) {
2601                         /*
2602                          * XXX Hack: Don't get stuck running service
2603                          * procedures. This is needed for sockfs when
2604                          * sending the unbind message out of the rput
2605                          * procedure - we don't want a put procedure
2606                          * to run service procedures.
2607                          */
2608                         putnext(wqp, mp);
2609                 } else {
2610                         stream_willservice(stp);
2611                         putnext(wqp, mp);
2612                         stream_runservice(stp);
2613                 }
2614                 return (0);
2615         }
2616         /*
2617          * Stream supports rwnext() for the write side.
2618          */
2619         if ((error = strmakedata(iosize, uiop, stp, flags, &mp)) != 0) {
2620                 freemsg(mctl);
2621                 /*
2622                  * map EAGAIN to ENOMEM since EAGAIN means "flow controlled".
2623                  */
2624                 return (error == EAGAIN ? ENOMEM : error);
2625         }
2626         if (mctl != NULL) {
2627                 if (mctl->b_cont == NULL)
2628                         mctl->b_cont = mp;
2629                 else if (mp != NULL)
2630                         linkb(mctl, mp);
2631                 mp = mctl;
2632         } else if (mp == NULL) {
2633                 return (0);
2634         }
```

```
2636         mp->b_flag |= b_flag;
2637         mp->b_band = (uchar_t)pri;

2639         (void) uiodup(uiop, &uiod.d_uio, uiod.d_iov,
2640             sizeof (uiod.d_iov) / sizeof (*uiod.d_iov));
2641         uiod.d_uio.uio_offset = 0;
2642         uiod.d_mp = mp;
2643         error = rwnext(wqp, &uiod);
2644         if (! uiod.d_mp) {
2645                 uioskip(uiop, *iosize);
2646                 return (error);
2647         }
2648         ASSERT(mp == uiod.d_mp);
2649         if (error == EINVAL) {
2650                 /*
2651                  * The stream plumbing must have changed while
2652                  * we were away, so just turn off rwnext()s.
2653                  */
2654                 error = 0;
2655         } else if (error == EBUSY || error == EWOULDBLOCK) {
2656                 /*
2657                  * Couldn't enter a perimeter or took a page fault,
2658                  * so fall-back to putnext().
2659                  */
2660                 error = 0;
2661         } else {
2662                 freemsg(mp);
2663                 return (error);
2664         }
2665         /* Have to check canput before consuming data from the uio */
2666         if (pri == 0) {
2667                 if (!canputnext(wqp) && !(flags & MSG_IGNFLOW)) {
2668                         freemsg(mp);
2669                         return (EWOULDBLOCK);
2670                 }
2671         } else {
2672                 if (!bcanputnext(wqp, pri) && !(flags & MSG_IGNFLOW)) {
2673                         freemsg(mp);
2674                         return (EWOULDBLOCK);
2675                 }
2676         }
2677         ASSERT(mp == uiod.d_mp);
2678         /* Copyin data from the uio */
2679         if ((error = struioget(wqp, mp, &uiod, 0)) != 0) {
2680                 freemsg(mp);
2681                 return (error);
2682         }
2683         uioskip(uiop, *iosize);
2684         if (flags & MSG_IGNFLOW) {
2685                 /*
2686                  * XXX Hack: Don't get stuck running service procedures.
2687                  * This is needed for sockfs when sending the unbind message
2688                  * out of the rput procedure - we don't want a put procedure
2689                  * to run service procedures.
2690                  */
2691                 putnext(wqp, mp);
2692         } else {
2693                 stream_willservice(stp);
2694                 putnext(wqp, mp);
2695                 stream_runservice(stp);
2696         }
2697         return (0);
2698 }

2700 /*
2701  * Write attempts to break the write request into messages conforming
```



```
2702  * with the minimum and maximum packet sizes set downstream.
2703  *
2704  * Write will not block if downstream queue is full and
2705  * O_NDELAY is set, otherwise it will block waiting for the queue to get room.
2706  *
2707  * A write of zero bytes gets packaged into a zero length message and sent
2708  * downstream like any other message.
2709  *
2710  * If buffers of the requested sizes are not available, the write will
2711  * sleep until the buffers become available.
2712  *
2713  * Write (if specified) will supply a write offset in a message if it
2714  * makes sense. This can be specified by downstream modules as part of
2715  * a M_SETOPTS message.  Write will not supply the write offset if it
2716  * cannot supply any data in a buffer.  In other words, write will never
2717  * send down an empty packet due to a write offset.
2718  */
2719 /* ARGSUSED2 */
2720 int
2721 strwrite(struct vnode *vp, struct uio *uiop, cred_t *crp)
2722 {
2723         return (strwrite_common(vp, uiop, crp, 0));
2724 }

2726 /* ARGSUSED2 */
2727 int
2728 strwrite_common(struct vnode *vp, struct uio *uiop, cred_t *crp, int wflag)
2729 {
2730         struct stdata *stp;
2731         struct queue *wqp;
2732         ssize_t rmin, rmax;
2733         ssize_t iosize;
2734         int waitflag;
2735         int tempmode;
2736         int error = 0;
2737         int b_flag;

2739         ASSERT(vp->v_stream);
2740         stp = vp->v_stream;

2742         mutex_enter(&stp->sd_lock);

2744         if ((error = i_straccess(stp, JCWRITE)) != 0) {
2745                 mutex_exit(&stp->sd_lock);
2746                 return (error);
2747         }

2749         if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
2750                 error = strwriteable(stp, B_TRUE, B_TRUE);
2751                 if (error != 0) {
2752                         mutex_exit(&stp->sd_lock);
2753                         return (error);
2754                 }
2755         }

2757         mutex_exit(&stp->sd_lock);

2759         wqp = stp->sd_wrq;

2761         /* get these values from them cached in the stream head */
2762         rmin = stp->sd_qn_minpsz;
2763         rmax = stp->sd_qn_maxpsz;

2765         /*
2766          * Check the min/max packet size constraints.  If min packet size
2767          * is non-zero, the write cannot be split into multiple messages
```

```
2768          * and still guarantee the size constraints.
2769          */
2770         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_IN, "strwrite in:q %p", wqp);

2772         ASSERT((rmax >= 0) || (rmax == INFPSZ));
2773         if (rmax == 0) {
2774                 return (0);
2775         }
2776         if (rmin > 0) {
2777                 if (uiop->uio_resid < rmin) {
2778                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2779                             "strwrite out:q %p out %d error %d",
2780                             wqp, 0, ERANGE);
2781                         return (ERANGE);
2782                 }
2783                 if ((rmax != INFPSZ) && (uiop->uio_resid > rmax)) {
2784                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2785                             "strwrite out:q %p out %d error %d",
2786                             wqp, 1, ERANGE);
2787                         return (ERANGE);
2788                 }
2789         }

2791         /*
2792          * Do until count satisfied or error.
2793          */
2794         waitflag = WRITEWAIT | wflag;
2795         if (stp->sd_flag & OLDNDELAY)
2796                 tempmode = uiop->uio_fmode & ~FNDELAY;
2797         else
2798                 tempmode = uiop->uio_fmode;

2800         if (rmax == INFPSZ)
2801                 rmax = uiop->uio_resid;

2803         /*
2804          * Note that tempmode does not get used in strput/strmakedata
2805          * but only in strwaitq. The other routines use uio_fmode
2806          * unmodified.
2807          */

2809         /* LINTED: constant in conditional context */
2810         while (1) {     /* breaks when uio_resid reaches zero */
2811                 /*
2812                  * Determine the size of the next message to be
2813                  * packaged.  May have to break write into several
2814                  * messages based on max packet size.
2815                  */
2816                 iosize = MIN(uiop->uio_resid, rmax);

2818                 /*
2819                  * Put block downstream when flow control allows it.
2820                  */
2821                 if ((stp->sd_flag & STRDELIM) && (uiop->uio_resid == iosize))
2822                         b_flag = MSGDELIM;
2823                 else
2824                         b_flag = 0;

2826                 for (;;) {
2827                         int done = 0;

2829                         error = strput(stp, NULL, uiop, &iosize, b_flag, 0, 0);
2830                         if (error == 0)
2831                                 break;
2832                         if (error != EWOULDBLOCK)
2833                                 goto out;
```



```
2835                         mutex_enter(&stp->sd_lock);
2836                         /*
2837                          * Check for a missed wakeup.
2838                          * Needed since strput did not hold sd_lock across
2839                          * the canputnext.
2840                          */
2841                         if (canputnext(wqp)) {
2842                                 /* Try again */
2843                                 mutex_exit(&stp->sd_lock);
2844                                 continue;
2845                         }
2846                         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_WAIT,
2847                             "strwrite wait:q %p wait", wqp);
2848                         if ((error = strwaitq(stp, waitflag, (ssize_t)0,
2849                             tempmode, -1, &done)) != 0 || done) {
2850                                 mutex_exit(&stp->sd_lock);
2851                                 if ((vp->v_type == VFIFO) &&
2852                                     (uiop->uio_fmode & FNDELAY) &&
2853                                     (error == EAGAIN))
2854                                         error = 0;
2855                                 goto out;
2856                         }
2857                         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_WAKE,
2858                             "strwrite wake:q %p awakes", wqp);
2859                         if ((error = i_straccess(stp, JCWRITE)) != 0) {
2860                                 mutex_exit(&stp->sd_lock);
2861                                 goto out;
2862                         }
2863                         mutex_exit(&stp->sd_lock);
2864                 }
2865                 waitflag |= NOINTR;
2866                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRWRITE_RESID,
2867                     "strwrite resid:q %p uiop %p", wqp, uiop);
2868                 if (uiop->uio_resid) {
2869                         /* Recheck for errors - needed for sockets */
2870                         if ((stp->sd_wput_opt & SW_RECHECK_ERR) &&
2871                             (stp->sd_flag & (STWRERR|STRHUP|STPLEX))) {
2872                                 mutex_enter(&stp->sd_lock);
2873                                 error = strwriteable(stp, B_FALSE, B_TRUE);
2874                                 mutex_exit(&stp->sd_lock);
2875                                 if (error != 0)
2876                                         return (error);
2877                         }
2878                         continue;
2879                 }
2880                 break;
2881         }
2882 out:
2883         /*
2884          * For historical reasons, applications expect EAGAIN when a data
2885          * mblk_t cannot be allocated, so change ENOMEM back to EAGAIN.
2886          */
2887         if (error == ENOMEM)
2888                 error = EAGAIN;
2889         TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2890             "strwrite out:q %p out %d error %d", wqp, 2, error);
2891         return (error);
2892 }

2894 /*
2895  * Stream head write service routine.
2896  * Its job is to wake up any sleeping writers when a queue
2897  * downstream needs data (part of the flow control in putq and getq).
2898  * It also must wake anyone sleeping on a poll().
2899  * For stream head right below mux module, it must also invoke put procedure
```

```
2900  * of next downstream module.
2901  */
2902 int
2903 strwsrv(queue_t *q)
2904 {
2905         struct stdata *stp;
2906         queue_t *tq;
2907         qband_t *qbp;
2908         int i;
2909         qband_t *myqbp;
2910         int isevent;
2911         unsigned char   qbf[NBAND];     /* band flushing backenable flags */

2913         TRACE_1(TR_FAC_STREAMS_FR,
2914             TR_STRWSRV, "strwsrv:q %p", q);
2915         stp = (struct stdata *)q->q_ptr;
2916         ASSERT(qclaimed(q));
2917         mutex_enter(&stp->sd_lock);
2918         ASSERT(!(stp->sd_flag & STPLEX));

2920         if (stp->sd_flag & WSLEEP) {
2921                 stp->sd_flag &= ~WSLEEP;
2922                 cv_broadcast(&q->q_wait);
2923         }
2924         mutex_exit(&stp->sd_lock);

2926         /* The other end of a stream pipe went away. */
2927         if ((tq = q->q_next) == NULL) {
2928                 return (0);
2929         }

2931         /* Find the next module forward that has a service procedure */
2932         claimstr(q);
2933         tq = q->q_nfsrv;
2934         ASSERT(tq != NULL);

2936         if ((q->q_flag & QBACK)) {
2937                 if ((tq->q_flag & QFULL)) {
2938                         mutex_enter(QLOCK(tq));
2939                         if (!(tq->q_flag & QFULL)) {
2940                                 mutex_exit(QLOCK(tq));
2941                                 goto wakeup;
2942                         }
2943                         /*
2944                          * The queue must have become full again. Set QWANTW
2945                          * again so strwsrv will be back enabled when
2946                          * the queue becomes non-full next time.
2947                          */
2948                         tq->q_flag |= QWANTW;
2949                         mutex_exit(QLOCK(tq));
2950                 } else {
2951                 wakeup:
2952                         pollwakeup(&stp->sd_pollist, POLLWRNORM);
2953                         mutex_enter(&stp->sd_lock);
2954                         if (stp->sd_sigflags & S_WRNORM)
2955                                 strsendsig(stp->sd_siglist, S_WRNORM, 0, 0);
2956                         mutex_exit(&stp->sd_lock);
2957                 }
2958         }

2960         isevent = 0;
2961         i = 1;
2962         bzero((caddr_t)qbf, NBAND);
2963         mutex_enter(QLOCK(tq));
2964         if ((myqbp = q->q_bandp) != NULL)
2965                 for (qbp = tq->q_bandp; qbp && myqbp; qbp = qbp->qb_next) {
```



```
2966                         ASSERT(myqbp);
2967                         if ((myqbp->qb_flag & QB_BACK)) {
2968                                 if (qbp->qb_flag & QB_FULL) {
2969                                         /*
2970                                          * The band must have become full again.
2971                                          * Set QB_WANTW again so strwsrv will
2972                                          * be back enabled when the band becomes
2973                                          * non-full next time.
2974                                          */
2975                                         qbp->qb_flag |= QB_WANTW;
2976                                 } else {
2977                                         isevent = 1;
2978                                         qbf[i] = 1;
2979                                 }
2980                         }
2981                         myqbp = myqbp->qb_next;
2982                         i++;
2983                 }
2984         mutex_exit(QLOCK(tq));

2986         if (isevent) {
2987                 for (i = tq->q_nband; i; i--) {
2988                         if (qbf[i]) {
2989                                 pollwakeup(&stp->sd_pollist, POLLWRBAND);
2990                                 mutex_enter(&stp->sd_lock);
2991                                 if (stp->sd_sigflags & S_WRBAND)
2992                                         strsendsig(stp->sd_siglist, S_WRBAND,
2993                                             (uchar_t)i, 0);
2994                                 mutex_exit(&stp->sd_lock);
2995                         }
2996                 }
2997         }

2999         releasestr(q);
3000         return (0);
3001 }

3003 /*
3004  * Special case of strcopyin/strcopyout for copying
3005  * struct strioctl that can deal with both data
3006  * models.
3007  */

3009 #ifdef  _LP64

3011 static int
3012 strcopyin_strioctl(void *from, void *to, int flag, int copyflag)
3013 {
3014         struct  strioctl32 strioc32;
3015         struct  strioctl *striocp;

3017         if (copyflag & U_TO_K) {
3018                 ASSERT((copyflag & K_TO_K) == 0);

3020                 if ((flag & FMODELS) == DATAMODEL_ILP32) {
3021                         if (copyin(from, &strioc32, sizeof (strioc32)))
3022                                 return (EFAULT);

3024                         striocp = (struct strioctl *)to;
3025                         striocp->ic_cmd = strioc32.ic_cmd;
3026                         striocp->ic_timout = strioc32.ic_timout;
3027                         striocp->ic_len = strioc32.ic_len;
3028                         striocp->ic_dp  = (char *)(uintptr_t)strioc32.ic_dp;

3030                 } else { /* NATIVE data model */
3031                         if (copyin(from, to, sizeof (struct strioctl))) {
```

```
3032                                return (EFAULT);
3033                        } else {
3034                                return (0);
3035                        }
3036                }
3037        } else {
3038                ASSERT(copyflag & K_TO_K);
3039                bcopy(from, to, sizeof (struct strioctl));
3040        }
3041        return (0);
3042 }

3044 static int
3045 strcopyout_strioctl(void *from, void *to, int flag, int copyflag)
3046 {
3047        struct  strioctl32 strioc32;
3048        struct  strioctl *striocp;

3050        if (copyflag & U_TO_K) {
3051                ASSERT((copyflag & K_TO_K) == 0);

3053                if ((flag & FMODELS) == DATAMODEL_ILP32) {
3054                        striocp = (struct strioctl *)from;
3055                        strioc32.ic_cmd = striocp->ic_cmd;
3056                        strioc32.ic_timout = striocp->ic_timout;
3057                        strioc32.ic_len = striocp->ic_len;
3058                        strioc32.ic_dp  = (caddr32_t)(uintptr_t)striocp->ic_dp;
3059                        ASSERT((char *)(uintptr_t)strioc32.ic_dp ==
3060                            striocp->ic_dp);

3062                        if (copyout(&strioc32, to, sizeof (strioc32)))
3063                                return (EFAULT);

3065                } else { /* NATIVE data model */
3066                        if (copyout(from, to, sizeof (struct strioctl))) {
3067                                return (EFAULT);
3068                        } else {
3069                                return (0);
3070                        }
3071                }
3072        } else {
3073                ASSERT(copyflag & K_TO_K);
3074                bcopy(from, to, sizeof (struct strioctl));
3075        }
3076        return (0);
3077 }

3079 #else   /* ! _LP64 */

3081 /* ARGSUSED2 */
3082 static int
3083 strcopyin_strioctl(void *from, void *to, int flag, int copyflag)
3084 {
3085        return (strcopyin(from, to, sizeof (struct strioctl), copyflag));
3086 }

3088 /* ARGSUSED2 */
3089 static int
3090 strcopyout_strioctl(void *from, void *to, int flag, int copyflag)
3091 {
3092        return (strcopyout(from, to, sizeof (struct strioctl), copyflag));
3093 }

3095 #endif  /* _LP64 */

3097 /*
```



```
3098  * Determine type of job control semantics expected by user.  The
3099  * possibilities are:
3100  *      JCREAD  - Behaves like read() on fd; send SIGTTIN
3101  *      JCWRITE - Behaves like write() on fd; send SIGTTOU if TOSTOP set
3102  *      JCSETP  - Sets a value in the stream; send SIGTTOU, ignore TOSTOP
3103  *      JCGETP  - Gets a value in the stream; no signals.
3104  * See straccess in strsubr.c for usage of these values.
3105  *
3106  * This routine also returns -1 for I_STR as a special case; the
3107  * caller must call again with the real ioctl number for
3108  * classification.
3109  */
3110 static int
3111 job_control_type(int cmd)
3112 {
3113        switch (cmd) {
3114        case I_STR:
3115                return (-1);

3117        case I_RECVFD:
3118        case I_E_RECVFD:
3119                return (JCREAD);

3121        case I_FDINSERT:
3122        case I_SENDFD:
3123                return (JCWRITE);

3125        case TCSETA:
3126        case TCSETAW:
3127        case TCSETAF:
3128        case TCSBRK:
3129        case TCXONC:
3130        case TCFLSH:
3131        case TCDSET:    /* Obsolete */
3132        case TIOCSWINSZ:
3133        case TCSETS:
3134        case TCSETSW:
3135        case TCSETSF:
3136        case TIOCSETD:
3137        case TIOCHPCL:
3138        case TIOCSETP:
3139        case TIOCSETN:
3140        case TIOCEXCL:
3141        case TIOCNXCL:
3142        case TIOCFLUSH:
3143        case TIOCSETC:
3144        case TIOCLBIS:
3145        case TIOCLBIC:
3146        case TIOCLSET:
3147        case TIOCSBRK:
3148        case TIOCCBRK:
3149        case TIOCSDTR:
3150        case TIOCCDTR:
3151        case TIOCSLTC:
3152        case TIOCSTOP:
3153        case TIOCSTART:
3154        case TIOCSTI:
3155        case TIOCSPGRP:
3156        case TIOCMSET:
3157        case TIOCMBIS:
3158        case TIOCMBIC:
3159        case TIOCREMOTE:
3160        case TIOCSIGNAL:
3161        case LDSETT:
3162        case LDSMAP:    /* Obsolete */
3163        case DIOCSETP:
```

```
3164         case I_FLUSH:
3165         case I_SRDOPT:
3166         case I_SETSIG:
3167         case I_SWROPT:
3168         case I_FLUSHBAND:
3169         case I_SETCLTIME:
3170         case I_SERROPT:
3171         case I_ESETSIG:
3172         case FIONBIO:
3173         case FIOASYNC:
3174         case FIOSETOWN:
3175         case JBOOT:     /* Obsolete */
3176         case JTERM:     /* Obsolete */
3177         case JTIMOM:    /* Obsolete */
3178         case JZOMBOOT:  /* Obsolete */
3179         case JAGENT:    /* Obsolete */
3180         case JTRUN:     /* Obsolete */
3181         case JXTPROTO:  /* Obsolete */
3182                 return (JCSETP);
3183         }

3185         return (JCGETP);
3186 }

3188 /*
3189  * ioctl for streams
3190  */
3191 int
3192 strioctl(struct vnode *vp, int cmd, intptr_t arg, int flag, int copyflag,
3193     cred_t *crp, int *rvalp)
3194 {
3195         struct stdata *stp;
3196         struct strcmd *scp;
3197         struct strioctl strioc;
3198         struct uio uio;
3199         struct iovec iov;
3200         int access;
3201         mblk_t *mp;
3202         int error = 0;
3203         int done = 0;
3204         ssize_t rmin, rmax;
3205         queue_t *wrq;
3206         queue_t *rdq;
3207         boolean_t kioctl = B_FALSE;
3208         uint32_t auditing = AU_AUDITING();

3210         if (flag & FKIOCTL) {
3211                 copyflag = K_TO_K;
3212                 kioctl = B_TRUE;
3213         }
3214         ASSERT(vp->v_stream);
3215         ASSERT(copyflag == U_TO_K || copyflag == K_TO_K);
3216         stp = vp->v_stream;

3218         TRACE_3(TR_FAC_STREAMS_FR, TR_IOCTL_ENTER,
3219             "strioctl:stp %p cmd %X arg %lX", stp, cmd, arg);

3221         /*
3222          * If the copy is kernel to kernel, make sure that the FNATIVE
3223          * flag is set.  After this it would be a serious error to have
3224          * no model flag.
3225          */
3226         if (copyflag == K_TO_K)
3227                 flag = (flag & ~FMODELS) | FNATIVE;

3229         ASSERT((flag & FMODELS) != 0);
```



```
3231         wrq = stp->sd_wrq;
3232         rdq = _RD(wrq);

3234         access = job_control_type(cmd);

3236         /* We should never see these here, should be handled by iwscn */
3237         if (cmd == SRIOCSREDIR || cmd == SRIOCISREDIR)
3238                 return (EINVAL);

3240         mutex_enter(&stp->sd_lock);
3241         if ((access != -1) && ((error = i_straccess(stp, access)) != 0)) {
3242                 mutex_exit(&stp->sd_lock);
3243                 return (error);
3244         }
3245         mutex_exit(&stp->sd_lock);

3247         /*
3248          * Check for sgttyb-related ioctls first, and complain as
3249          * necessary.
3250          */
3251         switch (cmd) {
3252         case TIOCGETP:
3253         case TIOCSETP:
3254         case TIOCSETN:
3255                 if (sgttyb_handling >= 2 && !sgttyb_complaint) {
3256                         sgttyb_complaint = B_TRUE;
3257                         cmn_err(CE_NOTE,
3258                             "application used obsolete TIOC[GS]ET");
3259                 }
3260                 if (sgttyb_handling >= 3) {
3261                         tsignal(curthread, SIGSYS);
3262                         return (EIO);
3263                 }
3264                 break;
3265         }

3267         mutex_enter(&stp->sd_lock);

3269         switch (cmd) {
3270         case I_RECVFD:
3271         case I_E_RECVFD:
3272         case I_PEEK:
3273         case I_NREAD:
3274         case FIONREAD:
3275         case FIORDCHK:
3276         case I_ATMARK:
3277         case FIONBIO:
3278         case FIOASYNC:
3279                 if (stp->sd_flag & (STRDERR|STPLEX)) {
3280                         error = strgeterr(stp, STRDERR|STPLEX, 0);
3281                         if (error != 0) {
3282                                 mutex_exit(&stp->sd_lock);
3283                                 return (error);
3284                         }
3285                 }
3286                 break;

3288         default:
3289                 if (stp->sd_flag & (STRDERR|STWRERR|STPLEX)) {
3290                         error = strgeterr(stp, STRDERR|STWRERR|STPLEX, 0);
3291                         if (error != 0) {
3292                                 mutex_exit(&stp->sd_lock);
3293                                 return (error);
3294                         }
3295                 }
```

```
3296         }

3298         mutex_exit(&stp->sd_lock);

3300         switch (cmd) {
3301         default:
3302                 /*
3303                  * The stream head has hardcoded knowledge of a
3304                  * miscellaneous collection of terminal-, keyboard- and
3305                  * mouse-related ioctls, enumerated below.  This hardcoded
3306                  * knowledge allows the stream head to automatically
3307                  * convert transparent ioctl requests made by userland
3308                  * programs into I_STR ioctls which many old STREAMS
3309                  * modules and drivers require.
3310                  *
3311                  * No new ioctls should ever be added to this list.
3312                  * Instead, the STREAMS module or driver should be written
3313                  * to either handle transparent ioctls or require any
3314                  * userland programs to use I_STR ioctls (by returning
3315                  * EINVAL to any transparent ioctl requests).
3316                  *
3317                  * More importantly, removing ioctls from this list should
3318                  * be done with the utmost care, since our STREAMS modules
3319                  * and drivers *count* on the stream head performing this
3320                  * conversion, and thus may panic while processing
3321                  * transparent ioctl request for one of these ioctls (keep
3322                  * in mind that third party modules and drivers may have
3323                  * similar problems).
3324                  */
3325                 if (((cmd & IOCTYPE) == LDIOC) ||
3326                     ((cmd & IOCTYPE) == tIOC) ||
3327                     ((cmd & IOCTYPE) == TIOC) ||
3328                     ((cmd & IOCTYPE) == KIOC) ||
3329                     ((cmd & IOCTYPE) == MSIOC) ||
3330                     ((cmd & IOCTYPE) == VUIOC)) {
3331                         /*
3332                          * The ioctl is a tty ioctl - set up strioc buffer
3333                          * and call strdoioctl() to do the work.
3334                          */
3335                         if (stp->sd_flag & STRHUP)
3336                                 return (ENXIO);
3337                         strioc.ic_cmd = cmd;
3338                         strioc.ic_timout = INFTIM;

3340                         switch (cmd) {

3342                         case TCXONC:
3343                         case TCSBRK:
3344                         case TCFLSH:
3345                         case TCDSET:
3346                                 {
3347                                 int native_arg = (int)arg;
3348                                 strioc.ic_len = sizeof (int);
3349                                 strioc.ic_dp = (char *)&native_arg;
3350                                 return (strdoioctl(stp, &strioc, flag,
3351                                     K_TO_K, crp, rvalp));
3352                                 }

3354                         case TCSETA:
3355                         case TCSETAW:
3356                         case TCSETAF:
3357                                 strioc.ic_len = sizeof (struct termio);
3358                                 strioc.ic_dp = (char *)arg;
3359                                 return (strdoioctl(stp, &strioc, flag,
3360                                     copyflag, crp, rvalp));
```



```
3362                         case TCSETS:
3363                         case TCSETSW:
3364                         case TCSETSF:
3365                                 strioc.ic_len = sizeof (struct termios);
3366                                 strioc.ic_dp = (char *)arg;
3367                                 return (strdoioctl(stp, &strioc, flag,
3368                                     copyflag, crp, rvalp));

3370                         case LDSETT:
3371                                 strioc.ic_len = sizeof (struct termcb);
3372                                 strioc.ic_dp = (char *)arg;
3373                                 return (strdoioctl(stp, &strioc, flag,
3374                                     copyflag, crp, rvalp));

3376                         case TIOCSETP:
3377                                 strioc.ic_len = sizeof (struct sgttyb);
3378                                 strioc.ic_dp = (char *)arg;
3379                                 return (strdoioctl(stp, &strioc, flag,
3380                                     copyflag, crp, rvalp));

3382                         case TIOCSTI:
3383                                 if ((flag & FREAD) == 0 &&
3384                                     secpolicy_sti(crp) != 0) {
3385                                         return (EPERM);
3386                                 }
3387                                 mutex_enter(&stp->sd_lock);
3388                                 mutex_enter(&curproc->p_splock);
3389                                 if (stp->sd_sidp != curproc->p_sessp->s_sidp &&
3390                                     secpolicy_sti(crp) != 0) {
3391                                         mutex_exit(&curproc->p_splock);
3392                                         mutex_exit(&stp->sd_lock);
3393                                         return (EACCES);
3394                                 }
3395                                 mutex_exit(&curproc->p_splock);
3396                                 mutex_exit(&stp->sd_lock);

3398                                 strioc.ic_len = sizeof (char);
3399                                 strioc.ic_dp = (char *)arg;
3400                                 return (strdoioctl(stp, &strioc, flag,
3401                                     copyflag, crp, rvalp));

3403                         case TIOCSWINSZ:
3404                                 strioc.ic_len = sizeof (struct winsize);
3405                                 strioc.ic_dp = (char *)arg;
3406                                 return (strdoioctl(stp, &strioc, flag,
3407                                     copyflag, crp, rvalp));

3409                         case TIOCSSIZE:
3410                                 strioc.ic_len = sizeof (struct ttysize);
3411                                 strioc.ic_dp = (char *)arg;
3412                                 return (strdoioctl(stp, &strioc, flag,
3413                                     copyflag, crp, rvalp));

3415                         case TIOCSSOFTCAR:
3416                         case KIOCTRANS:
3417                         case KIOCTRANSABLE:
3418                         case KIOCCMD:
3419                         case KIOCSDIRECT:
3420                         case KIOCSCOMPAT:
3421                         case KIOCSKABORTEN:
3422                         case KIOCSRPTDELAY:
3423                         case KIOCSRPTRATE:
3424                         case VUIDSFORMAT:
3425                         case TIOCSPPS:
3426                                 strioc.ic_len = sizeof (int);
3427                                 strioc.ic_dp = (char *)arg;
```

```
3428                                    return (strdoioctl(stp, &strioc, flag,
3429                                        copyflag, crp, rvalp));
3430
3431                            case KIOCSETKEY:
3432                            case KIOCGETKEY:
3433                                    strioc.ic_len = sizeof (struct kiockey);
3434                                    strioc.ic_dp = (char *)arg;
3435                                    return (strdoioctl(stp, &strioc, flag,
3436                                        copyflag, crp, rvalp));
3437
3438                            case KIOCSKEY:
3439                            case KIOCGKEY:
3440                                    strioc.ic_len = sizeof (struct kiockeymap);
3441                                    strioc.ic_dp = (char *)arg;
3442                                    return (strdoioctl(stp, &strioc, flag,
3443                                        copyflag, crp, rvalp));
3444
3445                            case KIOCSLED:
3446                                    /* arg is a pointer to char */
3447                                    strioc.ic_len = sizeof (char);
3448                                    strioc.ic_dp = (char *)arg;
3449                                    return (strdoioctl(stp, &strioc, flag,
3450                                        copyflag, crp, rvalp));
3451
3452                            case MSIOSETPARMS:
3453                                    strioc.ic_len = sizeof (Ms_parms);
3454                                    strioc.ic_dp = (char *)arg;
3455                                    return (strdoioctl(stp, &strioc, flag,
3456                                        copyflag, crp, rvalp));
3457
3458                            case VUIDSADDR:
3459                            case VUIDGADDR:
3460                                    strioc.ic_len = sizeof (struct vuid_addr_probe);
3461                                    strioc.ic_dp = (char *)arg;
3462                                    return (strdoioctl(stp, &strioc, flag,
3463                                        copyflag, crp, rvalp));
3464
3465                            /*
3466                             * These M_IOCTL's don't require any data to be sent
3467                             * downstream, and the driver will allocate and link
3468                             * on its own mblk_t upon M_IOCACK -- thus we set
3469                             * ic_len to zero and set ic_dp to arg so we know
3470                             * where to copyout to later.
3471                             */
3472                            case TIOCGSOFTCAR:
3473                            case TIOCGWINSZ:
3474                            case TIOCGSIZE:
3475                            case KIOCGTRANS:
3476                            case KIOCGTRANSABLE:
3477                            case KIOCTYPE:
3478                            case KIOCGDIRECT:
3479                            case KIOCGCOMPAT:
3480                            case KIOCLAYOUT:
3481                            case KIOCGLED:
3482                            case MSIOGETPARMS:
3483                            case MSIOBUTTONS:
3484                            case VUIDGFORMAT:
3485                            case TIOCGPPS:
3486                            case TIOCGPPSEV:
3487                            case TCGETA:
3488                            case TCGETS:
3489                            case LDGETT:
3490                            case TIOCGETP:
3491                            case KIOCGRPTDELAY:
3492                            case KIOCGRPTRATE:
3493                                    strioc.ic_len = 0;
```



```
3494                                    strioc.ic_dp = (char *)arg;
3495                                    return (strdoioctl(stp, &strioc, flag,
3496                                        copyflag, crp, rvalp));
3497                            }
3498                    }
3499
3500                    /*
3501                     * Unknown cmd - send it down as a transparent ioctl.
3502                     */
3503                    strioc.ic_cmd = cmd;
3504                    strioc.ic_timout = INFTIM;
3505                    strioc.ic_len = TRANSPARENT;
3506                    strioc.ic_dp = (char *)&arg;
3507
3508                    return (strdoioctl(stp, &strioc, flag, copyflag, crp, rvalp));
3509
3510            case I_STR:
3511                    /*
3512                     * Stream ioctl.  Read in an strioctl buffer from the user
3513                     * along with any data specified and send it downstream.
3514                     * Strdoioctl will wait allow only one ioctl message at
3515                     * a time, and waits for the acknowledgement.
3516                     */
3517
3518                    if (stp->sd_flag & STRHUP)
3519                            return (ENXIO);
3520
3521                    error = strcopyin_strioctl((void *)arg, &strioc, flag,
3522                        copyflag);
3523                    if (error != 0)
3524                            return (error);
3525
3526                    if ((strioc.ic_len < 0) || (strioc.ic_timout < -1))
3527                            return (EINVAL);
3528
3529                    access = job_control_type(strioc.ic_cmd);
3530                    mutex_enter(&stp->sd_lock);
3531                    if ((access != -1) &&
3532                        ((error = i_straccess(stp, access)) != 0)) {
3533                            mutex_exit(&stp->sd_lock);
3534                            return (error);
3535                    }
3536                    mutex_exit(&stp->sd_lock);
3537
3538                    /*
3539                     * The I_STR facility provides a trap door for malicious
3540                     * code to send down bogus streamio(7I) ioctl commands to
3541                     * unsuspecting STREAMS modules and drivers which expect to
3542                     * only get these messages from the stream head.
3543                     * Explicitly prohibit any streamio ioctls which can be
3544                     * passed downstream by the stream head.  Note that we do
3545                     * not block all streamio ioctls because the ioctl
3546                     * numberspace is not well managed and thus it's possible
3547                     * that a module or driver's ioctl numbers may accidentally
3548                     * collide with them.
3549                     */
3550                    switch (strioc.ic_cmd) {
3551                    case I_LINK:
3552                    case I_PLINK:
3553                    case I_UNLINK:
3554                    case I_PUNLINK:
3555                    case _I_GETPEERCRED:
3556                    case _I_PLINK_LH:
3557                            return (EINVAL);
3558                    }
```

```
3560                error = strdoioctl(stp, &strioc, flag, copyflag, crp, rvalp);
3561                if (error == 0) {
3562                        error = strcopyout_strioctl(&strioc, (void *)arg,
3563                            flag, copyflag);
3564                }
3565                return (error);

3567        case _I_CMD:
3568                /*
3569                 * Like I_STR, but without using M_IOC* messages and without
3570                 * copyins/copyouts beyond the passed-in argument.
3571                 */
3572                if (stp->sd_flag & STRHUP)
3573                        return (ENXIO);

3575                if ((scp = kmem_alloc(sizeof (strcmd_t), KM_NOSLEEP)) == NULL)
3576                        return (ENOMEM);

3578                if (copyin((void *)arg, scp, sizeof (strcmd_t))) {
3579                        kmem_free(scp, sizeof (strcmd_t));
3580                        return (EFAULT);
3581                }

3583                access = job_control_type(scp->sc_cmd);
3584                mutex_enter(&stp->sd_lock);
3585                if (access != -1 && (error = i_straccess(stp, access)) != 0) {
3586                        mutex_exit(&stp->sd_lock);
3587                        kmem_free(scp, sizeof (strcmd_t));
3588                        return (error);
3589                }
3590                mutex_exit(&stp->sd_lock);

3592                *rvalp = 0;
3593                if ((error = strdocmd(stp, scp, crp)) == 0) {
3594                        if (copyout(scp, (void *)arg, sizeof (strcmd_t)))
3595                                error = EFAULT;
3596                }
3597                kmem_free(scp, sizeof (strcmd_t));
3598                return (error);

3600        case I_NREAD:
3601                /*
3602                 * Return number of bytes of data in first message
3603                 * in queue in "arg" and return the number of messages
3604                 * in queue in return value.
3605                 */
3606        {
3607                size_t  size;
3608                int     retval;
3609                int     count = 0;

3611                mutex_enter(QLOCK(rdq));

3613                size = msgdsize(rdq->q_first);
3614                for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3615                        count++;

3617                mutex_exit(QLOCK(rdq));
3618                if (stp->sd_struiordq) {
3619                        infod_t infod;

3621                        infod.d_cmd = INFOD_COUNT;
3622                        infod.d_count = 0;
3623                        if (count == 0) {
3624                                infod.d_cmd |= INFOD_FIRSTBYTES;
3625                                infod.d_bytes = 0;
```



```
3626                        }
3627                        infod.d_res = 0;
3628                        (void) infonext(rdq, &infod);
3629                        count += infod.d_count;
3630                        if (infod.d_res & INFOD_FIRSTBYTES)
3631                                size = infod.d_bytes;
3632                }

3634                /*
3635                 * Drop down from size_t to the "int" required by the
3636                 * interface.  Cap at INT_MAX.
3637                 */
3638                retval = MIN(size, INT_MAX);
3639                error = strcopyout(&retval, (void *)arg, sizeof (retval),
3640                    copyflag);
3641                if (!error)
3642                        *rvalp = count;
3643                return (error);
3644        }

3646        case FIONREAD:
3647                /*
3648                 * Return number of bytes of data in all data messages
3649                 * in queue in "arg".
3650                 */
3651        {
3652                size_t  size = 0;
3653                int     retval;

3655                mutex_enter(QLOCK(rdq));
3656                for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3657                        size += msgdsize(mp);
3658                mutex_exit(QLOCK(rdq));

3660                if (stp->sd_struiordq) {
3661                        infod_t infod;

3663                        infod.d_cmd = INFOD_BYTES;
3664                        infod.d_res = 0;
3665                        infod.d_bytes = 0;
3666                        (void) infonext(rdq, &infod);
3667                        size += infod.d_bytes;
3668                }

3670                /*
3671                 * Drop down from size_t to the "int" required by the
3672                 * interface.  Cap at INT_MAX.
3673                 */
3674                retval = MIN(size, INT_MAX);
3675                error = strcopyout(&retval, (void *)arg, sizeof (retval),
3676                    copyflag);

3678                *rvalp = 0;
3679                return (error);
3680        }
3681        case FIORDCHK:
3682                /*
3683                 * FIORDCHK does not use arg value (like FIONREAD),
3684                 * instead a count is returned. I_NREAD value may
3685                 * not be accurate but safe. The real thing to do is
3686                 * to add the msgdsizes of all data  messages until
3687                 * a non-data message.
3688                 */
3689        {
3690                size_t size = 0;
```

```
3692                mutex_enter(QLOCK(rdq));
3693                for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3694                        size += msgdsize(mp);
3695                mutex_exit(QLOCK(rdq));

3697                if (stp->sd_struiordq) {
3698                        infod_t infod;

3700                        infod.d_cmd = INFOD_BYTES;
3701                        infod.d_res = 0;
3702                        infod.d_bytes = 0;
3703                        (void) infonext(rdq, &infod);
3704                        size += infod.d_bytes;
3705                }

3707                /*
3708                 * Since ioctl returns an int, and memory sizes under
3709                 * LP64 may not fit, we return INT_MAX if the count was
3710                 * actually greater.
3711                 */
3712                *rvalp = MIN(size, INT_MAX);
3713                return (0);
3714        }

3716        case I_FIND:
3717                /*
3718                 * Get module name.
3719                 */
3720        {
3721                char mname[FMNAMESZ + 1];
3722                queue_t *q;

3724                error = (copyflag & U_TO_K ? copyinstr : copystr)((void *)arg,
3725                    mname, FMNAMESZ + 1, NULL);
3726                if (error)
3727                        return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

3729                /*
3730                 * Return EINVAL if we're handed a bogus module name.
3731                 */
3732                if (fmodsw_find(mname, FMODSW_LOAD) == NULL) {
3733                        TRACE_0(TR_FAC_STREAMS_FR,
3734                            TR_I_CANT_FIND, "couldn't I_FIND");
3735                        return (EINVAL);
3736                }

3738                *rvalp = 0;

3740                /* Look downstream to see if module is there. */
3741                claimstr(stp->sd_wrq);
3742                for (q = stp->sd_wrq->q_next; q; q = q->q_next) {
3743                        if (q->q_flag & QREADR) {
3744                                q = NULL;
3745                                break;
3746                        }
3747                        if (strcmp(mname, Q2NAME(q)) == 0)
3748                                break;
3749                }
3750                releasestr(stp->sd_wrq);

3752                *rvalp = (q ? 1 : 0);
3753                return (error);
3754        }

3756        case I_PUSH:
3757        case __I_PUSH_NOCTTY:
```



```
3758                /*
3759                 * Push a module.
3760                 * For the case __I_PUSH_NOCTTY push a module but
3761                 * do not allocate controlling tty. See bugid 4025044
3762                 */

3764        {
3765                char mname[FMNAMESZ + 1];
3766                fmodsw_impl_t *fp;
3767                dev_t dummydev;

3769                if (stp->sd_flag & STRHUP)
3770                        return (ENXIO);

3772                /*
3773                 * Get module name and look up in fmodsw.
3774                 */
3775                error = (copyflag & U_TO_K ? copyinstr : copystr)((void *)arg,
3776                    mname, FMNAMESZ + 1, NULL);
3777                if (error)
3778                        return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

3780                if ((fp = fmodsw_find(mname, FMODSW_HOLD | FMODSW_LOAD)) ==
3781                    NULL)
3782                        return (EINVAL);

3784                TRACE_2(TR_FAC_STREAMS_FR, TR_I_PUSH,
3785                    "I_PUSH:fp %p stp %p", fp, stp);

3787                if (error = strstartplumb(stp, flag, cmd)) {
3788                        fmodsw_rele(fp);
3789                        return (error);
3790                }

3792                /*
3793                 * See if any more modules can be pushed on this stream.
3794                 * Note that this check must be done after strstartplumb()
3795                 * since otherwise multiple threads issuing I_PUSHes on
3796                 * the same stream will be able to exceed nstrpush.
3797                 */
3798                mutex_enter(&stp->sd_lock);
3799                if (stp->sd_pushcnt >= nstrpush) {
3800                        fmodsw_rele(fp);
3801                        strendplumb(stp);
3802                        mutex_exit(&stp->sd_lock);
3803                        return (EINVAL);
3804                }
3805                mutex_exit(&stp->sd_lock);

3807                /*
3808                 * Push new module and call its open routine
3809                 * via qattach().  Modules don't change device
3810                 * numbers, so just ignore dummydev here.
3811                 */
3812                dummydev = vp->v_rdev;
3813                if ((error = qattach(rdq, &dummydev, 0, crp, fp,
3814                    B_FALSE)) == 0) {
3815                        if (vp->v_type == VCHR && /* sorry, no pipes allowed */
3816                            (cmd == I_PUSH) && (stp->sd_flag & STRISTTY)) {
3817                                /*
3818                                 * try to allocate it as a controlling terminal
3819                                 */
3820                                (void) strctty(stp);
3821                        }
3822                }
```

```
3824                mutex_enter(&stp->sd_lock);
3826                /*
3827                 * As a performance concern we are caching the values of
3828                 * q_minpsz and q_maxpsz of the module below the stream
3829                 * head in the stream head.
3830                 */
3831                mutex_enter(QLOCK(stp->sd_wrq->q_next));
3832                rmin = stp->sd_wrq->q_next->q_minpsz;
3833                rmax = stp->sd_wrq->q_next->q_maxpsz;
3834                mutex_exit(QLOCK(stp->sd_wrq->q_next));
3836                /* Do this processing here as a performance concern */
3837                if (strmsgsz != 0) {
3838                        if (rmax == INFPSZ)
3839                                rmax = strmsgsz;
3840                        else  {
3841                                if (vp->v_type == VFIFO)
3842                                        rmax = MIN(PIPE_BUF, rmax);
3843                                else    rmax = MIN(strmsgsz, rmax);
3844                        }
3845                }
3847                mutex_enter(QLOCK(wrq));
3848                stp->sd_qn_minpsz = rmin;
3849                stp->sd_qn_maxpsz = rmax;
3850                mutex_exit(QLOCK(wrq));
3852                strendplumb(stp);
3853                mutex_exit(&stp->sd_lock);
3854                return (error);
3855        }
3857        case I_POP:
3858        {
3859                queue_t *q;
3861                if (stp->sd_flag & STRHUP)
3862                        return (ENXIO);
3863                if (!wrq->q_next)       /* for broken pipes */
3864                        return (EINVAL);
3866                if (error = strstartplumb(stp, flag, cmd))
3867                        return (error);
3869                /*
3870                 * If there is an anchor on this stream and popping
3871                 * the current module would attempt to pop through the
3872                 * anchor, then disallow the pop unless we have sufficient
3873                 * privileges; take the cheapest (non-locking) check
3874                 * first.
3875                 */
3876                if (secpolicy_ip_config(crp, B_TRUE) != 0 ||
3877                    (stp->sd_anchorzone != crgetzoneid(crp))) {
3878                        mutex_enter(&stp->sd_lock);
3879                        /*
3880                         * Anchors only apply if there's at least one
3881                         * module on the stream (sd_pushcnt > 0).
3882                         */
3883                        if (stp->sd_pushcnt > 0 &&
3884                            stp->sd_pushcnt == stp->sd_anchor &&
3885                            stp->sd_vnode->v_type != VFIFO) {
3886                                strendplumb(stp);
3887                                mutex_exit(&stp->sd_lock);
3888                                if (stp->sd_anchorzone != crgetzoneid(crp))
3889                                        return (EINVAL);
```



```
3890                                /* Audit and report error */
3891                                return (secpolicy_ip_config(crp, B_FALSE));
3892                        }
3893                        mutex_exit(&stp->sd_lock);
3894                }
3896                q = wrq->q_next;
3897                TRACE_2(TR_FAC_STREAMS_FR, TR_I_POP,
3898                    "I_POP:%p from %p", q, stp);
3899                if (q->q_next == NULL || (q->q_flag & (QREADR|QISDRV))) {
3900                        error = EINVAL;
3901                } else {
3902                        qdetach(_RD(q), 1, flag, crp, B_FALSE);
3903                        error = 0;
3904                }
3905                mutex_enter(&stp->sd_lock);
3907                /*
3908                 * As a performance concern we are caching the values of
3909                 * q_minpsz and q_maxpsz of the module below the stream
3910                 * head in the stream head.
3911                 */
3912                mutex_enter(QLOCK(wrq->q_next));
3913                rmin = wrq->q_next->q_minpsz;
3914                rmax = wrq->q_next->q_maxpsz;
3915                mutex_exit(QLOCK(wrq->q_next));
3917                /* Do this processing here as a performance concern */
3918                if (strmsgsz != 0) {
3919                        if (rmax == INFPSZ)
3920                                rmax = strmsgsz;
3921                        else  {
3922                                if (vp->v_type == VFIFO)
3923                                        rmax = MIN(PIPE_BUF, rmax);
3924                                else    rmax = MIN(strmsgsz, rmax);
3925                        }
3926                }
3928                mutex_enter(QLOCK(wrq));
3929                stp->sd_qn_minpsz = rmin;
3930                stp->sd_qn_maxpsz = rmax;
3931                mutex_exit(QLOCK(wrq));
3933                /* If we popped through the anchor, then reset the anchor. */
3934                if (stp->sd_pushcnt < stp->sd_anchor) {
3935                        stp->sd_anchor = 0;
3936                        stp->sd_anchorzone = 0;
3937                }
3938                strendplumb(stp);
3939                mutex_exit(&stp->sd_lock);
3940                return (error);
3941        }
3943        case _I_MUXID2FD:
3944        {
3945                /*
3946                 * Create a fd for a I_PLINK'ed lower stream with a given
3947                 * muxid.  With the fd, application can send down ioctls,
3948                 * like I_LIST, to the previously I_PLINK'ed stream.  Note
3949                 * that after getting the fd, the application has to do an
3950                 * I_PUNLINK on the muxid before it can do any operation
3951                 * on the lower stream.  This is required by spec1170.
3952                 *
3953                 * The fd used to do this ioctl should point to the same
3954                 * controlling device used to do the I_PLINK.  If it uses
3955                 * a different stream or an invalid muxid, I_MUXID2FD will
```

```
3956                 * fail.  The error code is set to EINVAL.
3957                 *
3958                 * The intended use of this interface is the following.
3959                 * An application I_PLINK'ed a stream and exits.  The fd
3960                 * to the lower stream is gone.  Another application
3961                 * wants to get a fd to the lower stream, it uses I_MUXID2FD.
3962                 */
3963                int muxid = (int)arg;
3964                int fd;
3965                linkinfo_t *linkp;
3966                struct file *fp;
3967                netstack_t *ns;
3968                str_stack_t *ss;

3970                /*
3971                 * Do not allow the wildcard muxid.  This ioctl is not
3972                 * intended to find arbitrary link.
3973                 */
3974                if (muxid == 0) {
3975                        return (EINVAL);
3976                }

3978                ns = netstack_find_by_cred(crp);
3979                ASSERT(ns != NULL);
3980                ss = ns->netstack_str;
3981                ASSERT(ss != NULL);

3983                mutex_enter(&muxifier);
3984                linkp = findlinks(vp->v_stream, muxid, LINKPERSIST, ss);
3985                if (linkp == NULL) {
3986                        mutex_exit(&muxifier);
3987                        netstack_rele(ss->ss_netstack);
3988                        return (EINVAL);
3989                }

3991                if ((fd = ufalloc(0)) == -1) {
3992                        mutex_exit(&muxifier);
3993                        netstack_rele(ss->ss_netstack);
3994                        return (EMFILE);
3995                }
3996                fp = linkp->li_fpdown;
3997                mutex_enter(&fp->f_tlock);
3998                fp->f_count++;
3999                mutex_exit(&fp->f_tlock);
4000                mutex_exit(&muxifier);
4001                setf(fd, fp);
4002                *rvalp = fd;
4003                netstack_rele(ss->ss_netstack);
4004                return (0);
4005        }

4007        case _I_INSERT:
4008        {
4009                /*
4010                 * To insert a module to a given position in a stream.
4011                 * In the first release, only allow privileged user
4012                 * to use this ioctl. Furthermore, the insert is only allowed
4013                 * below an anchor if the zoneid is the same as the zoneid
4014                 * which created the anchor.
4015                 *
4016                 * Note that we do not plan to support this ioctl
4017                 * on pipes in the first release.  We want to learn more
4018                 * about the implications of these ioctls before extending
4019                 * their support.  And we do not think these features are
4020                 * valuable for pipes.
4021                 */
```



```
4022                STRUCT_DECL(strmodconf, strmodinsert);
4023                char mod_name[FMNAMESZ + 1];
4024                fmodsw_impl_t *fp;
4025                dev_t dummydev;
4026                queue_t *tmp_wrq;
4027                int pos;
4028                boolean_t is_insert;

4030                STRUCT_INIT(strmodinsert, flag);
4031                if (stp->sd_flag & STRHUP)
4032                        return (ENXIO);
4033                if (STRMATED(stp))
4034                        return (EINVAL);
4035                if ((error = secpolicy_net_config(crp, B_FALSE)) != 0)
4036                        return (error);
4037                if (stp->sd_anchor != 0 &&
4038                    stp->sd_anchorzone != crgetzoneid(crp))
4039                        return (EINVAL);

4041                error = strcopyin((void *)arg, STRUCT_BUF(strmodinsert),
4042                    STRUCT_SIZE(strmodinsert), copyflag);
4043                if (error)
4044                        return (error);

4046                /*
4047                 * Get module name and look up in fmodsw.
4048                 */
4049                error = (copyflag & U_TO_K ? copyinstr :
4050                    copystr)(STRUCT_FGETP(strmodinsert, mod_name),
4051                    mod_name, FMNAMESZ + 1, NULL);
4052                if (error)
4053                        return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

4055                if ((fp = fmodsw_find(mod_name, FMODSW_HOLD | FMODSW_LOAD)) ==
4056                    NULL)
4057                        return (EINVAL);

4059                if (error = strstartplumb(stp, flag, cmd)) {
4060                        fmodsw_rele(fp);
4061                        return (error);
4062                }

4064                /*
4065                 * Is this _I_INSERT just like an I_PUSH?  We need to know
4066                 * this because we do some optimizations if this is a
4067                 * module being pushed.
4068                 */
4069                pos = STRUCT_FGET(strmodinsert, pos);
4070                is_insert = (pos != 0);

4072                /*
4073                 * Make sure pos is valid.  Even though it is not an I_PUSH,
4074                 * we impose the same limit on the number of modules in a
4075                 * stream.
4076                 */
4077                mutex_enter(&stp->sd_lock);
4078                if (stp->sd_pushcnt >= nstrpush || pos < 0 ||
4079                    pos > stp->sd_pushcnt) {
4080                        fmodsw_rele(fp);
4081                        strendplumb(stp);
4082                        mutex_exit(&stp->sd_lock);
4083                        return (EINVAL);
4084                }
4085                if (stp->sd_anchor != 0) {
4086                        /*
4087                         * Is this insert below the anchor?
```

