

new/usr/src/cmd/cmd-inet/usr.bin/netstat/Makefile

1

1938 Wed Jul 13 01:32:34 2016

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 LOCALOBSJ= netstat.o
31 LOCALOBSJ= netstat.o unix.o
32 COMMONOBSJ= compat.o
```

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

```
37 LOCALSRCS= $(LOCALOBSJ:%.o=%.c)
38 COMMONSRCS= $(CMDINETCOMMONDIR)/$(COMMONOBSJ:%.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 OBJ= $(LOCALOBSJ) $(COMMONOBSJ) $(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 += -ldhcapagent -lsocket -lnsl -lkstat -ltsnet -ltsol
```

```
53 .KEEP_STATE:
```

```
55 all: $(PROG) $(NPROG)
```

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

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

new/usr/src/cmd/cmd-inet/usr.bin/netstat/Makefile

2

```
61 $(POST_PROCESS)
```

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

```
67 install: all $(ROOTPROG)
```

```
69 clean:
70 $(RM) $(OBJ)
```

```
72 lint: lint_SRCS
```

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

```

*****
203932 Wed Jul 13 01:32:34 2016
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 <dhcpageant_ipc.h>
95 #include <dhcpageant_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] = 0xffffffffful, \
107                                (v6)._S6_un._S6_u32[1] = 0xffffffffful, \
108                                (v6)._S6_un._S6_u32[2] = 0xffffffffful, \
109                                (v6)._S6_un._S6_u32[3] = (v4))

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

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;

```

```

127     int             mib_id;
128     int             length;
129     void            *valp;
130 } mib_item_t;
_____
146 typedef struct proc_info {
147     char *pr_user;
148     char *pr_fname;
149     char *pr_psargs;
150 } proc_info_t;

152 #endif /* ! codereview */
153 static mib_item_t *mibget(int sd);
154 static void mibfree(mib_item_t *firstitem);
155 static int mibopen(void);
156 static void mib_get_constants(mib_item_t *item);
157 static mib_item_t *mib_item_dup(mib_item_t *item);
158 static mib_item_t *mib_item_diff(mib_item_t *item1,
159     mib_item_t *item2);
160 static void mib_item_destroy(mib_item_t **item);

162 static boolean_t octetstrmatch(const Octet_t *a, const Octet_t *b);
163 static char *octetstr(const Octet_t *op, int code,
164     char *dst, uint_t dstlen);
165 static char *pr_addr(uint_t addr,
166     char *dst, uint_t dstlen);
167 static char *pr_addrnz(ipaddr_t addr, char *dst, uint_t dstlen);
168 static char *pr_addr6(const in6_addr_t *addr,
169     char *dst, uint_t dstlen);
170 static char *pr_mask(uint_t addr,
171     char *dst, uint_t dstlen);
172 static char *pr_prefix6(const struct in6_addr *addr,
173     uint_t prefixlen, char *dst, uint_t dstlen);
174 static char *pr_ap(uint_t addr, uint_t port,
175     char *proto, char *dst, uint_t dstlen);
176 static char *pr_ap6(const in6_addr_t *addr, uint_t port,
177     char *proto, char *dst, uint_t dstlen);
178 static char *pr_net(uint_t addr, uint_t mask,
179     char *dst, uint_t dstlen);
180 static char *pr_netaddr(uint_t addr, uint_t mask,
181     char *dst, uint_t dstlen);
182 static char *fmodestr(uint_t fmode);
183 static char *portname(uint_t port, char *proto,
184     char *dst, uint_t dstlen);

186 static const char *mitcp_state(int code,
187     const mib2_transportMLPEntry_t *attr);
188 static const char *miudp_state(int code,
189     const mib2_transportMLPEntry_t *attr);

191 static void stat_report(mib_item_t *item);
192 static void mrt_stat_report(mib_item_t *item);
193 static void arp_report(mib_item_t *item);
194 static void ndp_report(mib_item_t *item);
195 static void mrt_report(mib_item_t *item);
196 static void if_stat_total(struct ifstat *oldstats,
197     struct ifstat *newstats, struct ifstat *sumstats);
198 static void if_report(mib_item_t *item, char *ifname,
199     int iflag_only, boolean_t once_only);
200 static void if_report_ip4(mib2_ipAddrEntry_t *ap,
201     char ifname[], char loginname[],
202     struct ifstat *statptr, boolean_t ksp_not_null);
203 static void if_report_ip6(mib2_ipv6AddrEntry_t *ap6,
204     char ifname[], char loginname[],
205     struct ifstat *statptr, boolean_t ksp_not_null);

```

```

206 static void ire_report(const mib_item_t *item);
207 static void tcp_report(const mib_item_t *item);
208 static void udp_report(const mib_item_t *item);
209 static void uds_report(kstat_ctl_t *);
210 #endif /* ! codereview */
211 static void group_report(mib_item_t *item);
212 static void dce_report(mib_item_t *item);
213 static void print_ip_stats(mib2_ip_t *ip);
214 static void print_icmp_stats(mib2_icmp_t *icmp);
215 static void print_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6);
216 static void print_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6);
217 static void print_sctp_stats(mib2_sctp_t *tcp);
218 static void print_tcp_stats(mib2_tcp_t *tcp);
219 static void print_udp_stats(mib2_udp_t *udp);
220 static void print_rawip_stats(mib2_rawip_t *rawip);
221 static void print_igmp_stats(struct igmpstat *igps);
222 static void print_mrt_stats(struct mrtstat *mrts);
223 static void sctp_report(const mib_item_t *item);
224 static void sum_ip6_stats(mib2_ipv6IfStatsEntry_t *ip6,
225     mib2_ipv6IfStatsEntry_t *sum6);
226 static void sum_icmp6_stats(mib2_ipv6IfIcmpEntry_t *icmp6,
227     mib2_ipv6IfIcmpEntry_t *sum6);
228 static void m_report(void);
229 static void dhcp_report(char *);

231 static uint64_t kstat_named_value(kstat_t *, char *);
232 static kid_t safe_kstat_read(kstat_ctl_t *, kstat_t *, void *);
233 static int isnum(char *);
234 static char *plural(int n);
235 static char *plurality(int n);
236 static char *plurales(int n);
237 static void process_filter(char *arg);
238 static char *ifindex2str(uint_t, char *);
239 static boolean_t family_selected(int family);

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

247 #define PLURAL(n) plural((int)n)
248 #define PLURALY(n) plurality((int)n)
249 #define PLURALES(n) plurales((int)n)
250 #define IFLAGMOD(flg, val1, val2) if (flg == val1) flg = val2
251 #define MDIFF(diff, elem2, elem1, member) (diff)->member = \
252     (elem2)->member - (elem1)->member