```
4088                         * Pushcnt hasn't been increased yet hence
4089                         * we test for greater than here, and greater or
4090                         * equal after qattach.
4091                         */
4092                        if (pos > (stp->sd_pushcnt - stp->sd_anchor) &&
4093                            stp->sd_anchorzone != crgetzoneid(crp)) {
4094                                fmodsw_rele(fp);
4095                                strendplumb(stp);
4096                                mutex_exit(&stp->sd_lock);
4097                                return (EPERM);
4098                        }
4099                }

4101                mutex_exit(&stp->sd_lock);

4103                /*
4104                 * First find the correct position this module to
4105                 * be inserted.  We don't need to call claimstr()
4106                 * as the stream should not be changing at this point.
4107                 *
4108                 * Insert new module and call its open routine
4109                 * via qattach().  Modules don't change device
4110                 * numbers, so just ignore dummydev here.
4111                 */
4112                for (tmp_wrq = stp->sd_wrq; pos > 0;
4113                    tmp_wrq = tmp_wrq->q_next, pos--) {
4114                        ASSERT(SAMESTR(tmp_wrq));
4115                }
4116                dummydev = vp->v_rdev;
4117                if ((error = qattach(_RD(tmp_wrq), &dummydev, 0, crp,
4118                    fp, is_insert)) != 0) {
4119                        mutex_enter(&stp->sd_lock);
4120                        strendplumb(stp);
4121                        mutex_exit(&stp->sd_lock);
4122                        return (error);
4123                }

4125                mutex_enter(&stp->sd_lock);

4127                /*
4128                 * As a performance concern we are caching the values of
4129                 * q_minpsz and q_maxpsz of the module below the stream
4130                 * head in the stream head.
4131                 */
4132                if (!is_insert) {
4133                        mutex_enter(QLOCK(stp->sd_wrq->q_next));
4134                        rmin = stp->sd_wrq->q_next->q_minpsz;
4135                        rmax = stp->sd_wrq->q_next->q_maxpsz;
4136                        mutex_exit(QLOCK(stp->sd_wrq->q_next));

4138                        /* Do this processing here as a performance concern */
4139                        if (strmsgsz != 0) {
4140                                if (rmax == INFPSZ) {
4141                                        rmax = strmsgsz;
4142                                } else  {
4143                                        rmax = MIN(strmsgsz, rmax);
4144                                }
4145                        }

4147                        mutex_enter(QLOCK(wrq));
4148                        stp->sd_qn_minpsz = rmin;
4149                        stp->sd_qn_maxpsz = rmax;
4150                        mutex_exit(QLOCK(wrq));
4151                }

4153                /*
```



```
4154                 * Need to update the anchor value if this module is
4155                 * inserted below the anchor point.
4156                 */
4157                if (stp->sd_anchor != 0) {
4158                        pos = STRUCT_FGET(strmodinsert, pos);
4159                        if (pos >= (stp->sd_pushcnt - stp->sd_anchor))
4160                                stp->sd_anchor++;
4161                }

4163                strendplumb(stp);
4164                mutex_exit(&stp->sd_lock);
4165                return (0);
4166        }

4168        case _I_REMOVE:
4169        {
4170                /*
4171                 * To remove a module with a given name in a stream.  The
4172                 * caller of this ioctl needs to provide both the name and
4173                 * the position of the module to be removed.  This eliminates
4174                 * the ambiguity of removal if a module is inserted/pushed
4175                 * multiple times in a stream.  In the first release, only
4176                 * allow privileged user to use this ioctl.
4177                 * Furthermore, the remove is only allowed
4178                 * below an anchor if the zoneid is the same as the zoneid
4179                 * which created the anchor.
4180                 *
4181                 * Note that we do not plan to support this ioctl
4182                 * on pipes in the first release.  We want to learn more
4183                 * about the implications of these ioctls before extending
4184                 * their support.  And we do not think these features are
4185                 * valuable for pipes.
4186                 *
4187                 * Also note that _I_REMOVE cannot be used to remove a
4188                 * driver or the stream head.
4189                 */
4190                STRUCT_DECL(strmodconf, strmodremove);
4191                queue_t *q;
4192                int pos;
4193                char mod_name[FMNAMESZ + 1];
4194                boolean_t is_remove;

4196                STRUCT_INIT(strmodremove, flag);
4197                if (stp->sd_flag & STRHUP)
4198                        return (ENXIO);
4199                if (STRMATED(stp))
4200                        return (EINVAL);
4201                if ((error = secpolicy_net_config(crp, B_FALSE)) != 0)
4202                        return (error);
4203                if (stp->sd_anchor != 0 &&
4204                    stp->sd_anchorzone != crgetzoneid(crp))
4205                        return (EINVAL);

4207                error = strcopyin((void *)arg, STRUCT_BUF(strmodremove),
4208                    STRUCT_SIZE(strmodremove), copyflag);
4209                if (error)
4210                        return (error);

4212                error = (copyflag & U_TO_K ? copyinstr :
4213                    copystr)(STRUCT_FGETP(strmodremove, mod_name),
4214                    mod_name, FMNAMESZ + 1, NULL);
4215                if (error)
4216                        return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);

4218                if ((error = strstartplumb(stp, flag, cmd)) != 0)
4219                        return (error);
```

```
4221                /*
4222                 * Match the name of given module to the name of module at
4223                 * the given position.
4224                 */
4225                pos = STRUCT_FGET(strmodremove, pos);

4227                is_remove = (pos != 0);
4228                for (q = stp->sd_wrq->q_next; SAMESTR(q) && pos > 0;
4229                    q = q->q_next, pos--)
4230                        ;
4231                if (pos > 0 || !SAMESTR(q) ||
4232                    strcmp(Q2NAME(q), mod_name) != 0) {
4233                        mutex_enter(&stp->sd_lock);
4234                        strendplumb(stp);
4235                        mutex_exit(&stp->sd_lock);
4236                        return (EINVAL);
4237                }

4239                /*
4240                 * If the position is at or below an anchor, then the zoneid
4241                 * must match the zoneid that created the anchor.
4242                 */
4243                if (stp->sd_anchor != 0) {
4244                        pos = STRUCT_FGET(strmodremove, pos);
4245                        if (pos >= (stp->sd_pushcnt - stp->sd_anchor) &&
4246                            stp->sd_anchorzone != crgetzoneid(crp)) {
4247                                mutex_enter(&stp->sd_lock);
4248                                strendplumb(stp);
4249                                mutex_exit(&stp->sd_lock);
4250                                return (EPERM);
4251                        }
4252                }


4255                ASSERT(!(q->q_flag & QREADR));
4256                qdetach(_RD(q), 1, flag, crp, is_remove);

4258                mutex_enter(&stp->sd_lock);

4260                /*
4261                 * As a performance concern we are caching the values of
4262                 * q_minpsz and q_maxpsz of the module below the stream
4263                 * head in the stream head.
4264                 */
4265                if (!is_remove) {
4266                        mutex_enter(QLOCK(wrq->q_next));
4267                        rmin = wrq->q_next->q_minpsz;
4268                        rmax = wrq->q_next->q_maxpsz;
4269                        mutex_exit(QLOCK(wrq->q_next));

4271                        /* Do this processing here as a performance concern */
4272                        if (strmsgsz != 0) {
4273                                if (rmax == INFPSZ)
4274                                        rmax = strmsgsz;
4275                                else  {
4276                                        if (vp->v_type == VFIFO)
4277                                                rmax = MIN(PIPE_BUF, rmax);
4278                                        else    rmax = MIN(strmsgsz, rmax);
4279                                }
4280                        }

4282                        mutex_enter(QLOCK(wrq));
4283                        stp->sd_qn_minpsz = rmin;
4284                        stp->sd_qn_maxpsz = rmax;
4285                        mutex_exit(QLOCK(wrq));
```



```
4286                }

4288                /*
4289                 * Need to update the anchor value if this module is removed
4290                 * at or below the anchor point.  If the removed module is at
4291                 * the anchor point, remove the anchor for this stream if
4292                 * there is no module above the anchor point.  Otherwise, if
4293                 * the removed module is below the anchor point, decrement the
4294                 * anchor point by 1.
4295                 */
4296                if (stp->sd_anchor != 0) {
4297                        pos = STRUCT_FGET(strmodremove, pos);
4298                        if (pos == stp->sd_pushcnt - stp->sd_anchor + 1)
4299                                stp->sd_anchor = 0;
4300                        else if (pos > (stp->sd_pushcnt - stp->sd_anchor + 1))
4301                                stp->sd_anchor--;
4302                }

4304                strendplumb(stp);
4305                mutex_exit(&stp->sd_lock);
4306                return (0);
4307        }

4309        case I_ANCHOR:
4310                /*
4311                 * Set the anchor position on the stream to reside at
4312                 * the top module (in other words, the top module
4313                 * cannot be popped).  Anchors with a FIFO make no
4314                 * obvious sense, so they're not allowed.
4315                 */
4316                mutex_enter(&stp->sd_lock);

4318                if (stp->sd_vnode->v_type == VFIFO) {
4319                        mutex_exit(&stp->sd_lock);
4320                        return (EINVAL);
4321                }
4322                /* Only allow the same zoneid to update the anchor */
4323                if (stp->sd_anchor != 0 &&
4324                    stp->sd_anchorzone != crgetzoneid(crp)) {
4325                        mutex_exit(&stp->sd_lock);
4326                        return (EINVAL);
4327                }
4328                stp->sd_anchor = stp->sd_pushcnt;
4329                stp->sd_anchorzone = crgetzoneid(crp);
4330                mutex_exit(&stp->sd_lock);
4331                return (0);

4333        case I_LOOK:
4334                /*
4335                 * Get name of first module downstream.
4336                 * If no module, return an error.
4337                 */
4338                claimstr(wrq);
4339                if (_SAMESTR(wrq) && wrq->q_next->q_next != NULL) {
4340                        char *name = Q2NAME(wrq->q_next);

4342                        error = strcopyout(name, (void *)arg, strlen(name) + 1,
4343                            copyflag);
4344                        releasestr(wrq);
4345                        return (error);
4346                }
4347                releasestr(wrq);
4348                return (EINVAL);

4350        case I_LINK:
4351        case I_PLINK:
```

```
4352                 /*
4353                  * Link a multiplexor.
4354                  */
4355                 return (mlink(vp, cmd, (int)arg, crp, rvalp, 0));

4357         case _I_PLINK_LH:
4358                 /*
4359                  * Link a multiplexor: Call must originate from kernel.
4360                  */
4361                 if (kioctl)
4362                         return (ldi_mlink_lh(vp, cmd, arg, crp, rvalp));

4364                 return (EINVAL);
4365         case I_UNLINK:
4366         case I_PUNLINK:
4367                 /*
4368                  * Unlink a multiplexor.
4369                  * If arg is -1, unlink all links for which this is the
4370                  * controlling stream.  Otherwise, arg is an index number
4371                  * for a link to be removed.
4372                  */
4373         {
4374                 struct linkinfo *linkp;
4375                 int native_arg = (int)arg;
4376                 int type;
4377                 netstack_t *ns;
4378                 str_stack_t *ss;

4380                 TRACE_1(TR_FAC_STREAMS_FR,
4381                     TR_I_UNLINK, "I_UNLINK/I_PUNLINK:%p", stp);
4382                 if (vp->v_type == VFIFO) {
4383                         return (EINVAL);
4384                 }
4385                 if (cmd == I_UNLINK)
4386                         type = LINKNORMAL;
4387                 else    /* I_PUNLINK */
4388                         type = LINKPERSIST;
4389                 if (native_arg == 0) {
4390                         return (EINVAL);
4391                 }
4392                 ns = netstack_find_by_cred(crp);
4393                 ASSERT(ns != NULL);
4394                 ss = ns->netstack_str;
4395                 ASSERT(ss != NULL);

4397                 if (native_arg == MUXID_ALL)
4398                         error = munlinkall(stp, type, crp, rvalp, ss);
4399                 else {
4400                         mutex_enter(&muxifier);
4401                         if (!(linkp = findlinks(stp, (int)arg, type, ss))) {
4402                                 /* invalid user supplied index number */
4403                                 mutex_exit(&muxifier);
4404                                 netstack_rele(ss->ss_netstack);
4405                                 return (EINVAL);
4406                         }
4407                         /* munlink drops the muxifier lock */
4408                         error = munlink(stp, linkp, type, crp, rvalp, ss);
4409                 }
4410                 netstack_rele(ss->ss_netstack);
4411                 return (error);
4412         }

4414         case I_FLUSH:
4415                 /*
4416                  * send a flush message downstream
4417                  * flush message can indicate
```



```
4418                  * FLUSHR - flush read queue
4419                  * FLUSHW - flush write queue
4420                  * FLUSHRW - flush read/write queue
4421                  */
4422                 if (stp->sd_flag & STRHUP)
4423                         return (ENXIO);
4424                 if (arg & ~FLUSHRW)
4425                         return (EINVAL);

4427                 for (;;) {
4428                         if (putnextctl1(stp->sd_wrq, M_FLUSH, (int)arg)) {
4429                                 break;
4430                         }
4431                         if (error = strwaitbuf(1, BPRI_HI)) {
4432                                 return (error);
4433                         }
4434                 }

4436                 /*
4437                  * Send down an unsupported ioctl and wait for the nack
4438                  * in order to allow the M_FLUSH to propagate back
4439                  * up to the stream head.
4440                  * Replaces if (qready()) runqueues();
4441                  */
4442                 strioc.ic_cmd = -1;     /* The unsupported ioctl */
4443                 strioc.ic_timout = 0;
4444                 strioc.ic_len = 0;
4445                 strioc.ic_dp = NULL;
4446                 (void) strdoioctl(stp, &strioc, flag, K_TO_K, crp, rvalp);
4447                 *rvalp = 0;
4448                 return (0);

4450         case I_FLUSHBAND:
4451         {
4452                 struct bandinfo binfo;

4454                 error = strcopyin((void *)arg, &binfo, sizeof (binfo),
4455                     copyflag);
4456                 if (error)
4457                         return (error);
4458                 if (stp->sd_flag & STRHUP)
4459                         return (ENXIO);
4460                 if (binfo.bi_flag & ~FLUSHRW)
4461                         return (EINVAL);
4462                 while (!(mp = allocb(2, BPRI_HI))) {
4463                         if (error = strwaitbuf(2, BPRI_HI))
4464                                 return (error);
4465                 }
4466                 mp->b_datap->db_type = M_FLUSH;
4467                 *mp->b_wptr++ = binfo.bi_flag | FLUSHBAND;
4468                 *mp->b_wptr++ = binfo.bi_pri;
4469                 putnext(stp->sd_wrq, mp);
4470                 /*
4471                  * Send down an unsupported ioctl and wait for the nack
4472                  * in order to allow the M_FLUSH to propagate back
4473                  * up to the stream head.
4474                  * Replaces if (qready()) runqueues();
4475                  */
4476                 strioc.ic_cmd = -1;     /* The unsupported ioctl */
4477                 strioc.ic_timout = 0;
4478                 strioc.ic_len = 0;
4479                 strioc.ic_dp = NULL;
4480                 (void) strdoioctl(stp, &strioc, flag, K_TO_K, crp, rvalp);
4481                 *rvalp = 0;
4482                 return (0);
4483         }
```

```
4485         case I_SRDOPT:
4486                 /*
4487                  * Set read options
4488                  *
4489                  * RNORM - default stream mode
4490                  * RMSGN - message no discard
4491                  * RMSGD - message discard
4492                  * RPROTNORM - fail read with EBADMSG for M_[PC]PROTOs
4493                  * RPROTDAT - convert M_[PC]PROTOs to M_DATAs
4494                  * RPROTDIS - discard M_[PC]PROTOs and retain M_DATAs
4495                  */
4496                 if (arg & ~(RMODEMASK | RPROTMASK))
4497                         return (EINVAL);

4499                 if ((arg & (RMSGD|RMSGN)) == (RMSGD|RMSGN))
4500                         return (EINVAL);

4502                 mutex_enter(&stp->sd_lock);
4503                 switch (arg & RMODEMASK) {
4504                 case RNORM:
4505                         stp->sd_read_opt &= ~(RD_MSGDIS | RD_MSGNODIS);
4506                         break;
4507                 case RMSGD:
4508                         stp->sd_read_opt = (stp->sd_read_opt & ~RD_MSGNODIS) |
4509                             RD_MSGDIS;
4510                         break;
4511                 case RMSGN:
4512                         stp->sd_read_opt = (stp->sd_read_opt & ~RD_MSGDIS) |
4513                             RD_MSGNODIS;
4514                         break;
4515                 }

4517                 switch (arg & RPROTMASK) {
4518                 case RPROTNORM:
4519                         stp->sd_read_opt &= ~(RD_PROTDAT | RD_PROTDIS);
4520                         break;

4522                 case RPROTDAT:
4523                         stp->sd_read_opt = ((stp->sd_read_opt & ~RD_PROTDIS) |
4524                             RD_PROTDAT);
4525                         break;

4527                 case RPROTDIS:
4528                         stp->sd_read_opt = ((stp->sd_read_opt & ~RD_PROTDAT) |
4529                             RD_PROTDIS);
4530                         break;
4531                 }
4532                 mutex_exit(&stp->sd_lock);
4533                 return (0);

4535         case I_GRDOPT:
4536                 /*
4537                  * Get read option and return the value
4538                  * to spot pointed to by arg
4539                  */
4540         {
4541                 int rdopt;

4543                 rdopt = ((stp->sd_read_opt & RD_MSGDIS) ? RMSGD :
4544                     ((stp->sd_read_opt & RD_MSGNODIS) ? RMSGN : RNORM));
4545                 rdopt |= ((stp->sd_read_opt & RD_PROTDAT) ? RPROTDAT :
4546                     ((stp->sd_read_opt & RD_PROTDIS) ? RPROTDIS : RPROTNORM));

4548                 return (strcopyout(&rdopt, (void *)arg, sizeof (int),
4549                     copyflag));
```



```
4550         }

4552         case I_SERROPT:
4553                 /*
4554                  * Set error options
4555                  *
4556                  * RERRNORM - persistent read errors
4557                  * RERRNONPERSIST - non-persistent read errors
4558                  * WERRNORM - persistent write errors
4559                  * WERRNONPERSIST - non-persistent write errors
4560                  */
4561                 if (arg & ~(RERRMASK | WERRMASK))
4562                         return (EINVAL);

4564                 mutex_enter(&stp->sd_lock);
4565                 switch (arg & RERRMASK) {
4566                 case RERRNORM:
4567                         stp->sd_flag &= ~STRDERRNONPERSIST;
4568                         break;
4569                 case RERRNONPERSIST:
4570                         stp->sd_flag |= STRDERRNONPERSIST;
4571                         break;
4572                 }
4573                 switch (arg & WERRMASK) {
4574                 case WERRNORM:
4575                         stp->sd_flag &= ~STWRERRNONPERSIST;
4576                         break;
4577                 case WERRNONPERSIST:
4578                         stp->sd_flag |= STWRERRNONPERSIST;
4579                         break;
4580                 }
4581                 mutex_exit(&stp->sd_lock);
4582                 return (0);

4584         case I_GERROPT:
4585                 /*
4586                  * Get error option and return the value
4587                  * to spot pointed to by arg
4588                  */
4589         {
4590                 int erropt = 0;

4592                 erropt |= (stp->sd_flag & STRDERRNONPERSIST) ? RERRNONPERSIST :
4593                     RERRNORM;
4594                 erropt |= (stp->sd_flag & STWRERRNONPERSIST) ? WERRNONPERSIST :
4595                     WERRNORM;
4596                 return (strcopyout(&erropt, (void *)arg, sizeof (int),
4597                     copyflag));
4598         }

4600         case I_SETSIG:
4601                 /*
4602                  * Register the calling proc to receive the SIGPOLL
4603                  * signal based on the events given in arg.  If
4604                  * arg is zero, remove the proc from register list.
4605                  */
4606         {
4607                 strsig_t *ssp, *pssp;
4608                 struct pid *pidp;

4610                 pssp = NULL;
4611                 pidp = curproc->p_pidp;
4612                 /*
4613                  * Hold sd_lock to prevent traversal of sd_siglist while
4614                  * it is modified.
4615                  */
```

```
4616                mutex_enter(&stp->sd_lock);
4617                for (ssp = stp->sd_siglist; ssp && (ssp->ss_pidp != pidp);
4618                    pssp = ssp, ssp = ssp->ss_next)
4619                        ;
4621                if (arg) {
4622                        if (arg & ~(S_INPUT|S_HIPRI|S_MSG|S_HANGUP|S_ERROR|
4623                            S_RDNORM|S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)) {
4624                                mutex_exit(&stp->sd_lock);
4625                                return (EINVAL);
4626                        }
4627                        if ((arg & S_BANDURG) && !(arg & S_RDBAND)) {
4628                                mutex_exit(&stp->sd_lock);
4629                                return (EINVAL);
4630                        }
4632                        /*
4633                         * If proc not already registered, add it
4634                         * to list.
4635                         */
4636                        if (!ssp) {
4637                                ssp = kmem_alloc(sizeof (strsig_t), KM_SLEEP);
4638                                ssp->ss_pidp = pidp;
4639                                ssp->ss_pid = pidp->pid_id;
4640                                ssp->ss_next = NULL;
4641                                if (pssp)
4642                                        pssp->ss_next = ssp;
4643                                else
4644                                        stp->sd_siglist = ssp;
4645                                mutex_enter(&pidlock);
4646                                PID_HOLD(pidp);
4647                                mutex_exit(&pidlock);
4648                        }
4650                        /*
4651                         * Set events.
4652                         */
4653                        ssp->ss_events = (int)arg;
4654                } else {
4655                        /*
4656                         * Remove proc from register list.
4657                         */
4658                        if (ssp) {
4659                                mutex_enter(&pidlock);
4660                                PID_RELE(pidp);
4661                                mutex_exit(&pidlock);
4662                                if (pssp)
4663                                        pssp->ss_next = ssp->ss_next;
4664                                else
4665                                        stp->sd_siglist = ssp->ss_next;
4666                                kmem_free(ssp, sizeof (strsig_t));
4667                        } else {
4668                                mutex_exit(&stp->sd_lock);
4669                                return (EINVAL);
4670                        }
4671                }
4673                /*
4674                 * Recalculate OR of sig events.
4675                 */
4676                stp->sd_sigflags = 0;
4677                for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4678                        stp->sd_sigflags |= ssp->ss_events;
4679                mutex_exit(&stp->sd_lock);
4680                return (0);
4681        }
```



```
4683        case I_GETSIG:
4684                /*
4685                 * Return (in arg) the current registration of events
4686                 * for which the calling proc is to be signaled.
4687                 */
4688        {
4689                struct strsig *ssp;
4690                struct pid  *pidp;
4692                pidp = curproc->p_pidp;
4693                mutex_enter(&stp->sd_lock);
4694                for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4695                        if (ssp->ss_pidp == pidp) {
4696                                error = strcopyout(&ssp->ss_events, (void *)arg,
4697                                    sizeof (int), copyflag);
4698                                mutex_exit(&stp->sd_lock);
4699                                return (error);
4700                        }
4701                mutex_exit(&stp->sd_lock);
4702                return (EINVAL);
4703        }
4705        case I_ESETSIG:
4706                /*
4707                 * Register the ss_pid to receive the SIGPOLL
4708                 * signal based on the events is ss_events arg.  If
4709                 * ss_events is zero, remove the proc from register list.
4710                 */
4711        {
4712                struct strsig *ssp, *pssp;
4713                struct proc *proc;
4714                struct pid  *pidp;
4715                pid_t pid;
4716                struct strsigset ss;
4718                error = strcopyin((void *)arg, &ss, sizeof (ss), copyflag);
4719                if (error)
4720                        return (error);
4722                pid = ss.ss_pid;
4724                if (ss.ss_events != 0) {
4725                        /*
4726                         * Permissions check by sending signal 0.
4727                         * Note that when kill fails it does a set_errno
4728                         * causing the system call to fail.
4729                         */
4730                        error = kill(pid, 0);
4731                        if (error) {
4732                                return (error);
4733                        }
4734                }
4735                mutex_enter(&pidlock);
4736                if (pid == 0)
4737                        proc = curproc;
4738                else if (pid < 0)
4739                        proc = pgfind(-pid);
4740                else
4741                        proc = prfind(pid);
4742                if (proc == NULL) {
4743                        mutex_exit(&pidlock);
4744                        return (ESRCH);
4745                }
4746                if (pid < 0)
4747                        pidp = proc->p_pgidp;
```

```
4748                else
4749                        pidp = proc->p_pidp;
4750                ASSERT(pidp);
4751                /*
4752                 * Get a hold on the pid structure while referencing it.
4753                 * There is a separate PID_HOLD should it be inserted
4754                 * in the list below.
4755                 */
4756                PID_HOLD(pidp);
4757                mutex_exit(&pidlock);

4759                pssp = NULL;
4760                /*
4761                 * Hold sd_lock to prevent traversal of sd_siglist while
4762                 * it is modified.
4763                 */
4764                mutex_enter(&stp->sd_lock);
4765                for (ssp = stp->sd_siglist; ssp && (ssp->ss_pid != pid);
4766                    pssp = ssp, ssp = ssp->ss_next)
4767                        ;

4769                if (ss.ss_events) {
4770                        if (ss.ss_events &
4771                            ~(S_INPUT|S_HIPRI|S_MSG|S_HANGUP|S_ERROR|
4772                            S_RDNORM|S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)) {
4773                                mutex_exit(&stp->sd_lock);
4774                                mutex_enter(&pidlock);
4775                                PID_RELE(pidp);
4776                                mutex_exit(&pidlock);
4777                                return (EINVAL);
4778                        }
4779                        if ((ss.ss_events & S_BANDURG) &&
4780                            !(ss.ss_events & S_RDBAND)) {
4781                                mutex_exit(&stp->sd_lock);
4782                                mutex_enter(&pidlock);
4783                                PID_RELE(pidp);
4784                                mutex_exit(&pidlock);
4785                                return (EINVAL);
4786                        }

4788                        /*
4789                         * If proc not already registered, add it
4790                         * to list.
4791                         */
4792                        if (!ssp) {
4793                                ssp = kmem_alloc(sizeof (strsig_t), KM_SLEEP);
4794                                ssp->ss_pidp = pidp;
4795                                ssp->ss_pid = pid;
4796                                ssp->ss_next = NULL;
4797                                if (pssp)
4798                                        pssp->ss_next = ssp;
4799                                else
4800                                        stp->sd_siglist = ssp;
4801                                mutex_enter(&pidlock);
4802                                PID_HOLD(pidp);
4803                                mutex_exit(&pidlock);
4804                        }

4806                        /*
4807                         * Set events.
4808                         */
4809                        ssp->ss_events = ss.ss_events;
4810                } else {
4811                        /*
4812                         * Remove proc from register list.
4813                         */
```



```
4814                        if (ssp) {
4815                                mutex_enter(&pidlock);
4816                                PID_RELE(pidp);
4817                                mutex_exit(&pidlock);
4818                                if (pssp)
4819                                        pssp->ss_next = ssp->ss_next;
4820                                else
4821                                        stp->sd_siglist = ssp->ss_next;
4822                                kmem_free(ssp, sizeof (strsig_t));
4823                        } else {
4824                                mutex_exit(&stp->sd_lock);
4825                                mutex_enter(&pidlock);
4826                                PID_RELE(pidp);
4827                                mutex_exit(&pidlock);
4828                                return (EINVAL);
4829                        }
4830                }

4832                /*
4833                 * Recalculate OR of sig events.
4834                 */
4835                stp->sd_sigflags = 0;
4836                for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4837                        stp->sd_sigflags |= ssp->ss_events;
4838                mutex_exit(&stp->sd_lock);
4839                mutex_enter(&pidlock);
4840                PID_RELE(pidp);
4841                mutex_exit(&pidlock);
4842                return (0);
4843        }

4845        case I_EGETSIG:
4846                /*
4847                 * Return (in arg) the current registration of events
4848                 * for which the calling proc is to be signaled.
4849                 */
4850        {
4851                struct strsig *ssp;
4852                struct proc *proc;
4853                pid_t pid;
4854                struct pid  *pidp;
4855                struct strsigset ss;

4857                error = strcopyin((void *)arg, &ss, sizeof (ss), copyflag);
4858                if (error)
4859                        return (error);

4861                pid = ss.ss_pid;
4862                mutex_enter(&pidlock);
4863                if (pid == 0)
4864                        proc = curproc;
4865                else if (pid < 0)
4866                        proc = pgfind(-pid);
4867                else
4868                        proc = prfind(pid);
4869                if (proc == NULL) {
4870                        mutex_exit(&pidlock);
4871                        return (ESRCH);
4872                }
4873                if (pid < 0)
4874                        pidp = proc->p_pgidp;
4875                else
4876                        pidp = proc->p_pidp;

4878                /* Prevent the pidp from being reassigned */
4879                PID_HOLD(pidp);
```

```
4880                mutex_exit(&pidlock);

4882                mutex_enter(&stp->sd_lock);
4883                for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4884                        if (ssp->ss_pid == pid) {
4885                                ss.ss_pid = ssp->ss_pid;
4886                                ss.ss_events = ssp->ss_events;
4887                                error = strcopyout(&ss, (void *)arg,
4888                                    sizeof (struct strsigset), copyflag);
4889                                mutex_exit(&stp->sd_lock);
4890                                mutex_enter(&pidlock);
4891                                PID_RELE(pidp);
4892                                mutex_exit(&pidlock);
4893                                return (error);
4894                        }
4895                mutex_exit(&stp->sd_lock);
4896                mutex_enter(&pidlock);
4897                PID_RELE(pidp);
4898                mutex_exit(&pidlock);
4899                return (EINVAL);
4900        }

4902        case I_PEEK:
4903        {
4904                STRUCT_DECL(strpeek, strpeek);
4905                size_t n;
4906                mblk_t *fmp, *tmp_mp = NULL;

4908                STRUCT_INIT(strpeek, flag);

4910                error = strcopyin((void *)arg, STRUCT_BUF(strpeek),
4911                    STRUCT_SIZE(strpeek), copyflag);
4912                if (error)
4913                        return (error);

4915                mutex_enter(QLOCK(rdq));
4916                /*
4917                 * Skip the invalid messages
4918                 */
4919                for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
4920                        if (mp->b_datap->db_type != M_SIG)
4921                                break;

4923                /*
4924                 * If user has requested to peek at a high priority message
4925                 * and first message is not, return 0
4926                 */
4927                if (mp != NULL) {
4928                        if ((STRUCT_FGET(strpeek, flags) & RS_HIPRI) &&
4929                            queclass(mp) == QNORM) {
4930                                *rvalp = 0;
4931                                mutex_exit(QLOCK(rdq));
4932                                return (0);
4933                        }
4934                } else if (stp->sd_struiordq == NULL ||
4935                    (STRUCT_FGET(strpeek, flags) & RS_HIPRI)) {
4936                        /*
4937                         * No mblks to look at at the streamhead and
4938                         * 1). This isn't a synch stream or
4939                         * 2). This is a synch stream but caller wants high
4940                         *      priority messages which is not supported by
4941                         *      the synch stream. (it only supports QNORM)
4942                         */
4943                        *rvalp = 0;
4944                        mutex_exit(QLOCK(rdq));
4945                        return (0);
```



```
4946                }

4948                fmp = mp;

4950                if (mp && mp->b_datap->db_type == M_PASSFP) {
4951                        mutex_exit(QLOCK(rdq));
4952                        return (EBADMSG);
4953                }

4955                ASSERT(mp == NULL || mp->b_datap->db_type == M_PCPROTO ||
4956                    mp->b_datap->db_type == M_PROTO ||
4957                    mp->b_datap->db_type == M_DATA);

4959                if (mp && mp->b_datap->db_type == M_PCPROTO) {
4960                        STRUCT_FSET(strpeek, flags, RS_HIPRI);
4961                } else {
4962                        STRUCT_FSET(strpeek, flags, 0);
4963                }


4966                if (mp && ((tmp_mp = dupmsg(mp)) == NULL)) {
4967                        mutex_exit(QLOCK(rdq));
4968                        return (ENOSR);
4969                }
4970                mutex_exit(QLOCK(rdq));

4972                /*
4973                 * set mp = tmp_mp, so that I_PEEK processing can continue.
4974                 * tmp_mp is used to free the dup'd message.
4975                 */
4976                mp = tmp_mp;

4978                uio.uio_fmode = 0;
4979                uio.uio_extflg = UIO_COPY_CACHED;
4980                uio.uio_segflg = (copyflag == U_TO_K) ? UIO_USERSPACE :
4981                    UIO_SYSSPACE;
4982                uio.uio_limit = 0;
4983                /*
4984                 * First process PROTO blocks, if any.
4985                 * If user doesn't want to get ctl info by setting maxlen <= 0,
4986                 * then set len to -1/0 and skip control blocks part.
4987                 */
4988                if (STRUCT_FGET(strpeek, ctlbuf.maxlen) < 0)
4989                        STRUCT_FSET(strpeek, ctlbuf.len, -1);
4990                else if (STRUCT_FGET(strpeek, ctlbuf.maxlen) == 0)
4991                        STRUCT_FSET(strpeek, ctlbuf.len, 0);
4992                else {
4993                        int     ctl_part = 0;

4995                        iov.iov_base = STRUCT_FGETP(strpeek, ctlbuf.buf);
4996                        iov.iov_len = STRUCT_FGET(strpeek, ctlbuf.maxlen);
4997                        uio.uio_iov = &iov;
4998                        uio.uio_resid = iov.iov_len;
4999                        uio.uio_loffset = 0;
5000                        uio.uio_iovcnt = 1;
5001                        while (mp && mp->b_datap->db_type != M_DATA &&
5002                            uio.uio_resid >= 0) {
5003                                ASSERT(STRUCT_FGET(strpeek, flags) == 0 ?
5004                                    mp->b_datap->db_type == M_PROTO :
5005                                    mp->b_datap->db_type == M_PCPROTO);

5007                                if ((n = MIN(uio.uio_resid,
5008                                    mp->b_wptr - mp->b_rptr)) != 0 &&
5009                                    (error = uiomove((char *)mp->b_rptr, n,
5010                                    UIO_READ, &uio)) != 0) {
5011                                        freemsg(tmp_mp);
```

```
5012                                    return (error);
5013                            }
5014                            ctl_part = 1;
5015                            mp = mp->b_cont;
5016                    }
5017                    /* No ctl message */
5018                    if (ctl_part == 0)
5019                            STRUCT_FSET(strpeek, ctlbuf.len, -1);
5020                    else
5021                            STRUCT_FSET(strpeek, ctlbuf.len,
5022                                STRUCT_FGET(strpeek, ctlbuf.maxlen) -
5023                                uio.uio_resid);
5024            }

5026            /*
5027             * Now process DATA blocks, if any.
5028             * If user doesn't want to get data info by setting maxlen <= 0,
5029             * then set len to -1/0 and skip data blocks part.
5030             */
5031            if (STRUCT_FGET(strpeek, databuf.maxlen) < 0)
5032                    STRUCT_FSET(strpeek, databuf.len, -1);
5033            else if (STRUCT_FGET(strpeek, databuf.maxlen) == 0)
5034                    STRUCT_FSET(strpeek, databuf.len, 0);
5035            else {
5036                    int     data_part = 0;

5038                    iov.iov_base = STRUCT_FGETP(strpeek, databuf.buf);
5039                    iov.iov_len = STRUCT_FGET(strpeek, databuf.maxlen);
5040                    uio.uio_iov = &iov;
5041                    uio.uio_resid = iov.iov_len;
5042                    uio.uio_loffset = 0;
5043                    uio.uio_iovcnt = 1;
5044                    while (mp && uio.uio_resid) {
5045                            if (mp->b_datap->db_type == M_DATA) {
5046                                    if ((n = MIN(uio.uio_resid,
5047                                        mp->b_wptr - mp->b_rptr)) != 0 &&
5048                                        (error = uiomove((char *)mp->b_rptr,
5049                                        n, UIO_READ, &uio)) != 0) {
5050                                            freemsg(tmp_mp);
5051                                            return (error);
5052                                    }
5053                                    data_part = 1;
5054                            }
5055                            ASSERT(data_part == 0 ||
5056                                mp->b_datap->db_type == M_DATA);
5057                            mp = mp->b_cont;
5058                    }
5059                    /* No data message */
5060                    if (data_part == 0)
5061                            STRUCT_FSET(strpeek, databuf.len, -1);
5062                    else
5063                            STRUCT_FSET(strpeek, databuf.len,
5064                                STRUCT_FGET(strpeek, databuf.maxlen) -
5065                                uio.uio_resid);
5066            }
5067            freemsg(tmp_mp);

5069            /*
5070             * It is a synch stream and user wants to get
5071             * data (maxlen > 0).
5072             * uio setup is done by the codes that process DATA
5073             * blocks above.
5074             */
5075            if ((fmp == NULL) && STRUCT_FGET(strpeek, databuf.maxlen) > 0) {
5076                    infod_t infod;
```



```
5078                    infod.d_cmd = INFOD_COPYOUT;
5079                    infod.d_res = 0;
5080                    infod.d_uiop = &uio;
5081                    error = infonext(rdq, &infod);
5082                    if (error == EINVAL || error == EBUSY)
5083                            error = 0;
5084                    if (error)
5085                            return (error);
5086                    STRUCT_FSET(strpeek, databuf.len, STRUCT_FGET(strpeek,
5087                        databuf.maxlen) - uio.uio_resid);
5088                    if (STRUCT_FGET(strpeek, databuf.len) == 0) {
5089                            /*
5090                             * No data found by the infonext().
5091                             */
5092                            STRUCT_FSET(strpeek, databuf.len, -1);
5093                    }
5094            }
5095            error = strcopyout(STRUCT_BUF(strpeek), (void *)arg,
5096                STRUCT_SIZE(strpeek), copyflag);
5097            if (error) {
5098                    return (error);
5099            }
5100            /*
5101             * If there is no message retrieved, set return code to 0
5102             * otherwise, set it to 1.
5103             */
5104            if (STRUCT_FGET(strpeek, ctlbuf.len) == -1 &&
5105                STRUCT_FGET(strpeek, databuf.len) == -1)
5106                    *rvalp = 0;
5107            else
5108                    *rvalp = 1;
5109            return (0);
5110    }

5112    case I_FDINSERT:
5113    {
5114            STRUCT_DECL(strfdinsert, strfdinsert);
5115            struct file *resftp;
5116            struct stdata *resstp;
5117            t_uscalar_t     ival;
5118            ssize_t msgsize;
5119            struct strbuf mctl;

5121            STRUCT_INIT(strfdinsert, flag);
5122            if (stp->sd_flag & STRHUP)
5123                    return (ENXIO);
5124            /*
5125             * STRDERR, STWRERR and STPLEX tested above.
5126             */
5127            error = strcopyin((void *)arg, STRUCT_BUF(strfdinsert),
5128                STRUCT_SIZE(strfdinsert), copyflag);
5129            if (error)
5130                    return (error);

5132            if (STRUCT_FGET(strfdinsert, offset) < 0 ||
5133                (STRUCT_FGET(strfdinsert, offset) %
5134                sizeof (t_uscalar_t)) != 0)
5135                    return (EINVAL);
5136            if ((resftp = getf(STRUCT_FGET(strfdinsert, fildes))) != NULL) {
5137                    if ((resstp = resftp->f_vnode->v_stream) == NULL) {
5138                            releasef(STRUCT_FGET(strfdinsert, fildes));
5139                            return (EINVAL);
5140                    }
5141            } else
5142                    return (EINVAL);
```

```
5144                mutex_enter(&resstp->sd_lock);
5145                if (resstp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
5146                        error = strgeterr(resstp,
5147                            STRDERR|STWRERR|STRHUP|STPLEX, 0);
5148                        if (error != 0) {
5149                                mutex_exit(&resstp->sd_lock);
5150                                releasef(STRUCT_FGET(strfdinsert, fildes));
5151                                return (error);
5152                        }
5153                }
5154                mutex_exit(&resstp->sd_lock);

5156 #ifdef  _ILP32
5157                {
5158                        queue_t *q;
5159                        queue_t *mate = NULL;

5161                        /* get read queue of stream terminus */
5162                        claimstr(resstp->sd_wrq);
5163                        for (q = resstp->sd_wrq->q_next; q->q_next != NULL;
5164                            q = q->q_next)
5165                                if (!STRMATED(resstp) && STREAM(q) != resstp &&
5166                                    mate == NULL) {
5167                                        ASSERT(q->q_qinfo->qi_srvp);
5168                                        ASSERT(_OTHERQ(q)->q_qinfo->qi_srvp);
5169                                        claimstr(q);
5170                                        mate = q;
5171                                }
5172                        q = _RD(q);
5173                        if (mate)
5174                                releasestr(mate);
5175                        releasestr(resstp->sd_wrq);
5176                        ival = (t_uscalar_t)q;
5177                }
5178 #else
5179                ival = (t_uscalar_t)getminor(resftp->f_vnode->v_rdev);
5180 #endif  /* _ILP32 */

5182                if (STRUCT_FGET(strfdinsert, ctlbuf.len) <
5183                    STRUCT_FGET(strfdinsert, offset) + sizeof (t_uscalar_t)) {
5184                        releasef(STRUCT_FGET(strfdinsert, fildes));
5185                        return (EINVAL);
5186                }

5188                /*
5189                 * Check for legal flag value.
5190                 */
5191                if (STRUCT_FGET(strfdinsert, flags) & ~RS_HIPRI) {
5192                        releasef(STRUCT_FGET(strfdinsert, fildes));
5193                        return (EINVAL);
5194                }

5196                /* get these values from those cached in the stream head */
5197                mutex_enter(QLOCK(stp->sd_wrq));
5198                rmin = stp->sd_qn_minpsz;
5199                rmax = stp->sd_qn_maxpsz;
5200                mutex_exit(QLOCK(stp->sd_wrq));

5202                /*
5203                 * Make sure ctl and data sizes together fall within
5204                 * the limits of the max and min receive packet sizes
5205                 * and do not exceed system limit.  A negative data
5206                 * length means that no data part is to be sent.
5207                 */
5208                ASSERT((rmax >= 0) || (rmax == INFPSZ));
5209                if (rmax == 0) {
```



```
5210                        releasef(STRUCT_FGET(strfdinsert, fildes));
5211                        return (ERANGE);
5212                }
5213                if ((msgsize = STRUCT_FGET(strfdinsert, databuf.len)) < 0)
5214                        msgsize = 0;
5215                if ((msgsize < rmin) ||
5216                    ((msgsize > rmax) && (rmax != INFPSZ)) ||
5217                    (STRUCT_FGET(strfdinsert, ctlbuf.len) > strctlsz)) {
5218                        releasef(STRUCT_FGET(strfdinsert, fildes));
5219                        return (ERANGE);
5220                }

5222                mutex_enter(&stp->sd_lock);
5223                while (!(STRUCT_FGET(strfdinsert, flags) & RS_HIPRI) &&
5224                    !canputnext(stp->sd_wrq)) {
5225                        if ((error = strwaitq(stp, WRITEWAIT, (ssize_t)0,
5226                            flag, -1, &done)) != 0 || done) {
5227                                mutex_exit(&stp->sd_lock);
5228                                releasef(STRUCT_FGET(strfdinsert, fildes));
5229                                return (error);
5230                        }
5231                        if ((error = i_straccess(stp, access)) != 0) {
5232                                mutex_exit(&stp->sd_lock);
5233                                releasef(
5234                                    STRUCT_FGET(strfdinsert, fildes));
5235                                return (error);
5236                        }
5237                }
5238                mutex_exit(&stp->sd_lock);

5240                /*
5241                 * Copy strfdinsert.ctlbuf into native form of
5242                 * ctlbuf to pass down into strmakemsg().
5243                 */
5244                mctl.maxlen = STRUCT_FGET(strfdinsert, ctlbuf.maxlen);
5245                mctl.len = STRUCT_FGET(strfdinsert, ctlbuf.len);
5246                mctl.buf = STRUCT_FGETP(strfdinsert, ctlbuf.buf);

5248                iov.iov_base = STRUCT_FGETP(strfdinsert, databuf.buf);
5249                iov.iov_len = STRUCT_FGET(strfdinsert, databuf.len);
5250                uio.uio_iov = &iov;
5251                uio.uio_iovcnt = 1;
5252                uio.uio_loffset = 0;
5253                uio.uio_segflg = (copyflag == U_TO_K) ? UIO_USERSPACE :
5254                    UIO_SYSSPACE;
5255                uio.uio_fmode = 0;
5256                uio.uio_extflg = UIO_COPY_CACHED;
5257                uio.uio_resid = iov.iov_len;
5258                if ((error = strmakemsg(&mctl,
5259                    &msgsize, &uio, stp,
5260                    STRUCT_FGET(strfdinsert, flags), &mp)) != 0 || !mp) {
5261                        STRUCT_FSET(strfdinsert, databuf.len, msgsize);
5262                        releasef(STRUCT_FGET(strfdinsert, fildes));
5263                        return (error);
5264                }

5266                STRUCT_FSET(strfdinsert, databuf.len, msgsize);

5268                /*
5269                 * Place the possibly reencoded queue pointer 'offset' bytes
5270                 * from the start of the control portion of the message.
5271                 */
5272                *((t_uscalar_t *)(mp->b_rptr +
5273                    STRUCT_FGET(strfdinsert, offset))) = ival;

5275                /*
```

```
5276                 * Put message downstream.
5277                 */
5278                stream_willservice(stp);
5279                putnext(stp->sd_wrq, mp);
5280                stream_runservice(stp);
5281                releasef(STRUCT_FGET(strfdinsert, fildes));
5282                return (error);
5283        }

5285        case I_SENDFD:
5286        {
5287                struct file *fp;

5289                if ((fp = getf((int)arg)) == NULL)
5290                        return (EBADF);
5291                error = do_sendfp(stp, fp, crp);
5292                if (auditing) {
5293                        audit_fdsend((int)arg, fp, error);
5294                }
5295                releasef((int)arg);
5296                return (error);
5297        }

5299        case I_RECVFD:
5300        case I_E_RECVFD:
5301        {
5302                struct k_strrecvfd *srf;
5303                int i, fd;

5305                mutex_enter(&stp->sd_lock);
5306                while (!(mp = getq(rdq))) {
5307                        if (stp->sd_flag & (STRHUP|STREOF)) {
5308                                mutex_exit(&stp->sd_lock);
5309                                return (ENXIO);
5310                        }
5311                        if ((error = strwaitq(stp, GETWAIT, (ssize_t)0,
5312                            flag, -1, &done)) != 0 || done) {
5313                                mutex_exit(&stp->sd_lock);
5314                                return (error);
5315                        }
5316                        if ((error = i_straccess(stp, access)) != 0) {
5317                                mutex_exit(&stp->sd_lock);
5318                                return (error);
5319                        }
5320                }
5321                if (mp->b_datap->db_type != M_PASSFP) {
5322                        putback(stp, rdq, mp, mp->b_band);
5323                        mutex_exit(&stp->sd_lock);
5324                        return (EBADMSG);
5325                }
5326                mutex_exit(&stp->sd_lock);

5328                srf = (struct k_strrecvfd *)mp->b_rptr;
5329                if ((fd = ufalloc(0)) == -1) {
5330                        mutex_enter(&stp->sd_lock);
5331                        putback(stp, rdq, mp, mp->b_band);
5332                        mutex_exit(&stp->sd_lock);
5333                        return (EMFILE);
5334                }
5335                if (cmd == I_RECVFD) {
5336                        struct o_strrecvfd      ostrfd;

5338                        /* check to see if uid/gid values are too large. */

5340                        if (srf->uid > (o_uid_t)USHRT_MAX ||
5341                            srf->gid > (o_gid_t)USHRT_MAX) {
```



```
5342                                mutex_enter(&stp->sd_lock);
5343                                putback(stp, rdq, mp, mp->b_band);
5344                                mutex_exit(&stp->sd_lock);
5345                                setf(fd, NULL); /* release fd entry */
5346                                return (EOVERFLOW);
5347                        }

5349                        ostrfd.fd = fd;
5350                        ostrfd.uid = (o_uid_t)srf->uid;
5351                        ostrfd.gid = (o_gid_t)srf->gid;

5353                        /* Null the filler bits */
5354                        for (i = 0; i < 8; i++)
5355                                ostrfd.fill[i] = 0;

5357                        error = strcopyout(&ostrfd, (void *)arg,
5358                            sizeof (struct o_strrecvfd), copyflag);
5359                } else {                /* I_E_RECVFD */
5360                        struct strrecvfd        strfd;

5362                        strfd.fd = fd;
5363                        strfd.uid = srf->uid;
5364                        strfd.gid = srf->gid;

5366                        /* null the filler bits */
5367                        for (i = 0; i < 8; i++)
5368                                strfd.fill[i] = 0;

5370                        error = strcopyout(&strfd, (void *)arg,
5371                            sizeof (struct strrecvfd), copyflag);
5372                }

5374                if (error) {
5375                        setf(fd, NULL); /* release fd entry */
5376                        mutex_enter(&stp->sd_lock);
5377                        putback(stp, rdq, mp, mp->b_band);
5378                        mutex_exit(&stp->sd_lock);
5379                        return (error);
5380                }
5381                if (auditing) {
5382                        audit_fdrecv(fd, srf->fp);
5383                }

5385                /*
5386                 * Always increment f_count since the freemsg() below will
5387                 * always call free_passfp() which performs a closef().
5388                 */
5389                mutex_enter(&srf->fp->f_tlock);
5390                srf->fp->f_count++;
5391                mutex_exit(&srf->fp->f_tlock);
5392                setf(fd, srf->fp);
5393                freemsg(mp);
5394                return (0);
5395        }

5397        case I_SWROPT:
5398                /*
5399                 * Set/clear the write options. arg is a bit
5400                 * mask with any of the following bits set...
5401                 *      SNDZERO - send zero length message
5402                 *      SNDPIPE - send sigpipe to process if
5403                 *              sd_werror is set and process is
5404                 *              doing a write or putmsg.
5405                 * The new stream head write options should reflect
5406                 * what is in arg.
5407                 */
```

```
5408                if (arg & ~(SNDZERO|SNDPIPE))
5409                        return (EINVAL);

5411                mutex_enter(&stp->sd_lock);
5412                stp->sd_wput_opt &= ~(SW_SIGPIPE|SW_SNDZERO);
5413                if (arg & SNDZERO)
5414                        stp->sd_wput_opt |= SW_SNDZERO;
5415                if (arg & SNDPIPE)
5416                        stp->sd_wput_opt |= SW_SIGPIPE;
5417                mutex_exit(&stp->sd_lock);
5418                return (0);

5420        case I_GWROPT:
5421        {
5422                int wropt = 0;

5424                if (stp->sd_wput_opt & SW_SNDZERO)
5425                        wropt |= SNDZERO;
5426                if (stp->sd_wput_opt & SW_SIGPIPE)
5427                        wropt |= SNDPIPE;
5428                return (strcopyout(&wropt, (void *)arg, sizeof (wropt),
5429                    copyflag));
5430        }

5432        case I_LIST:
5433                /*
5434                 * Returns all the modules found on this stream,
5435                 * upto the driver. If argument is NULL, return the
5436                 * number of modules (including driver). If argument
5437                 * is not NULL, copy the names into the structure
5438                 * provided.
5439                 */

5441        {
5442                queue_t *q;
5443                char *qname;
5444                int i, nmods;
5445                struct str_mlist *mlist;
5446                STRUCT_DECL(str_list, strlist);

5448                if (arg == NULL) { /* Return number of modules plus driver */
5449                        if (stp->sd_vnode->v_type == VFIFO)
5450                                *rvalp = stp->sd_pushcnt;
5451                        else
5452                                *rvalp = stp->sd_pushcnt + 1;
5453                        return (0);
5454                }

5456                STRUCT_INIT(strlist, flag);

5458                error = strcopyin((void *)arg, STRUCT_BUF(strlist),
5459                    STRUCT_SIZE(strlist), copyflag);
5460                if (error != 0)
5461                        return (error);

5463                mlist = STRUCT_FGETP(strlist, sl_modlist);
5464                nmods = STRUCT_FGET(strlist, sl_nmods);
5465                if (nmods <= 0)
5466                        return (EINVAL);

5468                claimstr(stp->sd_wrq);
5469                q = stp->sd_wrq;
5470                for (i = 0; i < nmods && _SAMESTR(q); i++, q = q->q_next) {
5471                        qname = Q2NAME(q->q_next);
5472                        error = strcopyout(qname, &mlist[i], strlen(qname) + 1,
5473                            copyflag);
```



```
5474                        if (error != 0) {
5475                                releasestr(stp->sd_wrq);
5476                                return (error);
5477                        }
5478                }
5479                releasestr(stp->sd_wrq);
5480                return (strcopyout(&i, (void *)arg, sizeof (int), copyflag));
5481        }

5483        case I_CKBAND:
5484        {
5485                queue_t *q;
5486                qband_t *qbp;

5488                if ((arg < 0) || (arg >= NBAND))
5489                        return (EINVAL);
5490                q = _RD(stp->sd_wrq);
5491                mutex_enter(QLOCK(q));
5492                if (arg > (int)q->q_nband) {
5493                        *rvalp = 0;
5494                } else {
5495                        if (arg == 0) {
5496                                if (q->q_first)
5497                                        *rvalp = 1;
5498                                else
5499                                        *rvalp = 0;
5500                        } else {
5501                                qbp = q->q_bandp;
5502                                while (--arg > 0)
5503                                        qbp = qbp->qb_next;
5504                                if (qbp->qb_first)
5505                                        *rvalp = 1;
5506                                else
5507                                        *rvalp = 0;
5508                        }
5509                }
5510                mutex_exit(QLOCK(q));
5511                return (0);
5512        }

5514        case I_GETBAND:
5515        {
5516                int intpri;
5517                queue_t *q;

5519                q = _RD(stp->sd_wrq);
5520                mutex_enter(QLOCK(q));
5521                mp = q->q_first;
5522                if (!mp) {
5523                        mutex_exit(QLOCK(q));
5524                        return (ENODATA);
5525                }
5526                intpri = (int)mp->b_band;
5527                error = strcopyout(&intpri, (void *)arg, sizeof (int),
5528                    copyflag);
5529                mutex_exit(QLOCK(q));
5530                return (error);
5531        }

5533        case I_ATMARK:
5534        {
5535                queue_t *q;

5537                if (arg & ~(ANYMARK|LASTMARK))
5538                        return (EINVAL);
5539                q = _RD(stp->sd_wrq);
```

```
5540                mutex_enter(&stp->sd_lock);
5541                if ((stp->sd_flag & STRATMARK) && (arg == ANYMARK)) {
5542                        *rvalp = 1;
5543                } else {
5544                        mutex_enter(QLOCK(q));
5545                        mp = q->q_first;

5547                        if (mp == NULL)
5548                                *rvalp = 0;
5549                        else if ((arg == ANYMARK) && (mp->b_flag & MSGMARK))
5550                                *rvalp = 1;
5551                        else if ((arg == LASTMARK) && (mp == stp->sd_mark))
5552                                *rvalp = 1;
5553                        else
5554                                *rvalp = 0;
5555                        mutex_exit(QLOCK(q));
5556                }
5557                mutex_exit(&stp->sd_lock);
5558                return (0);
5559        }

5561        case I_CANPUT:
5562        {
5563                char band;

5565                if ((arg < 0) || (arg >= NBAND))
5566                        return (EINVAL);
5567                band = (char)arg;
5568                *rvalp = bcanputnext(stp->sd_wrq, band);
5569                return (0);
5570        }

5572        case I_SETCLTIME:
5573        {
5574                int closetime;

5576                error = strcopyin((void *)arg, &closetime, sizeof (int),
5577                    copyflag);
5578                if (error)
5579                        return (error);
5580                if (closetime < 0)
5581                        return (EINVAL);

5583                stp->sd_closetime = closetime;
5584                return (0);
5585        }

5587        case I_GETCLTIME:
5588        {
5589                int closetime;

5591                closetime = stp->sd_closetime;
5592                return (strcopyout(&closetime, (void *)arg, sizeof (int),
5593                    copyflag));
5594        }

5596        case TIOCGSID:
5597        {
5598                pid_t sid;

5600                mutex_enter(&stp->sd_lock);
5601                if (stp->sd_sidp == NULL) {
5602                        mutex_exit(&stp->sd_lock);
5603                        return (ENOTTY);
5604                }
5605                sid = stp->sd_sidp->pid_id;
```



```
5606                mutex_exit(&stp->sd_lock);
5607                return (strcopyout(&sid, (void *)arg, sizeof (pid_t),
5608                    copyflag));
5609        }

5611        case TIOCSPGRP:
5612        {
5613                pid_t pgrp;
5614                proc_t *q;
5615                pid_t   sid, fg_pgid, bg_pgid;

5617                if (error = strcopyin((void *)arg, &pgrp, sizeof (pid_t),
5618                    copyflag))
5619                        return (error);
5620                mutex_enter(&stp->sd_lock);
5621                mutex_enter(&pidlock);
5622                if (stp->sd_sidp != ttoproc(curthread)->p_sessp->s_sidp) {
5623                        mutex_exit(&pidlock);
5624                        mutex_exit(&stp->sd_lock);
5625                        return (ENOTTY);
5626                }
5627                if (pgrp == stp->sd_pgidp->pid_id) {
5628                        mutex_exit(&pidlock);
5629                        mutex_exit(&stp->sd_lock);
5630                        return (0);
5631                }
5632                if (pgrp <= 0 || pgrp >= maxpid) {
5633                        mutex_exit(&pidlock);
5634                        mutex_exit(&stp->sd_lock);
5635                        return (EINVAL);
5636                }
5637                if ((q = pgfind(pgrp)) == NULL ||
5638                    q->p_sessp != ttoproc(curthread)->p_sessp) {
5639                        mutex_exit(&pidlock);
5640                        mutex_exit(&stp->sd_lock);
5641                        return (EPERM);
5642                }
5643                sid = stp->sd_sidp->pid_id;
5644                fg_pgid = q->p_pgrp;
5645                bg_pgid = stp->sd_pgidp->pid_id;
5646                CL_SET_PROCESS_GROUP(curthread, sid, bg_pgid, fg_pgid);
5647                PID_RELE(stp->sd_pgidp);
5648                ctty_clear_sighuped();
5649                stp->sd_pgidp = q->p_pgidp;
5650                PID_HOLD(stp->sd_pgidp);
5651                mutex_exit(&pidlock);
5652                mutex_exit(&stp->sd_lock);
5653                return (0);
5654        }

5656        case TIOCGPGRP:
5657        {
5658                pid_t pgrp;

5660                mutex_enter(&stp->sd_lock);
5661                if (stp->sd_sidp == NULL) {
5662                        mutex_exit(&stp->sd_lock);
5663                        return (ENOTTY);
5664                }
5665                pgrp = stp->sd_pgidp->pid_id;
5666                mutex_exit(&stp->sd_lock);
5667                return (strcopyout(&pgrp, (void *)arg, sizeof (pid_t),
5668                    copyflag));
5669        }

5671        case TIOCSCTTY:
```

```
5672         {
5673                 return (strctty(stp));
5674         }

5676         case TIOCNOTTY:
5677         {
5678                 /* freectty() always assumes curproc. */
5679                 if (freectty(B_FALSE) != 0)
5680                         return (0);
5681                 return (ENOTTY);
5682         }

5684         case FIONBIO:
5685         case FIOASYNC:
5686                 return (0);     /* handled by the upper layer */
5687         case F_FORKED: {
5688                 if (crp != kcred)
5689                         return (-1);
5690                 sh_insert_pid(stp, (proc_t *)arg);
5691                 return (0);
5692         }
5693         case F_CLOSED: {
5694                 if (crp != kcred)
5695                         return (-1);
5696                 sh_remove_pid(stp, (proc_t *)arg);
5697                 return (0);
5698         }
5699 #endif /* ! codereview */
5700         }
5701 }

5703 /*
5704  * Custom free routine used for M_PASSFP messages.
5705  */
5706 static void
5707 free_passfp(struct k_strrecvfd *srf)
5708 {
5709         (void) closef(srf->fp);
5710         kmem_free(srf, sizeof (struct k_strrecvfd) + sizeof (frtn_t));
5711 }

5713 /* ARGSUSED */
5714 int
5715 do_sendfp(struct stdata *stp, struct file *fp, struct cred *cr)
5716 {
5717         queue_t *qp, *nextqp;
5718         struct k_strrecvfd *srf;
5719         mblk_t *mp;
5720         frtn_t *frtnp;
5721         size_t bufsize;
5722         queue_t *mate = NULL;
5723         syncq_t *sq = NULL;
5724         int retval = 0;

5726         if (stp->sd_flag & STRHUP)
5727                 return (ENXIO);

5729         claimstr(stp->sd_wrq);

5731         /* Fastpath, we have a pipe, and we are already mated, use it. */
5732         if (STRMATED(stp)) {
5733                 qp = _RD(stp->sd_mate->sd_wrq);
5734                 claimstr(qp);
5735                 mate = qp;
5736         } else { /* Not already mated. */
```



```
5738                 /*
5739                  * Walk the stream to the end of this one.
5740                  * assumes that the claimstr() will prevent
5741                  * plumbing between the stream head and the
5742                  * driver from changing
5743                  */
5744                 qp = stp->sd_wrq;

5746                 /*
5747                  * Loop until we reach the end of this stream.
5748                  * On completion, qp points to the write queue
5749                  * at the end of the stream, or the read queue
5750                  * at the stream head if this is a fifo.
5751                  */
5752                 while (((qp = qp->q_next) != NULL) && _SAMESTR(qp))
5753                         ;

5755                 /*
5756                  * Just in case we get a q_next which is NULL, but
5757                  * not at the end of the stream.  This is actually
5758                  * broken, so we set an assert to catch it in
5759                  * debug, and set an error and return if not debug.
5760                  */
5761                 ASSERT(qp);
5762                 if (qp == NULL) {
5763                         releasestr(stp->sd_wrq);
5764                         return (EINVAL);
5765                 }

5767                 /*
5768                  * Enter the syncq for the driver, so (hopefully)
5769                  * the queue values will not change on us.
5770                  * XXXX - This will only prevent the race IFF only
5771                  *   the write side modifies the q_next member, and
5772                  *   the put procedure is protected by at least
5773                  *   MT_PERQ.
5774                  */
5775                 if ((sq = qp->q_syncq) != NULL)
5776                         entersq(sq, SQ_PUT);

5778                 /* Now get the q_next value from this qp. */
5779                 nextqp = qp->q_next;

5781                 /*
5782                  * If nextqp exists and the other stream is different
5783                  * from this one claim the stream, set the mate, and
5784                  * get the read queue at the stream head of the other
5785                  * stream.  Assumes that nextqp was at least valid when
5786                  * we got it.  Hopefully the entersq of the driver
5787                  * will prevent it from changing on us.
5788                  */
5789                 if ((nextqp != NULL) && (STREAM(nextqp) != stp)) {
5790                         ASSERT(qp->q_qinfo->qi_srvp);
5791                         ASSERT(_OTHERQ(qp)->q_qinfo->qi_srvp);
5792                         ASSERT(_OTHERQ(qp->q_next)->q_qinfo->qi_srvp);
5793                         claimstr(nextqp);

5795                         /* Make sure we still have a q_next */
5796                         if (nextqp != qp->q_next) {
5797                                 releasestr(stp->sd_wrq);
5798                                 releasestr(nextqp);
5799                                 return (EINVAL);
5800                         }

5802                         qp = _RD(STREAM(nextqp)->sd_wrq);
5803                         mate = qp;
```

```
5804                }
5805                /* If we entered the synq above, leave it. */
5806                if (sq != NULL)
5807                        leavesq(sq, SQ_PUT);
5808        } /*  STRMATED(STP)  */

5810        /* XXX prevents substitution of the ops vector */
5811        if (qp->q_qinfo != &strdata && qp->q_qinfo != &fifo_strdata) {
5812                retval = EINVAL;
5813                goto out;
5814        }

5816        if (qp->q_flag & QFULL) {
5817                retval = EAGAIN;
5818                goto out;
5819        }

5821        /*
5822         * Since M_PASSFP messages include a file descriptor, we use
5823         * esballoc() and specify a custom free routine (free_passfp()) that
5824         * will close the descriptor as part of freeing the message.  For
5825         * convenience, we stash the frtn_t right after the data block.
5826         */
5827        bufsize = sizeof (struct k_strrecvfd) + sizeof (frtn_t);
5828        srf = kmem_alloc(bufsize, KM_NOSLEEP);
5829        if (srf == NULL) {
5830                retval = EAGAIN;
5831                goto out;
5832        }

5834        frtnp = (frtn_t *)(srf + 1);
5835        frtnp->free_arg = (caddr_t)srf;
5836        frtnp->free_func = free_passfp;

5838        mp = esballoc((uchar_t *)srf, bufsize, BPRI_MED, frtnp);
5839        if (mp == NULL) {
5840                kmem_free(srf, bufsize);
5841                retval = EAGAIN;
5842                goto out;
5843        }
5844        mp->b_wptr += sizeof (struct k_strrecvfd);
5845        mp->b_datap->db_type = M_PASSFP;

5847        srf->fp = fp;
5848        srf->uid = crgetuid(curthread->t_cred);
5849        srf->gid = crgetgid(curthread->t_cred);
5850        mutex_enter(&fp->f_tlock);
5851        fp->f_count++;
5852        mutex_exit(&fp->f_tlock);

5854        put(qp, mp);
5855 out:
5856        releasestr(stp->sd_wrq);
5857        if (mate)
5858                releasestr(mate);
5859        return (retval);
5860 }

5862 /*
5863  * Send an ioctl message downstream and wait for acknowledgement.
5864  * flags may be set to either U_TO_K or K_TO_K and a combination
5865  * of STR_NOERROR or STR_NOSIG
5866  * STR_NOSIG: Signals are essentially ignored or held and have
5867  *      no effect for the duration of the call.
5868  * STR_NOERROR: Ignores stream head read, write and hup errors.
5869  *      Additionally, if an existing ioctl times out, it is assumed
```



```
5870  *      lost and and this ioctl will continue as if the previous ioctl had
5871  *      finished.  ETIME may be returned if this ioctl times out (i.e.
5872  *      ic_timout is not INFTIM).  Non-stream head errors may be returned if
5873  *      the ioc_error indicates that the driver/module had problems,
5874  *      an EFAULT was found when accessing user data, a lack of
5875  *      resources, etc.
5876  */
5877 int
5878 strdoioctl(
5879         struct stdata *stp,
5880         struct strioctl *strioc,
5881         int fflags,             /* file flags with model info */
5882         int flag,
5883         cred_t *crp,
5884         int *rvalp)
5885 {
5886         mblk_t *bp;
5887         struct iocblk *iocbp;
5888         struct copyreq *reqp;
5889         struct copyresp *resp;
5890         int id;
5891         int transparent = 0;
5892         int error = 0;
5893         int len = 0;
5894         caddr_t taddr;
5895         int copyflag = (flag & (U_TO_K | K_TO_K));
5896         int sigflag = (flag & STR_NOSIG);
5897         int errs;
5898         uint_t waitflags;
5899         boolean_t set_iocwaitne = B_FALSE;

5901         ASSERT(copyflag == U_TO_K || copyflag == K_TO_K);
5902         ASSERT((fflags & FMODELS) != 0);

5904         TRACE_2(TR_FAC_STREAMS_FR,
5905             TR_STRDOIOCTL,
5906             "strdoioctl:stp %p strioc %p", stp, strioc);
5907         if (strioc->ic_len == TRANSPARENT) {    /* send arg in M_DATA block */
5908                 transparent = 1;
5909                 strioc->ic_len = sizeof (intptr_t);
5910         }

5912         if (strioc->ic_len < 0 || (strmsgsz > 0 && strioc->ic_len > strmsgsz))
5913                 return (EINVAL);

5915         if ((bp = allocb_cred_wait(sizeof (union ioctypes), sigflag, &error,
5916             crp, curproc->p_pid)) == NULL)
5917                         return (error);

5919         bzero(bp->b_wptr, sizeof (union ioctypes));

5921         iocbp = (struct iocblk *)bp->b_wptr;
5922         iocbp->ioc_count = strioc->ic_len;
5923         iocbp->ioc_cmd = strioc->ic_cmd;
5924         iocbp->ioc_flag = (fflags & FMODELS);

5926         crhold(crp);
5927         iocbp->ioc_cr = crp;
5928         DB_TYPE(bp) = M_IOCTL;
5929         bp->b_wptr += sizeof (struct iocblk);

5931         if (flag & STR_NOERROR)
5932                 errs = STPLEX;
5933         else
5934                 errs = STRHUP|STRDERR|STWRERR|STPLEX;
```

```
5936         /*
5937          * If there is data to copy into ioctl block, do so.
5938          */
5939         if (iocbp->ioc_count > 0) {
5940                 if (transparent)
5941                         /*
5942                          * Note: STR_NOERROR does not have an effect
5943                          * in putiocd()
5944                          */
5945                         id = K_TO_K | sigflag;
5946                 else
5947                         id = flag;
5948                 if ((error = putiocd(bp, strioc->ic_dp, id, crp)) != 0) {
5949                         freemsg(bp);
5950                         crfree(crp);
5951                         return (error);
5952                 }
5953 
5954                 /*
5955                  * We could have slept copying in user pages.
5956                  * Recheck the stream head state (the other end
5957                  * of a pipe could have gone away).
5958                  */
5959                 if (stp->sd_flag & errs) {
5960                         mutex_enter(&stp->sd_lock);
5961                         error = strgeterr(stp, errs, 0);
5962                         mutex_exit(&stp->sd_lock);
5963                         if (error != 0) {
5964                                 freemsg(bp);
5965                                 crfree(crp);
5966                                 return (error);
5967                         }
5968                 }
5969         }
5970         if (transparent)
5971                 iocbp->ioc_count = TRANSPARENT;
5972 
5973         /*
5974          * Block for up to STRTIMOUT milliseconds if there is an outstanding
5975          * ioctl for this stream already running.  All processes
5976          * sleeping here will be awakened as a result of an ACK
5977          * or NAK being received for the outstanding ioctl, or
5978          * as a result of the timer expiring on the outstanding
5979          * ioctl (a failure), or as a result of any waiting
5980          * process's timer expiring (also a failure).
5981          */
5982 
5983         error = 0;
5984         mutex_enter(&stp->sd_lock);
5985         while ((stp->sd_flag & IOCWAIT) ||
5986             (!set_iocwaitne && (stp->sd_flag & IOCWAITNE))) {
5987                 clock_t cv_rval;
5988 
5989                 TRACE_0(TR_FAC_STREAMS_FR,
5990                     TR_STRDOIOCTL_WAIT,
5991                     "strdoioctl sleeps - IOCWAIT");
5992                 cv_rval = str_cv_wait(&stp->sd_iocmonitor, &stp->sd_lock,
5993                     STRTIMOUT, sigflag);
5994                 if (cv_rval <= 0) {
5995                         if (cv_rval == 0) {
5996                                 error = EINTR;
5997                         } else {
5998                                 if (flag & STR_NOERROR) {
5999                                         /*
6000                                          * Terminating current ioctl in
6001                                          * progress -- assume it got lost and
```



```
6002                                          * wake up the other thread so that the
6003                                          * operation completes.
6004                                          */
6005                                         if (!(stp->sd_flag & IOCWAITNE)) {
6006                                                 set_iocwaitne = B_TRUE;
6007                                                 stp->sd_flag |= IOCWAITNE;
6008                                                 cv_broadcast(&stp->sd_monitor);
6009                                         }
6010                                         /*
6011                                          * Otherwise, there's a running
6012                                          * STR_NOERROR -- we have no choice
6013                                          * here but to wait forever (or until
6014                                          * interrupted).
6015                                          */
6016                                 } else {
6017                                         /*
6018                                          * pending ioctl has caused
6019                                          * us to time out
6020                                          */
6021                                         error = ETIME;
6022                                 }
6023                         }
6024                 } else if ((stp->sd_flag & errs)) {
6025                         error = strgeterr(stp, errs, 0);
6026                 }
6027                 if (error) {
6028                         mutex_exit(&stp->sd_lock);
6029                         freemsg(bp);
6030                         crfree(crp);
6031                         return (error);
6032                 }
6033         }
6034 
6035         /*
6036          * Have control of ioctl mechanism.
6037          * Send down ioctl packet and wait for response.
6038          */
6039         if (stp->sd_iocblk != (mblk_t *)-1) {
6040                 freemsg(stp->sd_iocblk);
6041         }
6042         stp->sd_iocblk = NULL;
6043 
6044         /*
6045          * If this is marked with 'noerror' (internal; mostly
6046          * I_{P,}{UN,}LINK), then make sure nobody else is able to get
6047          * in here by setting IOCWAITNE.
6048          */
6049         waitflags = IOCWAIT;
6050         if (flag & STR_NOERROR)
6051                 waitflags |= IOCWAITNE;
6052 
6053         stp->sd_flag |= waitflags;
6054 
6055         /*
6056          * Assign sequence number.
6057          */
6058         iocbp->ioc_id = stp->sd_iocid = getiocseqno();
6059 
6060         mutex_exit(&stp->sd_lock);
6061 
6062         TRACE_1(TR_FAC_STREAMS_FR,
6063             TR_STRDOIOCTL_PUT, "strdoioctl put: stp %p", stp);
6064         stream_willservice(stp);
6065         putnext(stp->sd_wrq, bp);
6066         stream_runservice(stp);
```

```
6068         /*
6069          * Timed wait for acknowledgment.  The wait time is limited by the
6070          * timeout value, which must be a positive integer (number of
6071          * milliseconds) to wait, or 0 (use default value of STRTIMOUT
6072          * milliseconds), or -1 (wait forever).  This will be awakened
6073          * either by an ACK/NAK message arriving, the timer expiring, or
6074          * the timer expiring on another ioctl waiting for control of the
6075          * mechanism.
6076          */
6077 waitioc:
6078         mutex_enter(&stp->sd_lock);


6081         /*
6082          * If the reply has already arrived, don't sleep.  If awakened from
6083          * the sleep, fail only if the reply has not arrived by then.
6084          * Otherwise, process the reply.
6085          */
6086         while (!stp->sd_iocblk) {
6087                 clock_t cv_rval;

6089                 if (stp->sd_flag & errs) {
6090                         error = strgeterr(stp, errs, 0);
6091                         if (error != 0) {
6092                                 stp->sd_flag &= ~waitflags;
6093                                 cv_broadcast(&stp->sd_iocmonitor);
6094                                 mutex_exit(&stp->sd_lock);
6095                                 crfree(crp);
6096                                 return (error);
6097                         }
6098                 }

6100                 TRACE_0(TR_FAC_STREAMS_FR,
6101                     TR_STRDOIOCTL_WAIT2,
6102                     "strdoioctl sleeps awaiting reply");
6103                 ASSERT(error == 0);

6105                 cv_rval = str_cv_wait(&stp->sd_monitor, &stp->sd_lock,
6106                     (strioc->ic_timout ?
6107                     strioc->ic_timout * 1000 : STRTIMOUT), sigflag);

6109                 /*
6110                  * There are four possible cases here: interrupt, timeout,
6111                  * wakeup by IOCWAITNE (above), or wakeup by strrput_nondata (a
6112                  * valid M_IOCTL reply).
6113                  *
6114                  * If we've been awakened by a STR_NOERROR ioctl on some other
6115                  * thread, then sd_iocblk will still be NULL, and IOCWAITNE
6116                  * will be set.  Pretend as if we just timed out.  Note that
6117                  * this other thread waited at least STRTIMOUT before trying to
6118                  * awaken our thread, so this is indistinguishable (even for
6119                  * INFTIM) from the case where we failed with ETIME waiting on
6120                  * IOCWAIT in the prior loop.
6121                  */
6122                 if (cv_rval > 0 && !(flag & STR_NOERROR) &&
6123                     stp->sd_iocblk == NULL && (stp->sd_flag & IOCWAITNE)) {
6124                         cv_rval = -1;
6125                 }

6127                 /*
6128                  * note: STR_NOERROR does not protect
6129                  * us here.. use ic_timout < 0
6130                  */
6131                 if (cv_rval <= 0) {
6132                         if (cv_rval == 0) {
6133                                 error = EINTR;
```



```
6134                         } else {
6135                                 error =  ETIME;
6136                         }
6137                         /*
6138                          * A message could have come in after we were scheduled
6139                          * but before we were actually run.
6140                          */
6141                         bp = stp->sd_iocblk;
6142                         stp->sd_iocblk = NULL;
6143                         if (bp != NULL) {
6144                                 if ((bp->b_datap->db_type == M_COPYIN) ||
6145                                     (bp->b_datap->db_type == M_COPYOUT)) {
6146                                         mutex_exit(&stp->sd_lock);
6147                                         if (bp->b_cont) {
6148                                                 freemsg(bp->b_cont);
6149                                                 bp->b_cont = NULL;
6150                                         }
6151                                         bp->b_datap->db_type = M_IOCDATA;
6152                                         bp->b_wptr = bp->b_rptr +
6153                                             sizeof (struct copyresp);
6154                                         resp = (struct copyresp *)bp->b_rptr;
6155                                         resp->cp_rval =
6156                                             (caddr_t)1; /* failure */
6157                                         stream_willservice(stp);
6158                                         putnext(stp->sd_wrq, bp);
6159                                         stream_runservice(stp);
6160                                         mutex_enter(&stp->sd_lock);
6161                                 } else {
6162                                         freemsg(bp);
6163                                 }
6164                         }
6165                         stp->sd_flag &= ~waitflags;
6166                         cv_broadcast(&stp->sd_iocmonitor);
6167                         mutex_exit(&stp->sd_lock);
6168                         crfree(crp);
6169                         return (error);
6170                 }
6171         }
6172         bp = stp->sd_iocblk;
6173         /*
6174          * Note: it is strictly impossible to get here with sd_iocblk set to
6175          * -1.  This is because the initial loop above doesn't allow any new
6176          * ioctls into the fray until all others have passed this point.
6177          */
6178         ASSERT(bp != NULL && bp != (mblk_t *)-1);
6179         TRACE_1(TR_FAC_STREAMS_FR,
6180             TR_STRDOIOCTL_ACK, "strdoioctl got reply: bp %p", bp);
6181         if ((bp->b_datap->db_type == M_IOCACK) ||
6182             (bp->b_datap->db_type == M_IOCNAK)) {
6183                 /* for detection of duplicate ioctl replies */
6184                 stp->sd_iocblk = (mblk_t *)-1;
6185                 stp->sd_flag &= ~waitflags;
6186                 cv_broadcast(&stp->sd_iocmonitor);
6187                 mutex_exit(&stp->sd_lock);
6188         } else {
6189                 /*
6190                  * flags not cleared here because we're still doing
6191                  * copy in/out for ioctl.
6192                  */
6193                 stp->sd_iocblk = NULL;
6194                 mutex_exit(&stp->sd_lock);
6195         }


6198         /*
6199          * Have received acknowledgment.
```

```
6200          */
6202         switch (bp->b_datap->db_type) {
6203         case M_IOCACK:
6204                 /*
6205                  * Positive ack.
6206                  */
6207                 iocbp = (struct iocblk *)bp->b_rptr;
6209                 /*
6210                  * Set error if indicated.
6211                  */
6212                 if (iocbp->ioc_error) {
6213                         error = iocbp->ioc_error;
6214                         break;
6215                 }
6217                 /*
6218                  * Set return value.
6219                  */
6220                 *rvalp = iocbp->ioc_rval;
6222                 /*
6223                  * Data may have been returned in ACK message (ioc_count > 0).
6224                  * If so, copy it out to the user's buffer.
6225                  */
6226                 if (iocbp->ioc_count && !transparent) {
6227                         if (error = getiocd(bp, strioc->ic_dp, copyflag))
6228                                 break;
6229                 }
6230                 if (!transparent) {
6231                         if (len)        /* an M_COPYOUT was used with I_STR */
6232                                 strioc->ic_len = len;
6233                         else
6234                                 strioc->ic_len = (int)iocbp->ioc_count;
6235                 }
6236                 break;
6238         case M_IOCNAK:
6239                 /*
6240                  * Negative ack.
6241                  *
6242                  * The only thing to do is set error as specified
6243                  * in neg ack packet.
6244                  */
6245                 iocbp = (struct iocblk *)bp->b_rptr;
6247                 error = (iocbp->ioc_error ? iocbp->ioc_error : EINVAL);
6248                 break;
6250         case M_COPYIN:
6251                 /*
6252                  * Driver or module has requested user ioctl data.
6253                  */
6254                 reqp = (struct copyreq *)bp->b_rptr;
6256                 /*
6257                  * M_COPYIN should *never* have a message attached, though
6258                  * it's harmless if it does -- thus, panic on a DEBUG
6259                  * kernel and just free it on a non-DEBUG build.
6260                  */
6261                 ASSERT(bp->b_cont == NULL);
6262                 if (bp->b_cont != NULL) {
6263                         freemsg(bp->b_cont);
6264                         bp->b_cont = NULL;
6265                 }
```



```
6267                 error = putiocd(bp, reqp->cq_addr, flag, crp);
6268                 if (error && bp->b_cont) {
6269                         freemsg(bp->b_cont);
6270                         bp->b_cont = NULL;
6271                 }
6273                 bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
6274                 bp->b_datap->db_type = M_IOCDATA;
6276                 mblk_setcred(bp, crp, curproc->p_pid);
6277                 resp = (struct copyresp *)bp->b_rptr;
6278                 resp->cp_rval = (caddr_t)(uintptr_t)error;
6279                 resp->cp_flag = (fflags & FMODELS);
6281                 stream_willservice(stp);
6282                 putnext(stp->sd_wrq, bp);
6283                 stream_runservice(stp);
6285                 if (error) {
6286                         mutex_enter(&stp->sd_lock);
6287                         stp->sd_flag &= ~waitflags;
6288                         cv_broadcast(&stp->sd_iocmonitor);
6289                         mutex_exit(&stp->sd_lock);
6290                         crfree(crp);
6291                         return (error);
6292                 }
6294                 goto waitioc;
6296         case M_COPYOUT:
6297                 /*
6298                  * Driver or module has ioctl data for a user.
6299                  */
6300                 reqp = (struct copyreq *)bp->b_rptr;
6301                 ASSERT(bp->b_cont != NULL);
6303                 /*
6304                  * Always (transparent or non-transparent )
6305                  * use the address specified in the request
6306                  */
6307                 taddr = reqp->cq_addr;
6308                 if (!transparent)
6309                         len = (int)reqp->cq_size;
6311                 /* copyout data to the provided address */
6312                 error = getiocd(bp, taddr, copyflag);
6314                 freemsg(bp->b_cont);
6315                 bp->b_cont = NULL;
6317                 bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
6318                 bp->b_datap->db_type = M_IOCDATA;
6320                 mblk_setcred(bp, crp, curproc->p_pid);
6321                 resp = (struct copyresp *)bp->b_rptr;
6322                 resp->cp_rval = (caddr_t)(uintptr_t)error;
6323                 resp->cp_flag = (fflags & FMODELS);
6325                 stream_willservice(stp);
6326                 putnext(stp->sd_wrq, bp);
6327                 stream_runservice(stp);
6329                 if (error) {
6330                         mutex_enter(&stp->sd_lock);
6331                         stp->sd_flag &= ~waitflags;
```

```
6332                        cv_broadcast(&stp->sd_iocmonitor);
6333                        mutex_exit(&stp->sd_lock);
6334                        crfree(crp);
6335                        return (error);
6336                }
6337                goto waitioc;

6339        default:
6340                ASSERT(0);
6341                mutex_enter(&stp->sd_lock);
6342                stp->sd_flag &= ~waitflags;
6343                cv_broadcast(&stp->sd_iocmonitor);
6344                mutex_exit(&stp->sd_lock);
6345                break;
6346        }

6348        freemsg(bp);
6349        crfree(crp);
6350        return (error);
6351 }

6353 /*
6354  * Send an M_CMD message downstream and wait for a reply.  This is a ptools
6355  * special used to retrieve information from modules/drivers a stream without
6356  * being subjected to flow control or interfering with pending messages on the
6357  * stream (e.g. an ioctl in flight).
6358  */
6359 int
6360 strdocmd(struct stdata *stp, struct strcmd *scp, cred_t *crp)
6361 {
6362        mblk_t *mp;
6363        struct cmdblk *cmdp;
6364        int error = 0;
6365        int errs = STRHUP|STRDERR|STWRERR|STPLEX;
6366        clock_t rval, timeout = STRTIMOUT;

6368        if (scp->sc_len < 0 || scp->sc_len > sizeof (scp->sc_buf) ||
6369            scp->sc_timeout < -1)
6370                return (EINVAL);

6372        if (scp->sc_timeout > 0)
6373                timeout = scp->sc_timeout * MILLISEC;

6375        if ((mp = allocb_cred(sizeof (struct cmdblk), crp,
6376            curproc->p_pid)) == NULL)
6377                return (ENOMEM);

6379        crhold(crp);

6381        cmdp = (struct cmdblk *)mp->b_wptr;
6382        cmdp->cb_cr = crp;
6383        cmdp->cb_cmd = scp->sc_cmd;
6384        cmdp->cb_len = scp->sc_len;
6385        cmdp->cb_error = 0;
6386        mp->b_wptr += sizeof (struct cmdblk);

6388        DB_TYPE(mp) = M_CMD;
6389        DB_CPID(mp) = curproc->p_pid;

6391        /*
6392         * Copy in the payload.
6393         */
6394        if (cmdp->cb_len > 0) {
6395                mp->b_cont = allocb_cred(sizeof (scp->sc_buf), crp,
6396                    curproc->p_pid);
6397                if (mp->b_cont == NULL) {
```



```
6398                        error = ENOMEM;
6399                        goto out;
6400                }

6402                /* cb_len comes from sc_len, which has already been checked */
6403                ASSERT(cmdp->cb_len <= sizeof (scp->sc_buf));
6404                (void) bcopy(scp->sc_buf, mp->b_cont->b_wptr, cmdp->cb_len);
6405                mp->b_cont->b_wptr += cmdp->cb_len;
6406                DB_CPID(mp->b_cont) = curproc->p_pid;
6407        }

6409        /*
6410         * Since this mechanism is strictly for ptools, and since only one
6411         * process can be grabbed at a time, we simply fail if there's
6412         * currently an operation pending.
6413         */
6414        mutex_enter(&stp->sd_lock);
6415        if (stp->sd_flag & STRCMDWAIT) {
6416                mutex_exit(&stp->sd_lock);
6417                error = EBUSY;
6418                goto out;
6419        }
6420        stp->sd_flag |= STRCMDWAIT;
6421        ASSERT(stp->sd_cmdblk == NULL);
6422        mutex_exit(&stp->sd_lock);

6424        putnext(stp->sd_wrq, mp);
6425        mp = NULL;

6427        /*
6428         * Timed wait for acknowledgment.  If the reply has already arrived,
6429         * don't sleep.  If awakened from the sleep, fail only if the reply
6430         * has not arrived by then.  Otherwise, process the reply.
6431         */
6432        mutex_enter(&stp->sd_lock);
6433        while (stp->sd_cmdblk == NULL) {
6434                if (stp->sd_flag & errs) {
6435                        if ((error = strgeterr(stp, errs, 0)) != 0)
6436                                goto waitout;
6437                }

6439                rval = str_cv_wait(&stp->sd_monitor, &stp->sd_lock, timeout, 0);
6440                if (stp->sd_cmdblk != NULL)
6441                        break;

6443                if (rval <= 0) {
6444                        error = (rval == 0) ? EINTR : ETIME;
6445                        goto waitout;
6446                }
6447        }

6449        /*
6450         * We received a reply.
6451         */
6452        mp = stp->sd_cmdblk;
6453        stp->sd_cmdblk = NULL;
6454        ASSERT(mp != NULL && DB_TYPE(mp) == M_CMD);
6455        ASSERT(stp->sd_flag & STRCMDWAIT);
6456        stp->sd_flag &= ~STRCMDWAIT;
6457        mutex_exit(&stp->sd_lock);

6459        cmdp = (struct cmdblk *)mp->b_rptr;
6460        if ((error = cmdp->cb_error) != 0)
6461                goto out;

6463        /*
```

```
6464          * Data may have been returned in the reply (cb_len > 0).
6465          * If so, copy it out to the user's buffer.
6466          */
6467         if (cmdp->cb_len > 0) {
6468                 if (mp->b_cont == NULL || MBLKL(mp->b_cont) < cmdp->cb_len) {
6469                         error = EPROTO;
6470                         goto out;
6471                 }

6473                 cmdp->cb_len = MIN(cmdp->cb_len, sizeof (scp->sc_buf));
6474                 (void) bcopy(mp->b_cont->b_rptr, scp->sc_buf, cmdp->cb_len);
6475         }
6476         scp->sc_len = cmdp->cb_len;
6477 out:
6478         freemsg(mp);
6479         crfree(crp);
6480         return (error);
6481 waitout:
6482         ASSERT(stp->sd_cmdblk == NULL);
6483         stp->sd_flag &= ~STRCMDWAIT;
6484         mutex_exit(&stp->sd_lock);
6485         crfree(crp);
6486         return (error);
6487 }

6489 /*
6490  * For the SunOS keyboard driver.
6491  * Return the next available "ioctl" sequence number.
6492  * Exported, so that streams modules can send "ioctl" messages
6493  * downstream from their open routine.
6494  */
6495 int
6496 getiocseqno(void)
6497 {
6498         int     i;

6500         mutex_enter(&strresources);
6501         i = ++ioc_id;
6502         mutex_exit(&strresources);
6503         return (i);
6504 }

6506 /*
6507  * Get the next message from the read queue.  If the message is
6508  * priority, STRPRI will have been set by strrput().  This flag
6509  * should be reset only when the entire message at the front of the
6510  * queue as been consumed.
6511  *
6512  * NOTE: strgetmsg and kstrgetmsg have much of the logic in common.
6513  */
6514 int
6515 strgetmsg(
6516         struct vnode *vp,
6517         struct strbuf *mctl,
6518         struct strbuf *mdata,
6519         unsigned char *prip,
6520         int *flagsp,
6521         int fmode,
6522         rval_t *rvp)
6523 {
6524         struct stdata *stp;
6525         mblk_t *bp, *nbp;
6526         mblk_t *savemp = NULL;
6527         mblk_t *savemptail = NULL;
6528         uint_t old_sd_flag;
6529         int flg;
```



```
6530         int more = 0;
6531         int error = 0;
6532         char first = 1;
6533         uint_t mark;            /* Contains MSG*MARK and _LASTMARK */
6534 #define _LASTMARK       0x8000  /* Distinct from MSG*MARK */
6535         unsigned char pri = 0;
6536         queue_t *q;
6537         int     pr = 0;                 /* Partial read successful */
6538         struct uio uios;
6539         struct uio *uiop = &uios;
6540         struct iovec iovs;
6541         unsigned char type;

6543         TRACE_1(TR_FAC_STREAMS_FR, TR_STRGETMSG_ENTER,
6544             "strgetmsg:%p", vp);

6546         ASSERT(vp->v_stream);
6547         stp = vp->v_stream;
6548         rvp->r_val1 = 0;

6550         mutex_enter(&stp->sd_lock);

6552         if ((error = i_straccess(stp, JCREAD)) != 0) {
6553                 mutex_exit(&stp->sd_lock);
6554                 return (error);
6555         }

6557         if (stp->sd_flag & (STRDERR|STPLEX)) {
6558                 error = strgeterr(stp, STRDERR|STPLEX, 0);
6559                 if (error != 0) {
6560                         mutex_exit(&stp->sd_lock);
6561                         return (error);
6562                 }
6563         }
6564         mutex_exit(&stp->sd_lock);

6566         switch (*flagsp) {
6567         case MSG_HIPRI:
6568                 if (*prip != 0)
6569                         return (EINVAL);
6570                 break;

6572         case MSG_ANY:
6573         case MSG_BAND:
6574                 break;

6576         default:
6577                 return (EINVAL);
6578         }
6579         /*
6580          * Setup uio and iov for data part
6581          */
6582         iovs.iov_base = mdata->buf;
6583         iovs.iov_len = mdata->maxlen;
6584         uios.uio_iov = &iovs;
6585         uios.uio_iovcnt = 1;
6586         uios.uio_loffset = 0;
6587         uios.uio_segflg = UIO_USERSPACE;
6588         uios.uio_fmode = 0;
6589         uios.uio_extflg = UIO_COPY_CACHED;
6590         uios.uio_resid = mdata->maxlen;
6591         uios.uio_offset = 0;

6593         q = _RD(stp->sd_wrq);
6594         mutex_enter(&stp->sd_lock);
6595         old_sd_flag = stp->sd_flag;
```

```
6596         mark = 0;
6597         for (;;) {
6598                 int done = 0;
6599                 mblk_t *q_first = q->q_first;

6601                 /*
6602                  * Get the next message of appropriate priority
6603                  * from the stream head.  If the caller is interested
6604                  * in band or hipri messages, then they should already
6605                  * be enqueued at the stream head.  On the other hand
6606                  * if the caller wants normal (band 0) messages, they
6607                  * might be deferred in a synchronous stream and they
6608                  * will need to be pulled up.
6609                  *
6610                  * After we have dequeued a message, we might find that
6611                  * it was a deferred M_SIG that was enqueued at the
6612                  * stream head.  It must now be posted as part of the
6613                  * read by calling strsignal_nolock().
6614                  *
6615                  * Also note that strrput does not enqueue an M_PCSIG,
6616                  * and there cannot be more than one hipri message,
6617                  * so there was no need to have the M_PCSIG case.
6618                  *
6619                  * At some time it might be nice to try and wrap the
6620                  * functionality of kstrgetmsg() and strgetmsg() into
6621                  * a common routine so to reduce the amount of replicated
6622                  * code (since they are extremely similar).
6623                  */
6624                 if (!(*flagsp & (MSG_HIPRI|MSG_BAND))) {
6625                         /* Asking for normal, band0 data */
6626                         bp = strget(stp, q, uiop, first, &error);
6627                         ASSERT(MUTEX_HELD(&stp->sd_lock));
6628                         if (bp != NULL) {
6629                                 if (DB_TYPE(bp) == M_SIG) {
6630                                         strsignal_nolock(stp, *bp->b_rptr,
6631                                             bp->b_band);
6632                                         freemsg(bp);
6633                                         continue;
6634                                 } else {
6635                                         break;
6636                                 }
6637                         }
6638                         if (error != 0)
6639                                 goto getmout;

6641                 /*
6642                  * We can't depend on the value of STRPRI here because
6643                  * the stream head may be in transit. Therefore, we
6644                  * must look at the type of the first message to
6645                  * determine if a high priority messages is waiting
6646                  */
6647                 } else if ((*flagsp & MSG_HIPRI) && q_first != NULL &&
6648                     DB_TYPE(q_first) >= QPCTL &&
6649                     (bp = getq_noenab(q, 0)) != NULL) {
6650                         /* Asked for HIPRI and got one */
6651                         ASSERT(DB_TYPE(bp) >= QPCTL);
6652                         break;
6653                 } else if ((*flagsp & MSG_BAND) && q_first != NULL &&
6654                     ((q_first->b_band >= *prip) || DB_TYPE(q_first) >= QPCTL) &&
6655                     (bp = getq_noenab(q, 0)) != NULL) {
6656                         /*
6657                          * Asked for at least band "prip" and got either at
6658                          * least that band or a hipri message.
6659                          */
6660                         ASSERT(bp->b_band >= *prip || DB_TYPE(bp) >= QPCTL);
6661                         if (DB_TYPE(bp) == M_SIG) {
```



```
6662                                 strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
6663                                 freemsg(bp);
6664                                 continue;
6665                         } else {
6666                                 break;
6667                         }
6668                 }

6670                 /* No data. Time to sleep? */
6671                 qbackenable(q, 0);

6673                 /*
6674                  * If STRHUP or STREOF, return 0 length control and data.
6675                  * If resid is 0, then a read(fd,buf,0) was done. Do not
6676                  * sleep to satisfy this request because by default we have
6677                  * zero bytes to return.
6678                  */
6679                 if ((stp->sd_flag & (STRHUP|STREOF)) || (mctl->maxlen == 0 &&
6680                     mdata->maxlen == 0)) {
6681                         mctl->len = mdata->len = 0;
6682                         *flagsp = 0;
6683                         mutex_exit(&stp->sd_lock);
6684                         return (0);
6685                 }
6686                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_WAIT,
6687                     "strgetmsg calls strwaitq:%p, %p",
6688                     vp, uiop);
6689                 if (((error = strwaitq(stp, GETWAIT, (ssize_t)0, fmode, -1,
6690                     &done)) != 0) || done) {
6691                         TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_DONE,
6692                             "strgetmsg error or done:%p, %p",
6693                             vp, uiop);
6694                         mutex_exit(&stp->sd_lock);
6695                         return (error);
6696                 }
6697                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_AWAKE,
6698                     "strgetmsg awakes:%p, %p", vp, uiop);
6699                 if ((error = i_straccess(stp, JCREAD)) != 0) {
6700                         mutex_exit(&stp->sd_lock);
6701                         return (error);
6702                 }
6703                 first = 0;
6704         }
6705         ASSERT(bp != NULL);
6706         /*
6707          * Extract any mark information. If the message is not completely
6708          * consumed this information will be put in the mblk
6709          * that is putback.
6710          * If MSGMARKNEXT is set and the message is completely consumed
6711          * the STRATMARK flag will be set below. Likewise, if
6712          * MSGNOTMARKNEXT is set and the message is
6713          * completely consumed STRNOTATMARK will be set.
6714          */
6715         mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
6716         ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
6717             (MSGMARKNEXT|MSGNOTMARKNEXT));
6718         if (mark != 0 && bp == stp->sd_mark) {
6719                 mark |= _LASTMARK;
6720                 stp->sd_mark = NULL;
6721         }
6722         /*
6723          * keep track of the original message type and priority
6724          */
6725         pri = bp->b_band;
6726         type = bp->b_datap->db_type;
6727         if (type == M_PASSFP) {
```

```
6728                if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
6729                        stp->sd_mark = bp;
6730                bp->b_flag |= mark & ~_LASTMARK;
6731                putback(stp, q, bp, pri);
6732                qbackenable(q, pri);
6733                mutex_exit(&stp->sd_lock);
6734                return (EBADMSG);
6735        }
6736        ASSERT(type != M_SIG);

6738        /*
6739         * Set this flag so strrput will not generate signals. Need to
6740         * make sure this flag is cleared before leaving this routine
6741         * else signals will stop being sent.
6742         */
6743        stp->sd_flag |= STRGETINPROG;
6744        mutex_exit(&stp->sd_lock);

6746        if (STREAM_NEEDSERVICE(stp))
6747                stream_runservice(stp);

6749        /*
6750         * Set HIPRI flag if message is priority.
6751         */
6752        if (type >= QPCTL)
6753                flg = MSG_HIPRI;
6754        else
6755                flg = MSG_BAND;

6757        /*
6758         * First process PROTO or PCPROTO blocks, if any.
6759         */
6760        if (mctl->maxlen >= 0 && type != M_DATA) {
6761                size_t  n, bcnt;
6762                char    *ubuf;

6764                bcnt = mctl->maxlen;
6765                ubuf = mctl->buf;
6766                while (bp != NULL && bp->b_datap->db_type != M_DATA) {
6767                        if ((n = MIN(bcnt, bp->b_wptr - bp->b_rptr)) != 0 &&
6768                            copyout(bp->b_rptr, ubuf, n)) {
6769                                error = EFAULT;
6770                                mutex_enter(&stp->sd_lock);
6771                                /*
6772                                 * clear stream head pri flag based on
6773                                 * first message type
6774                                 */
6775                                if (type >= QPCTL) {
6776                                        ASSERT(type == M_PCPROTO);
6777                                        stp->sd_flag &= ~STRPRI;
6778                                }
6779                                more = 0;
6780                                freemsg(bp);
6781                                goto getmout;
6782                        }
6783                        ubuf += n;
6784                        bp->b_rptr += n;
6785                        if (bp->b_rptr >= bp->b_wptr) {
6786                                nbp = bp;
6787                                bp = bp->b_cont;
6788                                freeb(nbp);
6789                        }
6790                        ASSERT(n <= bcnt);
6791                        bcnt -= n;
6792                        if (bcnt == 0)
6793                                break;
```



```
6794                }
6795                mctl->len = mctl->maxlen - bcnt;
6796        } else
6797                mctl->len = -1;

6799        if (bp && bp->b_datap->db_type != M_DATA) {
6800                /*
6801                 * More PROTO blocks in msg.
6802                 */
6803                more |= MORECTL;
6804                savemp = bp;
6805                while (bp && bp->b_datap->db_type != M_DATA) {
6806                        savemptail = bp;
6807                        bp = bp->b_cont;
6808                }
6809                savemptail->b_cont = NULL;
6810        }

6812        /*
6813         * Now process DATA blocks, if any.
6814         */
6815        if (mdata->maxlen >= 0 && bp) {
6816                /*
6817                 * struiocopyout will consume a potential zero-length
6818                 * M_DATA even if uio_resid is zero.
6819                 */
6820                size_t oldresid = uiop->uio_resid;

6822                bp = struiocopyout(bp, uiop, &error);
6823                if (error != 0) {
6824                        mutex_enter(&stp->sd_lock);
6825                        /*
6826                         * clear stream head hi pri flag based on
6827                         * first message
6828                         */
6829                        if (type >= QPCTL) {
6830                                ASSERT(type == M_PCPROTO);
6831                                stp->sd_flag &= ~STRPRI;
6832                        }
6833                        more = 0;
6834                        freemsg(savemp);
6835                        goto getmout;
6836                }
6837                /*
6838                 * (pr == 1) indicates a partial read.
6839                 */
6840                if (oldresid > uiop->uio_resid)
6841                        pr = 1;
6842                mdata->len = mdata->maxlen - uiop->uio_resid;
6843        } else
6844                mdata->len = -1;

6846        if (bp) {                       /* more data blocks in msg */
6847                more |= MOREDATA;
6848                if (savemp)
6849                        savemptail->b_cont = bp;
6850                else
6851                        savemp = bp;
6852        }

6854        mutex_enter(&stp->sd_lock);
6855        if (savemp) {
6856                if (pr && (savemp->b_datap->db_type == M_DATA) &&
6857                    msgnodata(savemp)) {
6858                        /*
6859                         * Avoid queuing a zero-length tail part of
```

```
6860                         * a message. pr=1 indicates that we read some of
6861                         * the message.
6862                         */
6863                        freemsg(savemp);
6864                        more &= ~MOREDATA;
6865                        /*
6866                         * clear stream head hi pri flag based on
6867                         * first message
6868                         */
6869                        if (type >= QPCTL) {
6870                                ASSERT(type == M_PCPROTO);
6871                                stp->sd_flag &= ~STRPRI;
6872                        }
6873                } else {
6874                        savemp->b_band = pri;
6875                        /*
6876                         * If the first message was HIPRI and the one we're
6877                         * putting back isn't, then clear STRPRI, otherwise
6878                         * set STRPRI again.  Note that we must set STRPRI
6879                         * again since the flush logic in strrput_nondata()
6880                         * may have cleared it while we had sd_lock dropped.
6881                         */
6882                        if (type >= QPCTL) {
6883                                ASSERT(type == M_PCPROTO);
6884                                if (queclass(savemp) < QPCTL)
6885                                        stp->sd_flag &= ~STRPRI;
6886                                else
6887                                        stp->sd_flag |= STRPRI;
6888                        } else if (queclass(savemp) >= QPCTL) {
6889                                /*
6890                                 * The first message was not a HIPRI message,
6891                                 * but the one we are about to putback is.
6892                                 * For simplicitly, we do not allow for HIPRI
6893                                 * messages to be embedded in the message
6894                                 * body, so just force it to same type as
6895                                 * first message.
6896                                 */
6897                                ASSERT(type == M_DATA || type == M_PROTO);
6898                                ASSERT(savemp->b_datap->db_type == M_PCPROTO);
6899                                savemp->b_datap->db_type = type;
6900                        }
6901                        if (mark != 0) {
6902                                savemp->b_flag |= mark & ~_LASTMARK;
6903                                if ((mark & _LASTMARK) &&
6904                                    (stp->sd_mark == NULL)) {
6905                                        /*
6906                                         * If another marked message arrived
6907                                         * while sd_lock was not held sd_mark
6908                                         * would be non-NULL.
6909                                         */
6910                                        stp->sd_mark = savemp;
6911                                }
6912                        }
6913                        putback(stp, q, savemp, pri);
6914                }
6915        } else {
6916                /*
6917                 * The complete message was consumed.
6918                 *
6919                 * If another M_PCPROTO arrived while sd_lock was not held
6920                 * it would have been discarded since STRPRI was still set.
6921                 *
6922                 * Move the MSG*MARKNEXT information
6923                 * to the stream head just in case
6924                 * the read queue becomes empty.
6925                 * clear stream head hi pri flag based on
```



```
6926                 * first message
6927                 *
6928                 * If the stream head was at the mark
6929                 * (STRATMARK) before we dropped sd_lock above
6930                 * and some data was consumed then we have
6931                 * moved past the mark thus STRATMARK is
6932                 * cleared. However, if a message arrived in
6933                 * strrput during the copyout above causing
6934                 * STRATMARK to be set we can not clear that
6935                 * flag.
6936                 */
6937                if (type >= QPCTL) {
6938                        ASSERT(type == M_PCPROTO);
6939                        stp->sd_flag &= ~STRPRI;
6940                }
6941                if (mark & (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
6942                        if (mark & MSGMARKNEXT) {
6943                                stp->sd_flag &= ~STRNOTATMARK;
6944                                stp->sd_flag |= STRATMARK;
6945                        } else if (mark & MSGNOTMARKNEXT) {
6946                                stp->sd_flag &= ~STRATMARK;
6947                                stp->sd_flag |= STRNOTATMARK;
6948                        } else {
6949                                stp->sd_flag &= ~(STRATMARK|STRNOTATMARK);
6950                        }
6951                } else if (pr && (old_sd_flag & STRATMARK)) {
6952                        stp->sd_flag &= ~STRATMARK;
6953                }
6954        }

6956        *flagsp = flg;
6957        *prip = pri;

6959        /*
6960         * Getmsg cleanup processing - if the state of the queue has changed
6961         * some signals may need to be sent and/or poll awakened.
6962         */
6963 getmout:
6964        qbackenable(q, pri);

6966        /*
6967         * We dropped the stream head lock above. Send all M_SIG messages
6968         * before processing stream head for SIGPOLL messages.
6969         */
6970        ASSERT(MUTEX_HELD(&stp->sd_lock));
6971        while ((bp = q->q_first) != NULL &&
6972            (bp->b_datap->db_type == M_SIG)) {
6973                /*
6974                 * sd_lock is held so the content of the read queue can not
6975                 * change.
6976                 */
6977                bp = getq(q);
6978                ASSERT(bp != NULL && bp->b_datap->db_type == M_SIG);

6980                strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
6981                mutex_exit(&stp->sd_lock);
6982                freemsg(bp);
6983                if (STREAM_NEEDSERVICE(stp))
6984                        stream_runservice(stp);
6985                mutex_enter(&stp->sd_lock);
6986        }

6988        /*
6989         * stream head cannot change while we make the determination
6990         * whether or not to send a signal. Drop the flag to allow strrput
6991         * to send firstmsgsigs again.
```

```
6992          */
6993         stp->sd_flag &= ~STRGETINPROG;

6995         /*
6996          * If the type of message at the front of the queue changed
6997          * due to the receive the appropriate signals and pollwakeup events
6998          * are generated. The type of changes are:
6999          *      Processed a hipri message, q_first is not hipri.
7000          *      Processed a band X message, and q_first is band Y.
7001          * The generated signals and pollwakeups are identical to what
7002          * strrput() generates should the message that is now on q_first
7003          * arrive to an empty read queue.
7004          *
7005          * Note: only strrput will send a signal for a hipri message.
7006          */
7007         if ((bp = q->q_first) != NULL && !(stp->sd_flag & STRPRI)) {
7008                 strsigset_t signals = 0;
7009                 strpollset_t pollwakeups = 0;