255 static boolean_t Aflag = B_FALSE; /* All sockets/ifs/rtnng-tbls */
256 static boolean_t Dflag = B_FALSE; /* DCE info */
257 static boolean_t Iflag = B_FALSE; /* IP Traffic Interfaces */
258 static boolean_t Mflag = B_FALSE; /* STREAMS Memory Statistics */
259 static boolean_t Nflag = B_FALSE; /* Numeric Network Addresses */
260 static boolean_t Rflag = B_FALSE; /* Routing Tables */
261 static boolean_t RSECflag = B_FALSE; /* Security attributes */
262 static boolean_t Sflag = B_FALSE; /* Per-protocol Statistics */
263 static boolean_t Vflag = B_FALSE; /* Verbose */
264 static boolean_t Uflag = B_FALSE; /* Show PID and UID info. */
265 #endif /* ! codereview */
266 static boolean_t Pflag = B_FALSE; /* Net to Media Tables */
267 static boolean_t Gflag = B_FALSE; /* Multicast group membership */
268 static boolean_t MMflag = B_FALSE; /* Multicast routing table */
269 static boolean_t DHCPflag = B_FALSE; /* DHCP statistics */
270 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 mfctlSize;

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     0x00010000 /* 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, "adimnrspMgvxf: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;

```



```

667         if (Gflag)
668             group_report(item);
669         if (Pflag) {
670             if (family_selected(AF_INET))
671                 arp_report(item);
672             if (family_selected(AF_INET6))
673                 ndp_report(item);
674         }
675         if (Dflag)
676             dce_report(item);
677         mib_item_destroy(&curritem);
678     }
679
680     /* netstat: AF_UNIX behaviour */
681     if (family_selected(AF_UNIX) &&
682         (!(Dflag || Iflag || Rflag || Sflag || Mflag ||
683          MMflag || Pflag || Gflag)))
684         uds_report(kc);
685     unixpr(kc);
686     (void) kstat_close(kc);
687
688     /* iteration handling code */
689     if (count > 0 && --count == 0)
690         break;
691     (void) sleep(interval);
692
693     /* re-populating of data structures */
694     if (family_selected(AF_INET) || family_selected(AF_INET6)) {
695         if (Sflag) {
696             /* previtem is a cut-down list */
697             previtem = mib_item_dup(item);
698             if (previtem == NULL)
699                 fatal(1, "can't process mib data, "
700                    "out of memory\n");
701         }
702         mibfree(item);
703         (void) close(sd);
704         if ((sd = mibopen()) == -1)
705             fatal(1, "can't open mib stream anymore\n");
706         if ((item = mibget(sd)) == NULL) {
707             (void) close(sd);
708             fatal(1, "mibget() failed\n");
709         }
710     }
711     if ((kc = kstat_open()) == NULL)
712         fail(1, "kstat_open(): can't open /dev/kstat");
713
714     } /* 'for' loop 1 ends */
715     mibfree(item);
716     (void) close(sd);
717     if (zone_security_label != NULL)
718         m_label_free(zone_security_label);
719
720     return (0);
721 }

```

unchanged portion omitted

```

992 /*
993 * mib_item_diff: takes two (mib_item_t *) linked lists
994 * item1 and item2 and computes the difference between
995 * differentiable values in item2 against item1 for every
996 * given member of item2; returns an mib_item_t * linked
997 * list of diff's, or a copy of item2 if item1 is NULL;
998 * will return NULL if system out of memory; works only
999 * for item->mib_id == 0
1000 */

```

```

1001 static mib_item_t *
1002 mib_item_diff(mib_item_t *item1, mib_item_t *item2)
1003 {
1004     480 mib_item_diff(mib_item_t *item1, mib_item_t *item2) {
1005         int nitems = 0; /* no. of items in item2 */
1006         mib_item_t *tempp2; /* walking copy of item2 */
1007         mib_item_t *tempp1; /* walking copy of item1 */
1008         mib_item_t *diffp;
1009         mib_item_t *diffptr; /* walking copy of diffp */
1010         mib_item_t *prevp = NULL;
1011
1012         if (item1 == NULL) {
1013             diffp = mib_item_dup(item2);
1014             return (diffp);
1015         }
1016
1017         for (tempp2 = item2;
1018             tempp2 = tempp2->next_item) {
1019             if (tempp2->mib_id == 0)
1020                 switch (tempp2->group) {
1021                     /*
1022                      * upon adding a case here, the same
1023                      * must also be added in the next
1024                      * switch statement, alongwith
1025                      * appropriate code
1026                      */
1027                     case MIB2_IP:
1028                     case MIB2_IP6:
1029                     case EXPER_DVMRP:
1030                     case EXPER_IGMP:
1031                     case MIB2_ICMP:
1032                     case MIB2_ICMP6:
1033                     case MIB2_TCP:
1034                     case MIB2_UDP:
1035                     case MIB2_SCTP:
1036                     case EXPER_RAWIP:
1037                         nitems++;
1038                 }
1039             }
1040         }
1041         tempp2 = NULL;
1042         if (nitems == 0) {
1043             diffp = mib_item_dup(item2);
1044             return (diffp);
1045         }
1046
1047         diffp = (mib_item_t *)calloc(nitems, sizeof (mib_item_t));
1048         if (diffp == NULL)
1049             return (NULL);
1050         diffptr = diffp;
1051         /* 'for' loop 1: */
1052         for (tempp2 = item2; tempp2 != NULL; tempp2 = tempp2->next_item) {
1053             if (tempp2->mib_id != 0)
1054                 continue; /* 'for' loop 1 */
1055             /* 'for' loop 2: */
1056             for (tempp1 = item1; tempp1 != NULL;
1057                 tempp1 = tempp1->next_item) {
1058                 if (!(tempp1->mib_id == 0 &&
1059                     tempp1->group == tempp2->group &&
1060                     tempp1->mib_id == tempp2->mib_id))
1061                     continue; /* 'for' loop 2 */
1062                 /* found comparable data sets */
1063                 if (prevp != NULL)
1064                     prevp->next_item = diffptr;
1065                 switch (tempp2->group) {
1066                     /*

```

```

1066     * Indenting note: Because of long variable names
1067     * in cases MIB2_IP6 and MIB2_ICMP6, their contents
1068     * have been indented by one tab space only
1069     */
1070     case MIB2_IP: {
1071         mib2_ip_t *i2 = (mib2_ip_t *)tempp2->valp;
1072         mib2_ip_t *i1 = (mib2_ip_t *)tempp1->valp;
1073         mib2_ip_t *d;