7011                 if (flg & MSG_HIPRI) {
7012                         /*
7013                          * Removed a hipri message. Regular data at
7014                          * the front of  the queue.
7015                          */
7016                         if (bp->b_band == 0) {
7017                                 signals = S_INPUT | S_RDNORM;
7018                                 pollwakeups = POLLIN | POLLRDNORM;
7019                         } else {
7020                                 signals = S_INPUT | S_RDBAND;
7021                                 pollwakeups = POLLIN | POLLRDBAND;
7022                         }
7023                 } else if (pri != bp->b_band) {
7024                         /*
7025                          * The band is different for the new q_first.
7026                          */
7027                         if (bp->b_band == 0) {
7028                                 signals = S_RDNORM;
7029                                 pollwakeups = POLLIN | POLLRDNORM;
7030                         } else {
7031                                 signals = S_RDBAND;
7032                                 pollwakeups = POLLIN | POLLRDBAND;
7033                         }
7034                 }

7036                 if (pollwakeups != 0) {
7037                         if (pollwakeups == (POLLIN | POLLRDNORM)) {
7038                                 if (!(stp->sd_rput_opt & SR_POLLIN))
7039                                         goto no_pollwake;
7040                                 stp->sd_rput_opt &= ~SR_POLLIN;
7041                         }
7042                         mutex_exit(&stp->sd_lock);
7043                         pollwakeup(&stp->sd_pollist, pollwakeups);
7044                         mutex_enter(&stp->sd_lock);
7045                 }
7046 no_pollwake:

7048                 if (stp->sd_sigflags & signals)
7049                         strsendsig(stp->sd_siglist, signals, bp->b_band, 0);
7050         }
7051         mutex_exit(&stp->sd_lock);

7053         rvp->r_val1 = more;
7054         return (error);
7055 #undef  _LASTMARK
7056 }
```



```
7058 /*
7059  * Get the next message from the read queue.  If the message is
7060  * priority, STRPRI will have been set by strrput().  This flag
7061  * should be reset only when the entire message at the front of the
7062  * queue as been consumed.
7063  *
7064  * If uiop is NULL all data is returned in mctlp.
7065  * Note that a NULL uiop implies that FNDELAY and FNONBLOCK are assumed
7066  * not enabled.
7067  * The timeout parameter is in milliseconds; -1 for infinity.
7068  * This routine handles the consolidation private flags:
7069  *      MSG_IGNERROR    Ignore any stream head error except STPLEX.
7070  *      MSG_DELAYERROR  Defer the error check until the queue is empty.
7071  *      MSG_HOLDSIG     Hold signals while waiting for data.
7072  *      MSG_IPEEK       Only peek at messages.
7073  *      MSG_DISCARDTAIL Discard the tail M_DATA part of the message
7074  *                      that doesn't fit.
7075  *      MSG_NOMARK      If the message is marked leave it on the queue.
7076  *
7077  * NOTE: strgetmsg and kstrgetmsg have much of the logic in common.
7078  */
7079 int
7080 kstrgetmsg(
7081         struct vnode *vp,
7082         mblk_t **mctlp,
7083         struct uio *uiop,
7084         unsigned char *prip,
7085         int *flagsp,
7086         clock_t timout,
7087         rval_t *rvp)
7088 {
7089         struct stdata *stp;
7090         mblk_t *bp, *nbp;
7091         mblk_t *savemp = NULL;
7092         mblk_t *savemptail = NULL;
7093         int flags;
7094         uint_t old_sd_flag;
7095         int flg;
7096         int more = 0;
7097         int error = 0;
7098         char first = 1;
7099         uint_t mark;            /* Contains MSG*MARK and _LASTMARK */
7100 #define _LASTMARK       0x8000  /* Distinct from MSG*MARK */
7101         unsigned char pri = 0;
7102         queue_t *q;
7103         int     pr = 0;                 /* Partial read successful */
7104         unsigned char type;

7106         TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_ENTER,
7107             "kstrgetmsg:%p", vp);

7109         ASSERT(vp->v_stream);
7110         stp = vp->v_stream;
7111         rvp->r_val1 = 0;

7113         mutex_enter(&stp->sd_lock);

7115         if ((error = i_straccess(stp, JCREAD)) != 0) {
7116                 mutex_exit(&stp->sd_lock);
7117                 return (error);
7118         }

7120         flags = *flagsp;
7121         if (stp->sd_flag & (STRDERR|STPLEX)) {
7122                 if ((stp->sd_flag & STPLEX) ||
7123                     (flags & (MSG_IGNERROR|MSG_DELAYERROR)) == 0) {
```

```
7124                        error = strgeterr(stp, STRDERR|STPLEX,
7125                            (flags & MSG_IPEEK));
7126                        if (error != 0) {
7127                                mutex_exit(&stp->sd_lock);
7128                                return (error);
7129                        }
7130                }
7131        }
7132        mutex_exit(&stp->sd_lock);

7134        switch (flags & (MSG_HIPRI|MSG_ANY|MSG_BAND)) {
7135        case MSG_HIPRI:
7136                if (*prip != 0)
7137                        return (EINVAL);
7138                break;

7140        case MSG_ANY:
7141        case MSG_BAND:
7142                break;

7144        default:
7145                return (EINVAL);
7146        }

7148 retry:
7149        q = _RD(stp->sd_wrq);
7150        mutex_enter(&stp->sd_lock);
7151        old_sd_flag = stp->sd_flag;
7152        mark = 0;
7153        for (;;) {
7154                int done = 0;
7155                int waitflag;
7156                int fmode;
7157                mblk_t *q_first = q->q_first;

7159                /*
7160                 * This section of the code operates just like the code
7161                 * in strgetmsg().  There is a comment there about what
7162                 * is going on here.
7163                 */
7164                if (!(flags & (MSG_HIPRI|MSG_BAND))) {
7165                        /* Asking for normal, band0 data */
7166                        bp = strget(stp, q, uiop, first, &error);
7167                        ASSERT(MUTEX_HELD(&stp->sd_lock));
7168                        if (bp != NULL) {
7169                                if (DB_TYPE(bp) == M_SIG) {
7170                                        strsignal_nolock(stp, *bp->b_rptr,
7171                                            bp->b_band);
7172                                        freemsg(bp);
7173                                        continue;
7174                                } else {
7175                                        break;
7176                                }
7177                        }
7178                        if (error != 0) {
7179                                goto getmout;
7180                        }
7181                /*
7182                 * We can't depend on the value of STRPRI here because
7183                 * the stream head may be in transit. Therefore, we
7184                 * must look at the type of the first message to
7185                 * determine if a high priority messages is waiting
7186                 */
7187                } else if ((flags & MSG_HIPRI) && q_first != NULL &&
7188                    DB_TYPE(q_first) >= QPCTL &&
7189                    (bp = getq_noenab(q, 0)) != NULL) {
```



```
7190                        ASSERT(DB_TYPE(bp) >= QPCTL);
7191                        break;
7192                } else if ((flags & MSG_BAND) && q_first != NULL &&
7193                    ((q_first->b_band >= *prip) || DB_TYPE(q_first) >= QPCTL) &&
7194                    (bp = getq_noenab(q, 0)) != NULL) {
7195                        /*
7196                         * Asked for at least band "prip" and got either at
7197                         * least that band or a hipri message.
7198                         */
7199                        ASSERT(bp->b_band >= *prip || DB_TYPE(bp) >= QPCTL);
7200                        if (DB_TYPE(bp) == M_SIG) {
7201                                strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
7202                                freemsg(bp);
7203                                continue;
7204                        } else {
7205                                break;
7206                        }
7207                }

7209                /* No data. Time to sleep? */
7210                qbackenable(q, 0);

7212                /*
7213                 * Delayed error notification?
7214                 */
7215                if ((stp->sd_flag & (STRDERR|STPLEX)) &&
7216                    (flags & (MSG_IGNERROR|MSG_DELAYERROR)) == MSG_DELAYERROR) {
7217                        error = strgeterr(stp, STRDERR|STPLEX,
7218                            (flags & MSG_IPEEK));
7219                        if (error != 0) {
7220                                mutex_exit(&stp->sd_lock);
7221                                return (error);
7222                        }
7223                }

7225                /*
7226                 * If STRHUP or STREOF, return 0 length control and data.
7227                 * If a read(fd,buf,0) has been done, do not sleep, just
7228                 * return.
7229                 *
7230                 * If mctlp == NULL and uiop == NULL, then the code will
7231                 * do the strwaitq. This is an understood way of saying
7232                 * sleep "polling" until a message is received.
7233                 */
7234                if ((stp->sd_flag & (STRHUP|STREOF)) ||
7235                    (uiop != NULL && uiop->uio_resid == 0)) {
7236                        if (mctlp != NULL)
7237                                *mctlp = NULL;
7238                        *flagsp = 0;
7239                        mutex_exit(&stp->sd_lock);
7240                        return (0);
7241                }

7243                waitflag = GETWAIT;
7244                if (flags &
7245                    (MSG_HOLDSIG|MSG_IGNERROR|MSG_IPEEK|MSG_DELAYERROR)) {
7246                        if (flags & MSG_HOLDSIG)
7247                                waitflag |= STR_NOSIG;
7248                        if (flags & MSG_IGNERROR)
7249                                waitflag |= STR_NOERROR;
7250                        if (flags & MSG_IPEEK)
7251                                waitflag |= STR_PEEK;
7252                        if (flags & MSG_DELAYERROR)
7253                                waitflag |= STR_DELAYERR;
7254                }
7255                if (uiop != NULL)
```

```
7256                        fmode = uiop->uio_fmode;
7257                else
7258                        fmode = 0;

7260                TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_WAIT,
7261                    "kstrgetmsg calls strwaitq:%p, %p",
7262                    vp, uiop);
7263                if (((error = strwaitq(stp, waitflag, (ssize_t)0,
7264                    fmode, timout, &done))) != 0 || done) {
7265                        TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_DONE,
7266                            "kstrgetmsg error or done:%p, %p",
7267                            vp, uiop);
7268                        mutex_exit(&stp->sd_lock);
7269                        return (error);
7270                }
7271                TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_AWAKE,
7272                    "kstrgetmsg awakes:%p, %p", vp, uiop);
7273                if ((error = i_straccess(stp, JCREAD)) != 0) {
7274                        mutex_exit(&stp->sd_lock);
7275                        return (error);
7276                }
7277                first = 0;
7278        }
7279        ASSERT(bp != NULL);
7280        /*
7281         * Extract any mark information. If the message is not completely
7282         * consumed this information will be put in the mblk
7283         * that is putback.
7284         * If MSGMARKNEXT is set and the message is completely consumed
7285         * the STRATMARK flag will be set below. Likewise, if
7286         * MSGNOTMARKNEXT is set and the message is
7287         * completely consumed STRNOTATMARK will be set.
7288         */
7289        mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
7290        ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
7291            (MSGMARKNEXT|MSGNOTMARKNEXT));
7292        pri = bp->b_band;
7293        if (mark != 0) {
7294                /*
7295                 * If the caller doesn't want the mark return.
7296                 * Used to implement MSG_WAITALL in sockets.
7297                 */
7298                if (flags & MSG_NOMARK) {
7299                        putback(stp, q, bp, pri);
7300                        qbackenable(q, pri);
7301                        mutex_exit(&stp->sd_lock);
7302                        return (EWOULDBLOCK);
7303                }
7304                if (bp == stp->sd_mark) {
7305                        mark |= _LASTMARK;
7306                        stp->sd_mark = NULL;
7307                }
7308        }

7310        /*
7311         * keep track of the first message type
7312         */
7313        type = bp->b_datap->db_type;

7315        if (bp->b_datap->db_type == M_PASSFP) {
7316                if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7317                        stp->sd_mark = bp;
7318                bp->b_flag |= mark & ~_LASTMARK;
7319                putback(stp, q, bp, pri);
7320                qbackenable(q, pri);
7321                mutex_exit(&stp->sd_lock);
```



```
7322                return (EBADMSG);
7323        }
7324        ASSERT(type != M_SIG);

7326        if (flags & MSG_IPEEK) {
7327                /*
7328                 * Clear any struioflag - we do the uiomove over again
7329                 * when peeking since it simplifies the code.
7330                 *
7331                 * Dup the message and put the original back on the queue.
7332                 * If dupmsg() fails, try again with copymsg() to see if
7333                 * there is indeed a shortage of memory.  dupmsg() may fail
7334                 * if db_ref in any of the messages reaches its limit.
7335                 */

7337                if ((nbp = dupmsg(bp)) == NULL && (nbp = copymsg(bp)) == NULL) {
7338                        /*
7339                         * Restore the state of the stream head since we
7340                         * need to drop sd_lock (strwaitbuf is sleeping).
7341                         */
7342                        size_t size = msgdsize(bp);

7344                        if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7345                                stp->sd_mark = bp;
7346                        bp->b_flag |= mark & ~_LASTMARK;
7347                        putback(stp, q, bp, pri);
7348                        mutex_exit(&stp->sd_lock);
7349                        error = strwaitbuf(size, BPRI_HI);
7350                        if (error) {
7351                                /*
7352                                 * There is no net change to the queue thus
7353                                 * no need to qbackenable.
7354                                 */
7355                                return (error);
7356                        }
7357                        goto retry;
7358                }

7360                if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7361                        stp->sd_mark = bp;
7362                bp->b_flag |= mark & ~_LASTMARK;
7363                putback(stp, q, bp, pri);
7364                bp = nbp;
7365        }

7367        /*
7368         * Set this flag so strrput will not generate signals. Need to
7369         * make sure this flag is cleared before leaving this routine
7370         * else signals will stop being sent.
7371         */
7372        stp->sd_flag |= STRGETINPROG;
7373        mutex_exit(&stp->sd_lock);

7375        if ((stp->sd_rputdatafunc != NULL) && (DB_TYPE(bp) == M_DATA)) {
7376                mblk_t *tmp, *prevmp;

7378                /*
7379                 * Put first non-data mblk back to stream head and
7380                 * cut the mblk chain so sd_rputdatafunc only sees
7381                 * M_DATA mblks. We can skip the first mblk since it
7382                 * is M_DATA according to the condition above.
7383                 */
7384                for (prevmp = bp, tmp = bp->b_cont; tmp != NULL;
7385                    prevmp = tmp, tmp = tmp->b_cont) {
7386                        if (DB_TYPE(tmp) != M_DATA) {
7387                                prevmp->b_cont = NULL;
```

```
7388                                mutex_enter(&stp->sd_lock);
7389                                putback(stp, q, tmp, tmp->b_band);
7390                                mutex_exit(&stp->sd_lock);
7391                                break;
7392                        }
7393                }

7395                bp = (stp->sd_rputdatafunc)(stp->sd_vnode, bp,
7396                    NULL, NULL, NULL, NULL);

7398                if (bp == NULL)
7399                        goto retry;
7400        }

7402        if (STREAM_NEEDSERVICE(stp))
7403                stream_runservice(stp);

7405        /*
7406         * Set HIPRI flag if message is priority.
7407         */
7408        if (type >= QPCTL)
7409                flg = MSG_HIPRI;
7410        else
7411                flg = MSG_BAND;

7413        /*
7414         * First process PROTO or PCPROTO blocks, if any.
7415         */
7416        if (mctlp != NULL && type != M_DATA) {
7417                mblk_t *nbp;

7419                *mctlp = bp;
7420                while (bp->b_cont && bp->b_cont->b_datap->db_type != M_DATA)
7421                        bp = bp->b_cont;
7422                nbp = bp->b_cont;
7423                bp->b_cont = NULL;
7424                bp = nbp;
7425        }

7427        if (bp && bp->b_datap->db_type != M_DATA) {
7428                /*
7429                 * More PROTO blocks in msg. Will only happen if mctlp is NULL.
7430                 */
7431                more |= MORECTL;
7432                savemp = bp;
7433                while (bp && bp->b_datap->db_type != M_DATA) {
7434                        savemptail = bp;
7435                        bp = bp->b_cont;
7436                }
7437                savemptail->b_cont = NULL;
7438        }

7440        /*
7441         * Now process DATA blocks, if any.
7442         */
7443        if (uiop == NULL) {
7444                /* Append data to tail of mctlp */

7446                if (mctlp != NULL) {
7447                        mblk_t **mpp = mctlp;

7449                        while (*mpp != NULL)
7450                                mpp = &((*mpp)->b_cont);
7451                        *mpp = bp;
7452                        bp = NULL;
7453                }
```



```
7454        } else if (uiop->uio_resid >= 0 && bp) {
7455                size_t oldresid = uiop->uio_resid;

7457                /*
7458                 * If a streams message is likely to consist
7459                 * of many small mblks, it is pulled up into
7460                 * one continuous chunk of memory.
7461                 * The size of the first mblk may be bogus because
7462                 * successive read() calls on the socket reduce
7463                 * the size of this mblk until it is exhausted
7464                 * and then the code walks on to the next. Thus
7465                 * the size of the mblk may not be the original size
7466                 * that was passed up, it's simply a remainder
7467                 * and hence can be very small without any
7468                 * implication that the packet is badly fragmented.
7469                 * So the size of the possible second mblk is
7470                 * used to spot a badly fragmented packet.
7471                 * see longer comment at top of page
7472                 * by mblk_pull_len declaration.
7473                 */

7475                if (bp->b_cont != NULL && MBLKL(bp->b_cont) < mblk_pull_len) {
7476                        (void) pullupmsg(bp, -1);
7477                }

7479                bp = struiocopyout(bp, uiop, &error);
7480                if (error != 0) {
7481                        if (mctlp != NULL) {
7482                                freemsg(*mctlp);
7483                                *mctlp = NULL;
7484                        } else
7485                                freemsg(savemp);
7486                        mutex_enter(&stp->sd_lock);
7487                        /*
7488                         * clear stream head hi pri flag based on
7489                         * first message
7490                         */
7491                        if (!(flags & MSG_IPEEK) && (type >= QPCTL)) {
7492                                ASSERT(type == M_PCPROTO);
7493                                stp->sd_flag &= ~STRPRI;
7494                        }
7495                        more = 0;
7496                        goto getmout;
7497                }
7498                /*
7499                 * (pr == 1) indicates a partial read.
7500                 */
7501                if (oldresid > uiop->uio_resid)
7502                        pr = 1;
7503        }

7505        if (bp) {                       /* more data blocks in msg */
7506                more |= MOREDATA;
7507                if (savemp)
7508                        savemptail->b_cont = bp;
7509                else
7510                        savemp = bp;
7511        }

7513        mutex_enter(&stp->sd_lock);
7514        if (savemp) {
7515                if (flags & (MSG_IPEEK|MSG_DISCARDTAIL)) {
7516                        /*
7517                         * When MSG_DISCARDTAIL is set or
7518                         * when peeking discard any tail. When peeking this
7519                         * is the tail of the dup that was copied out - the
```

```
7520                         * message has already been putback on the queue.
7521                         * Return MOREDATA to the caller even though the data
7522                         * is discarded. This is used by sockets (to
7523                         * set MSG_TRUNC).
7524                         */
7525                        freemsg(savemp);
7526                        if (!(flags & MSG_IPEEK) && (type >= QPCTL)) {
7527                                ASSERT(type == M_PCPROTO);
7528                                stp->sd_flag &= ~STRPRI;
7529                        }
7530                } else if (pr && (savemp->b_datap->db_type == M_DATA) &&
7531                    msgnodata(savemp)) {
7532                        /*
7533                         * Avoid queuing a zero-length tail part of
7534                         * a message. pr=1 indicates that we read some of
7535                         * the message.
7536                         */
7537                        freemsg(savemp);
7538                        more &= ~MOREDATA;
7539                        if (type >= QPCTL) {
7540                                ASSERT(type == M_PCPROTO);
7541                                stp->sd_flag &= ~STRPRI;
7542                        }
7543                } else {
7544                        savemp->b_band = pri;
7545                        /*
7546                         * If the first message was HIPRI and the one we're
7547                         * putting back isn't, then clear STRPRI, otherwise
7548                         * set STRPRI again.  Note that we must set STRPRI
7549                         * again since the flush logic in strrput_nondata()
7550                         * may have cleared it while we had sd_lock dropped.
7551                         */

7553                        if (type >= QPCTL) {
7554                                ASSERT(type == M_PCPROTO);
7555                                if (queclass(savemp) < QPCTL)
7556                                        stp->sd_flag &= ~STRPRI;
7557                                else
7558                                        stp->sd_flag |= STRPRI;
7559                        } else if (queclass(savemp) >= QPCTL) {
7560                                /*
7561                                 * The first message was not a HIPRI message,
7562                                 * but the one we are about to putback is.
7563                                 * For simplicitly, we do not allow for HIPRI
7564                                 * messages to be embedded in the message
7565                                 * body, so just force it to same type as
7566                                 * first message.
7567                                 */
7568                                ASSERT(type == M_DATA || type == M_PROTO);
7569                                ASSERT(savemp->b_datap->db_type == M_PCPROTO);
7570                                savemp->b_datap->db_type = type;
7571                        }
7572                        if (mark != 0) {
7573                                if ((mark & _LASTMARK) &&
7574                                    (stp->sd_mark == NULL)) {
7575                                        /*
7576                                         * If another marked message arrived
7577                                         * while sd_lock was not held sd_mark
7578                                         * would be non-NULL.
7579                                         */
7580                                        stp->sd_mark = savemp;
7581                                }
7582                                savemp->b_flag |= mark & ~_LASTMARK;
7583                        }
7584                        putback(stp, q, savemp, pri);
7585                }
```



```
7586        } else if (!(flags & MSG_IPEEK)) {
7587                /*
7588                 * The complete message was consumed.
7589                 *
7590                 * If another M_PCPROTO arrived while sd_lock was not held
7591                 * it would have been discarded since STRPRI was still set.
7592                 *
7593                 * Move the MSG*MARKNEXT information
7594                 * to the stream head just in case
7595                 * the read queue becomes empty.
7596                 * clear stream head hi pri flag based on
7597                 * first message
7598                 *
7599                 * If the stream head was at the mark
7600                 * (STRATMARK) before we dropped sd_lock above
7601                 * and some data was consumed then we have
7602                 * moved past the mark thus STRATMARK is
7603                 * cleared. However, if a message arrived in
7604                 * strrput during the copyout above causing
7605                 * STRATMARK to be set we can not clear that
7606                 * flag.
7607                 * XXX A "perimeter" would help by single-threading strrput,
7608                 * strread, strgetmsg and kstrgetmsg.
7609                 */
7610                if (type >= QPCTL) {
7611                        ASSERT(type == M_PCPROTO);
7612                        stp->sd_flag &= ~STRPRI;
7613                }
7614                if (mark & (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
7615                        if (mark & MSGMARKNEXT) {
7616                                stp->sd_flag &= ~STRNOTATMARK;
7617                                stp->sd_flag |= STRATMARK;
7618                        } else if (mark & MSGNOTMARKNEXT) {
7619                                stp->sd_flag &= ~STRATMARK;
7620                                stp->sd_flag |= STRNOTATMARK;
7621                        } else {
7622                                stp->sd_flag &= ~(STRATMARK|STRNOTATMARK);
7623                        }
7624                } else if (pr && (old_sd_flag & STRATMARK)) {
7625                        stp->sd_flag &= ~STRATMARK;
7626                }
7627        }

7629        *flagsp = flg;
7630        *prip = pri;

7632        /*
7633         * Getmsg cleanup processing - if the state of the queue has changed
7634         * some signals may need to be sent and/or poll awakened.
7635         */
7636 getmout:
7637        qbackenable(q, pri);

7639        /*
7640         * We dropped the stream head lock above. Send all M_SIG messages
7641         * before processing stream head for SIGPOLL messages.
7642         */
7643        ASSERT(MUTEX_HELD(&stp->sd_lock));
7644        while ((bp = q->q_first) != NULL &&
7645            (bp->b_datap->db_type == M_SIG)) {
7646                /*
7647                 * sd_lock is held so the content of the read queue can not
7648                 * change.
7649                 */
7650                bp = getq(q);
7651                ASSERT(bp != NULL && bp->b_datap->db_type == M_SIG);
```

```
7653                strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
7654                mutex_exit(&stp->sd_lock);
7655                freemsg(bp);
7656                if (STREAM_NEEDSERVICE(stp))
7657                        stream_runservice(stp);
7658                mutex_enter(&stp->sd_lock);
7659        }

7661        /*
7662         * stream head cannot change while we make the determination
7663         * whether or not to send a signal. Drop the flag to allow strrput
7664         * to send firstmsgsigs again.
7665         */
7666        stp->sd_flag &= ~STRGETINPROG;

7668        /*
7669         * If the type of message at the front of the queue changed
7670         * due to the receive the appropriate signals and pollwakeup events
7671         * are generated. The type of changes are:
7672         *      Processed a hipri message, q_first is not hipri.
7673         *      Processed a band X message, and q_first is band Y.
7674         * The generated signals and pollwakeups are identical to what
7675         * strrput() generates should the message that is now on q_first
7676         * arrive to an empty read queue.
7677         *
7678         * Note: only strrput will send a signal for a hipri message.
7679         */
7680        if ((bp = q->q_first) != NULL && !(stp->sd_flag & STRPRI)) {
7681                strsigset_t signals = 0;
7682                strpollset_t pollwakeups = 0;

7684                if (flg & MSG_HIPRI) {
7685                        /*
7686                         * Removed a hipri message. Regular data at
7687                         * the front of  the queue.
7688                         */
7689                        if (bp->b_band == 0) {
7690                                signals = S_INPUT | S_RDNORM;
7691                                pollwakeups = POLLIN | POLLRDNORM;
7692                        } else {
7693                                signals = S_INPUT | S_RDBAND;
7694                                pollwakeups = POLLIN | POLLRDBAND;
7695                        }
7696                } else if (pri != bp->b_band) {
7697                        /*
7698                         * The band is different for the new q_first.
7699                         */
7700                        if (bp->b_band == 0) {
7701                                signals = S_RDNORM;
7702                                pollwakeups = POLLIN | POLLRDNORM;
7703                        } else {
7704                                signals = S_RDBAND;
7705                                pollwakeups = POLLIN | POLLRDBAND;
7706                        }
7707                }

7709                if (pollwakeups != 0) {
7710                        if (pollwakeups == (POLLIN | POLLRDNORM)) {
7711                                if (!(stp->sd_rput_opt & SR_POLLIN))
7712                                        goto no_pollwake;
7713                                stp->sd_rput_opt &= ~SR_POLLIN;
7714                        }
7715                        mutex_exit(&stp->sd_lock);
7716                        pollwakeup(&stp->sd_pollist, pollwakeups);
7717                        mutex_enter(&stp->sd_lock);
```



```
7718                }
7719 no_pollwake:

7721                if (stp->sd_sigflags & signals)
7722                        strsendsig(stp->sd_siglist, signals, bp->b_band, 0);
7723        }
7724        mutex_exit(&stp->sd_lock);

7726        rvp->r_val1 = more;
7727        return (error);
7728 #undef  _LASTMARK
7729 }

7731 /*
7732  * Put a message downstream.
7733  *
7734  * NOTE: strputmsg and kstrputmsg have much of the logic in common.
7735  */
7736 int
7737 strputmsg(
7738        struct vnode *vp,
7739        struct strbuf *mctl,
7740        struct strbuf *mdata,
7741        unsigned char pri,
7742        int flag,
7743        int fmode)
7744 {
7745        struct stdata *stp;
7746        queue_t *wqp;
7747        mblk_t *mp;
7748        ssize_t msgsize;
7749        ssize_t rmin, rmax;
7750        int error;
7751        struct uio uios;
7752        struct uio *uiop = &uios;
7753        struct iovec iovs;
7754        int xpg4 = 0;

7756        ASSERT(vp->v_stream);
7757        stp = vp->v_stream;
7758        wqp = stp->sd_wrq;

7760        /*
7761         * If it is an XPG4 application, we need to send
7762         * SIGPIPE below
7763         */

7765        xpg4 = (flag & MSG_XPG4) ? 1 : 0;
7766        flag &= ~MSG_XPG4;

7768        if (AU_AUDITING())
7769                audit_strputmsg(vp, mctl, mdata, pri, flag, fmode);

7771        mutex_enter(&stp->sd_lock);

7773        if ((error = i_straccess(stp, JCWRITE)) != 0) {
7774                mutex_exit(&stp->sd_lock);
7775                return (error);
7776        }

7778        if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
7779                error = strwriteable(stp, B_FALSE, xpg4);
7780                if (error != 0) {
7781                        mutex_exit(&stp->sd_lock);
7782                        return (error);
7783                }
```

```
7784         }

7786         mutex_exit(&stp->sd_lock);

7788         /*
7789          * Check for legal flag value.
7790          */
7791         switch (flag) {
7792         case MSG_HIPRI:
7793                 if ((mctl->len < 0) || (pri != 0))
7794                         return (EINVAL);
7795                 break;
7796         case MSG_BAND:
7797                 break;

7799         default:
7800                 return (EINVAL);
7801         }

7803         TRACE_1(TR_FAC_STREAMS_FR, TR_STRPUTMSG_IN,
7804             "strputmsg in:stp %p", stp);

7806         /* get these values from those cached in the stream head */
7807         rmin = stp->sd_qn_minpsz;
7808         rmax = stp->sd_qn_maxpsz;

7810         /*
7811          * Make sure ctl and data sizes together fall within the
7812          * limits of the max and min receive packet sizes and do
7813          * not exceed system limit.
7814          */
7815         ASSERT((rmax >= 0) || (rmax == INFPSZ));
7816         if (rmax == 0) {
7817                 return (ERANGE);
7818         }
7819         /*
7820          * Use the MAXIMUM of sd_maxblk and q_maxpsz.
7821          * Needed to prevent partial failures in the strmakedata loop.
7822          */
7823         if (stp->sd_maxblk != INFPSZ && rmax != INFPSZ && rmax < stp->sd_maxblk)
7824                 rmax = stp->sd_maxblk;

7826         if ((msgsize = mdata->len) < 0) {
7827                 msgsize = 0;
7828                 rmin = 0;       /* no range check for NULL data part */
7829         }
7830         if ((msgsize < rmin) ||
7831             ((msgsize > rmax) && (rmax != INFPSZ)) ||
7832             (mctl->len > strctlsz)) {
7833                 return (ERANGE);
7834         }

7836         /*
7837          * Setup uio and iov for data part
7838          */
7839         iovs.iov_base = mdata->buf;
7840         iovs.iov_len = msgsize;
7841         uios.uio_iov = &iovs;
7842         uios.uio_iovcnt = 1;
7843         uios.uio_loffset = 0;
7844         uios.uio_segflg = UIO_USERSPACE;
7845         uios.uio_fmode = fmode;
7846         uios.uio_extflg = UIO_COPY_DEFAULT;
7847         uios.uio_resid = msgsize;
7848         uios.uio_offset = 0;
```



```
7850         /* Ignore flow control in strput for HIPRI */
7851         if (flag & MSG_HIPRI)
7852                 flag |= MSG_IGNFLOW;

7854         for (;;) {
7855                 int done = 0;

7857                 /*
7858                  * strput will always free the ctl mblk - even when strput
7859                  * fails.
7860                  */
7861                 if ((error = strmakectl(mctl, flag, fmode, &mp)) != 0) {
7862                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7863                             "strputmsg out:stp %p out %d error %d",
7864                             stp, 1, error);
7865                         return (error);
7866                 }
7867                 /*
7868                  * Verify that the whole message can be transferred by
7869                  * strput.
7870                  */
7871                 ASSERT(stp->sd_maxblk == INFPSZ ||
7872                     stp->sd_maxblk >= mdata->len);

7874                 msgsize = mdata->len;
7875                 error = strput(stp, mp, uiop, &msgsize, 0, pri, flag);
7876                 mdata->len = msgsize;

7878                 if (error == 0)
7879                         break;

7881                 if (error != EWOULDBLOCK)
7882                         goto out;

7884                 mutex_enter(&stp->sd_lock);
7885                 /*
7886                  * Check for a missed wakeup.
7887                  * Needed since strput did not hold sd_lock across
7888                  * the canputnext.
7889                  */
7890                 if (bcanputnext(wqp, pri)) {
7891                         /* Try again */
7892                         mutex_exit(&stp->sd_lock);
7893                         continue;
7894                 }
7895                 TRACE_2(TR_FAC_STREAMS_FR, TR_STRPUTMSG_WAIT,
7896                     "strputmsg wait:stp %p waits pri %d", stp, pri);
7897                 if (((error = strwaitq(stp, WRITEWAIT, (ssize_t)0, fmode, -1,
7898                     &done)) != 0) || done) {
7899                         mutex_exit(&stp->sd_lock);
7900                         TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7901                             "strputmsg out:q %p out %d error %d",
7902                             stp, 0, error);
7903                         return (error);
7904                 }
7905                 TRACE_1(TR_FAC_STREAMS_FR, TR_STRPUTMSG_WAKE,
7906                     "strputmsg wake:stp %p wakes", stp);
7907                 if ((error = i_straccess(stp, JCWRITE)) != 0) {
7908                         mutex_exit(&stp->sd_lock);
7909                         return (error);
7910                 }
7911                 mutex_exit(&stp->sd_lock);
7912         }
7913 out:
7914         /*
7915          * For historic reasons, applications expect EAGAIN
```

```
7916          * when data mblk could not be allocated. so change
7917          * ENOMEM back to EAGAIN
7918          */
7919         if (error == ENOMEM)
7920                 error = EAGAIN;
7921         TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7922             "strputmsg out:stp %p out %d error %d", stp, 2, error);
7923         return (error);
7924 }

7926 /*
7927  * Put a message downstream.
7928  * Can send only an M_PROTO/M_PCPROTO by passing in a NULL uiop.
7929  * The fmode flag (NDELAY, NONBLOCK) is the or of the flags in the uio
7930  * and the fmode parameter.
7931  *
7932  * This routine handles the consolidation private flags:
7933  *      MSG_IGNERROR    Ignore any stream head error except STPLEX.
7934  *      MSG_HOLDSIG     Hold signals while waiting for data.
7935  *      MSG_IGNFLOW     Don't check streams flow control.
7936  *
7937  * NOTE: strputmsg and kstrputmsg have much of the logic in common.
7938  */
7939 int
7940 kstrputmsg(
7941         struct vnode *vp,
7942         mblk_t *mctl,
7943         struct uio *uiop,
7944         ssize_t msgsize,
7945         unsigned char pri,
7946         int flag,
7947         int fmode)
7948 {
7949         struct stdata *stp;
7950         queue_t *wqp;
7951         ssize_t rmin, rmax;
7952         int error;

7954         ASSERT(vp->v_stream);
7955         stp = vp->v_stream;
7956         wqp = stp->sd_wrq;
7957         if (AU_AUDITING())
7958                 audit_strputmsg(vp, NULL, NULL, pri, flag, fmode);
7959         if (mctl == NULL)
7960                 return (EINVAL);

7962         mutex_enter(&stp->sd_lock);

7964         if ((error = i_straccess(stp, JCWRITE)) != 0) {
7965                 mutex_exit(&stp->sd_lock);
7966                 freemsg(mctl);
7967                 return (error);
7968         }

7970         if ((stp->sd_flag & STPLEX) || !(flag & MSG_IGNERROR)) {
7971                 if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
7972                         error = strwriteable(stp, B_FALSE, B_TRUE);
7973                         if (error != 0) {
7974                                 mutex_exit(&stp->sd_lock);
7975                                 freemsg(mctl);
7976                                 return (error);
7977                         }
7978                 }
7979         }

7981         mutex_exit(&stp->sd_lock);
```



```
7983         /*
7984          * Check for legal flag value.
7985          */
7986         switch (flag & (MSG_HIPRI|MSG_BAND|MSG_ANY)) {
7987         case MSG_HIPRI:
7988                 if (pri != 0) {
7989                         freemsg(mctl);
7990                         return (EINVAL);
7991                 }
7992                 break;
7993         case MSG_BAND:
7994                 break;
7995         default:
7996                 freemsg(mctl);
7997                 return (EINVAL);
7998         }

8000         TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_IN,
8001             "kstrputmsg in:stp %p", stp);

8003         /* get these values from those cached in the stream head */
8004         rmin = stp->sd_qn_minpsz;
8005         rmax = stp->sd_qn_maxpsz;

8007         /*
8008          * Make sure ctl and data sizes together fall within the
8009          * limits of the max and min receive packet sizes and do
8010          * not exceed system limit.
8011          */
8012         ASSERT((rmax >= 0) || (rmax == INFPSZ));
8013         if (rmax == 0) {
8014                 freemsg(mctl);
8015                 return (ERANGE);
8016         }
8017         /*
8018          * Use the MAXIMUM of sd_maxblk and q_maxpsz.
8019          * Needed to prevent partial failures in the strmakedata loop.
8020          */
8021         if (stp->sd_maxblk != INFPSZ && rmax != INFPSZ && rmax < stp->sd_maxblk)
8022                 rmax = stp->sd_maxblk;

8024         if (uiop == NULL) {
8025                 msgsize = -1;
8026                 rmin = -1;      /* no range check for NULL data part */
8027         } else {
8028                 /* Use uio flags as well as the fmode parameter flags */
8029                 fmode |= uiop->uio_fmode;

8031                 if ((msgsize < rmin) ||
8032                     ((msgsize > rmax) && (rmax != INFPSZ))) {
8033                         freemsg(mctl);
8034                         return (ERANGE);
8035                 }
8036         }

8038         /* Ignore flow control in strput for HIPRI */
8039         if (flag & MSG_HIPRI)
8040                 flag |= MSG_IGNFLOW;

8042         for (;;) {
8043                 int done = 0;
8044                 int waitflag;
8045                 mblk_t *mp;

8047                 /*
```

```
8048                  * strput will always free the ctl mblk - even when strput
8049                  * fails. If MSG_IGNFLOW is set then any error returned
8050                  * will cause us to break the loop, so we don't need a copy
8051                  * of the message. If MSG_IGNFLOW is not set, then we can
8052                  * get hit by flow control and be forced to try again. In
8053                  * this case we need to have a copy of the message. We
8054                  * do this using copymsg since the message may get modified
8055                  * by something below us.
8056                  *
8057                  * We've observed that many TPI providers do not check db_ref
8058                  * on the control messages but blindly reuse them for the
8059                  * T_OK_ACK/T_ERROR_ACK. Thus using copymsg is more
8060                  * friendly to such providers than using dupmsg. Also, note
8061                  * that sockfs uses MSG_IGNFLOW for all TPI control messages.
8062                  * Only data messages are subject to flow control, hence
8063                  * subject to this copymsg.
8064                  */
8065                 if (flag & MSG_IGNFLOW) {
8066                         mp = mctl;
8067                         mctl = NULL;
8068                 } else {
8069                         do {
8070                                 /*
8071                                  * If a message has a free pointer, the message
8072                                  * must be dupmsg to maintain this pointer.
8073                                  * Code using this facility must be sure
8074                                  * that modules below will not change the
8075                                  * contents of the dblk without checking db_ref
8076                                  * first. If db_ref is > 1, then the module
8077                                  * needs to do a copymsg first. Otherwise,
8078                                  * the contents of the dblk may become
8079                                  * inconsistent because the freesmg/freeb below
8080                                  * may end up calling atomic_add_32_nv.
8081                                  * The atomic_add_32_nv in freeb (accessing
8082                                  * all of db_ref, db_type, db_flags, and
8083                                  * db_struioflag) does not prevent other threads
8084                                  * from concurrently trying to modify e.g.
8085                                  * db_type.
8086                                  */
8087                                 if (mctl->b_datap->db_frtnp != NULL)
8088                                         mp = dupmsg(mctl);
8089                                 else
8090                                         mp = copymsg(mctl);
8092                                 if (mp != NULL)
8093                                         break;
8095                                 error = strwaitbuf(msgdsize(mctl), BPRI_MED);
8096                                 if (error) {
8097                                         freemsg(mctl);
8098                                         return (error);
8099                                 }
8100                         } while (mp == NULL);
8101                 }
8102                 /*
8103                  * Verify that all of msgsize can be transferred by
8104                  * strput.
8105                  */
8106                 ASSERT(stp->sd_maxblk == INFPSZ || stp->sd_maxblk >= msgsize);
8107                 error = strput(stp, mp, uiop, &msgsize, 0, pri, flag);
8108                 if (error == 0)
8109                         break;
8111                 if (error != EWOULDBLOCK)
8112                         goto out;
```



```
8114                 /*
8115                  * IF MSG_IGNFLOW is set we should have broken out of loop
8116                  * above.
8117                  */
8118                 ASSERT(!(flag & MSG_IGNFLOW));
8119                 mutex_enter(&stp->sd_lock);
8120                 /*
8121                  * Check for a missed wakeup.
8122                  * Needed since strput did not hold sd_lock across
8123                  * the canputnext.
8124                  */
8125                 if (bcanputnext(wqp, pri)) {
8126                         /* Try again */
8127                         mutex_exit(&stp->sd_lock);
8128                         continue;
8129                 }
8130                 TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_WAIT,
8131                     "kstrputmsg wait:stp %p waits pri %d", stp, pri);
8133                 waitflag = WRITEWAIT;
8134                 if (flag & (MSG_HOLDSIG|MSG_IGNERROR)) {
8135                         if (flag & MSG_HOLDSIG)
8136                                 waitflag |= STR_NOSIG;
8137                         if (flag & MSG_IGNERROR)
8138                                 waitflag |= STR_NOERROR;
8139                 }
8140                 if (((error = strwaitq(stp, waitflag,
8141                     (ssize_t)0, fmode, -1, &done)) != 0) || done) {
8142                         mutex_exit(&stp->sd_lock);
8143                         TRACE_3(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_OUT,
8144                             "kstrputmsg out:stp %p out %d error %d",
8145                             stp, 0, error);
8146                         freemsg(mctl);
8147                         return (error);
8148                 }
8149                 TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_WAKE,
8150                     "kstrputmsg wake:stp %p wakes", stp);
8151                 if ((error = i_straccess(stp, JCWRITE)) != 0) {
8152                         mutex_exit(&stp->sd_lock);
8153                         freemsg(mctl);
8154                         return (error);
8155                 }
8156                 mutex_exit(&stp->sd_lock);
8157         }
8158 out:
8159         freemsg(mctl);
8160         /*
8161          * For historic reasons, applications expect EAGAIN
8162          * when data mblk could not be allocated. so change
8163          * ENOMEM back to EAGAIN
8164          */
8165         if (error == ENOMEM)
8166                 error = EAGAIN;
8167         TRACE_3(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_OUT,
8168             "kstrputmsg out:stp %p out %d error %d", stp, 2, error);
8169         return (error);
8170 }
8172 /*
8173  * Determines whether the necessary conditions are set on a stream
8174  * for it to be readable, writeable, or have exceptions.
8175  *
8176  * strpoll handles the consolidation private events:
8177  *      POLLNOERR       Do not return POLLERR even if there are stream
8178  *                      head errors.
8179  *                      Used by sockfs.
```

```
 *      POLLRDDATA      Do not return POLLIN unless at least one message on
 *                      the queue contains one or more M_DATA mblks. Thus
 *                      when this flag is set a queue with only
 *                      M_PROTO/M_PCPROTO mblks does not return POLLIN.
 *                      Used by sockfs to ignore T_EXDATA_IND messages.
 *
 * Note: POLLRDDATA assumes that synch streams only return messages with
 * an M_DATA attached (i.e. not messages consisting of only
 * an M_PROTO/M_PCPROTO part).
 */
int
strpoll(
        struct stdata *stp,
        short events_arg,
        int anyyet,
        short *reventsp,
        struct pollhead **phpp)
{
        int events = (ushort_t)events_arg;
        int retevents = 0;
        mblk_t *mp;
        qband_t *qbp;
        long sd_flags = stp->sd_flag;
        int headlocked = 0;

        /*
         * For performance, a single 'if' tests for most possible edge
         * conditions in one shot
         */
        if (sd_flags & (STPLEX | STRDERR | STWRERR)) {
                if (sd_flags & STPLEX) {
                        *reventsp = POLLNVAL;
                        return (EINVAL);
                }
                if (((events & (POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI)) &&
                    (sd_flags & STRDERR)) ||
                    ((events & (POLLOUT | POLLWRNORM | POLLWRBAND)) &&
                    (sd_flags & STWRERR))) {
                        if (!(events & POLLNOERR)) {
                                *reventsp = POLLERR;
                                return (0);
                        }
                }
        }
        if (sd_flags & STRHUP) {
                retevents |= POLLHUP;
        } else if (events & (POLLWRNORM | POLLWRBAND)) {
                queue_t *tq;
                queue_t *qp = stp->sd_wrq;

                claimstr(qp);
                /* Find next module forward that has a service procedure */
                tq = qp->q_next->q_nfsrv;
                ASSERT(tq != NULL);

                polllock(&stp->sd_pollist, QLOCK(tq));
                if (events & POLLWRNORM) {
                        queue_t *sqp;

                        if (tq->q_flag & QFULL)
                                /* ensure backq svc procedure runs */
                                tq->q_flag |= QWANTW;
                        else if ((sqp = stp->sd_struiowrq) != NULL) {
                                /* Check sync stream barrier write q */
                                mutex_exit(QLOCK(tq));
                                polllock(&stp->sd_pollist, QLOCK(sqp));
```



```
                                if (sqp->q_flag & QFULL)
                                        /* ensure pollwakeup() is done */
                                        sqp->q_flag |= QWANTWSYNC;
                                else
                                        retevents |= POLLOUT;
                                /* More write events to process ??? */
                                if (! (events & POLLWRBAND)) {
                                        mutex_exit(QLOCK(sqp));
                                        releasestr(qp);
                                        goto chkrd;
                                }
                                mutex_exit(QLOCK(sqp));
                                polllock(&stp->sd_pollist, QLOCK(tq));
                        } else
                                retevents |= POLLOUT;
                }
                if (events & POLLWRBAND) {
                        qbp = tq->q_bandp;
                        if (qbp) {
                                while (qbp) {
                                        if (qbp->qb_flag & QB_FULL)
                                                qbp->qb_flag |= QB_WANTW;
                                        else
                                                retevents |= POLLWRBAND;
                                        qbp = qbp->qb_next;
                                }
                        } else {
                                retevents |= POLLWRBAND;
                        }
                }
                mutex_exit(QLOCK(tq));
                releasestr(qp);
        }
chkrd:
        if (sd_flags & STRPRI) {
                retevents |= (events & POLLPRI);
        } else if (events & (POLLRDNORM | POLLRDBAND | POLLIN)) {
                queue_t *qp = _RD(stp->sd_wrq);
                int normevents = (events & (POLLIN | POLLRDNORM));

                /*
                 * Note: Need to do polllock() here since ps_lock may be
                 * held. See bug 4191544.
                 */
                polllock(&stp->sd_pollist, &stp->sd_lock);
                headlocked = 1;
                mp = qp->q_first;
                while (mp) {
                        /*
                         * For POLLRDDATA we scan b_cont and b_next until we
                         * find an M_DATA.
                         */
                        if ((events & POLLRDDATA) &&
                            mp->b_datap->db_type != M_DATA) {
                                mblk_t *nmp = mp->b_cont;

                                while (nmp != NULL &&
                                    nmp->b_datap->db_type != M_DATA)
                                        nmp = nmp->b_cont;
                                if (nmp == NULL) {
                                        mp = mp->b_next;
                                        continue;
                                }
                        }
                        if (mp->b_band == 0)
                                retevents |= normevents;
```

```
8312                        else
8313                                retevents |= (events & (POLLIN | POLLRDBAND));
8314                        break;
8315                }
8316                if (! (retevents & normevents) &&
8317                    (stp->sd_wakeq & RSLEEP)) {
8318                        /*
8319                         * Sync stream barrier read queue has data.
8320                         */
8321                        retevents |= normevents;
8322                }
8323                /* Treat eof as normal data */
8324                if (sd_flags & STREOF)
8325                        retevents |= normevents;
8326        }

8328        *reventsp = (short)retevents;
8329        if (retevents) {
8330                if (headlocked)
8331                        mutex_exit(&stp->sd_lock);
8332                return (0);
8333        }

8335        /*
8336         * If poll() has not found any events yet, set up event cell
8337         * to wake up the poll if a requested event occurs on this
8338         * stream.  Check for collisions with outstanding poll requests.
8339         */
8340        if (!anyyet) {
8341                *phpp = &stp->sd_pollist;
8342                if (headlocked == 0) {
8343                        polllock(&stp->sd_pollist, &stp->sd_lock);
8344                        headlocked = 1;
8345                }
8346                stp->sd_rput_opt |= SR_POLLIN;
8347        }
8348        if (headlocked)
8349                mutex_exit(&stp->sd_lock);
8350        return (0);
8351 }

8353 /*
8354  * The purpose of putback() is to assure sleeping polls/reads
8355  * are awakened when there are no new messages arriving at the,
8356  * stream head, and a message is placed back on the read queue.
8357  *
8358  * sd_lock must be held when messages are placed back on stream
8359  * head.  (getq() holds sd_lock when it removes messages from
8360  * the queue)
8361  */

8363 static void
8364 putback(struct stdata *stp, queue_t *q, mblk_t *bp, int band)
8365 {
8366        mblk_t  *qfirst;
8367        ASSERT(MUTEX_HELD(&stp->sd_lock));

8369        /*
8370         * As a result of lock-step ordering around q_lock and sd_lock,
8371         * it's possible for function calls like putnext() and
8372         * canputnext() to get an inaccurate picture of how much
8373         * data is really being processed at the stream head.
8374         * We only consolidate with existing messages on the queue
8375         * if the length of the message we want to put back is smaller
8376         * than the queue hiwater mark.
8377         */
```



```
8378        if ((stp->sd_rput_opt & SR_CONSOL_DATA) &&
8379            (DB_TYPE(bp) == M_DATA) && ((qfirst = q->q_first) != NULL) &&
8380            (DB_TYPE(qfirst) == M_DATA) &&
8381            ((qfirst->b_flag & (MSGMARK|MSGDELIM)) == 0) &&
8382            ((bp->b_flag & (MSGMARK|MSGDELIM|MSGMARKNEXT)) == 0) &&
8383            (mp_cont_len(bp, NULL) < q->q_hiwat)) {
8384                /*
8385                 * We use the same logic as defined in strrput()
8386                 * but in reverse as we are putting back onto the
8387                 * queue and want to retain byte ordering.
8388                 * Consolidate M_DATA messages with M_DATA ONLY.
8389                 * strrput() allows the consolidation of M_DATA onto
8390                 * M_PROTO | M_PCPROTO but not the other way round.
8391                 *
8392                 * The consolidation does not take place if the message
8393                 * we are returning to the queue is marked with either
8394                 * of the marks or the delim flag or if q_first
8395                 * is marked with MSGMARK. The MSGMARK check is needed to
8396                 * handle the odd semantics of MSGMARK where essentially
8397                 * the whole message is to be treated as marked.
8398                 * Carry any MSGMARKNEXT and MSGNOTMARKNEXT from q_first
8399                 * to the front of the b_cont chain.
8400                 */
8401                rmvq_noenab(q, qfirst);

8403                /*
8404                 * The first message in the b_cont list
8405                 * tracks MSGMARKNEXT and MSGNOTMARKNEXT.
8406                 * We need to handle the case where we
8407                 * are appending:
8408                 *
8409                 * 1) a MSGMARKNEXT to a MSGNOTMARKNEXT.
8410                 * 2) a MSGMARKNEXT to a plain message.
8411                 * 3) a MSGNOTMARKNEXT to a plain message
8412                 * 4) a MSGNOTMARKNEXT to a MSGNOTMARKNEXT
8413                 *    message.
8414                 *
8415                 * Thus we never append a MSGMARKNEXT or
8416                 * MSGNOTMARKNEXT to a MSGMARKNEXT message.
8417                 */
8418                if (qfirst->b_flag & MSGMARKNEXT) {
8419                        bp->b_flag |= MSGMARKNEXT;
8420                        bp->b_flag &= ~MSGNOTMARKNEXT;
8421                        qfirst->b_flag &= ~MSGMARKNEXT;
8422                } else if (qfirst->b_flag & MSGNOTMARKNEXT) {
8423                        bp->b_flag |= MSGNOTMARKNEXT;
8424                        qfirst->b_flag &= ~MSGNOTMARKNEXT;
8425                }

8427                linkb(bp, qfirst);
8428        }
8429        (void) putbq(q, bp);

8431        /*
8432         * A message may have come in when the sd_lock was dropped in the
8433         * calling routine. If this is the case and STR*ATMARK info was
8434         * received, need to move that from the stream head to the q_last
8435         * so that SIOCATMARK can return the proper value.
8436         */
8437        if (stp->sd_flag & (STRATMARK | STRNOTATMARK)) {
8438                unsigned short *flagp = &q->q_last->b_flag;
8439                uint_t b_flag = (uint_t)*flagp;

8441                if (stp->sd_flag & STRATMARK) {
8442                        b_flag &= ~MSGNOTMARKNEXT;
8443                        b_flag |= MSGMARKNEXT;
```

```
8444                        stp->sd_flag &= ~STRATMARK;
8445                } else {
8446                        b_flag &= ~MSGMARKNEXT;
8447                        b_flag |= MSGNOTMARKNEXT;
8448                        stp->sd_flag &= ~STRNOTATMARK;
8449                }
8450                *flagp = (unsigned short) b_flag;
8451        }

8453 #ifdef  DEBUG
8454        /*
8455         * Make sure that the flags are not messed up.
8456         */
8457        {
8458                mblk_t *mp;
8459                mp = q->q_last;
8460                while (mp != NULL) {
8461                        ASSERT((mp->b_flag & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
8462                            (MSGMARKNEXT|MSGNOTMARKNEXT));
8463                        mp = mp->b_cont;
8464                }
8465        }
8466 #endif
8467        if (q->q_first == bp) {
8468                short pollevents;

8470                if (stp->sd_flag & RSLEEP) {
8471                        stp->sd_flag &= ~RSLEEP;
8472                        cv_broadcast(&q->q_wait);
8473                }
8474                if (stp->sd_flag & STRPRI) {
8475                        pollevents = POLLPRI;
8476                } else {
8477                        if (band == 0) {
8478                                if (!(stp->sd_rput_opt & SR_POLLIN))
8479                                        return;
8480                                stp->sd_rput_opt &= ~SR_POLLIN;
8481                                pollevents = POLLIN | POLLRDNORM;
8482                        } else {
8483                                pollevents = POLLIN | POLLRDBAND;
8484                        }
8485                }
8486                mutex_exit(&stp->sd_lock);
8487                pollwakeup(&stp->sd_pollist, pollevents);
8488                mutex_enter(&stp->sd_lock);
8489        }
8490 }

8492 /*
8493  * Return the held vnode attached to the stream head of a
8494  * given queue
8495  * It is the responsibility of the calling routine to ensure
8496  * that the queue does not go away (e.g. pop).
8497  */
8498 vnode_t *
8499 strq2vp(queue_t *qp)
8500 {
8501        vnode_t *vp;
8502        vp = STREAM(qp)->sd_vnode;
8503        ASSERT(vp != NULL);
8504        VN_HOLD(vp);
8505        return (vp);
8506 }

8508 /*
8509  * return the stream head write queue for the given vp
```



```
8510  * It is the responsibility of the calling routine to ensure
8511  * that the stream or vnode do not close.
8512  */
8513 queue_t *
8514 strvp2wq(vnode_t *vp)
8515 {
8516        ASSERT(vp->v_stream != NULL);
8517        return (vp->v_stream->sd_wrq);
8518 }

8520 /*
8521  * pollwakeup stream head
8522  * It is the responsibility of the calling routine to ensure
8523  * that the stream or vnode do not close.
8524  */
8525 void
8526 strpollwakeup(vnode_t *vp, short event)
8527 {
8528        ASSERT(vp->v_stream);
8529        pollwakeup(&vp->v_stream->sd_pollist, event);
8530 }

8532 /*
8533  * Mate the stream heads of two vnodes together. If the two vnodes are the
8534  * same, we just make the write-side point at the read-side -- otherwise,
8535  * we do a full mate.  Only works on vnodes associated with streams that are
8536  * still being built and thus have only a stream head.
8537  */
8538 void
8539 strmate(vnode_t *vp1, vnode_t *vp2)
8540 {
8541        queue_t *wrq1 = strvp2wq(vp1);
8542        queue_t *wrq2 = strvp2wq(vp2);

8544        /*
8545         * Verify that there are no modules on the stream yet.  We also
8546         * rely on the stream head always having a service procedure to
8547         * avoid tweaking q_nfsrv.
8548         */
8549        ASSERT(wrq1->q_next == NULL && wrq2->q_next == NULL);
8550        ASSERT(wrq1->q_qinfo->qi_srvp != NULL);
8551        ASSERT(wrq2->q_qinfo->qi_srvp != NULL);

8553        /*
8554         * If the queues are the same, just twist; otherwise do a full mate.
8555         */
8556        if (wrq1 == wrq2) {
8557                wrq1->q_next = _RD(wrq1);
8558        } else {
8559                wrq1->q_next = _RD(wrq2);
8560                wrq2->q_next = _RD(wrq1);
8561                STREAM(wrq1)->sd_mate = STREAM(wrq2);
8562                STREAM(wrq1)->sd_flag |= STRMATE;
8563                STREAM(wrq2)->sd_mate = STREAM(wrq1);
8564                STREAM(wrq2)->sd_flag |= STRMATE;
8565        }
8566 }

8568 /*
8569  * XXX will go away when console is correctly fixed.
8570  * Clean up the console PIDS, from previous I_SETSIG,
8571  * called only for cnopen which never calls strclean().
8572  */
8573 void
8574 str_cn_clean(struct vnode *vp)
8575 {
```

```
8576         strsig_t *ssp, *pssp, *tssp;
8577         struct stdata *stp;
8578         struct pid  *pidp;
8579         int update = 0;

8581         ASSERT(vp->v_stream);
8582         stp = vp->v_stream;
8583         pssp = NULL;
8584         mutex_enter(&stp->sd_lock);
8585         ssp = stp->sd_siglist;
8586         while (ssp) {
8587                 mutex_enter(&pidlock);
8588                 pidp = ssp->ss_pidp;
8589                 /*
8590                  * Get rid of PID if the proc is gone.
8591                  */
8592                 if (pidp->pid_prinactive) {
8593                         tssp = ssp->ss_next;
8594                         if (pssp)
8595                                 pssp->ss_next = tssp;
8596                         else
8597                                 stp->sd_siglist = tssp;
8598                         ASSERT(pidp->pid_ref <= 1);
8599                         PID_RELE(ssp->ss_pidp);
8600                         mutex_exit(&pidlock);
8601                         kmem_free(ssp, sizeof (strsig_t));
8602                         update = 1;
8603                         ssp = tssp;
8604                         continue;
8605                 } else
8606                         mutex_exit(&pidlock);
8607                 pssp = ssp;
8608                 ssp = ssp->ss_next;
8609         }
8610         if (update) {
8611                 stp->sd_sigflags = 0;
8612                 for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
8613                         stp->sd_sigflags |= ssp->ss_events;
8614         }
8615         mutex_exit(&stp->sd_lock);
8616 }

8618 /*
8619  * Return B_TRUE if there is data in the message, B_FALSE otherwise.
8620  */
8621 static boolean_t
8622 msghasdata(mblk_t *bp)
8623 {
8624         for (; bp; bp = bp->b_cont)
8625                 if (bp->b_datap->db_type == M_DATA) {
8626                         ASSERT(bp->b_wptr >= bp->b_rptr);
8627                         if (bp->b_wptr > bp->b_rptr)
8628                                 return (B_TRUE);
8629                 }
8630         return (B_FALSE);
8631 }
```

```
**********************************************************
  232312 Sun Aug  9 12:48:04 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;
3318         sh_insert_pid(stp, curproc);
3319 #endif /* ! codereview */

3321         return (stp);
3322 }

3324 /*
3325  * Free a stream head.
3326  */
3327 void
3328 shfree(stdata_t *stp)
3329 {
3330         pid_node_t *pn;

3332 #endif /* ! codereview */
3333         ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));

3335         stp->sd_wrq = NULL;

3337         mutex_enter(&stp->sd_qlock);
3338         while (stp->sd_svcflags & STRS_SCHEDULED) {
3339                 STRSTAT(strwaits);
3340                 cv_wait(&stp->sd_qcv, &stp->sd_qlock);
```



```
3341         }
3342         mutex_exit(&stp->sd_qlock);

3344         if (stp->sd_ciputctrl != NULL) {
3345                 ASSERT(stp->sd_nciputctrl == n_ciputctrl - 1);
3346                 SUMCHECK_CIPUTCTRL_COUNTS(stp->sd_ciputctrl,
3347                     stp->sd_nciputctrl, 0);
3348                 ASSERT(ciputctrl_cache != NULL);
3349                 kmem_cache_free(ciputctrl_cache, stp->sd_ciputctrl);
3350                 stp->sd_ciputctrl = NULL;
3351                 stp->sd_nciputctrl = 0;
3352         }
3353         ASSERT(stp->sd_qhead == NULL);
3354         ASSERT(stp->sd_qtail == NULL);
3355         ASSERT(stp->sd_nqueues == 0);

3357         mutex_enter(&stp->sd_pid_list_lock);
3358         while ((pn = list_head(&stp->sd_pid_list)) != NULL) {
3359                 list_remove(&stp->sd_pid_list, pn);
3360                 kmem_free(pn, sizeof (*pn));
3361         }
3362         mutex_exit(&stp->sd_pid_list_lock);

3364 #endif /* ! codereview */
3365         kmem_cache_free(stream_head_cache, stp);
3366 }

3368 void
3369 sh_insert_pid(struct stdata *stp, proc_t *p)
3370 {
3371         pid_node_t      *pn;

3373         mutex_enter(&stp->sd_pid_list_lock);
3374         pn = list_head(&stp->sd_pid_list);
3375         while (pn != NULL && pn->pn_pid != p->p_pidp->pid_id) {
3376                 pn = list_next(&stp->sd_pid_list, pn);
3377         }

3379         if (pn != NULL) {
3380                 pn->pn_count++;
3381         } else {
3382                 pn = kmem_zalloc(sizeof (*pn), KM_SLEEP);
3383                 list_link_init(&pn->pn_ref_link);
3384                 pn->pn_pid = p->p_pidp->pid_id;
3385                 pn->pn_count = 1;
3386                 list_insert_tail(&stp->sd_pid_list, pn);
3387         }
3388         mutex_exit(&stp->sd_pid_list_lock);
3389 }
3390 void
3391 sh_remove_pid(struct stdata *stp, proc_t *p)
3392 {
3393         pid_node_t *pn;

3395         mutex_enter(&stp->sd_pid_list_lock);
3396         pn = list_head(&stp->sd_pid_list);
3397         while (pn != NULL && pn->pn_pid != p->p_pidp->pid_id) {
3398                 pn = list_next(&stp->sd_pid_list, pn);
3399         }

3401         if (pn != NULL) {
3402                 if (pn->pn_count > 1)
3403                         pn->pn_count--;
3404                 else {
3405                         list_remove(&stp->sd_pid_list, pn);
3406                         kmem_free(pn, sizeof (*pn));
```

```
3407                }
3408        }
3409        mutex_exit(&stp->sd_pid_list_lock);
3410 }

3412 conn_pid_node_list_hdr_t *
3413 sh_get_pid_list(struct stdata *stp)
3414 {
3415        int sz, n = 0;
3416        pid_node_t                      *pn;
3417        conn_pid_node_t                 *cpn;
3418        conn_pid_node_list_hdr_t        *cph;

3420        mutex_enter(&stp->sd_pid_list_lock);

3422        n = list_size(&stp->sd_pid_list);
3423        sz = sizeof (conn_pid_node_list_hdr_t);
3424        sz += (n > 1)?((n - 1) * sizeof (conn_pid_node_t)):0;

3426        cph = kmem_zalloc(sz, KM_SLEEP);
3427        cph->cph_magic = CONN_PID_NODE_LIST_HDR_MAGIC;
3428        cph->cph_contents = CONN_PID_NODE_LIST_HDR_XTI;
3429        cph->cph_pn_cnt = n;
3430        cph->cph_tot_size = sz;
3431        cph->cph_flags = 0;
3432        cph->cph_optional1 = 0;
3433        cph->cph_optional2 = 0;

3435        if (cph->cph_pn_cnt > 0) {
3436                cpn = cph->cph_cpns;
3437                pn = list_head(&stp->sd_pid_list);
3438                while (pn != NULL) {
3439                        PIDNODE2CONNPIDNODE(pn, cpn);
3440                        pn = list_next(&stp->sd_pid_list, pn);
3441                        cpn++;
3442                }
3443        }

3445        mutex_exit(&stp->sd_pid_list_lock);
3446        return (cph);
3447 #endif /* ! codereview */
3448 }

3450 /*
3451  * Allocate a pair of queues and a syncq for the pair
3452  */
3453 queue_t *
3454 allocq(void)
3455 {
3456        queinfo_t *qip;
3457        queue_t *qp, *wqp;
3458        syncq_t *sq;

3460        qip = kmem_cache_alloc(queue_cache, KM_SLEEP);

3462        qp = &qip->qu_rqueue;
3463        wqp = &qip->qu_wqueue;
3464        sq = &qip->qu_syncq;

3466        qp->q_last      = NULL;
3467        qp->q_next      = NULL;
3468        qp->q_ptr       = NULL;
3469        qp->q_flag      = QUSE | QREADR;
3470        qp->q_bandp     = NULL;
3471        qp->q_stream    = NULL;
3472        qp->q_syncq     = sq;
```



```
3473        qp->q_nband     = 0;
3474        qp->q_nfsrv     = NULL;
3475        qp->q_draining  = 0;
3476        qp->q_syncqmsgs = 0;
3477        qp->q_spri      = 0;
3478        qp->q_qtstamp   = 0;
3479        qp->q_sqtstamp  = 0;
3480        qp->q_fp        = NULL;

3482        wqp->q_last     = NULL;
3483        wqp->q_next     = NULL;
3484        wqp->q_ptr      = NULL;
3485        wqp->q_flag     = QUSE;
3486        wqp->q_bandp    = NULL;
3487        wqp->q_stream   = NULL;
3488        wqp->q_syncq    = sq;
3489        wqp->q_nband    = 0;
3490        wqp->q_nfsrv    = NULL;
3491        wqp->q_draining = 0;
3492        wqp->q_syncqmsgs = 0;
3493        wqp->q_qtstamp  = 0;
3494        wqp->q_sqtstamp = 0;
3495        wqp->q_spri     = 0;

3497        sq->sq_count    = 0;
3498        sq->sq_rmqcount = 0;
3499        sq->sq_flags    = 0;
3500        sq->sq_type     = 0;
3501        sq->sq_callbflags = 0;
3502        sq->sq_cancelid = 0;
3503        sq->sq_ciputctrl = NULL;
3504        sq->sq_nciputctrl = 0;
3505        sq->sq_needexcl = 0;
3506        sq->sq_svcflags = 0;

3508        return (qp);
3509 }

3511 /*
3512  * Free a pair of queues and the "attached" syncq.
3513  * Discard any messages left on the syncq(s), remove the syncq(s) from the
3514  * outer perimeter, and free the syncq(s) if they are not the "attached" syncq.
3515  */
3516 void
3517 freeq(queue_t *qp)
3518 {
3519        qband_t *qbp, *nqbp;
3520        syncq_t *sq, *outer;
3521        queue_t *wqp = _WR(qp);

3523        ASSERT(qp->q_flag & QREADR);

3525        /*
3526         * If a previously dispatched taskq job is scheduled to run
3527         * sync_service() or a service routine is scheduled for the
3528         * queues about to be freed, wait here until all service is
3529         * done on the queue and all associated queues and syncqs.
3530         */
3531        wait_svc(qp);

3533        (void) flush_syncq(qp->q_syncq, qp);
3534        (void) flush_syncq(wqp->q_syncq, wqp);
3535        ASSERT(qp->q_syncqmsgs == 0 && wqp->q_syncqmsgs == 0);

3537        /*
3538         * Flush the queues before q_next is set to NULL This is needed
```

```
3539          * in order to backenable any downstream queue before we go away.
3540          * Note: we are already removed from the stream so that the
3541          * backenabling will not cause any messages to be delivered to our
3542          * put procedures.
3543          */
3544         flushq(qp, FLUSHALL);
3545         flushq(wqp, FLUSHALL);

3547         /* Tidy up - removeq only does a half-remove from stream */
3548         qp->q_next = wqp->q_next = NULL;
3549         ASSERT(!(qp->q_flag & QENAB));
3550         ASSERT(!(wqp->q_flag & QENAB));

3552         outer = qp->q_syncq->sq_outer;
3553         if (outer != NULL) {
3554                 outer_remove(outer, qp->q_syncq);
3555                 if (wqp->q_syncq != qp->q_syncq)
3556                         outer_remove(outer, wqp->q_syncq);
3557         }
3558         /*
3559          * Free any syncqs that are outside what allocq returned.
3560          */
3561         if (qp->q_syncq != SQ(qp) && !(qp->q_flag & QPERMOD))
3562                 free_syncq(qp->q_syncq);
3563         if (qp->q_syncq != wqp->q_syncq && wqp->q_syncq != SQ(qp))
3564                 free_syncq(wqp->q_syncq);

3566         ASSERT((qp->q_sqflags & (Q_SQQUEUED | Q_SQDRAINING)) == 0);
3567         ASSERT((wqp->q_sqflags & (Q_SQQUEUED | Q_SQDRAINING)) == 0);
3568         ASSERT(MUTEX_NOT_HELD(QLOCK(qp)));
3569         ASSERT(MUTEX_NOT_HELD(QLOCK(wqp)));
3570         sq = SQ(qp);
3571         ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
3572         ASSERT(sq->sq_head == NULL && sq->sq_tail == NULL);
3573         ASSERT(sq->sq_outer == NULL);
3574         ASSERT(sq->sq_onext == NULL && sq->sq_oprev == NULL);
3575         ASSERT(sq->sq_callbpend == NULL);
3576         ASSERT(sq->sq_needexcl == 0);

3578         if (sq->sq_ciputctrl != NULL) {
3579                 ASSERT(sq->sq_nciputctrl == n_ciputctrl - 1);
3580                 SUMCHECK_CIPUTCTRL_COUNTS(sq->sq_ciputctrl,
3581                     sq->sq_nciputctrl, 0);
3582                 ASSERT(ciputctrl_cache != NULL);
3583                 kmem_cache_free(ciputctrl_cache, sq->sq_ciputctrl);
3584                 sq->sq_ciputctrl = NULL;
3585                 sq->sq_nciputctrl = 0;
3586         }

3588         ASSERT(qp->q_first == NULL && wqp->q_first == NULL);
3589         ASSERT(qp->q_count == 0 && wqp->q_count == 0);
3590         ASSERT(qp->q_mblkcnt == 0 && wqp->q_mblkcnt == 0);

3592         qp->q_flag &= ~QUSE;
3593         wqp->q_flag &= ~QUSE;

3595         /* NOTE: Uncomment the assert below once bugid 1159635 is fixed. */
3596         /* ASSERT((qp->q_flag & QWANTW) == 0 && (wqp->q_flag & QWANTW) == 0); */

3598         qbp = qp->q_bandp;
3599         while (qbp) {
3600                 nqbp = qbp->qb_next;
3601                 freeband(qbp);
3602                 qbp = nqbp;
3603         }
3604         qbp = wqp->q_bandp;
```



```
3605         while (qbp) {
3606                 nqbp = qbp->qb_next;
3607                 freeband(qbp);
3608                 qbp = nqbp;
3609         }
3610         kmem_cache_free(queue_cache, qp);
3611 }

3613 /*
3614  * Allocate a qband structure.
3615  */
3616 qband_t *
3617 allocband(void)
3618 {
3619         qband_t *qbp;

3621         qbp = kmem_cache_alloc(qband_cache, KM_NOSLEEP);
3622         if (qbp == NULL)
3623                 return (NULL);

3625         qbp->qb_next    = NULL;
3626         qbp->qb_count   = 0;
3627         qbp->qb_mblkcnt = 0;
3628         qbp->qb_first   = NULL;
3629         qbp->qb_last    = NULL;
3630         qbp->qb_flag    = 0;

3632         return (qbp);
3633 }

3635 /*
3636  * Free a qband structure.
3637  */
3638 void
3639 freeband(qband_t *qbp)
3640 {
3641         kmem_cache_free(qband_cache, qbp);
3642 }

3644 /*
3645  * Just like putnextctl(9F), except that allocb_wait() is used.
3646  *
3647  * Consolidation Private, and of course only callable from the stream head or
3648  * routines that may block.
3649  */
3650 int
3651 putnextctl_wait(queue_t *q, int type)
3652 {
3653         mblk_t *bp;
3654         int error;

3656         if ((datamsg(type) && (type != M_DELAY)) ||
3657             (bp = allocb_wait(0, BPRI_HI, 0, &error)) == NULL)
3658                 return (0);

3660         bp->b_datap->db_type = (unsigned char)type;
3661         putnext(q, bp);
3662         return (1);
3663 }

3665 /*
3666  * Run any possible bufcalls.
3667  */
3668 void
3669 runbufcalls(void)
3670 {
```

```
3671         strbufcall_t *bcp;

3673         mutex_enter(&bcall_monitor);
3674         mutex_enter(&strbcall_lock);

3676         if (strbcalls.bc_head) {
3677                 size_t count;
3678                 int nevent;

3680                 /*
3681                  * count how many events are on the list
3682                  * now so we can check to avoid looping
3683                  * in low memory situations
3684                  */
3685                 nevent = 0;
3686                 for (bcp = strbcalls.bc_head; bcp; bcp = bcp->bc_next)
3687                         nevent++;

3689                 /*
3690                  * get estimate of available memory from kmem_avail().
3691                  * awake all bufcall functions waiting for
3692                  * memory whose request could be satisfied
3693                  * by 'count' memory and let 'em fight for it.
3694                  */
3695                 count = kmem_avail();
3696                 while ((bcp = strbcalls.bc_head) != NULL && nevent) {
3697                         STRSTAT(bufcalls);
3698                         --nevent;
3699                         if (bcp->bc_size <= count) {
3700                                 bcp->bc_executor = curthread;
3701                                 mutex_exit(&strbcall_lock);
3702                                 (*bcp->bc_func)(bcp->bc_arg);
3703                                 mutex_enter(&strbcall_lock);
3704                                 bcp->bc_executor = NULL;
3705                                 cv_broadcast(&bcall_cv);
3706                                 strbcalls.bc_head = bcp->bc_next;
3707                                 kmem_free(bcp, sizeof (strbufcall_t));
3708                         } else {
3709                                 /*
3710                                  * too big, try again later - note
3711                                  * that nevent was decremented above
3712                                  * so we won't retry this one on this
3713                                  * iteration of the loop
3714                                  */
3715                                 if (bcp->bc_next != NULL) {
3716                                         strbcalls.bc_head = bcp->bc_next;
3717                                         bcp->bc_next = NULL;
3718                                         strbcalls.bc_tail->bc_next = bcp;
3719                                         strbcalls.bc_tail = bcp;
3720                                 }
3721                         }
3722                 }
3723                 if (strbcalls.bc_head == NULL)
3724                         strbcalls.bc_tail = NULL;
3725         }

3727         mutex_exit(&strbcall_lock);
3728         mutex_exit(&bcall_monitor);
3729 }


3732 /*
3733  * Actually run queue's service routine.
3734  */
3735 static void
3736 runservice(queue_t *q)
```



```
3737 {
3738         qband_t *qbp;

3740         ASSERT(q->q_qinfo->qi_srvp);
3741 again:
3742         entersq(q->q_syncq, SQ_SVC);
3743         TRACE_1(TR_FAC_STREAMS_FR, TR_QRUNSERVICE_START,
3744             "runservice starts:%p", q);

3746         if (!(q->q_flag & QWCLOSE))
3747                 (*q->q_qinfo->qi_srvp)(q);

3749         TRACE_1(TR_FAC_STREAMS_FR, TR_QRUNSERVICE_END,
3750             "runservice ends:(%p)", q);

3752         leavesq(q->q_syncq, SQ_SVC);

3754         mutex_enter(QLOCK(q));
3755         if (q->q_flag & QENAB) {
3756                 q->q_flag &= ~QENAB;
3757                 mutex_exit(QLOCK(q));
3758                 goto again;
3759         }
3760         q->q_flag &= ~QINSERVICE;
3761         q->q_flag &= ~QBACK;
3762         for (qbp = q->q_bandp; qbp; qbp = qbp->qb_next)
3763                 qbp->qb_flag &= ~QB_BACK;
3764         /*
3765          * Wakeup thread waiting for the service procedure
3766          * to be run (strclose and qdetach).
3767          */
3768         cv_broadcast(&q->q_wait);

3770         mutex_exit(QLOCK(q));
3771 }

3773 /*
3774  * Background processing of bufcalls.
3775  */
3776 void
3777 streams_bufcall_service(void)
3778 {
3779         callb_cpr_t     cprinfo;

3781         CALLB_CPR_INIT(&cprinfo, &strbcall_lock, callb_generic_cpr,
3782             "streams_bufcall_service");

3784         mutex_enter(&strbcall_lock);

3786         for (;;) {
3787                 if (strbcalls.bc_head != NULL && kmem_avail() > 0) {
3788                         mutex_exit(&strbcall_lock);
3789                         runbufcalls();
3790                         mutex_enter(&strbcall_lock);
3791                 }
3792                 if (strbcalls.bc_head != NULL) {
3793                         STRSTAT(bcwaits);
3794                         /* Wait for memory to become available */
3795                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
3796                         (void) cv_reltimedwait(&memavail_cv, &strbcall_lock,
3797                             SEC_TO_TICK(60), TR_CLOCK_TICK);
3798                         CALLB_CPR_SAFE_END(&cprinfo, &strbcall_lock);
3799                 }

3801                 /* Wait for new work to arrive */
3802                 if (strbcalls.bc_head == NULL) {
```

```
3803                        CALLB_CPR_SAFE_BEGIN(&cprinfo);
3804                        cv_wait(&strbcall_cv, &strbcall_lock);
3805                        CALLB_CPR_SAFE_END(&cprinfo, &strbcall_lock);
3806                }
3807        }
3808 }

3810 /*
3811  * Background processing of streams background tasks which failed
3812  * taskq_dispatch.
3813  */
3814 static void
3815 streams_qbkgrnd_service(void)
3816 {
3817        callb_cpr_t cprinfo;
3818        queue_t *q;

3820        CALLB_CPR_INIT(&cprinfo, &service_queue, callb_generic_cpr,
3821            "streams_bkgrnd_service");

3823        mutex_enter(&service_queue);

3825        for (;;) {
3826                /*
3827                 * Wait for work to arrive.
3828                 */
3829                while ((freebs_list == NULL) && (qhead == NULL)) {
3830                        CALLB_CPR_SAFE_BEGIN(&cprinfo);
3831                        cv_wait(&services_to_run, &service_queue);
3832                        CALLB_CPR_SAFE_END(&cprinfo, &service_queue);
3833                }
3834                /*
3835                 * Handle all pending freebs requests to free memory.
3836                 */
3837                while (freebs_list != NULL) {
3838                        mblk_t *mp = freebs_list;
3839                        freebs_list = mp->b_next;
3840                        mutex_exit(&service_queue);
3841                        mblk_free(mp);
3842                        mutex_enter(&service_queue);
3843                }
3844                /*
3845                 * Run pending queues.
3846                 */
3847                while (qhead != NULL) {
3848                        DQ(q, qhead, qtail, q_link);
3849                        ASSERT(q != NULL);
3850                        mutex_exit(&service_queue);
3851                        queue_service(q);
3852                        mutex_enter(&service_queue);
3853                }
3854                ASSERT(qhead == NULL && qtail == NULL);
3855        }
3856 }

3858 /*
3859  * Background processing of streams background tasks which failed
3860  * taskq_dispatch.
3861  */
3862 static void
3863 streams_sqbkgrnd_service(void)
3864 {
3865        callb_cpr_t cprinfo;
3866        syncq_t *sq;

3868        CALLB_CPR_INIT(&cprinfo, &service_queue, callb_generic_cpr,
```



```
3869            "streams_sqbkgrnd_service");

3871        mutex_enter(&service_queue);

3873        for (;;) {
3874                /*
3875                 * Wait for work to arrive.
3876                 */
3877                while (sqhead == NULL) {
3878                        CALLB_CPR_SAFE_BEGIN(&cprinfo);
3879                        cv_wait(&syncqs_to_run, &service_queue);
3880                        CALLB_CPR_SAFE_END(&cprinfo, &service_queue);
3881                }

3883                /*
3884                 * Run pending syncqs.
3885                 */
3886                while (sqhead != NULL) {
3887                        DQ(sq, sqhead, sqtail, sq_next);
3888                        ASSERT(sq != NULL);
3889                        ASSERT(sq->sq_svcflags & SQ_BGTHREAD);
3890                        mutex_exit(&service_queue);
3891                        syncq_service(sq);
3892                        mutex_enter(&service_queue);
3893                }
3894        }
3895 }

3897 /*
3898  * Disable the syncq and wait for background syncq processing to complete.
3899  * If the syncq is placed on the sqhead/sqtail queue, try to remove it from the
3900  * list.
3901  */
3902 void
3903 wait_sq_svc(syncq_t *sq)
3904 {
3905        mutex_enter(SQLOCK(sq));
3906        sq->sq_svcflags |= SQ_DISABLED;
3907        if (sq->sq_svcflags & SQ_BGTHREAD) {
3908                syncq_t *sq_chase;
3909                syncq_t *sq_curr;
3910                int removed;

3912                ASSERT(sq->sq_servcount == 1);
3913                mutex_enter(&service_queue);
3914                RMQ(sq, sqhead, sqtail, sq_next, sq_chase, sq_curr, removed);
3915                mutex_exit(&service_queue);
3916                if (removed) {
3917                        sq->sq_svcflags &= ~SQ_BGTHREAD;
3918                        sq->sq_servcount = 0;
3919                        STRSTAT(sqremoved);
3920                        goto done;
3921                }
3922        }
3923        while (sq->sq_servcount != 0) {
3924                sq->sq_flags |= SQ_WANTWAKEUP;
3925                cv_wait(&sq->sq_wait, SQLOCK(sq));
3926        }
3927 done:
3928        mutex_exit(SQLOCK(sq));
3929 }

3931 /*
3932  * Put a syncq on the list of syncq's to be serviced by the sqthread.
3933  * Add the argument to the end of the sqhead list and set the flag
3934  * indicating this syncq has been enabled.  If it has already been
```

```
3935  * enabled, don't do anything.
3936  * This routine assumes that SQLOCK is held.
3937  * NOTE that the lock order is to have the SQLOCK first,
3938  * so if the service_syncq lock is held, we need to release it
3939  * before acquiring the SQLOCK (mostly relevant for the background
3940  * thread, and this seems to be common among the STREAMS global locks).
3941  * Note that the sq_svcflags are protected by the SQLOCK.
3942  */
3943 void
3944 sqenable(syncq_t *sq)
3945 {
3946         /*
3947          * This is probably not important except for where I believe it
3948          * is being called.  At that point, it should be held (and it
3949          * is a pain to release it just for this routine, so don't do
3950          * it).
3951          */
3952         ASSERT(MUTEX_HELD(SQLOCK(sq)));

3954         IMPLY(sq->sq_servcount == 0, sq->sq_next == NULL);
3955         IMPLY(sq->sq_next != NULL, sq->sq_svcflags & SQ_BGTHREAD);

3957         /*
3958          * Do not put on list if background thread is scheduled or
3959          * syncq is disabled.
3960          */
3961         if (sq->sq_svcflags & (SQ_DISABLED | SQ_BGTHREAD))
3962                 return;

3964         /*
3965          * Check whether we should enable sq at all.
3966          * Non PERMOD syncqs may be drained by at most one thread.
3967          * PERMOD syncqs may be drained by several threads but we limit the
3968          * total amount to the lesser of
3969          *      Number of queues on the squeue and
3970          *      Number of CPUs.
3971          */
3972         if (sq->sq_servcount != 0) {
3973                 if (((sq->sq_type & SQ_PERMOD) == 0) ||
3974                     (sq->sq_servcount >= MIN(sq->sq_nqueues, ncpus_online))) {
3975                         STRSTAT(sqtoomany);
3976                         return;
3977                 }
3978         }

3980         sq->sq_tstamp = ddi_get_lbolt();
3981         STRSTAT(sqenables);

3983         /* Attempt a taskq dispatch */
3984         sq->sq_servid = (void *)taskq_dispatch(streams_taskq,
3985             (task_func_t *)syncq_service, sq, TQ_NOSLEEP | TQ_NOQUEUE);
3986         if (sq->sq_servid != NULL) {
3987                 sq->sq_servcount++;
3988                 return;
3989         }

3991         /*
3992          * This taskq dispatch failed, but a previous one may have succeeded.
3993          * Don't try to schedule on the background thread whilst there is
3994          * outstanding taskq processing.
3995          */
3996         if (sq->sq_servcount != 0)
3997                 return;

3999         /*
4000          * System is low on resources and can't perform a non-sleeping
```



```
4001          * dispatch. Schedule the syncq for a background thread and mark the
4002          * syncq to avoid any further taskq dispatch attempts.
4003          */
4004         mutex_enter(&service_queue);
4005         STRSTAT(taskqfails);
4006         ENQUEUE(sq, sqhead, sqtail, sq_next);
4007         sq->sq_svcflags |= SQ_BGTHREAD;
4008         sq->sq_servcount = 1;
4009         cv_signal(&syncqs_to_run);
4010         mutex_exit(&service_queue);
4011 }

4013 /*
4014  * Note: fifo_close() depends on the mblk_t on the queue being freed
4015  * asynchronously. The asynchronous freeing of messages breaks the
4016  * recursive call chain of fifo_close() while there are I_SENDFD type of
4017  * messages referring to other file pointers on the queue. Then when
4018  * closing pipes it can avoid stack overflow in case of daisy-chained
4019  * pipes, and also avoid deadlock in case of fifonode_t pairs (which
4020  * share the same fifolock_t).
4021  *
4022  * No need to kpreempt_disable to access cpu_seqid.  If we migrate and
4023  * the esb queue does not match the new CPU, that is OK.
4024  */
4025 void
4026 freebs_enqueue(mblk_t *mp, dblk_t *dbp)
4027 {
4028         int qindex = CPU->cpu_seqid >> esbq_log2_cpus_per_q;
4029         esb_queue_t *eqp;

4031         ASSERT(dbp->db_mblk == mp);
4032         ASSERT(qindex < esbq_nelem);

4034         eqp = system_esbq_array;
4035         if (eqp != NULL) {
4036                 eqp += qindex;
4037         } else {
4038                 mutex_enter(&esbq_lock);
4039                 if (kmem_ready && system_esbq_array == NULL)
4040                         system_esbq_array = (esb_queue_t *)kmem_zalloc(
4041                             esbq_nelem * sizeof (esb_queue_t), KM_NOSLEEP);
4042                 mutex_exit(&esbq_lock);
4043                 eqp = system_esbq_array;
4044                 if (eqp != NULL)
4045                         eqp += qindex;
4046                 else
4047                         eqp = &system_esbq;
4048         }

4050         /*
4051          * Check data sanity. The dblock should have non-empty free function.
4052          * It is better to panic here then later when the dblock is freed
4053          * asynchronously when the context is lost.
4054          */
4055         if (dbp->db_frtnp->free_func == NULL) {
4056                 panic("freebs_enqueue: dblock %p has a NULL free callback",
4057                     (void *)dbp);
4058         }

4060         mutex_enter(&eqp->eq_lock);
4061         /* queue the new mblk on the esballoc queue */
4062         if (eqp->eq_head == NULL) {
4063                 eqp->eq_head = eqp->eq_tail = mp;
4064         } else {
4065                 eqp->eq_tail->b_next = mp;
4066                 eqp->eq_tail = mp;
```

```
4067         }
4068         eqp->eq_len++;

4070         /* If we're the first thread to reach the threshold, process */
4071         if (eqp->eq_len >= esbq_max_qlen &&
4072             !(eqp->eq_flags & ESBQ_PROCESSING))
4073                 esballoc_process_queue(eqp);

4075         esballoc_set_timer(eqp, esbq_timeout);
4076         mutex_exit(&eqp->eq_lock);
4077 }

4079 static void
4080 esballoc_process_queue(esb_queue_t *eqp)
4081 {
4082         mblk_t  *mp;

4084         ASSERT(MUTEX_HELD(&eqp->eq_lock));

4086         eqp->eq_flags |= ESBQ_PROCESSING;

4088         do {
4089                 /*
4090                  * Detach the message chain for processing.
4091                  */
4092                 mp = eqp->eq_head;
4093                 eqp->eq_tail->b_next = NULL;
4094                 eqp->eq_head = eqp->eq_tail = NULL;
4095                 eqp->eq_len = 0;
4096                 mutex_exit(&eqp->eq_lock);

4098                 /*
4099                  * Process the message chain.
4100                  */
4101                 esballoc_enqueue_mblk(mp);
4102                 mutex_enter(&eqp->eq_lock);
4103         } while ((eqp->eq_len >= esbq_max_qlen) && (eqp->eq_len > 0));

4105         eqp->eq_flags &= ~ESBQ_PROCESSING;
4106 }

4108 /*
4109  * taskq callback routine to free esballoced mblk's
4110  */
4111 static void
4112 esballoc_mblk_free(mblk_t *mp)
4113 {
4114         mblk_t  *nextmp;

4116         for (; mp != NULL; mp = nextmp) {
4117                 nextmp = mp->b_next;
4118                 mp->b_next = NULL;
4119                 mblk_free(mp);
4120         }
4121 }

4123 static void
4124 esballoc_enqueue_mblk(mblk_t *mp)
4125 {

4127         if (taskq_dispatch(system_taskq, (task_func_t *)esballoc_mblk_free, mp,
4128             TQ_NOSLEEP) == NULL) {
4129                 mblk_t *first_mp = mp;
4130                 /*
4131                  * System is low on resources and can't perform a non-sleeping
4132                  * dispatch. Schedule for a background thread.
```



```
4133                  */
4134                 mutex_enter(&service_queue);
4135                 STRSTAT(taskqfails);

4137                 while (mp->b_next != NULL)
4138                         mp = mp->b_next;

4140                 mp->b_next = freebs_list;
4141                 freebs_list = first_mp;
4142                 cv_signal(&services_to_run);
4143                 mutex_exit(&service_queue);
4144         }
4145 }

4147 static void
4148 esballoc_timer(void *arg)
4149 {
4150         esb_queue_t *eqp = arg;

4152         mutex_enter(&eqp->eq_lock);
4153         eqp->eq_flags &= ~ESBQ_TIMER;

4155         if (!(eqp->eq_flags & ESBQ_PROCESSING) &&
4156             eqp->eq_len > 0)
4157                 esballoc_process_queue(eqp);

4159         esballoc_set_timer(eqp, esbq_timeout);
4160         mutex_exit(&eqp->eq_lock);
4161 }

4163 static void
4164 esballoc_set_timer(esb_queue_t *eqp, clock_t eq_timeout)
4165 {
4166         ASSERT(MUTEX_HELD(&eqp->eq_lock));

4168         if (eqp->eq_len > 0 && !(eqp->eq_flags & ESBQ_TIMER)) {
4169                 (void) timeout(esballoc_timer, eqp, eq_timeout);
4170                 eqp->eq_flags |= ESBQ_TIMER;
4171         }
4172 }

4174 /*
4175  * Setup esbq array length based upon NCPU scaled by CPUs per
4176  * queue. Use static system_esbq until kmem_ready and we can
4177  * create an array in freebs_enqueue().
4178  */
4179 void
4180 esballoc_queue_init(void)
4181 {
4182         esbq_log2_cpus_per_q = highbit(esbq_cpus_per_q - 1);
4183         esbq_cpus_per_q = 1 << esbq_log2_cpus_per_q;
4184         esbq_nelem = howmany(NCPU, esbq_cpus_per_q);
4185         system_esbq.eq_len = 0;
4186         system_esbq.eq_head = system_esbq.eq_tail = NULL;
4187         system_esbq.eq_flags = 0;
4188 }

4190 /*
4191  * Set the QBACK or QB_BACK flag in the given queue for
4192  * the given priority band.
4193  */
4194 void
4195 setqback(queue_t *q, unsigned char pri)
4196 {
4197         int i;
4198         qband_t *qbp;
```

```
4199         qband_t **qbpp;

4201         ASSERT(MUTEX_HELD(QLOCK(q)));
4202         if (pri != 0) {
4203                 if (pri > q->q_nband) {
4204                         qbpp = &q->q_bandp;
4205                         while (*qbpp)
4206                                 qbpp = &(*qbpp)->qb_next;
4207                         while (pri > q->q_nband) {
4208                                 if ((*qbpp = allocband()) == NULL) {
4209                                         cmn_err(CE_WARN,
4210                                             "setqback: can't allocate qband\n");
4211                                         return;
4212                                 }
4213                                 (*qbpp)->qb_hiwat = q->q_hiwat;
4214                                 (*qbpp)->qb_lowat = q->q_lowat;
4215                                 q->q_nband++;
4216                                 qbpp = &(*qbpp)->qb_next;
4217                         }
4218                 }
4219                 qbp = q->q_bandp;
4220                 i = pri;
4221                 while (--i)
4222                         qbp = qbp->qb_next;
4223                 qbp->qb_flag |= QB_BACK;
4224         } else {
4225                 q->q_flag |= QBACK;
4226         }
4227 }

4229 int
4230 strcopyin(void *from, void *to, size_t len, int copyflag)
4231 {
4232         if (copyflag & U_TO_K) {
4233                 ASSERT((copyflag & K_TO_K) == 0);
4234                 if (copyin(from, to, len))
4235                         return (EFAULT);
4236         } else {
4237                 ASSERT(copyflag & K_TO_K);
4238                 bcopy(from, to, len);
4239         }
4240         return (0);
4241 }

4243 int
4244 strcopyout(void *from, void *to, size_t len, int copyflag)
4245 {
4246         if (copyflag & U_TO_K) {
4247                 if (copyout(from, to, len))
4248                         return (EFAULT);
4249         } else {
4250                 ASSERT(copyflag & K_TO_K);
4251                 bcopy(from, to, len);
4252         }
4253         return (0);
4254 }

4256 /*
4257  * strsignal_nolock() posts a signal to the process(es) at the stream head.
4258  * It assumes that the stream head lock is already held, whereas strsignal()
4259  * acquires the lock first.  This routine was created because a few callers
4260  * release the stream head lock before calling only to re-acquire it after
4261  * it returns.
4262  */
4263 void
4264 strsignal_nolock(stdata_t *stp, int sig, uchar_t band)
```



```
4265 {
4266         ASSERT(MUTEX_HELD(&stp->sd_lock));
4267         switch (sig) {
4268         case SIGPOLL:
4269                 if (stp->sd_sigflags & S_MSG)
4270                         strsendsig(stp->sd_siglist, S_MSG, band, 0);
4271                 break;
4272         default:
4273                 if (stp->sd_pgidp)
4274                         pgsignal(stp->sd_pgidp, sig);
4275                 break;
4276         }
4277 }

4279 void
4280 strsignal(stdata_t *stp, int sig, int32_t band)
4281 {
4282         TRACE_3(TR_FAC_STREAMS_FR, TR_SENDSIG,
4283             "strsignal:%p, %X, %X", stp, sig, band);

4285         mutex_enter(&stp->sd_lock);
4286         switch (sig) {
4287         case SIGPOLL:
4288                 if (stp->sd_sigflags & S_MSG)
4289                         strsendsig(stp->sd_siglist, S_MSG, (uchar_t)band, 0);
4290                 break;

4292         default:
4293                 if (stp->sd_pgidp) {
4294                         pgsignal(stp->sd_pgidp, sig);
4295                 }
4296                 break;
4297         }
4298         mutex_exit(&stp->sd_lock);
4299 }

4301 void
4302 strhup(stdata_t *stp)
4303 {
4304         ASSERT(mutex_owned(&stp->sd_lock));
4305         pollwakeup(&stp->sd_pollist, POLLHUP);
4306         if (stp->sd_sigflags & S_HANGUP)
4307                 strsendsig(stp->sd_siglist, S_HANGUP, 0, 0);
4308 }

4310 /*
4311  * Backenable the first queue upstream from 'q' with a service procedure.
4312  */
4313 void
4314 backenable(queue_t *q, uchar_t pri)
4315 {
4316         queue_t *nq;

4318         /*
4319          * Our presence might not prevent other modules in our own
4320          * stream from popping/pushing since the caller of getq might not
4321          * have a claim on the queue (some drivers do a getq on somebody
4322          * else's queue - they know that the queue itself is not going away
4323          * but the framework has to guarantee q_next in that stream).
4324          */
4325         claimstr(q);

4327         /* Find nearest back queue with service proc */
4328         for (nq = backq(q); nq && !nq->q_qinfo->qi_srvp; nq = backq(nq)) {
4329                 ASSERT(STRMATED(q->q_stream) || STREAM(q) == STREAM(nq));
4330         }
```

```
4332         if (nq) {
4333                 kthread_t *freezer;
4334                 /*
4335                  * backenable can be called either with no locks held
4336                  * or with the stream frozen (the latter occurs when a module
4337                  * calls rmvq with the stream frozen). If the stream is frozen
4338                  * by the caller the caller will hold all qlocks in the stream.
4339                  * Note that a frozen stream doesn't freeze a mated stream,
4340                  * so we explicitly check for that.
4341                  */
4342                 freezer = STREAM(q)->sd_freezer;
4343                 if (freezer != curthread || STREAM(q) != STREAM(nq)) {
4344                         mutex_enter(QLOCK(nq));
4345                 }
4346 #ifdef DEBUG
4347                 else {
4348                         ASSERT(frozenstr(q));
4349                         ASSERT(MUTEX_HELD(QLOCK(q)));
4350                         ASSERT(MUTEX_HELD(QLOCK(nq)));
4351                 }
4352 #endif
4353                 setqback(nq, pri);
4354                 qenable_locked(nq);
4355                 if (freezer != curthread || STREAM(q) != STREAM(nq))
4356                         mutex_exit(QLOCK(nq));
4357         }
4358         releasestr(q);
4359 }

4361 /*
4362  * Return the appropriate errno when one of flags_to_check is set
4363  * in sd_flags. Uses the exported error routines if they are set.
4364  * Will return 0 if non error is set (or if the exported error routines
4365  * do not return an error).
4366  *
4367  * If there is both a read and write error to check, we prefer the read error.
4368  * Also, give preference to recorded errno's over the error functions.
4369  * The flags that are handled are:
4370  *      STPLEX          return EINVAL
4371  *      STRDERR         return sd_rerror (and clear if STRDERRNONPERSIST)
4372  *      STWRERR         return sd_werror (and clear if STWRERRNONPERSIST)
4373  *      STRHUP          return sd_werror
4374  *
4375  * If the caller indicates that the operation is a peek, a nonpersistent error
4376  * is not cleared.
4377  */
4378 int
4379 strgeterr(stdata_t *stp, int32_t flags_to_check, int ispeek)
4380 {
4381         int32_t sd_flag = stp->sd_flag & flags_to_check;
4382         int error = 0;

4384         ASSERT(MUTEX_HELD(&stp->sd_lock));
4385         ASSERT((flags_to_check & ~(STRDERR|STWRERR|STRHUP|STPLEX)) == 0);
4386         if (sd_flag & STPLEX)
4387                 error = EINVAL;
4388         else if (sd_flag & STRDERR) {
4389                 error = stp->sd_rerror;
4390                 if ((stp->sd_flag & STRDERRNONPERSIST) && !ispeek) {
4391                         /*
4392                          * Read errors are non-persistent i.e. discarded once
4393                          * returned to a non-peeking caller,
4394                          */
4395                         stp->sd_rerror = 0;
4396                         stp->sd_flag &= ~STRDERR;
```



```
4397                 }
4398                 if (error == 0 && stp->sd_rderrfunc != NULL) {
4399                         int clearerr = 0;

4401                         error = (*stp->sd_rderrfunc)(stp->sd_vnode, ispeek,
4402                             &clearerr);
4403                         if (clearerr) {
4404                                 stp->sd_flag &= ~STRDERR;
4405                                 stp->sd_rderrfunc = NULL;
4406                         }
4407                 }
4408         } else if (sd_flag & STWRERR) {
4409                 error = stp->sd_werror;
4410                 if ((stp->sd_flag & STWRERRNONPERSIST) && !ispeek) {
4411                         /*
4412                          * Write errors are non-persistent i.e. discarded once
4413                          * returned to a non-peeking caller,
4414                          */
4415                         stp->sd_werror = 0;
4416                         stp->sd_flag &= ~STWRERR;
4417                 }
4418                 if (error == 0 && stp->sd_wrerrfunc != NULL) {
4419                         int clearerr = 0;

4421                         error = (*stp->sd_wrerrfunc)(stp->sd_vnode, ispeek,
4422                             &clearerr);
4423                         if (clearerr) {
4424                                 stp->sd_flag &= ~STWRERR;
4425                                 stp->sd_wrerrfunc = NULL;
4426                         }
4427                 }
4428         } else if (sd_flag & STRHUP) {
4429                 /* sd_werror set when STRHUP */
4430                 error = stp->sd_werror;
4431         }
4432         return (error);
4433 }


4436 /*
4437  * Single-thread open/close/push/pop
4438  * for twisted streams also
4439  */
4440 int
4441 strstartplumb(stdata_t *stp, int flag, int cmd)
4442 {
4443         int waited = 1;
4444         int error = 0;

4446         if (STRMATED(stp)) {
4447                 struct stdata *stmatep = stp->sd_mate;

4449                 STRLOCKMATES(stp);
4450                 while (waited) {
4451                         waited = 0;
4452                         while (stmatep->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4453                                 if ((cmd == I_POP) &&
4454                                     (flag & (FNDELAY|FNONBLOCK))) {
4455                                         STRUNLOCKMATES(stp);
4456                                         return (EAGAIN);
4457                                 }
4458                                 waited = 1;
4459                                 mutex_exit(&stp->sd_lock);
4460                                 if (!cv_wait_sig(&stmatep->sd_monitor,
4461                                     &stmatep->sd_lock)) {
4462                                         mutex_exit(&stmatep->sd_lock);
```

```
4463                                    return (EINTR);
4464                            }
4465                            mutex_exit(&stmatep->sd_lock);
4466                            STRLOCKMATES(stp);
4467                    }
4468                    while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4469                            if ((cmd == I_POP) &&
4470                                (flag & (FNDELAY|FNONBLOCK))) {
4471                                    STRUNLOCKMATES(stp);
4472                                    return (EAGAIN);
4473                            }
4474                            waited = 1;
4475                            mutex_exit(&stmatep->sd_lock);
4476                            if (!cv_wait_sig(&stp->sd_monitor,
4477                                &stp->sd_lock)) {
4478                                    mutex_exit(&stp->sd_lock);
4479                                    return (EINTR);
4480                            }
4481                            mutex_exit(&stp->sd_lock);
4482                            STRLOCKMATES(stp);
4483                    }
4484                    if (stp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
4485                            error = strgeterr(stp,
4486                                STRDERR|STWRERR|STRHUP|STPLEX, 0);
4487                            if (error != 0) {
4488                                    STRUNLOCKMATES(stp);
4489                                    return (error);
4490                            }
4491                    }
4492            }
4493            stp->sd_flag |= STRPLUMB;
4494            STRUNLOCKMATES(stp);
4495    } else {
4496            mutex_enter(&stp->sd_lock);
4497            while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4498                    if (((cmd == I_POP) || (cmd == _I_REMOVE)) &&
4499                        (flag & (FNDELAY|FNONBLOCK))) {
4500                            mutex_exit(&stp->sd_lock);
4501                            return (EAGAIN);
4502                    }
4503                    if (!cv_wait_sig(&stp->sd_monitor, &stp->sd_lock)) {
4504                            mutex_exit(&stp->sd_lock);
4505                            return (EINTR);
4506                    }
4507                    if (stp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
4508                            error = strgeterr(stp,
4509                                STRDERR|STWRERR|STRHUP|STPLEX, 0);
4510                            if (error != 0) {
4511                                    mutex_exit(&stp->sd_lock);
4512                                    return (error);
4513                            }
4514                    }
4515            }
4516            stp->sd_flag |= STRPLUMB;
4517            mutex_exit(&stp->sd_lock);
4518    }
4519    return (0);
4520 }

4522 /*
4523  * Complete the plumbing operation associated with stream ‘stp’.
4524  */
4525 void
4526 strendplumb(stdata_t *stp)
4527 {
4528    ASSERT(MUTEX_HELD(&stp->sd_lock));
```



```
4529    ASSERT(stp->sd_flag & STRPLUMB);
4530    stp->sd_flag &= ~STRPLUMB;
4531    cv_broadcast(&stp->sd_monitor);
4532 }

4534 /*
4535  * This describes how the STREAMS framework handles synchronization
4536  * during open/push and close/pop.
4537  * The key interfaces for open and close are qprocson and qprocsoff,
4538  * respectively. While the close case in general is harder both open
4539  * have close have significant similarities.
4540  *
4541  * During close the STREAMS framework has to both ensure that there
4542  * are no stale references to the queue pair (and syncq) that
4543  * are being closed and also provide the guarantees that are documented
4544  * in qprocsoff(9F).
4545  * If there are stale references to the queue that is closing it can
4546  * result in kernel memory corruption or kernel panics.
4547  *
4548  * Note that is it up to the module/driver to ensure that it itself
4549  * does not have any stale references to the closing queues once its close
4550  * routine returns. This includes:
4551  *  - Cancelling any timeout/bufcall/qtimeout/qbufcall callback routines
4552  *    associated with the queues. For timeout and bufcall callbacks the
4553  *    module/driver also has to ensure (or wait for) any callbacks that
4554  *    are in progress.
4555  *  - If the module/driver is using esballoc it has to ensure that any
4556  *    esballoc free functions do not refer to a queue that has closed.
4557  *    (Note that in general the close routine can not wait for the esballoc’ed
4558  *    messages to be freed since that can cause a deadlock.)
4559  *  - Cancelling any interrupts that refer to the closing queues and
4560  *    also ensuring that there are no interrupts in progress that will
4561  *    refer to the closing queues once the close routine returns.
4562  *  - For multiplexors removing any driver global state that refers to
4563  *    the closing queue and also ensuring that there are no threads in
4564  *    the multiplexor that has picked up a queue pointer but not yet
4565  *    finished using it.
4566  *
4567  * In addition, a driver/module can only reference the q_next pointer
4568  * in its open, close, put, or service procedures or in a
4569  * qtimeout/qbufcall callback procedure executing "on" the correct
4570  * stream. Thus it can not reference the q_next pointer in an interrupt
4571  * routine or a timeout, bufcall or esballoc callback routine. Likewise
4572  * it can not reference q_next of a different queue e.g. in a mux that
4573  * passes messages from one queues put/service procedure to another queue.
4574  * In all the cases when the driver/module can not access the q_next
4575  * field it must use the *next* versions e.g. canputnext instead of
4576  * canput(q->q_next) and putnextctl instead of putctl(q->q_next, ...).
4577  *
4578  *
4579  * Assuming that the driver/module conforms to the above constraints
4580  * the STREAMS framework has to avoid stale references to q_next for all
4581  * the framework internal cases which include (but are not limited to):
4582  *  - Threads in canput/canputnext/backenable and elsewhere that are
4583  *    walking q_next.
4584  *  - Messages on a syncq that have a reference to the queue through b_queue.
4585  *  - Messages on an outer perimeter (syncq) that have a reference to the
4586  *    queue through b_queue.
4587  *  - Threads that use q_nfsrv (e.g. canput) to find a queue.
4588  *    Note that only canput and bcanput use q_nfsrv without any locking.
4589  *
4590  * The STREAMS framework providing the qprocsoff(9F) guarantees means that
4591  * after qprocsoff returns, the framework has to ensure that no threads can
4592  * enter the put or service routines for the closing read or write-side queue.
4593  * In addition to preventing "direct" entry into the put procedures
4594  * the framework also has to prevent messages being drained from
```