1075         diffptr->group = tempp2->group;
1076         diffptr->mib_id = tempp2->mib_id;
1077         diffptr->length = tempp2->length;
1078         d = (mib2_ip_t *)calloc(tempp2->length, 1);
1079         if (d == NULL)
1080             goto mibdiff_out_of_memory;
1081         diffptr->valp = d;
1082         d->ipForwarding = i2->ipForwarding;
1083         d->ipDefaultTTL = i2->ipDefaultTTL;
1084         MDIFF(d, i2, i1, ipInReceives);
1085         MDIFF(d, i2, i1, ipInHdrErrors);
1086         MDIFF(d, i2, i1, ipInAddrErrors);
1087         MDIFF(d, i2, i1, ipInCksumErrs);
1088         MDIFF(d, i2, i1, ipForwDatagrams);
1089         MDIFF(d, i2, i1, ipForwProhibits);
1090         MDIFF(d, i2, i1, ipInUnknownProtos);
1091         MDIFF(d, i2, i1, ipInDiscards);
1092         MDIFF(d, i2, i1, ipInDelivers);
1093         MDIFF(d, i2, i1, ipOutRequests);
1094         MDIFF(d, i2, i1, ipOutDiscards);
1095         MDIFF(d, i2, i1, ipOutNoRoutes);
1096         MDIFF(d, i2, i1, ipReasmTimeout);
1097         MDIFF(d, i2, i1, ipReasmReqds);
1098         MDIFF(d, i2, i1, ipReasmOKs);
1099         MDIFF(d, i2, i1, ipReasmFails);
1100         MDIFF(d, i2, i1, ipReasmDuplicates);
1101         MDIFF(d, i2, i1, ipReasmPartDups);
1102         MDIFF(d, i2, i1, ipFragOKs);
1103         MDIFF(d, i2, i1, ipFragFails);
1104         MDIFF(d, i2, i1, ipFragCreates);
1105         MDIFF(d, i2, i1, ipRoutingDiscards);
1106         MDIFF(d, i2, i1, tcpInErrs);
1107         MDIFF(d, i2, i1, udpNoPorts);
1108         MDIFF(d, i2, i1, udpInCksumErrs);
1109         MDIFF(d, i2, i1, udpInOverflows);
1110         MDIFF(d, i2, i1, rawipInOverflows);
1111         MDIFF(d, i2, i1, ipsecInSucceeded);
1112         MDIFF(d, i2, i1, ipsecInFailed);
1113         MDIFF(d, i2, i1, ipInIPv6);
1114         MDIFF(d, i2, i1, ipOutIPv6);
1115         MDIFF(d, i2, i1, ipOutSwitchIPv6);
1116         prevp = diffptr++;
1117         break;
1118     }
1119     case MIB2_IP6: {
1120         mib2_ipv6IfStatsEntry_t *i2;
1121         mib2_ipv6IfStatsEntry_t *i1;
1122         mib2_ipv6IfStatsEntry_t *d;

1124         i2 = (mib2_ipv6IfStatsEntry_t *)tempp2->valp;
1125         i1 = (mib2_ipv6IfStatsEntry_t *)tempp1->valp;
1126         diffptr->group = tempp2->group;
1127         diffptr->mib_id = tempp2->mib_id;
1128         diffptr->length = tempp2->length;
1129         d = (mib2_ipv6IfStatsEntry_t *)calloc(
1130             tempp2->length, 1);
1131         if (d == NULL)

```

```

1132             goto mibdiff_out_of_memory;
1133         diffptr->valp = d;
1134         d->ipv6Forwarding = i2->ipv6Forwarding;
1135         d->ipv6DefaultHopLimit =
1136             i2->ipv6DefaultHopLimit;

1138         MDIFF(d, i2, i1, ipv6InReceives);
1139         MDIFF(d, i2, i1, ipv6InHdrErrors);
1140         MDIFF(d, i2, i1, ipv6InTooBigErrors);
1141         MDIFF(d, i2, i1, ipv6InNoRoutes);
1142         MDIFF(d, i2, i1, ipv6InAddrErrors);
1143         MDIFF(d, i2, i1, ipv6InUnknownProtos);
1144         MDIFF(d, i2, i1, ipv6InTruncatedPkts);
1145         MDIFF(d, i2, i1, ipv6InDiscards);
1146         MDIFF(d, i2, i1, ipv6InDelivers);
1147         MDIFF(d, i2, i1, ipv6OutForwDatagrams);
1148         MDIFF(d, i2, i1, ipv6OutRequests);
1149         MDIFF(d, i2, i1, ipv6OutDiscards);
1150         MDIFF(d, i2, i1, ipv6OutNoRoutes);
1151         MDIFF(d, i2, i1, ipv6OutFragOKs);
1152         MDIFF(d, i2, i1, ipv6OutFragFails);
1153         MDIFF(d, i2, i1, ipv6OutFragCreates);
1154         MDIFF(d, i2, i1, ipv6ReasmReqds);
1155         MDIFF(d, i2, i1, ipv6ReasmOKs);
1156         MDIFF(d, i2, i1, ipv6ReasmFails);
1157         MDIFF(d, i2, i1, ipv6InMcastPkts);
1158         MDIFF(d, i2, i1, ipv6OutMcastPkts);
1159         MDIFF(d, i2, i1, ipv6ReasmDuplicates);
1160         MDIFF(d, i2, i1, ipv6ReasmPartDups);
1161         MDIFF(d, i2, i1, ipv6ForwProhibits);
1162         MDIFF(d, i2, i1, udpInCksumErrs);
1163         MDIFF(d, i2, i1, udpInOverflows);
1164         MDIFF(d, i2, i1, rawipInOverflows);
1165         MDIFF(d, i2, i1, ipv6InIPv4);
1166         MDIFF(d, i2, i1, ipv6OutIPv4);
1167         MDIFF(d, i2, i1, ipv6OutSwitchIPv4);
1168         prevp = diffptr++;
1169         break;
1170     }
1171     case EXPER_DVMRP: {
1172         struct mrtstat *m2;
1173         struct mrtstat *m1;
1174         struct mrtstat *d;

1176         m2 = (struct mrtstat *)tempp2->valp;
1177         m1 = (struct mrtstat *)tempp1->valp;
1178         diffptr->group = tempp2->group;
1179         diffptr->mib_id = tempp2->mib_id;
1180         diffptr->length = tempp2->length;
1181         d = (struct mrtstat *)calloc(tempp2->length, 1);
1182         if (d == NULL)
1183             goto mibdiff_out_of_memory;
1184         diffptr->valp = d;
1185         MDIFF(d, m2, m1, mrts_mfc_hits);
1186         MDIFF(d, m2, m1, mrts_mfc_misses);
1187         MDIFF(d, m2, m1, mrts_fwd_in);
1188         MDIFF(d, m2, m1, mrts_fwd_out);
1189         d->mrts_upcalls = m2->mrts_upcalls;
1190         MDIFF(d, m2, m1, mrts_fwd_drop);
1191         MDIFF(d, m2, m1, mrts_bad_tunnel);
1192         MDIFF(d, m2, m1, mrts_cant_tunnel);
1193         MDIFF(d, m2, m1, mrts_wrong_if);
1194         MDIFF(d, m2, m1, mrts_upq_ovflw);
1195         MDIFF(d, m2, m1, mrts_cache_cleanup);
1196         MDIFF(d, m2, m1, mrts_drop_sel);
1197         MDIFF(d, m2, m1, mrts_q_overflow);

```

```

1198         MDIFF(d, m2, ml, mrts_pkt2large);
1199         MDIFF(d, m2, ml, mrts_pim_badversion);
1200         MDIFF(d, m2, ml, mrts_pim_rcv_badcsum);
1201         MDIFF(d, m2, ml, mrts_pim_badregisters);
1202         MDIFF(d, m2, ml, mrts_pim_regforwards);
1203         MDIFF(d, m2, ml, mrts_pim_regsend_drops);
1204         MDIFF(d, m2, ml, mrts_pim_malformed);
1205         MDIFF(d, m2, ml, mrts_pim_nomemory);
1206         prevp = diffptr++;
1207         break;
1208     }
1209     case EXPER_IGMP: {
1210         struct igmpstat *i2;
1211         struct igmpstat *i1;
1212         struct igmpstat *d;

1214         i2 = (struct igmpstat *)tempp2->valp;
1215         i1 = (struct igmpstat *)tempp1->valp;
1216         diffptr->group = tempp2->group;
1217         diffptr->mib_id = tempp2->mib_id;
1218         diffptr->length = tempp2->length;
1219         d = (struct igmpstat *)calloc(
1220             tempp2->length, 1);
1221         if (d == NULL)
1222             goto mibdiff_out_of_memory;
1223         diffptr->valp = d;
1224         MDIFF(d, i2, i1, igps_rcv_total);
1225         MDIFF(d, i2, i1, igps_rcv_tooshort);
1226         MDIFF(d, i2, i1, igps_rcv_badsum);
1227         MDIFF(d, i2, i1, igps_rcv_queries);
1228         MDIFF(d, i2, i1, igps_rcv_badqueries);
1229         MDIFF(d, i2, i1, igps_rcv_reports);
1230         MDIFF(d, i2, i1, igps_rcv_badreports);
1231         MDIFF(d, i2, i1, igps_rcv_ourreports);
1232         MDIFF(d, i2, i1, igps_snd_reports);
1233         prevp = diffptr++;
1234         break;
1235     }
1236     case MIB2_ICMP: {
1237         mib2_icmp_t *i2;
1238         mib2_icmp_t *i1;
1239         mib2_icmp_t *d;

1241         i2 = (mib2_icmp_t *)tempp2->valp;
1242         i1 = (mib2_icmp_t *)tempp1->valp;
1243         diffptr->group = tempp2->group;
1244         diffptr->mib_id = tempp2->mib_id;
1245         diffptr->length = tempp2->length;
1246         d = (mib2_icmp_t *)calloc(tempp2->length, 1);
1247         if (d == NULL)
1248             goto mibdiff_out_of_memory;
1249         diffptr->valp = d;
1250         MDIFF(d, i2, i1, icmpInMsgs);
1251         MDIFF(d, i2, i1, icmpInErrors);
1252         MDIFF(d, i2, i1, icmpInCksumErrs);
1253         MDIFF(d, i2, i1, icmpInUnknowns);
1254         MDIFF(d, i2, i1, icmpInDestUnreachs);
1255         MDIFF(d, i2, i1, icmpInTimeExcds);
1256         MDIFF(d, i2, i1, icmpInParmProbs);
1257         MDIFF(d, i2, i1, icmpInSrcQuenchs);
1258         MDIFF(d, i2, i1, icmpInRedirects);
1259         MDIFF(d, i2, i1, icmpInBadRedirects);
1260         MDIFF(d, i2, i1, icmpInEchos);
1261         MDIFF(d, i2, i1, icmpInEchoReps);
1262         MDIFF(d, i2, i1, icmpInTimestamps);
1263         MDIFF(d, i2, i1, icmpInAddrMasks);

```

```

1264         MDIFF(d, i2, i1, icmpInAddrMaskReps);
1265         MDIFF(d, i2, i1, icmpInFragNeeded);
1266         MDIFF(d, i2, i1, icmpOutMsgs);
1267         MDIFF(d, i2, i1, icmpOutDrops);
1268         MDIFF(d, i2, i1, icmpOutErrors);
1269         MDIFF(d, i2, i1, icmpOutDestUnreachs);
1270         MDIFF(d, i2, i1, icmpOutTimeExcds);
1271         MDIFF(d, i2, i1, icmpOutParmProbs);
1272         MDIFF(d, i2, i1, icmpOutSrcQuenchs);
1273         MDIFF(d, i2, i1, icmpOutRedirects);
1274         MDIFF(d, i2, i1, icmpOutEchos);
1275         MDIFF(d, i2, i1, icmpOutEchoReps);
1276         MDIFF(d, i2, i1, icmpOutTimestamps);
1277         MDIFF(d, i2, i1, icmpOutTimeExcds);
1278         MDIFF(d, i2, i1, icmpOutAddrMasks);
1279         MDIFF(d, i2, i1, icmpOutAddrMaskReps);
1280         MDIFF(d, i2, i1, icmpOutFragNeeded);
1281         MDIFF(d, i2, i1, icmpInOverflows);
1282         prevp = diffptr++;
1283         break;
1284     }
1285     case MIB2_ICMP6: {
1286         mib2_ipv6IfIcmpEntry_t *i2;
1287         mib2_ipv6IfIcmpEntry_t *i1;
1288         mib2_ipv6IfIcmpEntry_t *d;