```
 * the syncq or the outer perimeter.
 * 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) {
```

```
4727                    syncq_t *sq = sqll->sqll_sq;
4728                    uint16_t count;

4730                    mutex_enter(SQLOCK(sq));
4731                    count = sq->sq_count;
4732                    ASSERT(sq->sq_rmqcount <= count);
4733                    SQ_PUTLOCKS_ENTER(sq);
4734                    SUM_SQ_PUTCOUNTS(sq, count);
4735                    if (count == sq->sq_rmqcount)
4736                            continue;

4738                    /* Failed - drop all locks that we have acquired so far */
4739                    if (STRMATED(stp)) {
4740                            STREAM_PUTLOCKS_EXIT(stp);
4741                            STREAM_PUTLOCKS_EXIT(stp->sd_mate);
4742                            STRUNLOCKMATES(stp);
4743                            mutex_exit(&stp->sd_reflock);
4744                            mutex_exit(&stp->sd_mate->sd_reflock);
4745                    } else {
4746                            STREAM_PUTLOCKS_EXIT(stp);
4747                            mutex_exit(&stp->sd_lock);
4748                            mutex_exit(&stp->sd_reflock);
4749                    }
4750                    for (sql2 = sqlist->sqlist_head; sql2 != sql;
4751                        sql2 = sql2->sql_next) {
4752                            SQ_PUTLOCKS_EXIT(sql2->sql_sq);
4753                            mutex_exit(SQLOCK(sql2->sql_sq));
4754                    }

4756                    /*
4757                     * The wait loop below may starve when there are many threads
4758                     * claiming the syncq. This is especially a problem with permod
4759                     * syncqs (IP). To lessen the impact of the problem we increment
4760                     * sq_needexcl and clear fastbits so that putnexts will slow
4761                     * down and call sqenable instead of draining right away.
4762                     */
4763                    sq->sq_needexcl++;
4764                    SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
4765                    while (count > sq->sq_rmqcount) {
4766                            sq->sq_flags |= SQ_WANTWAKEUP;
4767                            SQ_PUTLOCKS_EXIT(sq);
4768                            cv_wait(&sq->sq_wait, SQLOCK(sq));
4769                            count = sq->sq_count;
4770                            SQ_PUTLOCKS_ENTER(sq);
4771                            SUM_SQ_PUTCOUNTS(sq, count);
4772                    }
4773                    sq->sq_needexcl--;
4774                    if (sq->sq_needexcl == 0)
4775                            SQ_PUTCOUNT_SETFAST_LOCKED(sq);
4776                    SQ_PUTLOCKS_EXIT(sq);
4777                    ASSERT(count == sq->sq_rmqcount);
4778                    mutex_exit(SQLOCK(sq));
4779                    goto retry;
4780            }
4781    }

4783    /*
4784     * Drop all the locks that strlock acquired.
4785     */
4786    static void
4787    strunlock(struct stdata *stp, sqlist_t *sqlist)
4788    {
4789            syncql_t *sql;

4791            if (STRMATED(stp)) {
4792                    STREAM_PUTLOCKS_EXIT(stp);
```



```
4793                    STREAM_PUTLOCKS_EXIT(stp->sd_mate);
4794                    STRUNLOCKMATES(stp);
4795                    mutex_exit(&stp->sd_reflock);
4796                    mutex_exit(&stp->sd_mate->sd_reflock);
4797            } else {
4798                    STREAM_PUTLOCKS_EXIT(stp);
4799                    mutex_exit(&stp->sd_lock);
4800                    mutex_exit(&stp->sd_reflock);
4801            }

4803            if (sqlist == NULL)
4804                    return;

4806            for (sql = sqlist->sqlist_head; sql; sql = sql->sql_next) {
4807                    SQ_PUTLOCKS_EXIT(sql->sql_sq);
4808                    mutex_exit(SQLOCK(sql->sql_sq));
4809            }
4810    }

4812    /*
4813     * When the module has service procedure, we need check if the next
4814     * module which has service procedure is in flow control to trigger
4815     * the backenable.
4816     */
4817    static void
4818    backenable_insertedq(queue_t *q)
4819    {
4820            qband_t *qbp;

4822            claimstr(q);
4823            if (q->q_qinfo->qi_srvp != NULL && q->q_next != NULL) {
4824                    if (q->q_next->q_nfsrv->q_flag & QWANTW)
4825                            backenable(q, 0);

4827                    qbp = q->q_next->q_nfsrv->q_bandp;
4828                    for (; qbp != NULL; qbp = qbp->qb_next)
4829                            if ((qbp->qb_flag & QB_WANTW) && qbp->qb_first != NULL)
4830                                    backenable(q, qbp->qb_first->b_band);
4831            }
4832            releasestr(q);
4833    }

4835    /*
4836     * Given two read queues, insert a new single one after another.
4837     *
4838     * This routine acquires all the necessary locks in order to change
4839     * q_next and related pointer using strlock().
4840     * It depends on the stream head ensuring that there are no concurrent
4841     * insertq or removeq on the same stream. The stream head ensures this
4842     * using the flags STWOPEN, STRCLOSE, and STRPLUMB.
4843     *
4844     * Note that no syncq locks are held during the q_next change. This is
4845     * applied to all streams since, unlike removeq, there is no problem of stale
4846     * pointers when adding a module to the stream. Thus drivers/modules that do a
4847     * canput(rq->q_next) would never get a closed/freed queue pointer even if we
4848     * applied this optimization to all streams.
4849     */
4850    void
4851    insertq(struct stdata *stp, queue_t *new)
4852    {
4853            queue_t *after;
4854            queue_t *wafter;
4855            queue_t *wnew = _WR(new);
4856            boolean_t have_fifo = B_FALSE;

4858            if (new->q_flag & _QINSERTING) {
```

```
4859                 ASSERT(stp->sd_vnode->v_type != VFIFO);
4860                 after = new->q_next;
4861                 wafter = _WR(new->q_next);
4862         } else {
4863                 after = _RD(stp->sd_wrq);
4864                 wafter = stp->sd_wrq;
4865         }

4867         TRACE_2(TR_FAC_STREAMS_FR, TR_INSERTQ,
4868             "insertq:%p, %p", after, new);
4869         ASSERT(after->q_flag & QREADR);
4870         ASSERT(new->q_flag & QREADR);

4872         strlock(stp, NULL);

4874         /* Do we have a FIFO? */
4875         if (wafter->q_next == after) {
4876                 have_fifo = B_TRUE;
4877                 wnew->q_next = new;
4878         } else {
4879                 wnew->q_next = wafter->q_next;
4880         }
4881         new->q_next = after;

4883         set_nfsrv_ptr(new, wnew, after, wafter);
4884         /*
4885          * set_nfsrv_ptr() needs to know if this is an insertion or not,
4886          * so only reset this flag after calling it.
4887          */
4888         new->q_flag &= ~_QINSERTING;

4890         if (have_fifo) {
4891                 wafter->q_next = wnew;
4892         } else {
4893                 if (wafter->q_next)
4894                         _OTHERQ(wafter->q_next)->q_next = new;
4895                 wafter->q_next = wnew;
4896         }

4898         set_qend(new);
4899         /* The QEND flag might have to be updated for the upstream guy */
4900         set_qend(after);

4902         ASSERT(_SAMESTR(new) == O_SAMESTR(new));
4903         ASSERT(_SAMESTR(wnew) == O_SAMESTR(wnew));
4904         ASSERT(_SAMESTR(after) == O_SAMESTR(after));
4905         ASSERT(_SAMESTR(wafter) == O_SAMESTR(wafter));
4906         strsetuio(stp);

4908         /*
4909          * If this was a module insertion, bump the push count.
4910          */
4911         if (!(new->q_flag & QISDRV))
4912                 stp->sd_pushcnt++;

4914         strunlock(stp, NULL);

4916         /* check if the write Q needs backenable */
4917         backenable_insertedq(wnew);

4919         /* check if the read Q needs backenable */
4920         backenable_insertedq(new);
4921 }

4923 /*
4924  * Given a read queue, unlink it from any neighbors.
```



```
4925  *
4926  * This routine acquires all the necessary locks in order to
4927  * change q_next and related pointers and also guard against
4928  * stale references (e.g. through q_next) to the queue that
4929  * is being removed. It also plays part of the role in ensuring
4930  * that the module's/driver's put procedure doesn't get called
4931  * after qprocsoff returns.
4932  *
4933  * Removeq depends on the stream head ensuring that there are
4934  * no concurrent insertq or removeq on the same stream. The
4935  * stream head ensures this using the flags STWOPEN, STRCLOSE and
4936  * STRPLUMB.
4937  *
4938  * The set of locks needed to remove the queue is different in
4939  * different cases:
4940  *
4941  * Acquire sd_lock, sd_reflock, and all the syncq locks in the stream after
4942  * waiting for the syncq reference count to drop to 0 indicating that no
4943  * non-close threads are present anywhere in the stream. This ensures that any
4944  * module/driver can reference q_next in its open, close, put, or service
4945  * procedures.
4946  *
4947  * The sq_rmqcount counter tracks the number of threads inside removeq().
4948  * strlock() ensures that there is either no threads executing inside perimeter
4949  * or there is only a thread calling qprocsoff().
4950  *
4951  * strlock() compares the value of sq_count with the number of threads inside
4952  * removeq() and waits until sq_count is equal to sq_rmqcount. We need to wakeup
4953  * any threads waiting in strlock() when the sq_rmqcount increases.
4954  */

4956 void
4957 removeq(queue_t *qp)
4958 {
4959         queue_t *wqp = _WR(qp);
4960         struct stdata *stp = STREAM(qp);
4961         sqlist_t *sqlist = NULL;
4962         boolean_t isdriver;
4963         int moved;
4964         syncq_t *sq = qp->q_syncq;
4965         syncq_t *wsq = wqp->q_syncq;

4967         ASSERT(stp);

4969         TRACE_2(TR_FAC_STREAMS_FR, TR_REMOVEQ,
4970             "removeq:%p %p", qp, wqp);
4971         ASSERT(qp->q_flag&QREADR);

4973         /*
4974          * For queues using Synchronous streams, we must wait for all threads in
4975          * rwnext() to drain out before proceeding.
4976          */
4977         if (qp->q_flag & QSYNCSTR) {
4978                 /* First, we need wakeup any threads blocked in rwnext() */
4979                 mutex_enter(SQLOCK(sq));
4980                 if (sq->sq_flags & SQ_WANTWAKEUP) {
4981                         sq->sq_flags &= ~SQ_WANTWAKEUP;
4982                         cv_broadcast(&sq->sq_wait);
4983                 }
4984                 mutex_exit(SQLOCK(sq));

4986                 if (wsq != sq) {
4987                         mutex_enter(SQLOCK(wsq));
4988                         if (wsq->sq_flags & SQ_WANTWAKEUP) {
4989                                 wsq->sq_flags &= ~SQ_WANTWAKEUP;
4990                                 cv_broadcast(&wsq->sq_wait);
```

```
4991                        }
4992                        mutex_exit(SQLOCK(wsq));
4993                }

4995                mutex_enter(QLOCK(qp));
4996                while (qp->q_rwcnt > 0) {
4997                        qp->q_flag |= QWANTRMQSYNC;
4998                        cv_wait(&qp->q_wait, QLOCK(qp));
4999                }
5000                mutex_exit(QLOCK(qp));

5002                mutex_enter(QLOCK(wqp));
5003                while (wqp->q_rwcnt > 0) {
5004                        wqp->q_flag |= QWANTRMQSYNC;
5005                        cv_wait(&wqp->q_wait, QLOCK(wqp));
5006                }
5007                mutex_exit(QLOCK(wqp));
5008        }

5010        mutex_enter(SQLOCK(sq));
5011        sq->sq_rmqcount++;
5012        if (sq->sq_flags & SQ_WANTWAKEUP) {
5013                sq->sq_flags &= ~SQ_WANTWAKEUP;
5014                cv_broadcast(&sq->sq_wait);
5015        }
5016        mutex_exit(SQLOCK(sq));

5018        isdriver = (qp->q_flag & QISDRV);

5020        sqlist = sqlist_build(qp, stp, STRMATED(stp));
5021        strlock(stp, sqlist);

5023        reset_nfsrv_ptr(qp, wqp);

5025        ASSERT(wqp->q_next == NULL || backq(qp)->q_next == qp);
5026        ASSERT(qp->q_next == NULL || backq(wqp)->q_next == wqp);
5027        /* Do we have a FIFO? */
5028        if (wqp->q_next == qp) {
5029                stp->sd_wrq->q_next = _RD(stp->sd_wrq);
5030        } else {
5031                if (wqp->q_next)
5032                        backq(qp)->q_next = qp->q_next;
5033                if (qp->q_next)
5034                        backq(wqp)->q_next = wqp->q_next;
5035        }

5037        /* The QEND flag might have to be updated for the upstream guy */
5038        if (qp->q_next)
5039                set_qend(qp->q_next);

5041        ASSERT(_SAMESTR(stp->sd_wrq) == O_SAMESTR(stp->sd_wrq));
5042        ASSERT(_SAMESTR(_RD(stp->sd_wrq)) == O_SAMESTR(_RD(stp->sd_wrq)));

5044        /*
5045         * Move any messages destined for the put procedures to the next
5046         * syncq in line. Otherwise free them.
5047         */
5048        moved = 0;
5049        /*
5050         * Quick check to see whether there are any messages or events.
5051         */
5052        if (qp->q_syncqmsgs != 0 || (qp->q_syncq->sq_flags & SQ_EVENTS))
5053                moved += propagate_syncq(qp);
5054        if (wqp->q_syncqmsgs != 0 ||
5055            (wqp->q_syncq->sq_flags & SQ_EVENTS))
5056                moved += propagate_syncq(wqp);
```



```
5058        strsetuio(stp);

5060        /*
5061         * If this was a module removal, decrement the push count.
5062         */
5063        if (!isdriver)
5064                stp->sd_pushcnt--;

5066        strunlock(stp, sqlist);
5067        sqlist_free(sqlist);

5069        /*
5070         * Make sure any messages that were propagated are drained.
5071         * Also clear any QFULL bit caused by messages that were propagated.
5072         */

5074        if (qp->q_next != NULL) {
5075                clr_qfull(qp);
5076                /*
5077                 * For the driver calling qprocsoff, propagate_syncq
5078                 * frees all the messages instead of putting it in
5079                 * the stream head
5080                 */
5081                if (!isdriver && (moved > 0))
5082                        emptysq(qp->q_next->q_syncq);
5083        }
5084        if (wqp->q_next != NULL) {
5085                clr_qfull(wqp);
5086                /*
5087                 * We come here for any pop of a module except for the
5088                 * case of driver being removed. We don't call emptysq
5089                 * if we did not move any messages. This will avoid holding
5090                 * PERMOD syncq locks in emptysq
5091                 */
5092                if (moved > 0)
5093                        emptysq(wqp->q_next->q_syncq);
5094        }

5096        mutex_enter(SQLOCK(sq));
5097        sq->sq_rmqcount--;
5098        mutex_exit(SQLOCK(sq));
5099 }

5101 /*
5102  * Prevent further entry by setting a flag (like SQ_FROZEN, SQ_BLOCKED or
5103  * SQ_WRITER) on a syncq.
5104  * If maxcnt is not -1 it assumes that caller has "maxcnt" claim(s) on the
5105  * sync queue and waits until sq_count reaches maxcnt.
5106  *
5107  * If maxcnt is -1 there's no need to grab sq_putlocks since the caller
5108  * does not care about putnext threads that are in the middle of calling put
5109  * entry points.
5110  *
5111  * This routine is used for both inner and outer syncqs.
5112  */
5113 static void
5114 blocksq(syncq_t *sq, ushort_t flag, int maxcnt)
5115 {
5116         uint16_t count = 0;

5118         mutex_enter(SQLOCK(sq));
5119         /*
5120          * Wait for SQ_FROZEN/SQ_BLOCKED to be reset.
5121          * SQ_FROZEN will be set if there is a frozen stream that has a
5122          * queue which also refers to this "shared" syncq.
```

```
5123          * SQ_BLOCKED will be set if there is "off" queue which also
5124          * refers to this "shared" syncq.
5125          */
5126         if (maxcnt != -1) {
5127                 count = sq->sq_count;
5128                 SQ_PUTLOCKS_ENTER(sq);
5129                 SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
5130                 SUM_SQ_PUTCOUNTS(sq, count);
5131         }
5132         sq->sq_needexcl++;
5133         ASSERT(sq->sq_needexcl != 0);   /* wraparound */

5135         while ((sq->sq_flags & flag) ||
5136             (maxcnt != -1 && count > (unsigned)maxcnt)) {
5137                 sq->sq_flags |= SQ_WANTWAKEUP;
5138                 if (maxcnt != -1) {
5139                         SQ_PUTLOCKS_EXIT(sq);
5140                 }
5141                 cv_wait(&sq->sq_wait, SQLOCK(sq));
5142                 if (maxcnt != -1) {
5143                         count = sq->sq_count;
5144                         SQ_PUTLOCKS_ENTER(sq);
5145                         SUM_SQ_PUTCOUNTS(sq, count);
5146                 }
5147         }
5148         sq->sq_needexcl--;
5149         sq->sq_flags |= flag;
5150         ASSERT(maxcnt == -1 || count == maxcnt);
5151         if (maxcnt != -1) {
5152                 if (sq->sq_needexcl == 0) {
5153                         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
5154                 }
5155                 SQ_PUTLOCKS_EXIT(sq);
5156         } else if (sq->sq_needexcl == 0) {
5157                 SQ_PUTCOUNT_SETFAST(sq);
5158         }

5160         mutex_exit(SQLOCK(sq));
5161 }

5163 /*
5164  * Reset a flag that was set with blocksq.
5165  *
5166  * Can not use this routine to reset SQ_WRITER.
5167  *
5168  * If "isouter" is set then the syncq is assumed to be an outer perimeter
5169  * and drain_syncq is not called. Instead we rely on the qwriter_outer thread
5170  * to handle the queued qwriter operations.
5171  *
5172  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5173  * sq_putlocks are used.
5174  */
5175 static void
5176 unblocksq(syncq_t *sq, uint16_t resetflag, int isouter)
5177 {
5178         uint16_t flags;

5180         mutex_enter(SQLOCK(sq));
5181         ASSERT(resetflag != SQ_WRITER);
5182         ASSERT(sq->sq_flags & resetflag);
5183         flags = sq->sq_flags & ~resetflag;
5184         sq->sq_flags = flags;
5185         if (flags & (SQ_QUEUED | SQ_WANTWAKEUP)) {
5186                 if (flags & SQ_WANTWAKEUP) {
5187                         flags &= ~SQ_WANTWAKEUP;
5188                         cv_broadcast(&sq->sq_wait);
```



```
5189                 }
5190                 sq->sq_flags = flags;
5191                 if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5192                         if (!isouter) {
5193                                 /* drain_syncq drops SQLOCK */
5194                                 drain_syncq(sq);
5195                                 return;
5196                         }
5197                 }
5198         }
5199         mutex_exit(SQLOCK(sq));
5200 }

5202 /*
5203  * Reset a flag that was set with blocksq.
5204  * Does not drain the syncq. Use emptysq() for that.
5205  * Returns 1 if SQ_QUEUED is set. Otherwise 0.
5206  *
5207  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5208  * sq_putlocks are used.
5209  */
5210 static int
5211 dropsq(syncq_t *sq, uint16_t resetflag)
5212 {
5213         uint16_t flags;

5215         mutex_enter(SQLOCK(sq));
5216         ASSERT(sq->sq_flags & resetflag);
5217         flags = sq->sq_flags & ~resetflag;
5218         if (flags & SQ_WANTWAKEUP) {
5219                 flags &= ~SQ_WANTWAKEUP;
5220                 cv_broadcast(&sq->sq_wait);
5221         }
5222         sq->sq_flags = flags;
5223         mutex_exit(SQLOCK(sq));
5224         if (flags & SQ_QUEUED)
5225                 return (1);
5226         return (0);
5227 }

5229 /*
5230  * Empty all the messages on a syncq.
5231  *
5232  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5233  * sq_putlocks are used.
5234  */
5235 static void
5236 emptysq(syncq_t *sq)
5237 {
5238         uint16_t flags;

5240         mutex_enter(SQLOCK(sq));
5241         flags = sq->sq_flags;
5242         if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5243                 /*
5244                  * To prevent potential recursive invocation of drain_syncq we
5245                  * do not call drain_syncq if count is non-zero.
5246                  */
5247                 if (sq->sq_count == 0) {
5248                         /* drain_syncq() drops SQLOCK */
5249                         drain_syncq(sq);
5250                         return;
5251                 } else
5252                         sqenable(sq);
5253         }
5254         mutex_exit(SQLOCK(sq));
```

```
5255 }

5257 /*
5258  * Ordered insert while removing duplicates.
5259  */
5260 static void
5261 sqlist_insert(sqlist_t *sqlist, syncq_t *sqp)
5262 {
5263         syncql_t *sqlp, **prev_sqlpp, *new_sqlp;

5265         prev_sqlpp = &sqlist->sqlist_head;
5266         while ((sqlp = *prev_sqlpp) != NULL) {
5267                 if (sqlp->sql_sq >= sqp) {
5268                         if (sqlp->sql_sq == sqp)        /* duplicate */
5269                                 return;
5270                         break;
5271                 }
5272                 prev_sqlpp = &sqlp->sql_next;
5273         }
5274         new_sqlp = &sqlist->sqlist_array[sqlist->sqlist_index++];
5275         ASSERT((char *)new_sqlp < (char *)sqlist + sqlist->sqlist_size);
5276         new_sqlp->sql_next = sqlp;
5277         new_sqlp->sql_sq = sqp;
5278         *prev_sqlpp = new_sqlp;
5279 }

5281 /*
5282  * Walk the write side queues until we hit either the driver
5283  * or a twist in the stream (_SAMESTR will return false in both
5284  * these cases) then turn around and walk the read side queues
5285  * back up to the stream head.
5286  */
5287 static void
5288 sqlist_insertall(sqlist_t *sqlist, queue_t *q)
5289 {
5290         while (q != NULL) {
5291                 sqlist_insert(sqlist, q->q_syncq);

5293                 if (_SAMESTR(q))
5294                         q = q->q_next;
5295                 else if (!(q->q_flag & QREADR))
5296                         q = _RD(q);
5297                 else
5298                         q = NULL;
5299         }
5300 }

5302 /*
5303  * Allocate and build a list of all syncqs in a stream and the syncq(s)
5304  * associated with the "q" parameter. The resulting list is sorted in a
5305  * canonical order and is free of duplicates.
5306  * Assumes the passed queue is a _RD(q).
5307  */
5308 static sqlist_t *
5309 sqlist_build(queue_t *q, struct stdata *stp, boolean_t do_twist)
5310 {
5311         sqlist_t *sqlist = sqlist_alloc(stp, KM_SLEEP);

5313         /*
5314          * start with the current queue/qpair
5315          */
5316         ASSERT(q->q_flag & QREADR);

5318         sqlist_insert(sqlist, q->q_syncq);
5319         sqlist_insert(sqlist, _WR(q)->q_syncq);
```



```
5321         sqlist_insertall(sqlist, stp->sd_wrq);
5322         if (do_twist)
5323                 sqlist_insertall(sqlist, stp->sd_mate->sd_wrq);

5325         return (sqlist);
5326 }

5328 static sqlist_t *
5329 sqlist_alloc(struct stdata *stp, int kmflag)
5330 {
5331         size_t sqlist_size;
5332         sqlist_t *sqlist;

5334         /*
5335          * Allocate 2 syncql_t's for each pushed module. Note that
5336          * the sqlist_t structure already has 4 syncql_t's built in:
5337          * 2 for the stream head, and 2 for the driver/other stream head.
5338          */
5339         sqlist_size = 2 * sizeof (syncql_t) * stp->sd_pushcnt +
5340             sizeof (sqlist_t);
5341         if (STRMATED(stp))
5342                 sqlist_size += 2 * sizeof (syncql_t) * stp->sd_mate->sd_pushcnt;
5343         sqlist = kmem_alloc(sqlist_size, kmflag);

5345         sqlist->sqlist_head = NULL;
5346         sqlist->sqlist_size = sqlist_size;
5347         sqlist->sqlist_index = 0;

5349         return (sqlist);
5350 }

5352 /*
5353  * Free the list created by sqlist_alloc()
5354  */
5355 static void
5356 sqlist_free(sqlist_t *sqlist)
5357 {
5358         kmem_free(sqlist, sqlist->sqlist_size);
5359 }

5361 /*
5362  * Prevent any new entries into any syncq in this stream.
5363  * Used by freezestr.
5364  */
5365 void
5366 strblock(queue_t *q)
5367 {
5368         struct stdata   *stp;
5369         syncql_t        *sql;
5370         sqlist_t        *sqlist;

5372         q = _RD(q);

5374         stp = STREAM(q);
5375         ASSERT(stp != NULL);

5377         /*
5378          * Get a sorted list with all the duplicates removed containing
5379          * all the syncqs referenced by this stream.
5380          */
5381         sqlist = sqlist_build(q, stp, B_FALSE);
5382         for (sql = sqlist->sqlist_head; sql != NULL; sql = sql->sql_next)
5383                 blocksq(sql->sql_sq, SQ_FROZEN, -1);
5384         sqlist_free(sqlist);
5385 }
```

```
5387 /*
5388  * Release the block on new entries into this stream
5389  */
5390 void
5391 strunblock(queue_t *q)
5392 {
5393         struct stdata   *stp;
5394         syncql_t        *sql;
5395         sqlist_t        *sqlist;
5396         int             drain_needed;

5398         q = _RD(q);

5400         /*
5401          * Get a sorted list with all the duplicates removed containing
5402          * all the syncqs referenced by this stream.
5403          * Have to drop the SQ_FROZEN flag on all the syncqs before
5404          * starting to drain them; otherwise the draining might
5405          * cause a freezestr in some module on the stream (which
5406          * would deadlock).
5407          */
5408         stp = STREAM(q);
5409         ASSERT(stp != NULL);
5410         sqlist = sqlist_build(q, stp, B_FALSE);
5411         drain_needed = 0;
5412         for (sql = sqlist->sqlist_head; sql != NULL; sql = sql->sql_next)
5413                 drain_needed += dropsq(sql->sql_sq, SQ_FROZEN);
5414         if (drain_needed) {
5415                 for (sql = sqlist->sqlist_head; sql != NULL;
5416                     sql = sql->sql_next)
5417                         emptysq(sql->sql_sq);
5418         }
5419         sqlist_free(sqlist);
5420 }

5422 #ifdef DEBUG
5423 static int
5424 qprocsareon(queue_t *rq)
5425 {
5426         if (rq->q_next == NULL)
5427                 return (0);
5428         return (_WR(rq->q_next)->q_next == _WR(rq));
5429 }

5431 int
5432 qclaimed(queue_t *q)
5433 {
5434         uint_t count;

5436         count = q->q_syncq->sq_count;
5437         SUM_SQ_PUTCOUNTS(q->q_syncq, count);
5438         return (count != 0);
5439 }

5441 /*
5442  * Check if anyone has frozen this stream with freezestr
5443  */
5444 int
5445 frozenstr(queue_t *q)
5446 {
5447         return ((q->q_syncq->sq_flags & SQ_FROZEN) != 0);
5448 }
5449 #endif /* DEBUG */

5451 /*
5452  * Enter a queue.
```



```
5453  * Obsoleted interface. Should not be used.
5454  */
5455 void
5456 enterq(queue_t *q)
5457 {
5458         entersq(q->q_syncq, SQ_CALLBACK);
5459 }

5461 void
5462 leaveq(queue_t *q)
5463 {
5464         leavesq(q->q_syncq, SQ_CALLBACK);
5465 }

5467 /*
5468  * Enter a perimeter. c_inner and c_outer specifies which concurrency bits
5469  * to check.
5470  * Wait if SQ_QUEUED is set to preserve ordering between messages and qwriter
5471  * calls and the running of open, close and service procedures.
5472  *
5473  * If c_inner bit is set no need to grab sq_putlocks since we don't care
5474  * if other threads have entered or are entering put entry point.
5475  *
5476  * If c_inner bit is set it might have been possible to use
5477  * sq_putlocks/sq_putcounts instead of SQLOCK/sq_count (e.g. to optimize
5478  * open/close path for IP) but since the count may need to be decremented in
5479  * qwait() we wouldn't know which counter to decrement. Currently counter is
5480  * selected by current cpu_seqid and current CPU can change at any moment. XXX
5481  * in the future we might use curthread id bits to select the counter and this
5482  * would stay constant across routine calls.
5483  */
5484 void
5485 entersq(syncq_t *sq, int entrypoint)
5486 {
5487         uint16_t        count = 0;
5488         uint16_t        flags;
5489         uint16_t        waitflags = SQ_STAYAWAY | SQ_EVENTS | SQ_EXCL;
5490         uint16_t        type;
5491         uint_t          c_inner = entrypoint & SQ_CI;
5492         uint_t          c_outer = entrypoint & SQ_CO;

5494         /*
5495          * Increment ref count to keep closes out of this queue.
5496          */
5497         ASSERT(sq);
5498         ASSERT(c_inner && c_outer);
5499         mutex_enter(SQLOCK(sq));
5500         flags = sq->sq_flags;
5501         type = sq->sq_type;
5502         if (!(type & c_inner)) {
5503                 /* Make sure all putcounts now use slowlock. */
5504                 count = sq->sq_count;
5505                 SQ_PUTLOCKS_ENTER(sq);
5506                 SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
5507                 SUM_SQ_PUTCOUNTS(sq, count);
5508                 sq->sq_needexcl++;
5509                 ASSERT(sq->sq_needexcl != 0);   /* wraparound */
5510                 waitflags |= SQ_MESSAGES;
5511         }
5512         /*
5513          * Wait until we can enter the inner perimeter.
5514          * If we want exclusive access we wait until sq_count is 0.
5515          * We have to do this before entering the outer perimeter in order
5516          * to preserve put/close message ordering.
5517          */
5518         while ((flags & waitflags) || (!(type & c_inner) && count != 0)) {
```

```
5519                 sq->sq_flags = flags | SQ_WANTWAKEUP;
5520                 if (!(type & c_inner)) {
5521                         SQ_PUTLOCKS_EXIT(sq);
5522                 }
5523                 cv_wait(&sq->sq_wait, SQLOCK(sq));
5524                 if (!(type & c_inner)) {
5525                         count = sq->sq_count;
5526                         SQ_PUTLOCKS_ENTER(sq);
5527                         SUM_SQ_PUTCOUNTS(sq, count);
5528                 }
5529                 flags = sq->sq_flags;
5530         }

5532         if (!(type & c_inner)) {
5533                 ASSERT(sq->sq_needexcl > 0);
5534                 sq->sq_needexcl--;
5535                 if (sq->sq_needexcl == 0) {
5536                         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
5537                 }
5538         }

5540         /* Check if we need to enter the outer perimeter */
5541         if (!(type & c_outer)) {
5542                 /*
5543                  * We have to enter the outer perimeter exclusively before
5544                  * we can increment sq_count to avoid deadlock. This implies
5545                  * that we have to re-check sq_flags and sq_count.
5546                  *
5547                  * is it possible to have c_inner set when c_outer is not set?
5548                  */
5549                 if (!(type & c_inner)) {
5550                         SQ_PUTLOCKS_EXIT(sq);
5551                 }
5552                 mutex_exit(SQLOCK(sq));
5553                 outer_enter(sq->sq_outer, SQ_GOAWAY);
5554                 mutex_enter(SQLOCK(sq));
5555                 flags = sq->sq_flags;
5556                 /*
5557                  * there should be no need to recheck sq_putcounts
5558                  * because outer_enter() has already waited for them to clear
5559                  * after setting SQ_WRITER.
5560                  */
5561                 count = sq->sq_count;
5562 #ifdef DEBUG
5563                 /*
5564                  * SUMCHECK_SQ_PUTCOUNTS should return the sum instead
5565                  * of doing an ASSERT internally. Others should do
5566                  * something like
5567                  *       ASSERT(SUMCHECK_SQ_PUTCOUNTS(sq) == 0);
5568                  * without the need to #ifdef DEBUG it.
5569                  */
5570                 SUMCHECK_SQ_PUTCOUNTS(sq, 0);
5571 #endif
5572                 while ((flags & (SQ_EXCL|SQ_BLOCKED|SQ_FROZEN)) ||
5573                     (!(type & c_inner) && count != 0)) {
5574                         sq->sq_flags = flags | SQ_WANTWAKEUP;
5575                         cv_wait(&sq->sq_wait, SQLOCK(sq));
5576                         count = sq->sq_count;
5577                         flags = sq->sq_flags;
5578                 }
5579         }

5581         sq->sq_count++;
5582         ASSERT(sq->sq_count != 0);      /* Wraparound */
5583         if (!(type & c_inner)) {
5584                 /* Exclusive entry */
```



```
5585                 ASSERT(sq->sq_count == 1);
5586                 sq->sq_flags |= SQ_EXCL;
5587                 if (type & c_outer) {
5588                         SQ_PUTLOCKS_EXIT(sq);
5589                 }
5590         }
5591         mutex_exit(SQLOCK(sq));
5592 }

5594 /*
5595  * Leave a syncq. Announce to framework that closes may proceed.
5596  * c_inner and c_outer specify which concurrency bits to check.
5597  *
5598  * Must never be called from driver or module put entry point.
5599  *
5600  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5601  * sq_putlocks are used.
5602  */
5603 void
5604 leavesq(syncq_t *sq, int entrypoint)
5605 {
5606         uint16_t        flags;
5607         uint16_t        type;
5608         uint_t          c_outer = entrypoint & SQ_CO;
5609 #ifdef DEBUG
5610         uint_t          c_inner = entrypoint & SQ_CI;
5611 #endif

5613         /*
5614          * Decrement ref count, drain the syncq if possible, and wake up
5615          * any waiting close.
5616          */
5617         ASSERT(sq);
5618         ASSERT(c_inner && c_outer);
5619         mutex_enter(SQLOCK(sq));
5620         flags = sq->sq_flags;
5621         type = sq->sq_type;
5622         if (flags & (SQ_QUEUED|SQ_WANTWAKEUP|SQ_WANTEXWAKEUP)) {

5624                 if (flags & SQ_WANTWAKEUP) {
5625                         flags &= ~SQ_WANTWAKEUP;
5626                         cv_broadcast(&sq->sq_wait);
5627                 }
5628                 if (flags & SQ_WANTEXWAKEUP) {
5629                         flags &= ~SQ_WANTEXWAKEUP;
5630                         cv_broadcast(&sq->sq_exitwait);
5631                 }

5633                 if ((flags & SQ_QUEUED) && !(flags & SQ_STAYAWAY)) {
5634                         /*
5635                          * The syncq needs to be drained. "Exit" the syncq
5636                          * before calling drain_syncq.
5637                          */
5638                         ASSERT(sq->sq_count != 0);
5639                         sq->sq_count--;
5640                         ASSERT((flags & SQ_EXCL) || (type & c_inner));
5641                         sq->sq_flags = flags & ~SQ_EXCL;
5642                         drain_syncq(sq);
5643                         ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
5644                         /* Check if we need to exit the outer perimeter */
5645                         /* XXX will this ever be true? */
5646                         if (!(type & c_outer))
5647                                 outer_exit(sq->sq_outer);
5648                         return;
5649                 }
5650         }
```

```
5651         ASSERT(sq->sq_count != 0);
5652         sq->sq_count--;
5653         ASSERT((flags & SQ_EXCL) || (type & c_inner));
5654         sq->sq_flags = flags & ~SQ_EXCL;
5655         mutex_exit(SQLOCK(sq));

5657         /* Check if we need to exit the outer perimeter */
5658         if (!(sq->sq_type & c_outer))
5659                 outer_exit(sq->sq_outer);
5660 }

5662 /*
5663  * Prevent q_next from changing in this stream by incrementing sq_count.
5664  *
5665  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5666  * sq_putlocks are used.
5667  */
5668 void
5669 claimq(queue_t *qp)
5670 {
5671         syncq_t *sq = qp->q_syncq;

5673         mutex_enter(SQLOCK(sq));
5674         sq->sq_count++;
5675         ASSERT(sq->sq_count != 0);      /* Wraparound */
5676         mutex_exit(SQLOCK(sq));
5677 }

5679 /*
5680  * Undo claimq.
5681  *
5682  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5683  * sq_putlocks are used.
5684  */
5685 void
5686 releaseq(queue_t *qp)
5687 {
5688         syncq_t *sq = qp->q_syncq;
5689         uint16_t flags;

5691         mutex_enter(SQLOCK(sq));
5692         ASSERT(sq->sq_count > 0);
5693         sq->sq_count--;

5695         flags = sq->sq_flags;
5696         if (flags & (SQ_WANTWAKEUP|SQ_QUEUED)) {
5697                 if (flags & SQ_WANTWAKEUP) {
5698                         flags &= ~SQ_WANTWAKEUP;
5699                         cv_broadcast(&sq->sq_wait);
5700                 }
5701                 sq->sq_flags = flags;
5702                 if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5703                         /*
5704                          * To prevent potential recursive invocation of
5705                          * drain_syncq we do not call drain_syncq if count is
5706                          * non-zero.
5707                          */
5708                         if (sq->sq_count == 0) {
5709                                 drain_syncq(sq);
5710                                 return;
5711                         } else
5712                                 sqenable(sq);
5713                 }
5714         }
5715         mutex_exit(SQLOCK(sq));
5716 }
```



```
5718 /*
5719  * Prevent q_next from changing in this stream by incrementing sd_refcnt.
5720  */
5721 void
5722 claimstr(queue_t *qp)
5723 {
5724         struct stdata *stp = STREAM(qp);

5726         mutex_enter(&stp->sd_reflock);
5727         stp->sd_refcnt++;
5728         ASSERT(stp->sd_refcnt != 0);    /* Wraparound */
5729         mutex_exit(&stp->sd_reflock);
5730 }

5732 /*
5733  * Undo claimstr.
5734  */
5735 void
5736 releasestr(queue_t *qp)
5737 {
5738         struct stdata *stp = STREAM(qp);

5740         mutex_enter(&stp->sd_reflock);
5741         ASSERT(stp->sd_refcnt != 0);
5742         if (--stp->sd_refcnt == 0)
5743                 cv_broadcast(&stp->sd_refmonitor);
5744         mutex_exit(&stp->sd_reflock);
5745 }

5747 static syncq_t *
5748 new_syncq(void)
5749 {
5750         return (kmem_cache_alloc(syncq_cache, KM_SLEEP));
5751 }

5753 static void
5754 free_syncq(syncq_t *sq)
5755 {
5756         ASSERT(sq->sq_head == NULL);
5757         ASSERT(sq->sq_outer == NULL);
5758         ASSERT(sq->sq_callbpend == NULL);
5759         ASSERT((sq->sq_onext == NULL && sq->sq_oprev == NULL) ||
5760             (sq->sq_onext == sq && sq->sq_oprev == sq));

5762         if (sq->sq_ciputctrl != NULL) {
5763                 ASSERT(sq->sq_nciputctrl == n_ciputctrl - 1);
5764                 SUMCHECK_CIPUTCTRL_COUNTS(sq->sq_ciputctrl,
5765                     sq->sq_nciputctrl, 0);
5766                 ASSERT(ciputctrl_cache != NULL);
5767                 kmem_cache_free(ciputctrl_cache, sq->sq_ciputctrl);
5768         }

5770         sq->sq_tail = NULL;
5771         sq->sq_evhead = NULL;
5772         sq->sq_evtail = NULL;
5773         sq->sq_ciputctrl = NULL;
5774         sq->sq_nciputctrl = 0;
5775         sq->sq_count = 0;
5776         sq->sq_rmqcount = 0;
5777         sq->sq_callbflags = 0;
5778         sq->sq_cancelid = 0;
5779         sq->sq_next = NULL;
5780         sq->sq_needexcl = 0;
5781         sq->sq_svcflags = 0;
5782         sq->sq_nqueues = 0;
```

```
5783         sq->sq_pri = 0;
5784         sq->sq_onext = NULL;
5785         sq->sq_oprev = NULL;
5786         sq->sq_flags = 0;
5787         sq->sq_type = 0;
5788         sq->sq_servcount = 0;

5790         kmem_cache_free(syncq_cache, sq);
5791 }

5793 /* Outer perimeter code */

5795 /*
5796  * The outer syncq uses the fields and flags in the syncq slightly
5797  * differently from the inner syncqs.
5798  *      sq_count        Incremented when there are pending or running
5799  *                      writers at the outer perimeter to prevent the set of
5800  *                      inner syncqs that belong to the outer perimeter from
5801  *                      changing.
5802  *      sq_head/tail    List of deferred qwriter(OUTER) operations.
5803  *
5804  *      SQ_BLOCKED      Set to prevent traversing of sq_next,sq_prev while
5805  *                      inner syncqs are added to or removed from the
5806  *                      outer perimeter.
5807  *      SQ_QUEUED       sq_head/tail has messages or events queued.
5808  *
5809  *      SQ_WRITER       A thread is currently traversing all the inner syncqs
5810  *                      setting the SQ_WRITER flag.
5811  */

5813 /*
5814  * Get write access at the outer perimeter.
5815  * Note that read access is done by entersq, putnext, and put by simply
5816  * incrementing sq_count in the inner syncq.
5817  *
5818  * Waits until "flags" is no longer set in the outer to prevent multiple
5819  * threads from having write access at the same time. SQ_WRITER has to be part
5820  * of "flags".
5821  *
5822  * Increases sq_count on the outer syncq to keep away outer_insert/remove
5823  * until the outer_exit is finished.
5824  *
5825  * outer_enter is vulnerable to starvation since it does not prevent new
5826  * threads from entering the inner syncqs while it is waiting for sq_count to
5827  * go to zero.
5828  */
5829 void
5830 outer_enter(syncq_t *outer, uint16_t flags)
5831 {
5832         syncq_t *sq;
5833         int     wait_needed;
5834         uint16_t        count;

5836         ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
5837             outer->sq_oprev != NULL);
5838         ASSERT(flags & SQ_WRITER);

5840 retry:
5841         mutex_enter(SQLOCK(outer));
5842         while (outer->sq_flags & flags) {
5843                 outer->sq_flags |= SQ_WANTWAKEUP;
5844                 cv_wait(&outer->sq_wait, SQLOCK(outer));
5845         }

5847         ASSERT(!(outer->sq_flags & SQ_WRITER));
5848         outer->sq_flags |= SQ_WRITER;
```



```
5849         outer->sq_count++;
5850         ASSERT(outer->sq_count != 0);   /* wraparound */
5851         wait_needed = 0;
5852         /*
5853          * Set SQ_WRITER on all the inner syncqs while holding
5854          * the SQLOCK on the outer syncq. This ensures that the changing
5855          * of SQ_WRITER is atomic under the outer SQLOCK.
5856          */
5857         for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
5858                 mutex_enter(SQLOCK(sq));
5859                 count = sq->sq_count;
5860                 SQ_PUTLOCKS_ENTER(sq);
5861                 sq->sq_flags |= SQ_WRITER;
5862                 SUM_SQ_PUTCOUNTS(sq, count);
5863                 if (count != 0)
5864                         wait_needed = 1;
5865                 SQ_PUTLOCKS_EXIT(sq);
5866                 mutex_exit(SQLOCK(sq));
5867         }
5868         mutex_exit(SQLOCK(outer));

5870         /*
5871          * Get everybody out of the syncqs sequentially.
5872          * Note that we don't actually need to acquire the PUTLOCKS, since
5873          * we have already cleared the fastbit, and set QWRITER.  By
5874          * definition, the count can not increase since putnext will
5875          * take the slowlock path (and the purpose of acquiring the
5876          * putlocks was to make sure it didn't increase while we were
5877          * waiting).
5878          *
5879          * Note that we still acquire the PUTLOCKS to be safe.
5880          */
5881         if (wait_needed) {
5882                 for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
5883                         mutex_enter(SQLOCK(sq));
5884                         count = sq->sq_count;
5885                         SQ_PUTLOCKS_ENTER(sq);
5886                         SUM_SQ_PUTCOUNTS(sq, count);
5887                         while (count != 0) {
5888                                 sq->sq_flags |= SQ_WANTWAKEUP;
5889                                 SQ_PUTLOCKS_EXIT(sq);
5890                                 cv_wait(&sq->sq_wait, SQLOCK(sq));
5891                                 count = sq->sq_count;
5892                                 SQ_PUTLOCKS_ENTER(sq);
5893                                 SUM_SQ_PUTCOUNTS(sq, count);
5894                         }
5895                         SQ_PUTLOCKS_EXIT(sq);
5896                         mutex_exit(SQLOCK(sq));
5897                 }
5898                 /*
5899                  * Verify that none of the flags got set while we
5900                  * were waiting for the sq_counts to drop.
5901                  * If this happens we exit and retry entering the
5902                  * outer perimeter.
5903                  */
5904                 mutex_enter(SQLOCK(outer));
5905                 if (outer->sq_flags & (flags & ~SQ_WRITER)) {
5906                         mutex_exit(SQLOCK(outer));
5907                         outer_exit(outer);
5908                         goto retry;
5909                 }
5910                 mutex_exit(SQLOCK(outer));
5911         }
5912 }

5914 /*
```

```
 * Drop the write access at the outer perimeter.
 * 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)
```

```
6047 {
6048         ASSERT(MUTEX_HELD(SQLOCK(outer)));
6049
6050         mp->b_prev = (mblk_t *)func;
6051         mp->b_queue = q;
6052         mp->b_next = NULL;
6053         outer->sq_count++;      /* Decremented when dequeued */
6054         ASSERT(outer->sq_count != 0);   /* Wraparound */
6055         if (outer->sq_evhead == NULL) {
6056                 /* First message. */
6057                 outer->sq_evhead = outer->sq_evtail = mp;
6058                 outer->sq_flags |= SQ_EVENTS;
6059                 mutex_exit(SQLOCK(outer));
6060                 STRSTAT(qwr_outer);
6061                 (void) taskq_dispatch(streams_taskq,
6062                     (task_func_t *)qwriter_outer_service, outer, TQ_SLEEP);
6063         } else {
6064                 ASSERT(outer->sq_flags & SQ_EVENTS);
6065                 outer->sq_evtail->b_next = mp;
6066                 outer->sq_evtail = mp;
6067                 mutex_exit(SQLOCK(outer));
6068         }
6069 }
6070
6071 /*
6072  * Try and upgrade to write access at the outer perimeter. If this can
6073  * not be done without blocking then queue the callback to be done
6074  * by the qwriter_outer_thread.
6075  *
6076  * This routine can only be called from put or service procedures plus
6077  * asynchronous callback routines that have properly entered the queue (with
6078  * entersq). Thus qwriter(OUTER) assumes the caller has one claim on the syncq
6079  * associated with q.
6080  */
6081 void
6082 qwriter_outer(queue_t *q, mblk_t *mp, void (*func)())
6083 {
6084         syncq_t *osq, *sq, *outer;
6085         int     failed;
6086         uint16_t flags;
6087
6088         osq = q->q_syncq;
6089         outer = osq->sq_outer;
6090         if (outer == NULL)
6091                 panic("qwriter(PERIM_OUTER): no outer perimeter");
6092         ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
6093             outer->sq_oprev != NULL);
6094
6095         mutex_enter(SQLOCK(outer));
6096         flags = outer->sq_flags;
6097         /*
6098          * If some thread is traversing sq_next, or if we are blocked by
6099          * outer_insert or outer_remove, or if the we already have queued
6100          * callbacks, then queue this callback for later processing.
6101          *
6102          * Also queue the qwriter for an interrupt thread in order
6103          * to reduce the time spent running at high IPL.
6104          * to identify there are events.
6105          */
6106         if ((flags & SQ_GOAWAY) || (curthread->t_pri >= kpreemptpri)) {
6107                 /*
6108                  * Queue the become_writer request.
6109                  * The queueing is atomic under SQLOCK(outer) in order
6110                  * to synchronize with outer_exit.
6111                  * queue_writer will drop the outer SQLOCK
6112                  */
```



```
6113                 if (flags & SQ_BLOCKED) {
6114                         /* Must set SQ_WRITER on inner perimeter */
6115                         mutex_enter(SQLOCK(osq));
6116                         osq->sq_flags |= SQ_WRITER;
6117                         mutex_exit(SQLOCK(osq));
6118                 } else {
6119                         if (!(flags & SQ_WRITER)) {
6120                                 /*
6121                                  * The outer could have been SQ_BLOCKED thus
6122                                  * SQ_WRITER might not be set on the inner.
6123                                  */
6124                                 mutex_enter(SQLOCK(osq));
6125                                 osq->sq_flags |= SQ_WRITER;
6126                                 mutex_exit(SQLOCK(osq));
6127                         }
6128                         ASSERT(osq->sq_flags & SQ_WRITER);
6129                 }
6130                 queue_writer(outer, func, q, mp);
6131                 return;
6132         }
6133         /*
6134          * We are half-way to exclusive access to the outer perimeter.
6135          * Prevent any outer_enter, qwriter(OUTER), or outer_insert/remove
6136          * while the inner syncqs are traversed.
6137          */
6138         outer->sq_count++;
6139         ASSERT(outer->sq_count != 0);   /* wraparound */
6140         flags |= SQ_WRITER;
6141         /*
6142          * Check if we can run the function immediately. Mark all
6143          * syncqs with the writer flag to prevent new entries into
6144          * put and service procedures.
6145          *
6146          * Set SQ_WRITER on all the inner syncqs while holding
6147          * the SQLOCK on the outer syncq. This ensures that the changing
6148          * of SQ_WRITER is atomic under the outer SQLOCK.
6149          */
6150         failed = 0;
6151         for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
6152                 uint16_t count;
6153                 uint_t  maxcnt = (sq == osq) ? 1 : 0;
6154
6155                 mutex_enter(SQLOCK(sq));
6156                 count = sq->sq_count;
6157                 SQ_PUTLOCKS_ENTER(sq);
6158                 SUM_SQ_PUTCOUNTS(sq, count);
6159                 if (sq->sq_count > maxcnt)
6160                         failed = 1;
6161                 sq->sq_flags |= SQ_WRITER;
6162                 SQ_PUTLOCKS_EXIT(sq);
6163                 mutex_exit(SQLOCK(sq));
6164         }
6165         if (failed) {
6166                 /*
6167                  * Some other thread has a read claim on the outer perimeter.
6168                  * Queue the callback for deferred processing.
6169                  *
6170                  * queue_writer will set SQ_QUEUED before we drop SQ_WRITER
6171                  * so that other qwriter(OUTER) calls will queue their
6172                  * callbacks as well. queue_writer increments sq_count so we
6173                  * decrement to compensate for the our increment.
6174                  *
6175                  * Dropping SQ_WRITER enables the writer thread to work
6176                  * on this outer perimeter.
6177                  */
6178                 outer->sq_flags = flags;
```

```
6179                queue_writer(outer, func, q, mp);
6180                /* queue_writer dropper the lock */
6181                mutex_enter(SQLOCK(outer));
6182                ASSERT(outer->sq_count > 0);
6183                outer->sq_count--;
6184                ASSERT(outer->sq_flags & SQ_WRITER);
6185                flags = outer->sq_flags;
6186                flags &= ~SQ_WRITER;
6187                if (flags & SQ_WANTWAKEUP) {
6188                        flags &= ~SQ_WANTWAKEUP;
6189                        cv_broadcast(&outer->sq_wait);
6190                }
6191                outer->sq_flags = flags;
6192                mutex_exit(SQLOCK(outer));
6193                return;
6194        } else {
6195                outer->sq_flags = flags;
6196                mutex_exit(SQLOCK(outer));
6197        }
6198
6199        /* Can run it immediately */
6200        (*func)(q, mp);
6201
6202        outer_exit(outer);
6203 }
6204
6205 /*
6206  * Dequeue all writer callbacks from the outer perimeter and run them.
6207  */
6208 static void
6209 write_now(syncq_t *outer)
6210 {
6211        mblk_t          *mp;
6212        queue_t         *q;
6213        void    (*func)();
6214
6215        ASSERT(MUTEX_HELD(SQLOCK(outer)));
6216        ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
6217            outer->sq_oprev != NULL);
6218        while ((mp = outer->sq_evhead) != NULL) {
6219                /*
6220                 * queues cannot be placed on the queuelist on the outer
6221                 * perimeter.
6222                 */
6223                ASSERT(!(outer->sq_flags & SQ_MESSAGES));
6224                ASSERT((outer->sq_flags & SQ_EVENTS));
6225
6226                outer->sq_evhead = mp->b_next;
6227                if (outer->sq_evhead == NULL) {
6228                        outer->sq_evtail = NULL;
6229                        outer->sq_flags &= ~SQ_EVENTS;
6230                }
6231                ASSERT(outer->sq_count != 0);
6232                outer->sq_count--;      /* Incremented when enqueued. */
6233                mutex_exit(SQLOCK(outer));
6234                /*
6235                 * Drop the message if the queue is closing.
6236                 * Make sure that the queue is "claimed" when the callback
6237                 * is run in order to satisfy various ASSERTs.
6238                 */
6239                q = mp->b_queue;
6240                func = (void (*)())mp->b_prev;
6241                ASSERT(func != NULL);
6242                mp->b_next = mp->b_prev = NULL;
6243                if (q->q_flag & QWCLOSE) {
6244                        freemsg(mp);
```