1290         i2 = (mib2_ipv6IfIcmpEntry_t *)tempp2->valp;
1291         i1 = (mib2_ipv6IfIcmpEntry_t *)tempp1->valp;
1292         diffptr->group = tempp2->group;
1293         diffptr->mib_id = tempp2->mib_id;
1294         diffptr->length = tempp2->length;
1295         d = (mib2_ipv6IfIcmpEntry_t *)calloc(tempp2->length, 1);
1296         if (d == NULL)
1297             goto mibdiff_out_of_memory;
1298         diffptr->valp = d;
1299         MDIFF(d, i2, i1, ipv6IfIcmpInMsgs);
1300         MDIFF(d, i2, i1, ipv6IfIcmpInErrors);
1301         MDIFF(d, i2, i1, ipv6IfIcmpInDestUnreachs);
1302         MDIFF(d, i2, i1, ipv6IfIcmpInAdminProhbs);
1303         MDIFF(d, i2, i1, ipv6IfIcmpInTimeExcds);
1304         MDIFF(d, i2, i1, ipv6IfIcmpInParmProblems);
1305         MDIFF(d, i2, i1, ipv6IfIcmpInPktTooBigs);
1306         MDIFF(d, i2, i1, ipv6IfIcmpInEchos);
1307         MDIFF(d, i2, i1, ipv6IfIcmpInEchoReplies);
1308         MDIFF(d, i2, i1, ipv6IfIcmpInRouterSolicits);
1309         MDIFF(d, i2, i1, ipv6IfIcmpInRouterAdvertisements);
1310         MDIFF(d, i2, i1, ipv6IfIcmpInNeighborSolicits);
1311         MDIFF(d, i2, i1, ipv6IfIcmpInNeighborAdvertisements);
1312         MDIFF(d, i2, i1, ipv6IfIcmpInRedirects);
1313         MDIFF(d, i2, i1, ipv6IfIcmpInBadRedirects);
1314         MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembQueries);
1315         MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembResponses);
1316         MDIFF(d, i2, i1, ipv6IfIcmpInGroupMembReductions);
1317         MDIFF(d, i2, i1, ipv6IfIcmpInOverflows);
1318         MDIFF(d, i2, i1, ipv6IfIcmpOutMsgs);
1319         MDIFF(d, i2, i1, ipv6IfIcmpOutErrors);
1320         MDIFF(d, i2, i1, ipv6IfIcmpOutDestUnreachs);
1321         MDIFF(d, i2, i1, ipv6IfIcmpOutAdminProhbs);
1322         MDIFF(d, i2, i1, ipv6IfIcmpOutTimeExcds);
1323         MDIFF(d, i2, i1, ipv6IfIcmpOutParmProblems);
1324         MDIFF(d, i2, i1, ipv6IfIcmpOutPktTooBigs);
1325         MDIFF(d, i2, i1, ipv6IfIcmpOutEchos);
1326         MDIFF(d, i2, i1, ipv6IfIcmpOutEchoReplies);
1327         MDIFF(d, i2, i1, ipv6IfIcmpOutRouterSolicits);
1328         MDIFF(d, i2, i1, ipv6IfIcmpOutRouterAdvertisements);
1329         MDIFF(d, i2, i1, ipv6IfIcmpOutNeighborSolicits);

```

```

1330 MDIFF(d, i2, i1, ipv6IfIcmpOutNeighborAdvertisements);
1331 MDIFF(d, i2, i1, ipv6IfIcmpOutRedirects);
1332 MDIFF(d, i2, i1, ipv6IfIcmpOutGroupMembQueries);
1333 MDIFF(d, i2, i1, ipv6IfIcmpOutGroupMembResponses);
1334 MDIFF(d, i2, i1, ipv6IfIcmpOutGroupMembReductions);
1335 prevp = diffptr++;
1336 break;
1337 }
1338 case MIB2_TCP: {
1339     mib2_tcp_t *t2;
1340     mib2_tcp_t *t1;
1341     mib2_tcp_t *d;

1343     t2 = (mib2_tcp_t *)tempp2->valp;
1344     t1 = (mib2_tcp_t *)tempp1->valp;
1345     diffptr->group = tempp2->group;
1346     diffptr->mib_id = tempp2->mib_id;
1347     diffptr->length = tempp2->length;
1348     d = (mib2_tcp_t *)calloc(tempp2->length, 1);
1349     if (d == NULL)
1350         goto mibdiff_out_of_memory;
1351     diffptr->valp = d;
1352     d->tcpRtoMin = t2->tcpRtoMin;
1353     d->tcpRtoMax = t2->tcpRtoMax;
1354     d->tcpMaxConn = t2->tcpMaxConn;
1355     MDIFF(d, t2, t1, tcpActiveOpens);
1356     MDIFF(d, t2, t1, tcpPassiveOpens);
1357     MDIFF(d, t2, t1, tcpAttemptFails);
1358     MDIFF(d, t2, t1, tcpEstabResets);
1359     d->tcpCurrEstab = t2->tcpCurrEstab;
1360     MDIFF(d, t2, t1, tcpHCOutSegs);
1361     MDIFF(d, t2, t1, tcpOutDataSegs);
1362     MDIFF(d, t2, t1, tcpOutDataBytes);
1363     MDIFF(d, t2, t1, tcpRetransSegs);
1364     MDIFF(d, t2, t1, tcpRetransBytes);
1365     MDIFF(d, t2, t1, tcpOutAck);
1366     MDIFF(d, t2, t1, tcpOutAckDelayed);
1367     MDIFF(d, t2, t1, tcpOutUrg);
1368     MDIFF(d, t2, t1, tcpOutWinUpdate);
1369     MDIFF(d, t2, t1, tcpOutWinProbe);
1370     MDIFF(d, t2, t1, tcpOutControl);
1371     MDIFF(d, t2, t1, tcpOutRsts);
1372     MDIFF(d, t2, t1, tcpOutFastRetrans);
1373     MDIFF(d, t2, t1, tcpHCHInSegs);
1374     MDIFF(d, t2, t1, tcpInAckSegs);
1375     MDIFF(d, t2, t1, tcpInAckBytes);
1376     MDIFF(d, t2, t1, tcpInDupAck);
1377     MDIFF(d, t2, t1, tcpInAckUnsent);
1378     MDIFF(d, t2, t1, tcpInDataInorderSegs);
1379     MDIFF(d, t2, t1, tcpInDataInorderBytes);
1380     MDIFF(d, t2, t1, tcpInDataUnorderSegs);
1381     MDIFF(d, t2, t1, tcpInDataUnorderBytes);
1382     MDIFF(d, t2, t1, tcpInDataDupSegs);
1383     MDIFF(d, t2, t1, tcpInDataDupBytes);
1384     MDIFF(d, t2, t1, tcpInDataPartDupSegs);
1385     MDIFF(d, t2, t1, tcpInDataPartDupBytes);
1386     MDIFF(d, t2, t1, tcpInDataPastWinSegs);
1387     MDIFF(d, t2, t1, tcpInDataPastWinBytes);
1388     MDIFF(d, t2, t1, tcpInWinProbe);
1389     MDIFF(d, t2, t1, tcpInWinUpdate);
1390     MDIFF(d, t2, t1, tcpInClosed);
1391     MDIFF(d, t2, t1, tcpRttNoUpdate);
1392     MDIFF(d, t2, t1, tcpRttUpdate);
1393     MDIFF(d, t2, t1, tcpTimRetrans);
1394     MDIFF(d, t2, t1, tcpTimRetransDrop);
1395     MDIFF(d, t2, t1, tcpTimKeepalive);

```