```
6245                } else {
6246                        claimq(q);
6247                        (*func)(q, mp);
6248                        releaseq(q);
6249                }
6250                mutex_enter(SQLOCK(outer));
6251        }
6252        ASSERT(MUTEX_HELD(SQLOCK(outer)));
6253 }
6254
6255 /*
6256  * The list of messages on the inner syncq is effectively hashed
6257  * by destination queue.  These destination queues are doubly
6258  * linked lists (hopefully) in priority order.  Messages are then
6259  * put on the queue referenced by the q_sqhead/q_sqtail elements.
6260  * Additional messages are linked together by the b_next/b_prev
6261  * elements in the mblk, with (similar to putq()) the first message
6262  * having a NULL b_prev and the last message having a NULL b_next.
6263  *
6264  * Events, such as qwriter callbacks, are put onto a list in FIFO
6265  * order referenced by sq_evhead, and sq_evtail.  This is a singly
6266  * linked list, and messages here MUST be processed in the order queued.
6267  */
6268
6269 /*
6270  * Run the events on the syncq event list (sq_evhead).
6271  * Assumes there is only one claim on the syncq, it is
6272  * already exclusive (SQ_EXCL set), and the SQLOCK held.
6273  * Messages here are processed in order, with the SQ_EXCL bit
6274  * held all the way through till the last message is processed.
6275  */
6276 void
6277 sq_run_events(syncq_t *sq)
6278 {
6279        mblk_t          *bp;
6280        queue_t         *qp;
6281        uint16_t        flags = sq->sq_flags;
6282        void            (*func)();
6283
6284        ASSERT(MUTEX_HELD(SQLOCK(sq)));
6285        ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6286            sq->sq_oprev == NULL) ||
6287            (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6288            sq->sq_oprev != NULL));
6289
6290        ASSERT(flags & SQ_EXCL);
6291        ASSERT(sq->sq_count == 1);
6292
6293        /*
6294         * We need to process all of the events on this list.  It
6295         * is possible that new events will be added while we are
6296         * away processing a callback, so on every loop, we start
6297         * back at the beginning of the list.
6298         */
6299        /*
6300         * We have to reaccess sq_evhead since there is a
6301         * possibility of a new entry while we were running
6302         * the callback.
6303         */
6304        for (bp = sq->sq_evhead; bp != NULL; bp = sq->sq_evhead) {
6305                ASSERT(bp->b_queue->q_syncq == sq);
6306                ASSERT(sq->sq_flags & SQ_EVENTS);
6307
6308                qp = bp->b_queue;
6309                func = (void (*)())bp->b_prev;
6310                ASSERT(func != NULL);
```

```
6312                 /*
6313                  * Messages from the event queue must be taken off in
6314                  * FIFO order.
6315                  */
6316                 ASSERT(sq->sq_evhead == bp);
6317                 sq->sq_evhead = bp->b_next;

6319                 if (bp->b_next == NULL) {
6320                         /* Deleting last */
6321                         ASSERT(sq->sq_evtail == bp);
6322                         sq->sq_evtail = NULL;
6323                         sq->sq_flags &= ~SQ_EVENTS;
6324                 }
6325                 bp->b_prev = bp->b_next = NULL;
6326                 ASSERT(bp->b_datap->db_ref != 0);

6328                 mutex_exit(SQLOCK(sq));

6330                 (*func)(qp, bp);

6332                 mutex_enter(SQLOCK(sq));
6333                 /*
6334                  * re-read the flags, since they could have changed.
6335                  */
6336                 flags = sq->sq_flags;
6337                 ASSERT(flags & SQ_EXCL);
6338         }
6339         ASSERT(sq->sq_evhead == NULL && sq->sq_evtail == NULL);
6340         ASSERT(!(sq->sq_flags & SQ_EVENTS));

6342         if (flags & SQ_WANTWAKEUP) {
6343                 flags &= ~SQ_WANTWAKEUP;
6344                 cv_broadcast(&sq->sq_wait);
6345         }
6346         if (flags & SQ_WANTEXWAKEUP) {
6347                 flags &= ~SQ_WANTEXWAKEUP;
6348                 cv_broadcast(&sq->sq_exitwait);
6349         }
6350         sq->sq_flags = flags;
6351 }

6353 /*
6354  * Put messages on the event list.
6355  * If we can go exclusive now, do so and process the event list, otherwise
6356  * let the last claim service this list (or wake the sqthread).
6357  * This procedure assumes SQLOCK is held.  To run the event list, it
6358  * must be called with no claims.
6359  */
6360 static void
6361 sqfill_events(syncq_t *sq, queue_t *q, mblk_t *mp, void (*func)())
6362 {
6363         uint16_t count;

6365         ASSERT(MUTEX_HELD(SQLOCK(sq)));
6366         ASSERT(func != NULL);

6368         /*
6369          * This is a callback.  Add it to the list of callbacks
6370          * and see about upgrading.
6371          */
6372         mp->b_prev = (mblk_t *)func;
6373         mp->b_queue = q;
6374         mp->b_next = NULL;
6375         if (sq->sq_evhead == NULL) {
6376                 sq->sq_evhead = sq->sq_evtail = mp;
```



```
6377                 sq->sq_flags |= SQ_EVENTS;
6378         } else {
6379                 ASSERT(sq->sq_evtail != NULL);
6380                 ASSERT(sq->sq_evtail->b_next == NULL);
6381                 ASSERT(sq->sq_flags & SQ_EVENTS);
6382                 sq->sq_evtail->b_next = mp;
6383                 sq->sq_evtail = mp;
6384         }
6385         /*
6386          * We have set SQ_EVENTS, so threads will have to
6387          * unwind out of the perimeter, and new entries will
6388          * not grab a putlock.  But we still need to know
6389          * how many threads have already made a claim to the
6390          * syncq, so grab the putlocks, and sum the counts.
6391          * If there are no claims on the syncq, we can upgrade
6392          * to exclusive, and run the event list.
6393          * NOTE: We hold the SQLOCK, so we can just grab the
6394          * putlocks.
6395          */
6396         count = sq->sq_count;
6397         SQ_PUTLOCKS_ENTER(sq);
6398         SUM_SQ_PUTCOUNTS(sq, count);
6399         /*
6400          * We have no claim, so we need to check if there
6401          * are no others, then we can upgrade.
6402          */
6403         /*
6404          * There are currently no claims on
6405          * the syncq by this thread (at least on this entry). The thread who has
6406          * the claim should drain syncq.
6407          */
6408         if (count > 0) {
6409                 /*
6410                  * Can't upgrade - other threads inside.
6411                  */
6412                 SQ_PUTLOCKS_EXIT(sq);
6413                 mutex_exit(SQLOCK(sq));
6414                 return;
6415         }
6416         /*
6417          * Need to set SQ_EXCL and make a claim on the syncq.
6418          */
6419         ASSERT((sq->sq_flags & SQ_EXCL) == 0);
6420         sq->sq_flags |= SQ_EXCL;
6421         ASSERT(sq->sq_count == 0);
6422         sq->sq_count++;
6423         SQ_PUTLOCKS_EXIT(sq);

6425         /* Process the events list */
6426         sq_run_events(sq);

6428         /*
6429          * Release our claim...
6430          */
6431         sq->sq_count--;

6433         /*
6434          * And release SQ_EXCL.
6435          * We don't need to acquire the putlocks to release
6436          * SQ_EXCL, since we are exclusive, and hold the SQLOCK.
6437          */
6438         sq->sq_flags &= ~SQ_EXCL;

6440         /*
6441          * sq_run_events should have released SQ_EXCL
6442          */
```

```
6443         ASSERT(!(sq->sq_flags & SQ_EXCL));

6445         /*
6446          * If anything happened while we were running the
6447          * events (or was there before), we need to process
6448          * them now.  We shouldn't be exclusive sine we
6449          * released the perimeter above (plus, we asserted
6450          * for it).
6451          */
6452         if (!(sq->sq_flags & SQ_STAYAWAY) && (sq->sq_flags & SQ_QUEUED))
6453                 drain_syncq(sq);
6454         else
6455                 mutex_exit(SQLOCK(sq));
6456 }

6458 /*
6459  * Perform delayed processing. The caller has to make sure that it is safe
6460  * to enter the syncq (e.g. by checking that none of the SQ_STAYAWAY bits are
6461  * set).
6462  *
6463  * Assume that the caller has NO claims on the syncq.  However, a claim
6464  * on the syncq does not indicate that a thread is draining the syncq.
6465  * There may be more claims on the syncq than there are threads draining
6466  * (i.e.  #_threads_draining <= sq_count)
6467  *
6468  * drain_syncq has to terminate when one of the SQ_STAYAWAY bits gets set
6469  * in order to preserve qwriter(OUTER) ordering constraints.
6470  *
6471  * sq_putcount only needs to be checked when dispatching the queued
6472  * writer call for CIPUT sync queue, but this is handled in sq_run_events.
6473  */
6474 void
6475 drain_syncq(syncq_t *sq)
6476 {
6477         queue_t         *qp;
6478         uint16_t        count;
6479         uint16_t        type = sq->sq_type;
6480         uint16_t        flags = sq->sq_flags;
6481         boolean_t       bg_service = sq->sq_svcflags & SQ_SERVICE;

6483         TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_START,
6484             "drain_syncq start:%p", sq);
6485         ASSERT(MUTEX_HELD(SQLOCK(sq)));
6486         ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6487             sq->sq_oprev == NULL) ||
6488             (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6489             sq->sq_oprev != NULL));

6491         /*
6492          * Drop SQ_SERVICE flag.
6493          */
6494         if (bg_service)
6495                 sq->sq_svcflags &= ~SQ_SERVICE;

6497         /*
6498          * If SQ_EXCL is set, someone else is processing this syncq - let him
6499          * finish the job.
6500          */
6501         if (flags & SQ_EXCL) {
6502                 if (bg_service) {
6503                         ASSERT(sq->sq_servcount != 0);
6504                         sq->sq_servcount--;
6505                 }
6506                 mutex_exit(SQLOCK(sq));
6507                 return;
6508         }
```



```
6510         /*
6511          * This routine can be called by a background thread if
6512          * it was scheduled by a hi-priority thread.  SO, if there are
6513          * NOT messages queued, return (remember, we have the SQLOCK,
6514          * and it cannot change until we release it). Wakeup any waiters also.
6515          */
6516         if (!(flags & SQ_QUEUED)) {
6517                 if (flags & SQ_WANTWAKEUP) {
6518                         flags &= ~SQ_WANTWAKEUP;
6519                         cv_broadcast(&sq->sq_wait);
6520                 }
6521                 if (flags & SQ_WANTEXWAKEUP) {
6522                         flags &= ~SQ_WANTEXWAKEUP;
6523                         cv_broadcast(&sq->sq_exitwait);
6524                 }
6525                 sq->sq_flags = flags;
6526                 if (bg_service) {
6527                         ASSERT(sq->sq_servcount != 0);
6528                         sq->sq_servcount--;
6529                 }
6530                 mutex_exit(SQLOCK(sq));
6531                 return;
6532         }

6534         /*
6535          * If this is not a concurrent put perimeter, we need to
6536          * become exclusive to drain.  Also, if not CIPUT, we would
6537          * not have acquired a putlock, so we don't need to check
6538          * the putcounts.  If not entering with a claim, we test
6539          * for sq_count == 0.
6540          */
6541         type = sq->sq_type;
6542         if (!(type & SQ_CIPUT)) {
6543                 if (sq->sq_count > 1) {
6544                         if (bg_service) {
6545                                 ASSERT(sq->sq_servcount != 0);
6546                                 sq->sq_servcount--;
6547                         }
6548                         mutex_exit(SQLOCK(sq));
6549                         return;
6550                 }
6551                 sq->sq_flags |= SQ_EXCL;
6552         }

6554         /*
6555          * This is where we make a claim to the syncq.
6556          * This can either be done by incrementing a putlock, or
6557          * the sq_count.  But since we already have the SQLOCK
6558          * here, we just bump the sq_count.
6559          *
6560          * Note that after we make a claim, we need to let the code
6561          * fall through to the end of this routine to clean itself
6562          * up.  A return in the while loop will put the syncq in a
6563          * very bad state.
6564          */
6565         sq->sq_count++;
6566         ASSERT(sq->sq_count != 0);      /* wraparound */

6568         while ((flags = sq->sq_flags) & SQ_QUEUED) {
6569                 /*
6570                  * If we are told to stayaway or went exclusive,
6571                  * we are done.
6572                  */
6573                 if (flags & (SQ_STAYAWAY)) {
6574                         break;
```

```
6575                }

6577                /*
6578                 * If there are events to run, do so.
6579                 * We have one claim to the syncq, so if there are
6580                 * more than one, other threads are running.
6581                 */
6582                if (sq->sq_evhead != NULL) {
6583                        ASSERT(sq->sq_flags & SQ_EVENTS);

6585                        count = sq->sq_count;
6586                        SQ_PUTLOCKS_ENTER(sq);
6587                        SUM_SQ_PUTCOUNTS(sq, count);
6588                        if (count > 1) {
6589                                SQ_PUTLOCKS_EXIT(sq);
6590                                /* Can't upgrade - other threads inside */
6591                                break;
6592                        }
6593                        ASSERT((flags & SQ_EXCL) == 0);
6594                        sq->sq_flags = flags | SQ_EXCL;
6595                        SQ_PUTLOCKS_EXIT(sq);
6596                        /*
6597                         * we have the only claim, run the events,
6598                         * sq_run_events will clear the SQ_EXCL flag.
6599                         */
6600                        sq_run_events(sq);

6602                        /*
6603                         * If this is a CIPUT perimeter, we need
6604                         * to drop the SQ_EXCL flag so we can properly
6605                         * continue draining the syncq.
6606                         */
6607                        if (type & SQ_CIPUT) {
6608                                ASSERT(sq->sq_flags & SQ_EXCL);
6609                                sq->sq_flags &= ~SQ_EXCL;
6610                        }

6612                        /*
6613                         * And go back to the beginning just in case
6614                         * anything changed while we were away.
6615                         */
6616                        ASSERT((sq->sq_flags & SQ_EXCL) || (type & SQ_CIPUT));
6617                        continue;
6618                }

6620                ASSERT(sq->sq_evhead == NULL);
6621                ASSERT(!(sq->sq_flags & SQ_EVENTS));

6623                /*
6624                 * Find the queue that is not draining.
6625                 *
6626                 * q_draining is protected by QLOCK which we do not hold.
6627                 * But if it was set, then a thread was draining, and if it gets
6628                 * cleared, then it was because the thread has successfully
6629                 * drained the syncq, or a GOAWAY state occurred. For the GOAWAY
6630                 * state to happen, a thread needs the SQLOCK which we hold, and
6631                 * if there was such a flag, we would have already seen it.
6632                 */

6634                for (qp = sq->sq_head;
6635                    qp != NULL && (qp->q_draining ||
6636                    (qp->q_sqflags & Q_SQDRAINING));
6637                    qp = qp->q_sqnext)
6638                        ;

6640                if (qp == NULL)
```



```
6641                        break;

6643                /*
6644                 * We have a queue to work on, and we hold the
6645                 * SQLOCK and one claim, call qdrain_syncq.
6646                 * This means we need to release the SQLOCK and
6647                 * acquire the QLOCK (OK since we have a claim).
6648                 * Note that qdrain_syncq will actually dequeue
6649                 * this queue from the sq_head list when it is
6650                 * convinced all the work is done and release
6651                 * the QLOCK before returning.
6652                 */
6653                qp->q_sqflags |= Q_SQDRAINING;
6654                mutex_exit(SQLOCK(sq));
6655                mutex_enter(QLOCK(qp));
6656                qdrain_syncq(sq, qp);
6657                mutex_enter(SQLOCK(sq));

6659                /* The queue is drained */
6660                ASSERT(qp->q_sqflags & Q_SQDRAINING);
6661                qp->q_sqflags &= ~Q_SQDRAINING;
6662                /*
6663                 * NOTE: After this point qp should not be used since it may be
6664                 * closed.
6665                 */
6666        }

6668        ASSERT(MUTEX_HELD(SQLOCK(sq)));
6669        flags = sq->sq_flags;

6671        /*
6672         * sq->sq_head cannot change because we hold the
6673         * sqlock. However, a thread CAN decide that it is no longer
6674         * going to drain that queue.  However, this should be due to
6675         * a GOAWAY state, and we should see that here.
6676         *
6677         * This loop is not very efficient. One solution may be adding a second
6678         * pointer to the "draining" queue, but it is difficult to do when
6679         * queues are inserted in the middle due to priority ordering. Another
6680         * possibility is to yank the queue out of the sq list and put it onto
6681         * the "draining list" and then put it back if it can't be drained.
6682         */

6684        ASSERT((sq->sq_head == NULL) || (flags & SQ_GOAWAY) ||
6685            (type & SQ_CI) || sq->sq_head->q_draining);

6687        /* Drop SQ_EXCL for non-CIPUT perimeters */
6688        if (!(type & SQ_CIPUT))
6689                flags &= ~SQ_EXCL;
6690        ASSERT((flags & SQ_EXCL) == 0);

6692        /* Wake up any waiters. */
6693        if (flags & SQ_WANTWAKEUP) {
6694                flags &= ~SQ_WANTWAKEUP;
6695                cv_broadcast(&sq->sq_wait);
6696        }
6697        if (flags & SQ_WANTEXWAKEUP) {
6698                flags &= ~SQ_WANTEXWAKEUP;
6699                cv_broadcast(&sq->sq_exitwait);
6700        }
6701        sq->sq_flags = flags;

6703        ASSERT(sq->sq_count != 0);
6704        /* Release our claim. */
6705        sq->sq_count--;
```

```
6707         if (bg_service) {
6708                 ASSERT(sq->sq_servcount != 0);
6709                 sq->sq_servcount--;
6710         }

6712         mutex_exit(SQLOCK(sq));

6714         TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_END,
6715             "drain_syncq end:%p", sq);
6716 }


6719 /*
6720  *
6721  * qdrain_syncq can be called (currently) from only one of two places:
6722  *      drain_syncq
6723  *      putnext  (or some variation of it).
6724  * and eventually
6725  *      qwait(_sig)
6726  *
6727  * If called from drain_syncq, we found it in the list of queues needing
6728  * service, so there is work to be done (or it wouldn't be in the list).
6729  *
6730  * If called from some putnext variation, it was because the
6731  * perimeter is open, but messages are blocking a putnext and
6732  * there is not a thread working on it.  Now a thread could start
6733  * working on it while we are getting ready to do so ourself, but
6734  * the thread would set the q_draining flag, and we can spin out.
6735  *
6736  * As for qwait(_sig), I think I shall let it continue to call
6737  * drain_syncq directly (after all, it will get here eventually).
6738  *
6739  * qdrain_syncq has to terminate when:
6740  * - one of the SQ_STAYAWAY bits gets set to preserve qwriter(OUTER) ordering
6741  * - SQ_EVENTS gets set to preserve qwriter(INNER) ordering
6742  *
6743  * ASSUMES:
6744  *      One claim
6745  *      QLOCK held
6746  *      SQLOCK not held
6747  *      Will release QLOCK before returning
6748  */
6749 void
6750 qdrain_syncq(syncq_t *sq, queue_t *q)
6751 {
6752         mblk_t          *bp;
6753 #ifdef DEBUG
6754         uint16_t        count;
6755 #endif

6757         TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_START,
6758             "drain_syncq start:%p", sq);
6759         ASSERT(q->q_syncq == sq);
6760         ASSERT(MUTEX_HELD(QLOCK(q)));
6761         ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
6762         /*
6763          * For non-CIPUT perimeters, we should be called with the exclusive bit
6764          * set already. For CIPUT perimeters, we will be doing a concurrent
6765          * drain, so it better not be set.
6766          */
6767         ASSERT((sq->sq_flags & (SQ_EXCL|SQ_CIPUT)));
6768         ASSERT(!((sq->sq_type & SQ_CIPUT) && (sq->sq_flags & SQ_EXCL)));
6769         ASSERT((sq->sq_type & SQ_CIPUT) || (sq->sq_flags & SQ_EXCL));
6770         /*
6771          * All outer pointers are set, or none of them are
6772          */
```



```
6773         ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6774             sq->sq_oprev == NULL) ||
6775             (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6776             sq->sq_oprev != NULL));
6777 #ifdef DEBUG
6778         count = sq->sq_count;
6779         /*
6780          * This is OK without the putlocks, because we have one
6781          * claim either from the sq_count, or a putcount.  We could
6782          * get an erroneous value from other counts, but ours won't
6783          * change, so one way or another, we will have at least a
6784          * value of one.
6785          */
6786         SUM_SQ_PUTCOUNTS(sq, count);
6787         ASSERT(count >= 1);
6788 #endif /* DEBUG */

6790         /*
6791          * The first thing to do is find out if a thread is already draining
6792          * this queue. If so, we are done, just return.
6793          */
6794         if (q->q_draining) {
6795                 mutex_exit(QLOCK(q));
6796                 return;
6797         }

6799         /*
6800          * If the perimeter is exclusive, there is nothing we can do right now,
6801          * go away. Note that there is nothing to prevent this case from
6802          * changing right after this check, but the spin-out will catch it.
6803          */

6805         /* Tell other threads that we are draining this queue */
6806         q->q_draining = 1;      /* Protected by QLOCK */

6808         /*
6809          * If there is nothing to do, clear QFULL as necessary. This caters for
6810          * the case where an empty queue was enqueued onto the syncq.
6811          */
6812         if (q->q_sqhead == NULL) {
6813                 ASSERT(q->q_syncqmsgs == 0);
6814                 mutex_exit(QLOCK(q));
6815                 clr_qfull(q);
6816                 mutex_enter(QLOCK(q));
6817         }

6819         /*
6820          * Note that q_sqhead must be re-checked here in case another message
6821          * was enqueued whilst QLOCK was dropped during the call to clr_qfull.
6822          */
6823         for (bp = q->q_sqhead; bp != NULL; bp = q->q_sqhead) {
6824                 /*
6825                  * Because we can enter this routine just because a putnext is
6826                  * blocked, we need to spin out if the perimeter wants to go
6827                  * exclusive as well as just blocked. We need to spin out also
6828                  * if events are queued on the syncq.
6829                  * Don't check for SQ_EXCL, because non-CIPUT perimeters would
6830                  * set it, and it can't become exclusive while we hold a claim.
6831                  */
6832                 if (sq->sq_flags & (SQ_STAYAWAY | SQ_EVENTS)) {
6833                         break;
6834                 }

6836 #ifdef DEBUG
6837                 /*
6838                  * Since we are in qdrain_syncq, we already know the queue,
```

```
6839                 * but for sanity, we want to check this against the qp that
6840                 * was passed in by bp->b_queue.
6841                 */

6843                ASSERT(bp->b_queue == q);
6844                ASSERT(bp->b_queue->q_syncq == sq);
6845                bp->b_queue = NULL;

6847                /*
6848                 * We would have the following check in the DEBUG code:
6849                 *
6850                 * if (bp->b_prev != NULL)  {
6851                 *      ASSERT(bp->b_prev == (void (*)())q->q_qinfo->qi_putp);
6852                 * }
6853                 *
6854                 * This can't be done, however, since IP modifies qinfo
6855                 * structure at run-time (switching between IPv4 qinfo and IPv6
6856                 * qinfo), invalidating the check.
6857                 * So the assignment to func is left here, but the ASSERT itself
6858                 * is removed until the whole issue is resolved.
6859                 */
6860 #endif
6861                ASSERT(q->q_sqhead == bp);
6862                q->q_sqhead = bp->b_next;
6863                bp->b_prev = bp->b_next = NULL;
6864                ASSERT(q->q_syncqmsgs > 0);
6865                mutex_exit(QLOCK(q));

6867                ASSERT(bp->b_datap->db_ref != 0);

6869                (void) (*q->q_qinfo->qi_putp)(q, bp);

6871                mutex_enter(QLOCK(q));

6873                /*
6874                 * q_syncqmsgs should only be decremented after executing the
6875                 * put procedure to avoid message re-ordering. This is due to an
6876                 * optimisation in putnext() which can call the put procedure
6877                 * directly if it sees q_syncqmsgs == 0 (despite Q_SQQUEUED
6878                 * being set).
6879                 *
6880                 * We also need to clear QFULL in the next service procedure
6881                 * queue if this is the last message destined for that queue.
6882                 *
6883                 * It would make better sense to have some sort of tunable for
6884                 * the low water mark, but these semantics are not yet defined.
6885                 * So, alas, we use a constant.
6886                 */
6887                if (--q->q_syncqmsgs == 0) {
6888                        mutex_exit(QLOCK(q));
6889                        clr_qfull(q);
6890                        mutex_enter(QLOCK(q));
6891                }

6893                /*
6894                 * Always clear SQ_EXCL when CIPUT in order to handle
6895                 * qwriter(INNER). The putp() can call qwriter and get exclusive
6896                 * access IFF this is the only claim. So, we need to test for
6897                 * this possibility, acquire the mutex and clear the bit.
6898                 */
6899                if ((sq->sq_type & SQ_CIPUT) && (sq->sq_flags & SQ_EXCL)) {
6900                        mutex_enter(SQLOCK(sq));
6901                        sq->sq_flags &= ~SQ_EXCL;
6902                        mutex_exit(SQLOCK(sq));
6903                }
6904        }
```



```
6906        /*
6907         * We should either have no messages on this queue, or we were told to
6908         * goaway by a waiter (which we will wake up at the end of this
6909         * function).
6910         */
6911        ASSERT((q->q_sqhead == NULL) ||
6912            (sq->sq_flags & (SQ_STAYAWAY | SQ_EVENTS)));

6914        ASSERT(MUTEX_HELD(QLOCK(q)));
6915        ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));

6917        /* Remove the q from the syncq list if all the messages are drained. */
6918        if (q->q_sqhead == NULL) {
6919                ASSERT(q->q_syncqmsgs == 0);
6920                mutex_enter(SQLOCK(sq));
6921                if (q->q_sqflags & Q_SQQUEUED)
6922                        SQRM_Q(sq, q);
6923                mutex_exit(SQLOCK(sq));
6924                /*
6925                 * Since the queue is removed from the list, reset its priority.
6926                 */
6927                q->q_spri = 0;
6928        }

6930        /*
6931         * Remember, the q_draining flag is used to let another thread know
6932         * that there is a thread currently draining the messages for a queue.
6933         * Since we are now done with this queue (even if there may be messages
6934         * still there), we need to clear this flag so some thread will work on
6935         * it if needed.
6936         */
6937        ASSERT(q->q_draining);
6938        q->q_draining = 0;

6940        /* Called with a claim, so OK to drop all locks. */
6941        mutex_exit(QLOCK(q));

6943        TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_END,
6944            "drain_syncq end:%p", sq);
6945 }
6946 /* END OF QDRAIN_SYNCQ  */


6949 /*
6950  * This is the mate to qdrain_syncq, except that it is putting the message onto
6951  * the queue instead of draining. Since the message is destined for the queue
6952  * that is selected, there is no need to identify the function because the
6953  * message is intended for the put routine for the queue. For debug kernels,
6954  * this routine will do it anyway just in case.
6955  *
6956  * After the message is enqueued on the syncq, it calls putnext_tail()
6957  * which will schedule a background thread to actually process the message.
6958  *
6959  * Assumes that there is a claim on the syncq (sq->sq_count > 0) and
6960  * SQLOCK(sq) and QLOCK(q) are not held.
6961  */
6962 void
6963 qfill_syncq(syncq_t *sq, queue_t *q, mblk_t *mp)
6964 {
6965        ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
6966        ASSERT(MUTEX_NOT_HELD(QLOCK(q)));
6967        ASSERT(sq->sq_count > 0);
6968        ASSERT(q->q_syncq == sq);
6969        ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6970            sq->sq_oprev == NULL) ||
```

```
6971             (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6972             sq->sq_oprev != NULL));

6974         mutex_enter(QLOCK(q));

6976 #ifdef DEBUG
6977         /*
6978          * This is used for debug in the qfill_syncq/qdrain_syncq case
6979          * to trace the queue that the message is intended for.  Note
6980          * that the original use was to identify the queue and function
6981          * to call on the drain.  In the new syncq, we have the context
6982          * of the queue that we are draining, so call it's putproc and
6983          * don't rely on the saved values.  But for debug this is still
6984          * useful information.
6985          */
6986         mp->b_prev = (mblk_t *)q->q_qinfo->qi_putp;
6987         mp->b_queue = q;
6988         mp->b_next = NULL;
6989 #endif
6990         ASSERT(q->q_syncq == sq);
6991         /*
6992          * Enqueue the message on the list.
6993          * SQPUT_MP() accesses q_syncqmsgs.  We are already holding QLOCK to
6994          * protect it.  So it's ok to acquire SQLOCK after SQPUT_MP().
6995          */
6996         SQPUT_MP(q, mp);
6997         mutex_enter(SQLOCK(sq));

6999         /*
7000          * And queue on syncq for scheduling, if not already queued.
7001          * Note that we need the SQLOCK for this, and for testing flags
7002          * at the end to see if we will drain.  So grab it now, and
7003          * release it before we call qdrain_syncq or return.
7004          */
7005         if (!(q->q_sqflags & Q_SQQUEUED)) {
7006                 q->q_spri = curthread->t_pri;
7007                 SQPUT_Q(sq, q);
7008         }
7009 #ifdef DEBUG
7010         else {
7011                 /*
7012                  * All of these conditions MUST be true!
7013                  */
7014                 ASSERT(sq->sq_tail != NULL);
7015                 if (sq->sq_tail == sq->sq_head) {
7016                         ASSERT((q->q_sqprev == NULL) &&
7017                             (q->q_sqnext == NULL));
7018                 } else {
7019                         ASSERT((q->q_sqprev != NULL) ||
7020                             (q->q_sqnext != NULL));
7021                 }
7022                 ASSERT(sq->sq_flags & SQ_QUEUED);
7023                 ASSERT(q->q_syncqmsgs != 0);
7024                 ASSERT(q->q_sqflags & Q_SQQUEUED);
7025         }
7026 #endif
7027         mutex_exit(QLOCK(q));
7028         /*
7029          * SQLOCK is still held, so sq_count can be safely decremented.
7030          */
7031         sq->sq_count--;

7033         putnext_tail(sq, q, 0);
7034         /* Should not reference sq or q after this point. */
7035 }
```



```
7037 /*  End of qfill_syncq  */

7039 /*
7040  * Remove all messages from a syncq (if qp is NULL) or remove all messages
7041  * that would be put into qp by drain_syncq.
7042  * Used when deleting the syncq (qp == NULL) or when detaching
7043  * a queue (qp != NULL).
7044  * Return non-zero if one or more messages were freed.
7045  *
7046  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
7047  * sq_putlocks are used.
7048  *
7049  * NOTE: This function assumes that it is called from the close() context and
7050  * that all the queues in the syncq are going away. For this reason it doesn't
7051  * acquire QLOCK for modifying q_sqhead/q_sqtail fields. This assumption is
7052  * currently valid, but it is useful to rethink this function to behave properly
7053  * in other cases.
7054  */
7055 int
7056 flush_syncq(syncq_t *sq, queue_t *qp)
7057 {
7058         mblk_t          *bp, *mp_head, *mp_next, *mp_prev;
7059         queue_t         *q;
7060         int             ret = 0;

7062         mutex_enter(SQLOCK(sq));

7064         /*
7065          * Before we leave, we need to make sure there are no
7066          * events listed for this queue.  All events for this queue
7067          * will just be freed.
7068          */
7069         if (qp != NULL && sq->sq_evhead != NULL) {
7070                 ASSERT(sq->sq_flags & SQ_EVENTS);

7072                 mp_prev = NULL;
7073                 for (bp = sq->sq_evhead; bp != NULL; bp = mp_next) {
7074                         mp_next = bp->b_next;
7075                         if (bp->b_queue == qp) {
7076                                 /* Delete this message */
7077                                 if (mp_prev != NULL) {
7078                                         mp_prev->b_next = mp_next;
7079                                         /*
7080                                          * Update sq_evtail if the last element
7081                                          * is removed.
7082                                          */
7083                                         if (bp == sq->sq_evtail) {
7084                                                 ASSERT(mp_next == NULL);
7085                                                 sq->sq_evtail = mp_prev;
7086                                         }
7087                                 } else
7088                                         sq->sq_evhead = mp_next;
7089                                 if (sq->sq_evhead == NULL)
7090                                         sq->sq_flags &= ~SQ_EVENTS;
7091                                 bp->b_prev = bp->b_next = NULL;
7092                                 freemsg(bp);
7093                                 ret++;
7094                         } else {
7095                                 mp_prev = bp;
7096                         }
7097                 }
7098         }

7100         /*
7101          * Walk sq_head and:
7102          *      - match qp if qp is set, remove it's messages
```

```
7103          *      - all if qp is not set
7104          */
7105         q = sq->sq_head;
7106         while (q != NULL) {
7107                 ASSERT(q->q_syncq == sq);
7108                 if ((qp == NULL) || (qp == q)) {
7109                         /*
7110                          * Yank the messages as a list off the queue
7111                          */
7112                         mp_head = q->q_sqhead;
7113                         /*
7114                          * We do not have QLOCK(q) here (which is safe due to
7115                          * assumptions mentioned above). To obtain the lock we
7116                          * need to release SQLOCK which may allow lots of things
7117                          * to change upon us. This place requires more analysis.
7118                          */
7119                         q->q_sqhead = q->q_sqtail = NULL;
7120                         ASSERT(mp_head->b_queue &&
7121                             mp_head->b_queue->q_syncq == sq);
7122
7123                         /*
7124                          * Free each of the messages.
7125                          */
7126                         for (bp = mp_head; bp != NULL; bp = mp_next) {
7127                                 mp_next = bp->b_next;
7128                                 bp->b_prev = bp->b_next = NULL;
7129                                 freemsg(bp);
7130                                 ret++;
7131                         }
7132                         /*
7133                          * Now remove the queue from the syncq.
7134                          */
7135                         ASSERT(q->q_sqflags & Q_SQQUEUED);
7136                         SQRM_Q(sq, q);
7137                         q->q_spri = 0;
7138                         q->q_syncqmsgs = 0;
7139
7140                         /*
7141                          * If qp was specified, we are done with it and are
7142                          * going to drop SQLOCK(sq) and return. We wakeup syncq
7143                          * waiters while we still have the SQLOCK.
7144                          */
7145                         if ((qp != NULL) && (sq->sq_flags & SQ_WANTWAKEUP)) {
7146                                 sq->sq_flags &= ~SQ_WANTWAKEUP;
7147                                 cv_broadcast(&sq->sq_wait);
7148                         }
7149                         /* Drop SQLOCK across clr_qfull */
7150                         mutex_exit(SQLOCK(sq));
7151
7152                         /*
7153                          * We avoid doing the test that drain_syncq does and
7154                          * unconditionally clear qfull for every flushed
7155                          * message. Since flush_syncq is only called during
7156                          * close this should not be a problem.
7157                          */
7158                         clr_qfull(q);
7159                         if (qp != NULL) {
7160                                 return (ret);
7161                         } else {
7162                                 mutex_enter(SQLOCK(sq));
7163                                 /*
7164                                  * The head was removed by SQRM_Q above.
7165                                  * reread the new head and flush it.
7166                                  */
7167                                 q = sq->sq_head;
7168                         }
```



```
7169                 } else {
7170                         q = q->q_sqnext;
7171                 }
7172                 ASSERT(MUTEX_HELD(SQLOCK(sq)));
7173         }
7174
7175         if (sq->sq_flags & SQ_WANTWAKEUP) {
7176                 sq->sq_flags &= ~SQ_WANTWAKEUP;
7177                 cv_broadcast(&sq->sq_wait);
7178         }
7179
7180         mutex_exit(SQLOCK(sq));
7181         return (ret);
7182 }
7183
7184 /*
7185  * Propagate all messages from a syncq to the next syncq that are associated
7186  * with the specified queue. If the queue is attached to a driver or if the
7187  * messages have been added due to a qwriter(PERIM_INNER), free the messages.
7188  *
7189  * Assumes that the stream is strlock()'ed. We don't come here if there
7190  * are no messages to propagate.
7191  *
7192  * NOTE : If the queue is attached to a driver, all the messages are freed
7193  * as there is no point in propagating the messages from the driver syncq
7194  * to the closing stream head which will in turn get freed later.
7195  */
7196 static int
7197 propagate_syncq(queue_t *qp)
7198 {
7199         mblk_t          *bp, *head, *tail, *prev, *next;
7200         syncq_t         *sq;
7201         queue_t         *nqp;
7202         syncq_t         *nsq;
7203         boolean_t       isdriver;
7204         int             moved = 0;
7205         uint16_t        flags;
7206         pri_t           priority = curthread->t_pri;
7207 #ifdef DEBUG
7208         void            (*func)();
7209 #endif
7210
7211         sq = qp->q_syncq;
7212         ASSERT(MUTEX_HELD(SQLOCK(sq)));
7213         /* debug macro */
7214         SQ_PUTLOCKS_HELD(sq);
7215         /*
7216          * As entersq() does not increment the sq_count for
7217          * the write side, check sq_count for non-QPERQ
7218          * perimeters alone.
7219          */
7220         ASSERT((qp->q_flag & QPERQ) || (sq->sq_count >= 1));
7221
7222         /*
7223          * propagate_syncq() can be called because of either messages on the
7224          * queue syncq or because on events on the queue syncq. Do actual
7225          * message propagations if there are any messages.
7226          */
7227         if (qp->q_syncqmsgs) {
7228                 isdriver = (qp->q_flag & QISDRV);
7229
7230                 if (!isdriver) {
7231                         nqp = qp->q_next;
7232                         nsq = nqp->q_syncq;
7233                         ASSERT(MUTEX_HELD(SQLOCK(nsq)));
7234                         /* debug macro */
```

```
7235                        SQ_PUTLOCKS_HELD(nsq);
7236 #ifdef DEBUG
7237                        func = (void (*)())nqp->q_qinfo->qi_putp;
7238 #endif
7239                }

7241                SQRM_Q(sq, qp);
7242                priority = MAX(qp->q_spri, priority);
7243                qp->q_spri = 0;
7244                head = qp->q_sqhead;
7245                tail = qp->q_sqtail;
7246                qp->q_sqhead = qp->q_sqtail = NULL;
7247                qp->q_syncqmsgs = 0;

7249                /*
7250                 * Walk the list of messages, and free them if this is a driver,
7251                 * otherwise reset the b_prev and b_queue value to the new putp.
7252                 * Afterward, we will just add the head to the end of the next
7253                 * syncq, and point the tail to the end of this one.
7254                 */

7256                for (bp = head; bp != NULL; bp = next) {
7257                        next = bp->b_next;
7258                        if (isdriver) {
7259                                bp->b_prev = bp->b_next = NULL;
7260                                freemsg(bp);
7261                                continue;
7262                        }
7263                        /* Change the q values for this message */
7264                        bp->b_queue = nqp;
7265 #ifdef DEBUG
7266                        bp->b_prev = (mblk_t *)func;
7267 #endif
7268                        moved++;
7269                }
7270                /*
7271                 * Attach list of messages to the end of the new queue (if there
7272                 * is a list of messages).
7273                 */

7275                if (!isdriver && head != NULL) {
7276                        ASSERT(tail != NULL);
7277                        if (nqp->q_sqhead == NULL) {
7278                                nqp->q_sqhead = head;
7279                        } else {
7280                                ASSERT(nqp->q_sqtail != NULL);
7281                                nqp->q_sqtail->b_next = head;
7282                        }
7283                        nqp->q_sqtail = tail;
7284                        /*
7285                         * When messages are moved from high priority queue to
7286                         * another queue, the destination queue priority is
7287                         * upgraded.
7288                         */

7290                        if (priority > nqp->q_spri)
7291                                nqp->q_spri = priority;

7293                        SQPUT_Q(nsq, nqp);

7295                        nqp->q_syncqmsgs += moved;
7296                        ASSERT(nqp->q_syncqmsgs != 0);
7297                }
7298        }

7300        /*
```



```
7301         * Before we leave, we need to make sure there are no
7302         * events listed for this queue.  All events for this queue
7303         * will just be freed.
7304         */
7305        if (sq->sq_evhead != NULL) {
7306                ASSERT(sq->sq_flags & SQ_EVENTS);
7307                prev = NULL;
7308                for (bp = sq->sq_evhead; bp != NULL; bp = next) {
7309                        next = bp->b_next;
7310                        if (bp->b_queue == qp) {
7311                                /* Delete this message */
7312                                if (prev != NULL) {
7313                                        prev->b_next = next;
7314                                        /*
7315                                         * Update sq_evtail if the last element
7316                                         * is removed.
7317                                         */
7318                                        if (bp == sq->sq_evtail) {
7319                                                ASSERT(next == NULL);
7320                                                sq->sq_evtail = prev;
7321                                        }
7322                                } else
7323                                        sq->sq_evhead = next;
7324                                if (sq->sq_evhead == NULL)
7325                                        sq->sq_flags &= ~SQ_EVENTS;
7326                                bp->b_prev = bp->b_next = NULL;
7327                                freemsg(bp);
7328                        } else {
7329                                prev = bp;
7330                        }
7331                }
7332        }

7334        flags = sq->sq_flags;

7336        /* Wake up any waiter before leaving. */
7337        if (flags & SQ_WANTWAKEUP) {
7338                flags &= ~SQ_WANTWAKEUP;
7339                cv_broadcast(&sq->sq_wait);
7340        }
7341        sq->sq_flags = flags;

7343        return (moved);
7344 }

7346 /*
7347  * Try and upgrade to exclusive access at the inner perimeter. If this can
7348  * not be done without blocking then request will be queued on the syncq
7349  * and drain_syncq will run it later.
7350  *
7351  * This routine can only be called from put or service procedures plus
7352  * asynchronous callback routines that have properly entered the queue (with
7353  * entersq). Thus qwriter_inner assumes the caller has one claim on the syncq
7354  * associated with q.
7355  */
7356 void
7357 qwriter_inner(queue_t *q, mblk_t *mp, void (*func)())
7358 {
7359        syncq_t *sq = q->q_syncq;
7360        uint16_t count;

7362        mutex_enter(SQLOCK(sq));
7363        count = sq->sq_count;
7364        SQ_PUTLOCKS_ENTER(sq);
7365        SUM_SQ_PUTCOUNTS(sq, count);
7366        ASSERT(count >= 1);
```

```
7367         ASSERT(sq->sq_type & (SQ_CIPUT|SQ_CISVC));

7369         if (count == 1) {
7370                 /*
7371                  * Can upgrade. This case also handles nested qwriter calls
7372                  * (when the qwriter callback function calls qwriter). In that
7373                  * case SQ_EXCL is already set.
7374                  */
7375                 sq->sq_flags |= SQ_EXCL;
7376                 SQ_PUTLOCKS_EXIT(sq);
7377                 mutex_exit(SQLOCK(sq));
7378                 (*func)(q, mp);
7379                 /*
7380                  * Assumes that leavesq, putnext, and drain_syncq will reset
7381                  * SQ_EXCL for SQ_CIPUT/SQ_CISVC queues. We leave SQ_EXCL on
7382                  * until putnext, leavesq, or drain_syncq drops it.
7383                  * That way we handle nested qwriter(INNER) without dropping
7384                  * SQ_EXCL until the outermost qwriter callback routine is
7385                  * done.
7386                  */
7387                 return;
7388         }
7389         SQ_PUTLOCKS_EXIT(sq);
7390         sqfill_events(sq, q, mp, func);
7391 }

7393 /*
7394  * Synchronous callback support functions
7395  */

7397 /*
7398  * Allocate a callback parameter structure.
7399  * Assumes that caller initializes the flags and the id.
7400  * Acquires SQLOCK(sq) if non-NULL is returned.
7401  */
7402 callbparams_t *
7403 callbparams_alloc(syncq_t *sq, void (*func)(void *), void *arg, int kmflags)
7404 {
7405         callbparams_t *cbp;
7406         size_t size = sizeof (callbparams_t);

7408         cbp = kmem_alloc(size, kmflags & ~KM_PANIC);

7410         /*
7411          * Only try tryhard allocation if the caller is ready to panic.
7412          * Otherwise just fail.
7413          */
7414         if (cbp == NULL) {
7415                 if (kmflags & KM_PANIC)
7416                         cbp = kmem_alloc_tryhard(sizeof (callbparams_t),
7417                             &size, kmflags);
7418                 else
7419                         return (NULL);
7420         }

7422         ASSERT(size >= sizeof (callbparams_t));
7423         cbp->cbp_size = size;
7424         cbp->cbp_sq = sq;
7425         cbp->cbp_func = func;
7426         cbp->cbp_arg = arg;
7427         mutex_enter(SQLOCK(sq));
7428         cbp->cbp_next = sq->sq_callbpend;
7429         sq->sq_callbpend = cbp;
7430         return (cbp);
7431 }
```



```
7433 void
7434 callbparams_free(syncq_t *sq, callbparams_t *cbp)
7435 {
7436         callbparams_t **pp, *p;

7438         ASSERT(MUTEX_HELD(SQLOCK(sq)));

7440         for (pp = &sq->sq_callbpend; (p = *pp) != NULL; pp = &p->cbp_next) {
7441                 if (p == cbp) {
7442                         *pp = p->cbp_next;
7443                         kmem_free(p, p->cbp_size);
7444                         return;
7445                 }
7446         }
7447         (void) (STRLOG(0, 0, 0, SL_CONSOLE,
7448             "callbparams_free: not found\n"));
7449 }

7451 void
7452 callbparams_free_id(syncq_t *sq, callbparams_id_t id, int32_t flag)
7453 {
7454         callbparams_t **pp, *p;

7456         ASSERT(MUTEX_HELD(SQLOCK(sq)));

7458         for (pp = &sq->sq_callbpend; (p = *pp) != NULL; pp = &p->cbp_next) {
7459                 if (p->cbp_id == id && p->cbp_flags == flag) {
7460                         *pp = p->cbp_next;
7461                         kmem_free(p, p->cbp_size);
7462                         return;
7463                 }
7464         }
7465         (void) (STRLOG(0, 0, 0, SL_CONSOLE,
7466             "callbparams_free_id: not found\n"));
7467 }

7469 /*
7470  * Callback wrapper function used by once-only callbacks that can be
7471  * cancelled (qtimeout and qbufcall)
7472  * Contains inline version of entersq(sq, SQ_CALLBACK) that can be
7473  * cancelled by the qun* functions.
7474  */
7475 void
7476 qcallbwrapper(void *arg)
7477 {
7478         callbparams_t *cbp = arg;
7479         syncq_t *sq;
7480         uint16_t count = 0;
7481         uint16_t waitflags = SQ_STAYAWAY | SQ_EVENTS | SQ_EXCL;
7482         uint16_t type;

7484         sq = cbp->cbp_sq;
7485         mutex_enter(SQLOCK(sq));
7486         type = sq->sq_type;
7487         if (!(type & SQ_CICB)) {
7488                 count = sq->sq_count;
7489                 SQ_PUTLOCKS_ENTER(sq);
7490                 SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
7491                 SUM_SQ_PUTCOUNTS(sq, count);
7492                 sq->sq_needexcl++;
7493                 ASSERT(sq->sq_needexcl != 0);   /* wraparound */
7494                 waitflags |= SQ_MESSAGES;
7495         }
7496         /* Can not handle exclusive entry at outer perimeter */
7497         ASSERT(type & SQ_COCB);
```

```
7499         while ((sq->sq_flags & waitflags) || (!(type & SQ_CICB) &&count != 0)) {
7500                 if ((sq->sq_callbflags & cbp->cbp_flags) &&
7501                     (sq->sq_cancelid == cbp->cbp_id)) {
7502                         /* timeout has been cancelled */
7503                         sq->sq_callbflags |= SQ_CALLB_BYPASSED;
7504                         callbparams_free(sq, cbp);
7505                         if (!(type & SQ_CICB)) {
7506                                 ASSERT(sq->sq_needexcl > 0);
7507                                 sq->sq_needexcl--;
7508                                 if (sq->sq_needexcl == 0) {
7509                                         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
7510                                 }
7511                                 SQ_PUTLOCKS_EXIT(sq);
7512                         }
7513                         mutex_exit(SQLOCK(sq));
7514                         return;
7515                 }
7516                 sq->sq_flags |= SQ_WANTWAKEUP;
7517                 if (!(type & SQ_CICB)) {
7518                         SQ_PUTLOCKS_EXIT(sq);
7519                 }
7520                 cv_wait(&sq->sq_wait, SQLOCK(sq));
7521                 if (!(type & SQ_CICB)) {
7522                         count = sq->sq_count;
7523                         SQ_PUTLOCKS_ENTER(sq);
7524                         SUM_SQ_PUTCOUNTS(sq, count);
7525                 }
7526         }

7528         sq->sq_count++;
7529         ASSERT(sq->sq_count != 0);      /* Wraparound */
7530         if (!(type & SQ_CICB)) {
7531                 ASSERT(count == 0);
7532                 sq->sq_flags |= SQ_EXCL;
7533                 ASSERT(sq->sq_needexcl > 0);
7534                 sq->sq_needexcl--;
7535                 if (sq->sq_needexcl == 0) {
7536                         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
7537                 }
7538                 SQ_PUTLOCKS_EXIT(sq);
7539         }

7541         mutex_exit(SQLOCK(sq));

7543         cbp->cbp_func(cbp->cbp_arg);

7545         /*
7546          * We drop the lock only for leavesq to re-acquire it.
7547          * Possible optimization is inline of leavesq.
7548          */
7549         mutex_enter(SQLOCK(sq));
7550         callbparams_free(sq, cbp);
7551         mutex_exit(SQLOCK(sq));
7552         leavesq(sq, SQ_CALLBACK);
7553 }

7555 /*
7556  * No need to grab sq_putlocks here. See comment in strsubr.h that
7557  * explains when sq_putlocks are used.
7558  *
7559  * sq_count (or one of the sq_putcounts) has already been
7560  * decremented by the caller, and if SQ_QUEUED, we need to call
7561  * drain_syncq (the global syncq drain).
7562  * If putnext_tail is called with the SQ_EXCL bit set, we are in
7563  * one of two states, non-CIPUT perimeter, and we need to clear
7564  * it, or we went exclusive in the put procedure.  In any case,
```



```
7565  * we want to clear the bit now, and it is probably easier to do
7566  * this at the beginning of this function (remember, we hold
7567  * the SQLOCK).  Lastly, if there are other messages queued
7568  * on the syncq (and not for our destination), enable the syncq
7569  * for background work.
7570  */

7572 /* ARGSUSED */
7573 void
7574 putnext_tail(syncq_t *sq, queue_t *qp, uint32_t passflags)
7575 {
7576         uint16_t        flags = sq->sq_flags;

7578         ASSERT(MUTEX_HELD(SQLOCK(sq)));
7579         ASSERT(MUTEX_NOT_HELD(QLOCK(qp)));

7581         /* Clear SQ_EXCL if set in passflags */
7582         if (passflags & SQ_EXCL) {
7583                 flags &= ~SQ_EXCL;
7584         }
7585         if (flags & SQ_WANTWAKEUP) {
7586                 flags &= ~SQ_WANTWAKEUP;
7587                 cv_broadcast(&sq->sq_wait);
7588         }
7589         if (flags & SQ_WANTEXWAKEUP) {
7590                 flags &= ~SQ_WANTEXWAKEUP;
7591                 cv_broadcast(&sq->sq_exitwait);
7592         }
7593         sq->sq_flags = flags;

7595         /*
7596          * We have cleared SQ_EXCL if we were asked to, and started
7597          * the wakeup process for waiters.  If there are no writers
7598          * then we need to drain the syncq if we were told to, or
7599          * enable the background thread to do it.
7600          */
7601         if (!(flags & (SQ_STAYAWAY|SQ_EXCL))) {
7602                 if ((passflags & SQ_QUEUED) ||
7603                     (sq->sq_svcflags & SQ_DISABLED)) {
7604                         /* drain_syncq will take care of events in the list */
7605                         drain_syncq(sq);
7606                         return;
7607                 } else if (flags & SQ_QUEUED) {
7608                         sqenable(sq);
7609                 }
7610         }
7611         /* Drop the SQLOCK on exit */
7612         mutex_exit(SQLOCK(sq));
7613         TRACE_3(TR_FAC_STREAMS_FR, TR_PUTNEXT_END,
7614             "putnext_end:(%p, %p, %p) done", NULL, qp, sq);
7615 }

7617 void
7618 set_qend(queue_t *q)
7619 {
7620         mutex_enter(QLOCK(q));
7621         if (!O_SAMESTR(q))
7622                 q->q_flag |= QEND;
7623         else
7624                 q->q_flag &= ~QEND;
7625         mutex_exit(QLOCK(q));
7626         q = _OTHERQ(q);
7627         mutex_enter(QLOCK(q));
7628         if (!O_SAMESTR(q))
7629                 q->q_flag |= QEND;
7630         else
```

```
7631                q->q_flag &= ~QEND;
7632        mutex_exit(QLOCK(q));
7633 }

7635 /*
7636  * Set QFULL in next service procedure queue (that cares) if not already
7637  * set and if there are already more messages on the syncq than
7638  * sq_max_size.  If sq_max_size is 0, no flow control will be asserted on
7639  * any syncq.
7640  *
7641  * The fq here is the next queue with a service procedure.  This is where
7642  * we would fail canputnext, so this is where we need to set QFULL.
7643  * In the case when fq != q we need to take QLOCK(fq) to set QFULL flag.
7644  *
7645  * We already have QLOCK at this point. To avoid cross-locks with
7646  * freezestr() which grabs all QLOCKs and with strlock() which grabs both
7647  * SQLOCK and sd_reflock, we need to drop respective locks first.
7648  */
7649 void
7650 set_qfull(queue_t *q)
7651 {
7652         queue_t         *fq = NULL;

7654         ASSERT(MUTEX_HELD(QLOCK(q)));
7655         if ((sq_max_size != 0) && (!(q->q_nfsrv->q_flag & QFULL)) &&
7656             (q->q_syncqmsgs > sq_max_size)) {
7657                 if ((fq = q->q_nfsrv) == q) {
7658                         fq->q_flag |= QFULL;
7659                 } else {
7660                         mutex_exit(QLOCK(q));
7661                         mutex_enter(QLOCK(fq));
7662                         fq->q_flag |= QFULL;
7663                         mutex_exit(QLOCK(fq));
7664                         mutex_enter(QLOCK(q));
7665                 }
7666         }
7667 }

7669 void
7670 clr_qfull(queue_t *q)
7671 {
7672         queue_t *oq = q;

7674         q = q->q_nfsrv;
7675         /* Fast check if there is any work to do before getting the lock. */
7676         if ((q->q_flag & (QFULL|QWANTW)) == 0) {
7677                 return;
7678         }

7680         /*
7681          * Do not reset QFULL (and backenable) if the q_count is the reason
7682          * for QFULL being set.
7683          */
7684         mutex_enter(QLOCK(q));
7685         /*
7686          * If queue is empty i.e q_mblkcnt is zero, queue can not be full.
7687          * Hence clear the QFULL.
7688          * If both q_count and q_mblkcnt are less than the hiwat mark,
7689          * clear the QFULL.
7690          */
7691         if (q->q_mblkcnt == 0 || ((q->q_count < q->q_hiwat) &&
7692             (q->q_mblkcnt < q->q_hiwat))) {
7693                 q->q_flag &= ~QFULL;
7694                 /*
7695                  * A little more confusing, how about this way:
7696                  * if someone wants to write,
```



```
7697                  * AND
7698                  *    both counts are less than the lowat mark
7699                  *    OR
7700                  *    the lowat mark is zero
7701                  * THEN
7702                  * backenable
7703                  */
7704                 if ((q->q_flag & QWANTW) &&
7705                     (((q->q_count < q->q_lowat) &&
7706                     (q->q_mblkcnt < q->q_lowat)) || q->q_lowat == 0)) {
7707                         q->q_flag &= ~QWANTW;
7708                         mutex_exit(QLOCK(q));
7709                         backenable(oq, 0);
7710                 } else
7711                         mutex_exit(QLOCK(q));
7712         } else
7713                 mutex_exit(QLOCK(q));
7714 }

7716 /*
7717  * Set the forward service procedure pointer.
7718  *
7719  * Called at insert-time to cache a queue's next forward service procedure in
7720  * q_nfsrv; used by canput() and canputnext().  If the queue to be inserted
7721  * has a service procedure then q_nfsrv points to itself.  If the queue to be
7722  * inserted does not have a service procedure, then q_nfsrv points to the next
7723  * queue forward that has a service procedure.  If the queue is at the logical
7724  * end of the stream (driver for write side, stream head for the read side)
7725  * and does not have a service procedure, then q_nfsrv also points to itself.
7726  */
7727 void
7728 set_nfsrv_ptr(
7729         queue_t  *rnew,         /* read queue pointer to new module */
7730         queue_t  *wnew,         /* write queue pointer to new module */
7731         queue_t  *prev_rq,      /* read queue pointer to the module above */
7732         queue_t  *prev_wq)      /* write queue pointer to the module above */
7733 {
7734         queue_t *qp;

7736         if (prev_wq->q_next == NULL) {
7737                 /*
7738                  * Insert the driver, initialize the driver and stream head.
7739                  * In this case, prev_rq/prev_wq should be the stream head.
7740                  * _I_INSERT does not allow inserting a driver.  Make sure
7741                  * that it is not an insertion.
7742                  */
7743                 ASSERT(!(rnew->q_flag & _QINSERTING));
7744                 wnew->q_nfsrv = wnew;
7745                 if (rnew->q_qinfo->qi_srvp)
7746                         rnew->q_nfsrv = rnew;
7747                 else
7748                         rnew->q_nfsrv = prev_rq;
7749                 prev_rq->q_nfsrv = prev_rq;
7750                 prev_wq->q_nfsrv = prev_wq;
7751         } else {
7752                 /*
7753                  * set up read side q_nfsrv pointer.  This MUST be done
7754                  * before setting the write side, because the setting of
7755                  * the write side for a fifo may depend on it.
7756                  *
7757                  * Suppose we have a fifo that only has pipemod pushed.
7758                  * pipemod has no read or write service procedures, so
7759                  * nfsrv for both pipemod queues points to prev_rq (the
7760                  * stream read head).  Now push bufmod (which has only a
7761                  * read service procedure).  Doing the write side first,
7762                  * wnew->q_nfsrv is set to pipemod's writeq nfsrv, which
```

```
7763                  * is WRONG; the next queue forward from wnew with a
7764                  * service procedure will be rnew, not the stream read head.
7765                  * Since the downstream queue (which in the case of a fifo
7766                  * is the read queue rnew) can affect upstream queues, it
7767                  * needs to be done first.  Setting up the read side first
7768                  * sets nfsrv for both pipemod queues to rnew and then
7769                  * when the write side is set up, wnew-q_nfsrv will also
7770                  * point to rnew.
7771                  */
7772                 if (rnew->q_qinfo->qi_srvp) {
7773                         /*
7774                          * use _OTHERQ() because, if this is a pipe, next
7775                          * module may have been pushed from other end and
7776                          * q_next could be a read queue.
7777                          */
7778                         qp = _OTHERQ(prev_wq->q_next);
7779                         while (qp && qp->q_nfsrv != qp) {
7780                                 qp->q_nfsrv = rnew;
7781                                 qp = backq(qp);
7782                         }
7783                         rnew->q_nfsrv = rnew;
7784                 } else
7785                         rnew->q_nfsrv = prev_rq->q_nfsrv;
7786
7787                 /* set up write side q_nfsrv pointer */
7788                 if (wnew->q_qinfo->qi_srvp) {
7789                         wnew->q_nfsrv = wnew;
7790
7791                         /*
7792                          * For insertion, need to update nfsrv of the modules
7793                          * above which do not have a service routine.
7794                          */
7795                         if (rnew->q_flag & _QINSERTING) {
7796                                 for (qp = prev_wq;
7797                                     qp != NULL && qp->q_nfsrv != qp;
7798                                     qp = backq(qp)) {
7799                                         qp->q_nfsrv = wnew->q_nfsrv;
7800                                 }
7801                         }
7802                 } else {
7803                         if (prev_wq->q_next == prev_rq)
7804                                 /*
7805                                  * Since prev_wq/prev_rq are the middle of a
7806                                  * fifo, wnew/rnew will also be the middle of
7807                                  * a fifo and wnew's nfsrv is same as rnew's.
7808                                  */
7809                                 wnew->q_nfsrv = rnew->q_nfsrv;
7810                         else
7811                                 wnew->q_nfsrv = prev_wq->q_next->q_nfsrv;
7812                 }
7813         }
7814 }
7815
7816 /*
7817  * Reset the forward service procedure pointer; called at remove-time.
7818  */
7819 void
7820 reset_nfsrv_ptr(queue_t *rqp, queue_t *wqp)
7821 {
7822         queue_t *tmp_qp;
7823
7824         /* Reset the write side q_nfsrv pointer for _I_REMOVE */
7825         if ((rqp->q_flag & _QREMOVING) && (wqp->q_qinfo->qi_srvp != NULL)) {
7826                 for (tmp_qp = backq(wqp);
7827                     tmp_qp != NULL && tmp_qp->q_nfsrv == wqp;
7828                     tmp_qp = backq(tmp_qp)) {
```



```
7829                         tmp_qp->q_nfsrv = wqp->q_nfsrv;
7830                 }
7831         }
7832
7833         /* reset the read side q_nfsrv pointer */
7834         if (rqp->q_qinfo->qi_srvp) {
7835                 if (wqp->q_next) {      /* non-driver case */
7836                         tmp_qp = _OTHERQ(wqp->q_next);
7837                         while (tmp_qp && tmp_qp->q_nfsrv == rqp) {
7838                                 /* Note that rqp->q_next cannot be NULL */
7839                                 ASSERT(rqp->q_next != NULL);
7840                                 tmp_qp->q_nfsrv = rqp->q_next->q_nfsrv;
7841                                 tmp_qp = backq(tmp_qp);
7842                         }
7843                 }
7844         }
7845 }
7846
7847 /*
7848  * This routine should be called after all stream geometry changes to update
7849  * the stream head cached struio() rd/wr queue pointers. Note must be called
7850  * with the streamlock()ed.
7851  *
7852  * Note: only enables Synchronous STREAMS for a side of a Stream which has
7853  *       an explicit synchronous barrier module queue. That is, a queue that
7854  *       has specified a struio() type.
7855  */
7856 static void
7857 strsetuio(stdata_t *stp)
7858 {
7859         queue_t *wrq;
7860
7861         if (stp->sd_flag & STPLEX) {
7862                 /*
7863                  * Not streamhead, but a mux, so no Synchronous STREAMS.
7864                  */
7865                 stp->sd_struiowrq = NULL;
7866                 stp->sd_struiordq = NULL;
7867                 return;
7868         }
7869         /*
7870          * Scan the write queue(s) while synchronous
7871          * until we find a qinfo uio type specified.
7872          */
7873         wrq = stp->sd_wrq->q_next;
7874         while (wrq) {
7875                 if (wrq->q_struiot == STRUIOT_NONE) {
7876                         wrq = 0;
7877                         break;
7878                 }
7879                 if (wrq->q_struiot != STRUIOT_DONTCARE)
7880                         break;
7881                 if (! _SAMESTR(wrq)) {
7882                         wrq = 0;
7883                         break;
7884                 }
7885                 wrq = wrq->q_next;
7886         }
7887         stp->sd_struiowrq = wrq;
7888         /*
7889          * Scan the read queue(s) while synchronous
7890          * until we find a qinfo uio type specified.
7891          */
7892         wrq = stp->sd_wrq->q_next;
7893         while (wrq) {
7894                 if (_RD(wrq)->q_struiot == STRUIOT_NONE) {
```

```
7895                        wrq = 0;
7896                        break;
7897                }
7898                if (_RD(wrq)->q_struiot != STRUIOT_DONTCARE)
7899                        break;
7900                if (! _SAMESTR(wrq)) {
7901                        wrq = 0;
7902                        break;
7903                }
7904                wrq = wrq->q_next;
7905        }
7906        stp->sd_struiordq = wrq ? _RD(wrq) : 0;
7907 }

7909 /*
7910  * pass_wput, unblocks the passthru queues, so that
7911  * messages can arrive at muxs lower read queue, before
7912  * I_LINK/I_UNLINK is acked/nacked.
7913  */
7914 static void
7915 pass_wput(queue_t *q, mblk_t *mp)
7916 {
7917         syncq_t *sq;

7919         sq = _RD(q)->q_syncq;
7920         if (sq->sq_flags & SQ_BLOCKED)
7921                 unblocksq(sq, SQ_BLOCKED, 0);
7922         putnext(q, mp);
7923 }

7925 /*
7926  * Set up queues for the link/unlink.
7927  * Create a new queue and block it and then insert it
7928  * below the stream head on the lower stream.
7929  * This prevents any messages from arriving during the setq
7930  * as well as while the mux is processing the LINK/I_UNLINK.
7931  * The blocked passq is unblocked once the LINK/I_UNLINK has
7932  * been acked or nacked or if a message is generated and sent
7933  * down muxs write put procedure.
7934  * See pass_wput().
7935  *
7936  * After the new queue is inserted, all messages coming from below are
7937  * blocked. The call to strlock will ensure that all activity in the stream head
7938  * read queue syncq is stopped (sq_count drops to zero).
7939  */
7940 static queue_t *
7941 link_addpassthru(stdata_t *stpdown)
7942 {
7943         queue_t *passq;
7944         sqlist_t sqlist;

7946         passq = allocq();
7947         STREAM(passq) = STREAM(_WR(passq)) = stpdown;
7948         /* setq might sleep in allocator - avoid holding locks. */
7949         setq(passq, &passthru_rinit, &passthru_winit, NULL, QPERQ,
7950             SQ_CI|SQ_CO, B_FALSE);
7951         claimq(passq);
7952         blocksq(passq->q_syncq, SQ_BLOCKED, 1);
7953         insertq(STREAM(passq), passq);

7955         /*
7956          * Use strlock() to wait for the stream head sq_count to drop to zero
7957          * since we are going to change q_ptr in the stream head.  Note that
7958          * insertq() doesn't wait for any syncq counts to drop to zero.
7959          */
7960         sqlist.sqlist_head = NULL;
```



```
7961         sqlist.sqlist_index = 0;
7962         sqlist.sqlist_size = sizeof (sqlist_t);
7963         sqlist_insert(&sqlist, _RD(stpdown->sd_wrq)->q_syncq);
7964         strlock(stpdown, &sqlist);
7965         strunlock(stpdown, &sqlist);

7967         releaseq(passq);
7968         return (passq);
7969 }

7971 /*
7972  * Let messages flow up into the mux by removing
7973  * the passq.
7974  */
7975 static void
7976 link_rempassthru(queue_t *passq)
7977 {
7978         claimq(passq);
7979         removeq(passq);
7980         releaseq(passq);
7981         freeq(passq);
7982 }

7984 /*
7985  * Wait for the condition variable pointed to by 'cvp' to be signaled,
7986  * or for 'tim' milliseconds to elapse, whichever comes first.  If 'tim'
7987  * is negative, then there is no time limit.  If 'nosigs' is non-zero,
7988  * then the wait will be non-interruptible.
7989  *
7990  * Returns >0 if signaled, 0 if interrupted, or -1 upon timeout.
7991  */
7992 clock_t
7993 str_cv_wait(kcondvar_t *cvp, kmutex_t *mp, clock_t tim, int nosigs)
7994 {
7995         clock_t ret;

7997         if (tim < 0) {
7998                 if (nosigs) {
7999                         cv_wait(cvp, mp);
8000                         ret = 1;
8001                 } else {
8002                         ret = cv_wait_sig(cvp, mp);
8003                 }
8004         } else if (tim > 0) {
8005                 /*
8006                  * convert milliseconds to clock ticks
8007                  */
8008                 if (nosigs) {
8009                         ret = cv_reltimedwait(cvp, mp,
8010                             MSEC_TO_TICK_ROUNDUP(tim), TR_CLOCK_TICK);
8011                 } else {
8012                         ret = cv_reltimedwait_sig(cvp, mp,
8013                             MSEC_TO_TICK_ROUNDUP(tim), TR_CLOCK_TICK);
8014                 }
8015         } else {
8016                 ret = -1;
8017         }
8018         return (ret);
8019 }

8021 /*
8022  * Wait until the stream head can determine if it is at the mark but
8023  * don't wait forever to prevent a race condition between the "mark" state
8024  * in the stream head and any mark state in the caller/user of this routine.
8025  *
8026  * This is used by sockets and for a socket it would be incorrect
```

```
 * to return a failure for SIOCATMARK when there is no data in the receive
 * queue and the marked urgent data is traveling up the stream.
 *
 * This routine waits until the mark is known by waiting for one of these
 * three events:
 *      The stream head read queue becoming non-empty (including an EOF).
 *      The STRATMARK flag being set (due to a MSGMARKNEXT message).
 *      The STRNOTATMARK flag being set (which indicates that the transport
 *      has sent a MSGNOTMARKNEXT message to indicate that it is not at
 *      the mark).
 *
 * The routine returns 1 if the stream is at the mark; 0 if it can
 * be determined that the stream is not at the mark.
 * If the wait times out and it can't determine
 * whether or not the stream might be at the mark the routine will return -1.
 *
 * Note: This routine should only be used when a mark is pending i.e.,
 * in the socket case the SIGURG has been posted.
 * Note2: This can not wakeup just because synchronous streams indicate
 * that data is available since it is not possible to use the synchronous
 * streams interfaces to determine the b_flag value for the data queued below
 * the stream head.
 */
int
strwaitmark(vnode_t *vp)
{
        struct stdata *stp = vp->v_stream;
        queue_t *rq = _RD(stp->sd_wrq);
        int mark;

        mutex_enter(&stp->sd_lock);
        while (rq->q_first == NULL &&
            !(stp->sd_flag & (STRATMARK|STRNOTATMARK|STREOF))) {
                stp->sd_flag |= RSLEEP;

                /* Wait for 100 milliseconds for any state change. */
                if (str_cv_wait(&rq->q_wait, &stp->sd_lock, 100, 1) == -1) {
                        mutex_exit(&stp->sd_lock);
                        return (-1);
                }
        }
        if (stp->sd_flag & STRATMARK)
                mark = 1;
        else if (rq->q_first != NULL && (rq->q_first->b_flag & MSGMARK))
                mark = 1;
        else
                mark = 0;

        mutex_exit(&stp->sd_lock);
        return (mark);
}

/*
 * Set a read side error. If persist is set change the socket error
 * to persistent. If errfunc is set install the function as the exported
 * error handler.
 */
void
strsetrerror(vnode_t *vp, int error, int persist, errfunc_t errfunc)
{
        struct stdata *stp = vp->v_stream;

        mutex_enter(&stp->sd_lock);
        stp->sd_rerror = error;
        if (error == 0 && errfunc == NULL)
                stp->sd_flag &= ~STRDERR;
```



```
        else
                stp->sd_flag |= STRDERR;
        if (persist) {
                stp->sd_flag &= ~STRDERRNONPERSIST;
        } else {
                stp->sd_flag |= STRDERRNONPERSIST;
        }
        stp->sd_rderrfunc = errfunc;
        if (error != 0 || errfunc != NULL) {
                cv_broadcast(&_RD(stp->sd_wrq)->q_wait);        /* readers */
                cv_broadcast(&stp->sd_wrq->q_wait);             /* writers */
                cv_broadcast(&stp->sd_monitor);                 /* ioctllers */

                mutex_exit(&stp->sd_lock);
                pollwakeup(&stp->sd_pollist, POLLERR);
                mutex_enter(&stp->sd_lock);

                if (stp->sd_sigflags & S_ERROR)
                        strsendsig(stp->sd_siglist, S_ERROR, 0, error);
        }
        mutex_exit(&stp->sd_lock);
}

/*
 * Set a write side error. If persist is set change the socket error
 * to persistent.
 */
void
strsetwerror(vnode_t *vp, int error, int persist, errfunc_t errfunc)
{
        struct stdata *stp = vp->v_stream;

        mutex_enter(&stp->sd_lock);
        stp->sd_werror = error;
        if (error == 0 && errfunc == NULL)
                stp->sd_flag &= ~STWRERR;
        else
                stp->sd_flag |= STWRERR;
        if (persist) {
                stp->sd_flag &= ~STWRERRNONPERSIST;
        } else {
                stp->sd_flag |= STWRERRNONPERSIST;
        }
        stp->sd_wrerrfunc = errfunc;
        if (error != 0 || errfunc != NULL) {
                cv_broadcast(&_RD(stp->sd_wrq)->q_wait);        /* readers */
                cv_broadcast(&stp->sd_wrq->q_wait);             /* writers */
                cv_broadcast(&stp->sd_monitor);                 /* ioctllers */

                mutex_exit(&stp->sd_lock);
                pollwakeup(&stp->sd_pollist, POLLERR);
                mutex_enter(&stp->sd_lock);

                if (stp->sd_sigflags & S_ERROR)
                        strsendsig(stp->sd_siglist, S_ERROR, 0, error);
        }
        mutex_exit(&stp->sd_lock);
}

/*
 * Make the stream return 0 (EOF) when all data has been read.
 * No effect on write side.
 */
void
strseteof(vnode_t *vp, int eof)
{
```

```
8159         struct stdata *stp = vp->v_stream;

8161         mutex_enter(&stp->sd_lock);
8162         if (!eof) {
8163                 stp->sd_flag &= ~STREOF;
8164                 mutex_exit(&stp->sd_lock);
8165                 return;
8166         }
8167         stp->sd_flag |= STREOF;
8168         if (stp->sd_flag & RSLEEP) {
8169                 stp->sd_flag &= ~RSLEEP;
8170                 cv_broadcast(&_RD(stp->sd_wrq)->q_wait);
8171         }

8173         mutex_exit(&stp->sd_lock);
8174         pollwakeup(&stp->sd_pollist, POLLIN|POLLRDNORM);
8175         mutex_enter(&stp->sd_lock);

8177         if (stp->sd_sigflags & (S_INPUT|S_RDNORM))
8178                 strsendsig(stp->sd_siglist, S_INPUT|S_RDNORM, 0, 0);
8179         mutex_exit(&stp->sd_lock);
8180 }

8182 void
8183 strflushrq(vnode_t *vp, int flag)
8184 {
8185         struct stdata *stp = vp->v_stream;

8187         mutex_enter(&stp->sd_lock);
8188         flushq(_RD(stp->sd_wrq), flag);
8189         mutex_exit(&stp->sd_lock);
8190 }

8192 void
8193 strsetrputhooks(vnode_t *vp, uint_t flags,
8194                 msgfunc_t protofunc, msgfunc_t miscfunc)
8195 {
8196         struct stdata *stp = vp->v_stream;

8198         mutex_enter(&stp->sd_lock);

8200         if (protofunc == NULL)
8201                 stp->sd_rprotofunc = strrput_proto;
8202         else
8203                 stp->sd_rprotofunc = protofunc;

8205         if (miscfunc == NULL)
8206                 stp->sd_rmiscfunc = strrput_misc;
8207         else
8208                 stp->sd_rmiscfunc = miscfunc;

8210         if (flags & SH_CONSOL_DATA)
8211                 stp->sd_rput_opt |= SR_CONSOL_DATA;
8212         else
8213                 stp->sd_rput_opt &= ~SR_CONSOL_DATA;

8215         if (flags & SH_SIGALLDATA)
8216                 stp->sd_rput_opt |= SR_SIGALLDATA;
8217         else
8218                 stp->sd_rput_opt &= ~SR_SIGALLDATA;

8220         if (flags & SH_IGN_ZEROLEN)
8221                 stp->sd_rput_opt |= SR_IGN_ZEROLEN;
8222         else
8223                 stp->sd_rput_opt &= ~SR_IGN_ZEROLEN;
```



```
8225         mutex_exit(&stp->sd_lock);
8226 }

8228 void
8229 strsetwputhooks(vnode_t *vp, uint_t flags, clock_t closetime)
8230 {
8231         struct stdata *stp = vp->v_stream;

8233         mutex_enter(&stp->sd_lock);
8234         stp->sd_closetime = closetime;

8236         if (flags & SH_SIGPIPE)
8237                 stp->sd_wput_opt |= SW_SIGPIPE;
8238         else
8239                 stp->sd_wput_opt &= ~SW_SIGPIPE;
8240         if (flags & SH_RECHECK_ERR)
8241                 stp->sd_wput_opt |= SW_RECHECK_ERR;
8242         else
8243                 stp->sd_wput_opt &= ~SW_RECHECK_ERR;

8245         mutex_exit(&stp->sd_lock);
8246 }

8248 void
8249 strsetrwputdatahooks(vnode_t *vp, msgfunc_t rdatafunc, msgfunc_t wdatafunc)
8250 {
8251         struct stdata *stp = vp->v_stream;

8253         mutex_enter(&stp->sd_lock);

8255         stp->sd_rputdatafunc = rdatafunc;
8256         stp->sd_wputdatafunc = wdatafunc;

8258         mutex_exit(&stp->sd_lock);
8259 }

8261 /* Used within framework when the queue is already locked */
8262 void
8263 qenable_locked(queue_t *q)
8264 {
8265         stdata_t *stp = STREAM(q);

8267         ASSERT(MUTEX_HELD(QLOCK(q)));

8269         if (!q->q_qinfo->qi_srvp)
8270                 return;

8272         /*
8273          * Do not place on run queue if already enabled or closing.
8274          */
8275         if (q->q_flag & (QWCLOSE|QENAB))
8276                 return;

8278         /*
8279          * mark queue enabled and place on run list if it is not already being
8280          * serviced. If it is serviced, the runservice() function will detect
8281          * that QENAB is set and call service procedure before clearing
8282          * QINSERVICE flag.
8283          */
8284         q->q_flag |= QENAB;
8285         if (q->q_flag & QINSERVICE)
8286                 return;

8288         /* Record the time of qenable */
8289         q->q_qtstamp = ddi_get_lbolt();
```

```
8291         /*
8292          * Put the queue in the stp list and schedule it for background
8293          * processing if it is not already scheduled or if stream head does not
8294          * intent to process it in the foreground later by setting
8295          * STRS_WILLSERVICE flag.
8296          */
8297         mutex_enter(&stp->sd_qlock);
8298         /*
8299          * If there are already something on the list, stp flags should show
8300          * intention to drain it.
8301          */
8302         IMPLY(STREAM_NEEDSERVICE(stp),
8303             (stp->sd_svcflags & (STRS_WILLSERVICE | STRS_SCHEDULED)));

8305         ENQUEUE(q, stp->sd_qhead, stp->sd_qtail, q_link);
8306         stp->sd_nqueues++;

8308         /*
8309          * If no one will drain this stream we are the first producer and
8310          * need to schedule it for background thread.
8311          */
8312         if (!(stp->sd_svcflags & (STRS_WILLSERVICE | STRS_SCHEDULED))) {
8313                 /*
8314                  * No one will service this stream later, so we have to
8315                  * schedule it now.
8316                  */
8317                 STRSTAT(stenables);
8318                 stp->sd_svcflags |= STRS_SCHEDULED;
8319                 stp->sd_servid = (void *)taskq_dispatch(streams_taskq,
8320                     (task_func_t *)stream_service, stp, TQ_NOSLEEP|TQ_NOQUEUE);

8322                 if (stp->sd_servid == NULL) {
8323                         /*
8324                          * Task queue failed so fail over to the backup
8325                          * servicing thread.
8326                          */
8327                         STRSTAT(taskqfails);
8328                         /*
8329                          * It is safe to clear STRS_SCHEDULED flag because it
8330                          * was set by this thread above.
8331                          */
8332                         stp->sd_svcflags &= ~STRS_SCHEDULED;

8334                         /*
8335                          * Failover scheduling is protected by service_queue
8336                          * lock.
8337                          */
8338                         mutex_enter(&service_queue);
8339                         ASSERT((stp->sd_qhead == q) && (stp->sd_qtail == q));
8340                         ASSERT(q->q_link == NULL);
8341                         /*
8342                          * Append the queue to qhead/qtail list.
8343                          */
8344                         if (qhead == NULL)
8345                                 qhead = q;
8346                         else
8347                                 qtail->q_link = q;
8348                         qtail = q;
8349                         /*
8350                          * Clear stp queue list.
8351                          */
8352                         stp->sd_qhead = stp->sd_qtail = NULL;
8353                         stp->sd_nqueues = 0;
8354                         /*
8355                          * Wakeup background queue processing thread.
8356                          */
```



```
8357                         cv_signal(&services_to_run);
8358                         mutex_exit(&service_queue);
8359                 }
8360         }
8361         mutex_exit(&stp->sd_qlock);
8362 }

8364 static void
8365 queue_service(queue_t *q)
8366 {
8367         /*
8368          * The queue in the list should have
8369          * QENAB flag set and should not have
8370          * QINSERVICE flag set. QINSERVICE is
8371          * set when the queue is dequeued and
8372          * qenable_locked doesn't enqueue a
8373          * queue with QINSERVICE set.
8374          */

8376         ASSERT(!(q->q_flag & QINSERVICE));
8377         ASSERT((q->q_flag & QENAB));
8378         mutex_enter(QLOCK(q));
8379         q->q_flag &= ~QENAB;
8380         q->q_flag |= QINSERVICE;
8381         mutex_exit(QLOCK(q));
8382         runservice(q);
8383 }

8385 static void
8386 syncq_service(syncq_t *sq)
8387 {
8388         STRSTAT(syncqservice);
8389         mutex_enter(SQLOCK(sq));
8390         ASSERT(!(sq->sq_svcflags & SQ_SERVICE));
8391         ASSERT(sq->sq_servcount != 0);
8392         ASSERT(sq->sq_next == NULL);

8394         /* if we came here from the background thread, clear the flag */
8395         if (sq->sq_svcflags & SQ_BGTHREAD)
8396                 sq->sq_svcflags &= ~SQ_BGTHREAD;

8398         /* let drain_syncq know that it's being called in the background */
8399         sq->sq_svcflags |= SQ_SERVICE;
8400         drain_syncq(sq);
8401 }

8403 static void
8404 qwriter_outer_service(syncq_t *outer)
8405 {
8406         /*
8407          * Note that SQ_WRITER is used on the outer perimeter
8408          * to signal that a qwriter(OUTER) is either investigating
8409          * running or that it is actually running a function.
8410          */
8411         outer_enter(outer, SQ_BLOCKED|SQ_WRITER);

8413         /*
8414          * All inner syncq are empty and have SQ_WRITER set
8415          * to block entering the outer perimeter.
8416          *
8417          * We do not need to explicitly call write_now since
8418          * outer_exit does it for us.
8419          */
8420         outer_exit(outer);
8421 }
```

```
8423 static void
8424 mblk_free(mblk_t *mp)
8425 {
8426         dblk_t *dbp = mp->b_datap;
8427         frtn_t *frp = dbp->db_frtnp;

8429         mp->b_next = NULL;
8430         if (dbp->db_fthdr != NULL)
8431                 str_ftfree(dbp);

8433         ASSERT(dbp->db_fthdr == NULL);
8434         frp->free_func(frp->free_arg);
8435         ASSERT(dbp->db_mblk == mp);

8437         if (dbp->db_credp != NULL) {
8438                 crfree(dbp->db_credp);
8439                 dbp->db_credp = NULL;
8440         }
8441         dbp->db_cpid = -1;
8442         dbp->db_struioflag = 0;
8443         dbp->db_struioun.cksum.flags = 0;

8445         kmem_cache_free(dbp->db_cache, dbp);
8446 }

8448 /*
8449  * Background processing of the stream queue list.
8450  */
8451 static void
8452 stream_service(stdata_t *stp)
8453 {
8454         queue_t *q;

8456         mutex_enter(&stp->sd_qlock);

8458         STR_SERVICE(stp, q);

8460         stp->sd_svcflags &= ~STRS_SCHEDULED;
8461         stp->sd_servid = NULL;
8462         cv_signal(&stp->sd_qcv);
8463         mutex_exit(&stp->sd_qlock);
8464 }

8466 /*
8467  * Foreground processing of the stream queue list.
8468  */
8469 void
8470 stream_runservice(stdata_t *stp)
8471 {
8472         queue_t *q;

8474         mutex_enter(&stp->sd_qlock);
8475         STRSTAT(rservice);
8476         /*
8477          * We are going to drain this stream queue list, so qenable_locked will
8478          * not schedule it until we finish.
8479          */
8480         stp->sd_svcflags |= STRS_WILLSERVICE;

8482         STR_SERVICE(stp, q);

8484         stp->sd_svcflags &= ~STRS_WILLSERVICE;
8485         mutex_exit(&stp->sd_qlock);
8486         /*
8487          * Help backup background thread to drain the qhead/qtail list.
8488          */
```



```
8489         while (qhead != NULL) {
8490                 STRSTAT(qhelps);
8491                 mutex_enter(&service_queue);
8492                 DQ(q, qhead, qtail, q_link);
8493                 mutex_exit(&service_queue);
8494                 if (q != NULL)
8495                         queue_service(q);
8496         }
8497 }

8499 void
8500 stream_willservice(stdata_t *stp)
8501 {
8502         mutex_enter(&stp->sd_qlock);
8503         stp->sd_svcflags |= STRS_WILLSERVICE;
8504         mutex_exit(&stp->sd_qlock);
8505 }

8507 /*
8508  * Replace the cred currently in the mblk with a different one.
8509  * Also update db_cpid.
8510  */
8511 void
8512 mblk_setcred(mblk_t *mp, cred_t *cr, pid_t cpid)
8513 {
8514         dblk_t *dbp = mp->b_datap;
8515         cred_t *ocr = dbp->db_credp;

8517         ASSERT(cr != NULL);

8519         if (cr != ocr) {
8520                 crhold(dbp->db_credp = cr);
8521                 if (ocr != NULL)
8522                         crfree(ocr);
8523         }
8524         /* Don't overwrite with NOPID */
8525         if (cpid != NOPID)
8526                 dbp->db_cpid = cpid;
8527 }

8529 /*
8530  * If the src message has a cred, then replace the cred currently in the mblk
8531  * with it.
8532  * Also update db_cpid.
8533  */
8534 void
8535 mblk_copycred(mblk_t *mp, const mblk_t *src)
8536 {
8537         dblk_t *dbp = mp->b_datap;
8538         cred_t *cr, *ocr;
8539         pid_t cpid;

8541         cr = msg_getcred(src, &cpid);
8542         if (cr == NULL)
8543                 return;

8545         ocr = dbp->db_credp;
8546         if (cr != ocr) {
8547                 crhold(dbp->db_credp = cr);
8548                 if (ocr != NULL)
8549                         crfree(ocr);
8550         }
8551         /* Don't overwrite with NOPID */
8552         if (cpid != NOPID)
8553                 dbp->db_cpid = cpid;
8554 }
```

```
8556 int
8557 hcksum_assoc(mblk_t *mp,  multidata_t *mmd, pdesc_t *pd,
8558     uint32_t start, uint32_t stuff, uint32_t end, uint32_t value,
8559     uint32_t flags, int km_flags)
8560 {
8561         int rc = 0;

8563         ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_MULTIDATA);
8564         if (mp->b_datap->db_type == M_DATA) {
8565                 /* Associate values for M_DATA type */
8566                 DB_CKSUMSTART(mp) = (intptr_t)start;
8567                 DB_CKSUMSTUFF(mp) = (intptr_t)stuff;
8568                 DB_CKSUMEND(mp) = (intptr_t)end;
8569                 DB_CKSUMFLAGS(mp) = flags;
8570                 DB_CKSUM16(mp) = (uint16_t)value;

8572         } else {
8573                 pattrinfo_t pa_info;

8575                 ASSERT(mmd != NULL);

8577                 pa_info.type = PATTR_HCKSUM;
8578                 pa_info.len = sizeof (pattr_hcksum_t);

8580                 if (mmd_addpattr(mmd, pd, &pa_info, B_TRUE, km_flags) != NULL) {
8581                         pattr_hcksum_t *hck = (pattr_hcksum_t *)pa_info.buf;

8583                         hck->hcksum_start_offset = start;
8584                         hck->hcksum_stuff_offset = stuff;
8585                         hck->hcksum_end_offset = end;
8586                         hck->hcksum_cksum_val.inet_cksum = (uint16_t)value;
8587                         hck->hcksum_flags = flags;
8588                 } else {
8589                         rc = -1;
8590                 }
8591         }
8592         return (rc);
8593 }

8595 void
8596 hcksum_retrieve(mblk_t *mp, multidata_t *mmd, pdesc_t *pd,
8597     uint32_t *start, uint32_t *stuff, uint32_t *end,
8598     uint32_t *value, uint32_t *flags)
8599 {
8600         ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_MULTIDATA);
8601         if (mp->b_datap->db_type == M_DATA) {
8602                 if (flags != NULL) {
8603                         *flags = DB_CKSUMFLAGS(mp) & HCK_FLAGS;
8604                         if ((*flags & (HCK_PARTIALCKSUM |
8605                             HCK_FULLCKSUM)) != 0) {
8606                                 if (value != NULL)
8607                                         *value = (uint32_t)DB_CKSUM16(mp);
8608                                 if ((*flags & HCK_PARTIALCKSUM) != 0) {
8609                                         if (start != NULL)
8610                                                 *start =
8611                                                     (uint32_t)DB_CKSUMSTART(mp);
8612                                         if (stuff != NULL)
8613                                                 *stuff =
8614                                                     (uint32_t)DB_CKSUMSTUFF(mp);
8615                                         if (end != NULL)
8616                                                 *end =
8617                                                     (uint32_t)DB_CKSUMEND(mp);
8618                                 }
8619                         }
8620                 }
```



```
8621         } else {
8622                 pattrinfo_t hck_attr = {PATTR_HCKSUM};

8624                 ASSERT(mmd != NULL);

8626                 /* get hardware checksum attribute */
8627                 if (mmd_getpattr(mmd, pd, &hck_attr) != NULL) {
8628                         pattr_hcksum_t *hck = (pattr_hcksum_t *)hck_attr.buf;

8630                         ASSERT(hck_attr.len >= sizeof (pattr_hcksum_t));
8631                         if (flags != NULL)
8632                                 *flags = hck->hcksum_flags;
8633                         if (start != NULL)
8634                                 *start = hck->hcksum_start_offset;
8635                         if (stuff != NULL)
8636                                 *stuff = hck->hcksum_stuff_offset;
8637                         if (end != NULL)
8638                                 *end = hck->hcksum_end_offset;
8639                         if (value != NULL)
8640                                 *value = (uint32_t)
8641                                     hck->hcksum_cksum_val.inet_cksum;
8642                 }
8643         }
8644 }

8646 void
8647 lso_info_set(mblk_t *mp, uint32_t mss, uint32_t flags)
8648 {
8649         ASSERT(DB_TYPE(mp) == M_DATA);
8650         ASSERT((flags & ~HW_LSO_FLAGS) == 0);

8652         /* Set the flags */
8653         DB_LSOFLAGS(mp) |= flags;
8654         DB_LSOMSS(mp) = mss;
8655 }

8657 void
8658 lso_info_cleanup(mblk_t *mp)
8659 {
8660         ASSERT(DB_TYPE(mp) == M_DATA);

8662         /* Clear the flags */
8663         DB_LSOFLAGS(mp) &= ~HW_LSO_FLAGS;
8664         DB_LSOMSS(mp) = 0;
8665 }

8667 /*
8668  * Checksum buffer *bp for len bytes with psum partial checksum,
8669  * or 0 if none, and return the 16 bit partial checksum.
8670  */
8671 unsigned
8672 bcksum(uchar_t *bp, int len, unsigned int psum)
8673 {
8674         int odd = len & 1;
8675         extern unsigned int ip_ocsum();

8677         if (((intptr_t)bp & 1) == 0 && !odd) {
8678                 /*
8679                  * Bp is 16 bit aligned and len is multiple of 16 bit word.
8680                  */
8681                 return (ip_ocsum((ushort_t *)bp, len >> 1, psum));
8682         }
8683         if (((intptr_t)bp & 1) != 0) {
8684                 /*
8685                  * Bp isn't 16 bit aligned.
8686                  */
```

```
8687                 unsigned int tsum;

8689 #ifdef _LITTLE_ENDIAN
8690                 psum += *bp;
8691 #else
8692                 psum += *bp << 8;
8693 #endif
8694                 len--;
8695                 bp++;
8696                 tsum = ip_ocsum((ushort_t *)bp, len >> 1, 0);
8697                 psum += (tsum << 8) & 0xffff | (tsum >> 8);
8698                 if (len & 1) {
8699                         bp += len - 1;
8700 #ifdef _LITTLE_ENDIAN
8701                         psum += *bp << 8;
8702 #else
8703                         psum += *bp;
8704 #endif
8705                 }
8706         } else {
8707                 /*
8708                  * Bp is 16 bit aligned.
8709                  */
8710                 psum = ip_ocsum((ushort_t *)bp, len >> 1, psum);
8711                 if (odd) {
8712                         bp += len - 1;
8713 #ifdef _LITTLE_ENDIAN
8714                         psum += *bp;
8715 #else
8716                         psum += *bp << 8;
8717 #endif
8718                 }
8719         }
8720         /*
8721          * Normalize psum to 16 bits before returning the new partial
8722          * checksum. The max psum value before normalization is 0x3FDFE.
8723          */
8724         return ((psum >> 16) + (psum & 0xFFFF));
8725 }

8727 boolean_t
8728 is_vmloaned_mblk(mblk_t *mp, multidata_t *mmd, pdesc_t *pd)
8729 {
8730         boolean_t rc;

8732         ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_MULTIDATA);
8733         if (DB_TYPE(mp) == M_DATA) {
8734                 rc = (((mp)->b_datap->db_struioflag & STRUIO_ZC) != 0);
8735         } else {
8736                 pattrinfo_t zcopy_attr = {PATTR_ZCOPY};

8738                 ASSERT(mmd != NULL);
8739                 rc = (mmd_getpattr(mmd, pd, &zcopy_attr) != NULL);
8740         }
8741         return (rc);
8742 }

8744 void
8745 freemsgchain(mblk_t *mp)
8746 {
8747         mblk_t  *next;

8749         while (mp != NULL) {
8750                 next = mp->b_next;
8751                 mp->b_next = NULL;
```



```
8753                 freemsg(mp);
8754                 mp = next;
8755         }
8756 }

8758 mblk_t *
8759 copymsgchain(mblk_t *mp)
8760 {
8761         mblk_t  *nmp = NULL;
8762         mblk_t  **nmpp = &nmp;

8764         for (; mp != NULL; mp = mp->b_next) {
8765                 if ((*nmpp = copymsg(mp)) == NULL) {
8766                         freemsgchain(nmp);
8767                         return (NULL);
8768                 }

8770                 nmpp = &((*nmpp)->b_next);
8771         }

8773         return (nmp);
8774 }

8776 /* NOTE: Do not add code after this point. */
8777 #undef QLOCK

8779 /*
8780  * Replacement for QLOCK macro for those that can't use it.
8781  */
8782 kmutex_t *
8783 QLOCK(queue_t *q)
8784 {
8785         return (&(q)->q_lock);
8786 }

8788 /*
8789  * Dummy runqueues/queuerun functions functions for backwards compatibility.
8790  */
8791 #undef runqueues
8792 void
8793 runqueues(void)
8794 {
8795 }

8797 #undef queuerun
8798 void
8799 queuerun(void)
8800 {
8801 }

8803 /*
8804  * Initialize the STR stack instance, which tracks autopush and persistent
8805  * links.
8806  */
8807 /* ARGSUSED */
8808 static void *
8809 str_stack_init(netstackid_t stackid, netstack_t *ns)
8810 {
8811         str_stack_t     *ss;
8812         int i;

8814         ss = (str_stack_t *)kmem_zalloc(sizeof (*ss), KM_SLEEP);
8815         ss->ss_netstack = ns;

8817         /*
8818          * set up autopush
```

```
8819          */
8820         sad_initspace(ss);

8822         /*
8823          * set up mux_node structures.
8824          */
8825         ss->ss_devcnt = devcnt; /* In case it should change before free */
8826         ss->ss_mux_nodes = kmem_zalloc((sizeof (struct mux_node) *
8827             ss->ss_devcnt), KM_SLEEP);
8828         for (i = 0; i < ss->ss_devcnt; i++)
8829                 ss->ss_mux_nodes[i].mn_imaj = i;
8830         return (ss);
8831 }

8833 /*
8834  * Note: run at zone shutdown and not destroy so that the PLINKs are
8835  * gone by the time other cleanup happens from the destroy callbacks.
8836  */
8837 static void
8838 str_stack_shutdown(netstackid_t stackid, void *arg)
8839 {
8840         str_stack_t *ss = (str_stack_t *)arg;
8841         int i;
8842         cred_t *cr;

8844         cr = zone_get_kcred(netstackid_to_zoneid(stackid));
8845         ASSERT(cr != NULL);

8847         /* Undo all the I_PLINKs for this zone */
8848         for (i = 0; i < ss->ss_devcnt; i++) {
8849                 struct mux_edge         *ep;
8850                 ldi_handle_t            lh;
8851                 ldi_ident_t             li;
8852                 int                     ret;
8853                 int                     rval;
8854                 dev_t                   rdev;

8856                 ep = ss->ss_mux_nodes[i].mn_outp;
8857                 if (ep == NULL)
8858                         continue;
8859                 ret = ldi_ident_from_major((major_t)i, &li);
8860                 if (ret != 0) {
8861                         continue;
8862                 }
8863                 rdev = ep->me_dev;
8864                 ret = ldi_open_by_dev(&rdev, OTYP_CHR, FREAD|FWRITE,
8865                     cr, &lh, li);
8866                 if (ret != 0) {
8867                         ldi_ident_release(li);
8868                         continue;
8869                 }

8871                 ret = ldi_ioctl(lh, I_PUNLINK, (intptr_t)MUXID_ALL, FKIOCTL,
8872                     cr, &rval);
8873                 if (ret) {
8874                         (void) ldi_close(lh, FREAD|FWRITE, cr);
8875                         ldi_ident_release(li);
8876                         continue;
8877                 }
8878                 (void) ldi_close(lh, FREAD|FWRITE, cr);

8880                 /* Close layered handles */
8881                 ldi_ident_release(li);
8882         }
8883         crfree(cr);
```



```
8885         sad_freespace(ss);

8887         kmem_free(ss->ss_mux_nodes, sizeof (struct mux_node) * ss->ss_devcnt);
8888         ss->ss_mux_nodes = NULL;
8889 }

8891 /*
8892  * Free the structure; str_stack_shutdown did the other cleanup work.
8893  */
8894 /* ARGSUSED */
8895 static void
8896 str_stack_fini(netstackid_t stackid, void *arg)
8897 {
8898         str_stack_t     *ss = (str_stack_t *)arg;

8900         kmem_free(ss, sizeof (*ss));
8901 }
```

```
**********************************************************
   22101 Sun Aug  9 12:48:05 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:
```

```
**********************************************************
   11441 Sun Aug  9 12:48:06 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_FORKED        29
169 #define F_CLOSED        30

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 Sun Aug  9 12:48:07 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_*

```
**********************************************************
    1909 Sun Aug  9 12:48:08 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 *);
uint64_t list_size(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 */
```

```
**********************************************************
    2074 Sun Aug  9 12:48:09 2015
new/usr/src/uts/common/sys/pidnode.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 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_NODE_LIST_HDR_MAGIC    0x5A7A0B1D

#define CONN_PID_NODE_LIST_HDR_NON      0
#define CONN_PID_NODE_LIST_HDR_SOC      1
#define CONN_PID_NODE_LIST_HDR_XTI      2
#define CONN_PID_NODE_LIST_HDR_UNX      3

typedef struct conn_pid_node_s {
        uint32_t        cpn_pid;
} conn_pid_node_t;

typedef struct conn_pid_node_list_hdr_s {
        uint32_t        cph_magic;      /* CONN_PID_NODE_LIST_HDR_MAGIC */
        uint32_t        cph_contents;   /* CONN_PID_NODE_LIST_HDR_* */
        uint64_t        cph_pn_cnt;     /* # of conn_pid_node_t(s) */
        uint64_t        cph_tot_size;   /* total size of hdr + nodes */
        uint32_t        cph_flags;      /* not used yet */
        uint32_t        cph_optional1;  /* an optional field, not used yet */
        uint64_t        cph_optional2;  /* an optional field, not used yet */
        conn_pid_node_t cph_cpns[1];    /* arrary of conn_pid_node_t */
} conn_pid_node_list_hdr_t;

#if defined(_KERNEL)

typedef struct pid_node_s {
        list_node_t     pn_ref_link;
        uint32_t        pn_count;
        uint32_t        pn_pid;
```



```
} pid_node_t;

#endif /* defined(_KERNEL) */

#define PIDNODE2CONNPIDNODE(pn, cpn)    (cpn)->cpn_pid = (pn)->pn_pid

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_PIDNODE_H */
#endif /* ! codereview */
```

```
**********************************************************
    8355 Sun Aug  9 12:48:09 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 *);
};
```

```
typedef sock_lower_handle_t (*so_proto_create_func_t)(int, int, int,
    sock_downcalls_t **, uint_t *, int *, int, cred_t *);

typedef struct sock_quiesce_arg {
        mblk_t *soqa_exdata_mp;
        mblk_t *soqa_urgmark_mp;
} sock_quiesce_arg_t;
typedef mblk_t *(*so_proto_quiesced_cb_t)(sock_upper_handle_t,
    sock_quiesce_arg_t *, struct T_capability_ack *, struct sockaddr *,
    socklen_t, struct sockaddr *, socklen_t, short);
typedef int (*so_proto_fallback_func_t)(sock_lower_handle_t, queue_t *,
    boolean_t, so_proto_quiesced_cb_t, sock_quiesce_arg_t *);

/*
 * These functions return EOPNOTSUPP and are intended for the sockfs
 * developer that doesn't wish to supply stubs for every function themselves.
 */
extern int sock_accept_notsupp(sock_lower_handle_t, sock_lower_handle_t,
    sock_upper_handle_t, cred_t *);
extern int sock_bind_notsupp(sock_lower_handle_t, struct sockaddr *,
    socklen_t, cred_t *);
extern int sock_listen_notsupp(sock_lower_handle_t, int, cred_t *);
extern int sock_connect_notsupp(sock_lower_handle_t,
    const struct sockaddr *, socklen_t, sock_connid_t *, cred_t *);
extern int sock_getpeername_notsupp(sock_lower_handle_t, struct sockaddr *,
    socklen_t *, cred_t *);
extern int sock_getsockname_notsupp(sock_lower_handle_t, struct sockaddr *,
    socklen_t *, cred_t *);
extern int sock_getsockopt_notsupp(sock_lower_handle_t, int, int, void *,
    socklen_t *, cred_t *);
extern int sock_setsockopt_notsupp(sock_lower_handle_t, int, int,
    const void *, socklen_t, cred_t *);
extern int sock_send_notsupp(sock_lower_handle_t, mblk_t *,
    struct nmsghdr *, cred_t *);
extern int sock_send_uio_notsupp(sock_lower_handle_t, uio_t *,
    struct nmsghdr *, cred_t *);
extern int sock_recv_uio_notsupp(sock_lower_handle_t, uio_t *,
    struct nmsghdr *, cred_t *);
extern short sock_poll_notsupp(sock_lower_handle_t, short, int, cred_t *);
extern int sock_shutdown_notsupp(sock_lower_handle_t, int, cred_t *);
extern void sock_clr_flowctrl_notsupp(sock_lower_handle_t);
extern int sock_ioctl_notsupp(sock_lower_handle_t, int, intptr_t, int,
    int32_t *, cred_t *);
extern int sock_close_notsupp(sock_lower_handle_t, int, cred_t *);

/*
 * Upcalls and related information
 */

/*
 * su_opctl() actions
 */
typedef enum sock_opctl_action {
        SOCK_OPCTL_ENAB_ACCEPT = 0,
        SOCK_OPCTL_SHUT_SEND,
        SOCK_OPCTL_SHUT_RECV
} sock_opctl_action_t;

struct sock_upcalls_s {
        sock_upper_handle_t (*su_newconn)(sock_upper_handle_t,
            sock_lower_handle_t, sock_downcalls_t *, cred_t *, pid_t,
            sock_upcalls_t **);
        void    (*su_connected)(sock_upper_handle_t, sock_connid_t, cred_t *,
            pid_t);
        int     (*su_disconnected)(sock_upper_handle_t, sock_connid_t, int);
```



```
        void    (*su_opctl)(sock_upper_handle_t, sock_opctl_action_t,
            uintptr_t);
        ssize_t (*su_recv)(sock_upper_handle_t, mblk_t *, size_t, int,
                int *, boolean_t *);
        void    (*su_set_proto_props)(sock_upper_handle_t,
                struct sock_proto_props *);
        void    (*su_txq_full)(sock_upper_handle_t, boolean_t);
        void    (*su_signal_oob)(sock_upper_handle_t, ssize_t);
        void    (*su_zcopy_notify)(sock_upper_handle_t);
        void    (*su_set_error)(sock_upper_handle_t, int);
        void    (*su_closed)(sock_upper_handle_t);
        conn_pid_node_list_hdr_t *      (*su_get_sock_pid_list)
                        (sock_upper_handle_t);
#endif /* ! codereview */
};

#define SOCK_UC_VERSION         sizeof (sock_upcalls_t)
#define SOCK_DC_VERSION         sizeof (sock_downcalls_t)

#define SOCKET_RECVHIWATER      (48 * 1024)
#define SOCKET_RECVLOWATER      1024

#define SOCKET_NO_RCVTIMER      0
#define SOCKET_TIMER_INTERVAL   50

#endif /* _KERNEL */

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_SOCKET_PROTO_H_ */
```

```
**********************************************************
   35371 Sun Aug  9 12:48:10 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.
 */

#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;
        conn_pid_node_t si_pns[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];
        char            se_strdev[MAXPATHLEN];
} sockconfig_socktable_entry_t;

typedef struct sockconfig_socktable {
```

```
1052         uint_t          num_of_entries;
1053         sockconfig_socktable_entry_t *st_entries;
1054 } sockconfig_socktable_t;

1056 #ifdef  _SYSCALL32

1058 typedef struct sof_socktuple32 {
1059         int32_t sofst_family;
1060         int32_t sofst_type;
1061         int32_t sofst_protocol;
1062 } sof_socktuple32_t;

1064 struct sockconfig_filter_props32 {
1065         caddr32_t       sfp_modname;
1066         boolean_t       sfp_autoattach;
1067         sof_hint_t      sfp_hint;
1068         caddr32_t       sfp_hintarg;
1069         uint32_t        sfp_socktuple_cnt;
1070         caddr32_t       sfp_socktuple;
1071 };

1073 typedef struct sockconfig_socktable32 {
1074         uint_t          num_of_entries;
1075         caddr32_t       st_entries;
1076 } sockconfig_socktable32_t;

1078 #endif  /* _SYSCALL32 */

1080 #define SOCKMOD_PATH    "socketmod"     /* dir where sockmods are stored */

1082 #ifdef  __cplusplus
1083 }
1084 #endif

1086 #endif  /* _SYS_SOCKETVAR_H */
```

```
**********************************************************
   48810 Sun Aug  9 12:48:10 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 *, proc_t *);
extern void sh_remove_pid(struct stdata *, proc_t *);
extern conn_pid_node_list_hdr_t *sh_get_pid_list(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 */
```

```
**********************************************************
   21151 Sun Aug  9 12:48:11 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_FORKED,
184                                 (intptr_t) p, FKIOCTL,
185                                 kcred, 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
228                         if (fp->f_vnode != NULL) {
229                                 (void) VOP_IOCTL(fp->f_vnode, F_FORKED,
230                                         (intptr_t) p, FKIOCTL,
231                                         kcred, NULL, NULL);
232                         }
233
234 #endif /* ! codereview */
235                         if ((error = closeandsetf(iarg, fp)) == 0) {
236                                 if (cmd == F_DUP2FD_CLOEXEC) {
237                                         f_setfd(iarg, FD_CLOEXEC);
238                                 }
239                                 retval = iarg;
240                         } else {
241                                 mutex_enter(&fp->f_tlock);
242                                 if (fp->f_count > 1) {
243                                         fp->f_count--;
244                                         mutex_exit(&fp->f_tlock);
245                                         if (fp->f_vnode != NULL) {
246                                                 VOP_IOCTL(fp->f_vnode, F_CLOSED,
247                                                         (intptr_t) p, FKIOCTL,
248                                                         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 }
```