```

1396 MDIFF(d, t2, t1, tcpTimKeepaliveProbe);
1397 MDIFF(d, t2, t1, tcpTimKeepaliveDrop);
1398 MDIFF(d, t2, t1, tcpListenDrop);
1399 MDIFF(d, t2, t1, tcpListenDropQ0);
1400 MDIFF(d, t2, t1, tcpHalfOpenDrop);
1401 MDIFF(d, t2, t1, tcpOutSackRetransSegs);
1402 prevp = diffptr++;
1403 break;
1404 }
1405 case MIB2_UDP: {
1406     mib2_udp_t *u2;
1407     mib2_udp_t *u1;
1408     mib2_udp_t *d;

1410     u2 = (mib2_udp_t *)tempp2->valp;
1411     u1 = (mib2_udp_t *)tempp1->valp;
1412     diffptr->group = tempp2->group;
1413     diffptr->mib_id = tempp2->mib_id;
1414     diffptr->length = tempp2->length;
1415     d = (mib2_udp_t *)calloc(tempp2->length, 1);
1416     if (d == NULL)
1417         goto mibdiff_out_of_memory;
1418     diffptr->valp = d;
1419     MDIFF(d, u2, u1, udpHCInDatagrams);
1420     MDIFF(d, u2, u1, udpInErrors);
1421     MDIFF(d, u2, u1, udpHCOutDatagrams);
1422     MDIFF(d, u2, u1, udpOutErrors);
1423     prevp = diffptr++;
1424     break;
1425 }
1426 case MIB2_SCTP: {
1427     mib2_sctp_t *s2;
1428     mib2_sctp_t *s1;
1429     mib2_sctp_t *d;

1431     s2 = (mib2_sctp_t *)tempp2->valp;
1432     s1 = (mib2_sctp_t *)tempp1->valp;
1433     diffptr->group = tempp2->group;
1434     diffptr->mib_id = tempp2->mib_id;
1435     diffptr->length = tempp2->length;
1436     d = (mib2_sctp_t *)calloc(tempp2->length, 1);
1437     if (d == NULL)
1438         goto mibdiff_out_of_memory;
1439     diffptr->valp = d;
1440     d->sctpRtoAlgorithm = s2->sctpRtoAlgorithm;
1441     d->sctpRtoMin = s2->sctpRtoMin;
1442     d->sctpRtoMax = s2->sctpRtoMax;
1443     d->sctpRtoInitial = s2->sctpRtoInitial;
1444     d->sctpMaxAssocs = s2->sctpMaxAssocs;
1445     d->sctpValCookieLife = s2->sctpValCookieLife;
1446     d->sctpMaxInitRetr = s2->sctpMaxInitRetr;
1447     d->sctpCurrEstab = s2->sctpCurrEstab;
1448     MDIFF(d, s2, s1, sctpActiveEstab);
1449     MDIFF(d, s2, s1, sctpPassiveEstab);
1450     MDIFF(d, s2, s1, sctpAborted);
1451     MDIFF(d, s2, s1, sctpShutdowns);
1452     MDIFF(d, s2, s1, sctpOutOfBlue);
1453     MDIFF(d, s2, s1, sctpChecksumError);
1454     MDIFF(d, s2, s1, sctpOutCtrlChunks);
1455     MDIFF(d, s2, s1, sctpOutOrderChunks);
1456     MDIFF(d, s2, s1, sctpOutUnorderChunks);
1457     MDIFF(d, s2, s1, sctpRetransChunks);
1458     MDIFF(d, s2, s1, sctpOutAck);
1459     MDIFF(d, s2, s1, sctpOutAckDelayed);
1460     MDIFF(d, s2, s1, sctpOutWinUpdate);
1461     MDIFF(d, s2, s1, sctpOutFastRetrans);

```

```

1462         MDIFF(d, s2, sl, sctpOutWinProbe);
1463         MDIFF(d, s2, sl, sctpInCtrlChunks);
1464         MDIFF(d, s2, sl, sctpInOrderChunks);
1465         MDIFF(d, s2, sl, sctpInUnorderChunks);
1466         MDIFF(d, s2, sl, sctpInAck);
1467         MDIFF(d, s2, sl, sctpInDupAck);
1468         MDIFF(d, s2, sl, sctpInAckUnsent);
1469         MDIFF(d, s2, sl, sctpFragUsrMsgs);
1470         MDIFF(d, s2, sl, sctpReasmUsrMsgs);
1471         MDIFF(d, s2, sl, sctpOutSCTPPkts);
1472         MDIFF(d, s2, sl, sctpInSCTPPkts);
1473         MDIFF(d, s2, sl, sctpInInvalidCookie);
1474         MDIFF(d, s2, sl, sctpTimRetrans);
1475         MDIFF(d, s2, sl, sctpTimRetransDrop);
1476         MDIFF(d, s2, sl, sctpTimHeartBeatProbe);
1477         MDIFF(d, s2, sl, sctpTimHeartBeatDrop);
1478         MDIFF(d, s2, sl, sctpListenDrop);
1479         MDIFF(d, s2, sl, sctpInClosed);
1480         prevp = diffptr++;
1481         break;
1482     }
1483     case EXPER_RAWIP: {
1484         mib2_rawip_t *r2;
1485         mib2_rawip_t *r1;
1486         mib2_rawip_t *d;

1488         r2 = (mib2_rawip_t *)temp2->valp;
1489         r1 = (mib2_rawip_t *)temp1->valp;
1490         diffptr->group = temp2->group;
1491         diffptr->mib_id = temp2->mib_id;
1492         diffptr->length = temp2->length;
1493         d = (mib2_rawip_t *)calloc(temp2->length, 1);
1494         if (d == NULL)
1495             goto mibdiff_out_of_memory;
1496         diffptr->valp = d;
1497         MDIFF(d, r2, r1, rawipInDatagrams);
1498         MDIFF(d, r2, r1, rawipInErrors);
1499         MDIFF(d, r2, r1, rawipInCksumErrs);
1500         MDIFF(d, r2, r1, rawipOutDatagrams);
1501         MDIFF(d, r2, r1, rawipOutErrors);
1502         prevp = diffptr++;
1503         break;
1504     }
1505     /*
1506     * there are more "group" types but they aren't
1507     * required for the -s and -Ms options
1508     */
1509 } /* 'for' loop 2 ends */
1510 temp1 = NULL;
1511 /* 'for' loop 1 ends */
1512 temp2 = NULL;
1513 diffptr--;
1514 diffptr->next_item = NULL;
1515 return (diffp);
1516

1518 mibdiff_out_of_memory:
1519     mib_item_destroy(&diffp);
1520     return (NULL);
1521 }

1523 /*
1524 * mib_item_destroy: cleans up a mib_item_t *
1525 * that was created by calling mib_item_dup or
1526 * mib_item_diff
1527 */

```

```

1528 static void
1529 mib_item_destroy(mib_item_t **itemp)
1530 {
1531     mib_item_destroy(mib_item_t **itemp) {
1532         int nitems = 0;
1533         int c = 0;
1534         mib_item_t *tempp;

1535         if (itemp == NULL || *itemp == NULL)
1536             return;

1538         for (tempp = *itemp; tempp != NULL; tempp = tempp->next_item)
1539             if (tempp->mib_id == 0)
1540                 nitems++;
1541         else
1542             return; /* cannot destroy! */

1544         if (nitems == 0)
1545             return; /* cannot destroy! */

1547         for (c = nitems - 1; c >= 0; c--) {
1548             if ((itemp[0][c]).valp != NULL)
1549                 free((itemp[0][c]).valp);
1550         }
1551         free(*itemp);

1553         *itemp = NULL;
1554     }
1555     _____ unchanged_portion_omitted _____

3294 static void
3295 if_report_ip4(mib2_ipAddrEntry_t *ap, char ifname[], char logintname[],
3296              struct ifstat *statptr, boolean_t ksp_not_null)
3297 {
3298     if_report_ip4(mib2_ipAddrEntry_t *ap,
3299                 char ifname[], char logintname[], struct ifstat *statptr,
3300                 boolean_t ksp_not_null) {

3299     char abuf[MAXHOSTNAMELEN + 1];
3300     char dstbuf[MAXHOSTNAMELEN + 1];

3302     if (ksp_not_null) {
3303         (void) printf("%-5s %-4u ",
3304                     ifname, ap->ipAdEntInfo.ae_mtu);
3305         if (ap->ipAdEntInfo.ae_flags & IFF_POINTOPOINT)
3306             (void) pr_addr(ap->ipAdEntInfo.ae_pp_dst_addr,
3307                            abuf, sizeof (abuf));
3308     } else
3309         (void) pr_netaddr(ap->ipAdEntAddr,
3310                          ap->ipAdEntNetMask, abuf, sizeof (abuf));
3311     (void) printf("%-13s %-14s %-6llu %-5llu %-6llu %-5llu "
3312                 "%-6llu %-6llu\n",
3313                 abuf, pr_addr(ap->ipAdEntAddr, dstbuf, sizeof (dstbuf)),
3314                 statptr->ipackets, statptr->ierrors,
3315                 statptr->opackets, statptr->errors,
3316                 statptr->collisions, 0LL);
3317 }
3318 /*
3319 * Print logical interface info if Aflag set (including logical unit 0)
3320 */
3321 if (Aflag) {
3322     *statptr = zerostat;
3323     statptr->ipackets = ap->ipAdEntInfo.ae_ibcnt;
3324     statptr->opackets = ap->ipAdEntInfo.ae_obcnt;

3326     (void) printf("%-5s %-4u ", logintname, ap->ipAdEntInfo.ae_mtu);

```

```

3327     if (ap->ipAdEntInfo.ae_flags & IFF_POINTOPOINT)
3328         (void) pr_addr(ap->ipAdEntInfo.ae_pp_dst_addr, abuf,
3329             sizeof (abuf));
3330     else
3331         (void) pr_netaddr(ap->ipAdEntAddr, ap->ipAdEntNetMask,
3332             abuf, sizeof (abuf));
3333
3334     (void) printf("%-13s %-14s %-6llu %-5s %-6s "
3335         "%-5s %-6s %-6llu\n", abuf,
3336         pr_addr(ap->ipAdEntAddr, dstbuf, sizeof (dstbuf)),
3337         statptr->ipackets, "N/A", "N/A", "N/A", "N/A",
3338         0LL);
3339 }
3340 }
3341
3342 static void
3343 if_report_ip6(mib2_ipv6AddrEntry_t *ap6, char ifname[], char logintname[],
3344     struct ifstat *statptr, boolean_t ksp_not_null)
3345 {
3346     if_report_ip6(mib2_ipv6AddrEntry_t *ap6,
3347         char ifname[], char logintname[], struct ifstat *statptr,
3348         boolean_t ksp_not_null) {
3349
3350     char abuf[MAXHOSTNAMELEN + 1];
3351     char dstbuf[MAXHOSTNAMELEN + 1];
3352
3353     if (ksp_not_null) {
3354         (void) printf("%-5s %-4u ", ifname, ap6->ipv6AddrInfo.ae_mtu);
3355         if (ap6->ipv6AddrInfo.ae_flags &
3356             IFF_POINTOPOINT) {
3357             (void) pr_addr6(&ap6->ipv6AddrInfo.ae_pp_dst_addr,
3358                 abuf, sizeof (abuf));
3359         } else {
3360             (void) pr_prefix6(&ap6->ipv6AddrAddress,
3361                 ap6->ipv6AddrPfxLength, abuf,
3362                 sizeof (abuf));
3363         }
3364         (void) printf("%-27s %-27s %-6llu %-5llu "
3365             "%-6llu %-5llu %-6llu\n",
3366             abuf, pr_addr6(&ap6->ipv6AddrAddress, dstbuf,
3367                 sizeof (dstbuf)),
3368             statptr->ipackets, statptr->ierrors, statptr->opackets,
3369             statptr->oerrors, statptr->collisions);
3370     }
3371     /*
3372      * Print logical interface info if Aflag set (including logical unit 0)
3373      */
3374     if (Aflag) {
3375         *statptr = zerostat;
3376         statptr->ipackets = ap6->ipv6AddrInfo.ae_ibcnt;
3377         statptr->opackets = ap6->ipv6AddrInfo.ae_obcnt;
3378
3379         (void) printf("%-5s %-4u ", logintname,
3380             ap6->ipv6AddrInfo.ae_mtu);
3381         if (ap6->ipv6AddrInfo.ae_flags & IFF_POINTOPOINT)
3382             (void) pr_addr6(&ap6->ipv6AddrInfo.ae_pp_dst_addr,
3383                 abuf, sizeof (abuf));
3384         else
3385             (void) pr_prefix6(&ap6->ipv6AddrAddress,
3386                 ap6->ipv6AddrPfxLength, abuf, sizeof (abuf));
3387         (void) printf("%-27s %-27s %-6llu %-5s %-6s %-5s %-6s\n",
3388             abuf, pr_addr6(&ap6->ipv6AddrAddress, dstbuf,
3389                 sizeof (dstbuf)),
3390             statptr->ipackets, "N/A", "N/A", "N/A", "N/A");
3391     }
3392 }

```

unchanged_portion_omitted

```

4758 /* ----- TCP_REPORT----- */
4759
4760 static const char tcp_hdr_v4[] =
4761     "\nTCP: IPv4\n";
4762 static const char tcp_hdr_v4_compat[] =
4763     "\nTCP\n";
4764 static const char tcp_hdr_v4_verbose[] =
4765     "Local/Remote Address Swind Snext Suna Rwind Rnext Rack "
4766     "Rto Mss State\n";
4767 -----
4768 -----\n";
4769 static const char tcp_hdr_v4_normal[] =
4770     " Local Address Remote Address Swind Send-Q Rwind Recv-Q "
4771     " State\n";
4772 -----
4773 -----\n";
4774 static const char tcp_hdr_v4_pid[] =
4775     " Local Address Remote Address User Pid Command Swind "
4776     " Send-Q Rwind Recv-Q State\n";
4777 -----
4778 -----\n";
4779 static const char tcp_hdr_v4_pid_verbose[] =
4780     "Local/Remote Address Swind Snext Suna Rwind Rnext Rack Rto "
4781     " Mss State User Pid Command\n";
4782 -----
4783 -----\n";
4784 #endif /* ! codereview */
4785
4786 static const char tcp_hdr_v6[] =
4787     "\nTCP: IPv6\n";
4788 static const char tcp_hdr_v6_verbose[] =
4789     "Local/Remote Address Swind Snext Suna Rwind Rnext "
4790     " Rack Rto Mss State If\n";
4791 -----
4792 -----\n";
4793 static const char tcp_hdr_v6_normal[] =
4794     " Local Address Remote Address "
4795     "Swind Send-Q Rwind Recv-Q State If\n";
4796 -----
4797 -----\n";
4798 static const char tcp_hdr_v6_pid[] =
4799     " Local Address Remote Address User "
4800     " Pid Command Swind Send-Q Rwind Recv-Q State If\n";
4801 -----
4802 -----\n";
4803 static const char tcp_hdr_v6_pid_verbose[] =
4804     "Local/Remote Address Swind Snext Suna Rwind Rnext "
4805     " Rack Rto Mss State If User Pid Command\n";
4806 -----
4807 -----\n";
4808 #endif /* ! codereview */
4809
4810 static boolean_t tcp_report_item_v4(const mib2_tcpConnEntry_t *,
4811     conn_pid_info_t *, boolean_t first,
4812     const mib2_transportMLPEntry_t *);
4813 static boolean_t tcp_report_item_v6(const mib2_tcp6ConnEntry_t *,
4814     conn_pid_info_t *, boolean_t first,
4815     const mib2_transportMLPEntry_t *);
4816
4822     boolean_t first, const mib2_transportMLPEntry_t *);
4823
4824 static void
4825 tcp_report(const mib_item_t *item)
4826 {

```

```

4821     int                                jtemp = 0;
4822     boolean_t                            print_hdr_once_v4 = B_TRUE;
4823     boolean_t                            print_hdr_once_v6 = B_TRUE;
4824     mib2_tcpConnEntry_t                  *tp;
4825     mib2_tcp6ConnEntry_t                 *tp6;
4826     mib2_transportMLPEntry_t             **v4_attrs, **v6_attrs;
4827     mib2_transportMLPEntry_t             **v4a, **v6a;
4828     mib2_transportMLPEntry_t             *aptr;
4829     conn_pid_info_t                      *cpi;
4830 #endif /* ! codereview */

4832     if (!protocol_selected(IPPROTO_TCP))
4833         return;

4835     /*
4836     * Preparation pass: the kernel returns separate entries for TCP
4837     * connection table entries and Multilevel Port attributes. We loop
4838     * through the attributes first and set up an array for each address
4839     * family.
4840     */
4841     v4_attrs = family_selected(AF_INET) && RSECflag ?
4842         gather_attrs(item, MIB2_TCP, MIB2_TCP_CONN, tcpConnEntrySize) :
4843         NULL;
4844     v6_attrs = family_selected(AF_INET6) && RSECflag ?
4845         gather_attrs(item, MIB2_TCP6, MIB2_TCP6_CONN, tcp6ConnEntrySize) :
4846         NULL;

4848     /* 'for' loop 1: */
4849     v4a = v4_attrs;
4850     v6a = v6_attrs;
4851     for (; item != NULL; item = item->next_item) {
4852         if (Xflag) {
4853             (void) printf("\n--- Entry %d ---\n", ++jtemp);
4854             (void) printf("Group = %d, mib_id = %d, "
4855                 "length = %d, valp = 0x%p\n",
4856                 item->group, item->mib_id,
4857                 item->length, item->valp);
4858         }

4860         if (!(item->group == MIB2_TCP &&
4861             item->mib_id == MIB2_TCP_CONN) ||
4862             (item->group == MIB2_TCP6 &&
4863             item->mib_id == MIB2_TCP6_CONN) ||
4864             (item->group == MIB2_TCP &&
4865             item->mib_id == EXPER_XPORT_PROC_INFO) ||
4866             (item->group == MIB2_TCP6 &&
4867             item->mib_id == EXPER_XPORT_PROC_INFO))
4868             item->mib_id == MIB2_TCP6_CONN)))
4869             continue; /* 'for' loop 1 */

4870         if (item->group == MIB2_TCP && !family_selected(AF_INET))
4871             continue; /* 'for' loop 1 */
4872         else if (item->group == MIB2_TCP6 && !family_selected(AF_INET6))
4873             continue; /* 'for' loop 1 */

4875         if ((!Uflag) && item->group == MIB2_TCP &&
4876             item->mib_id == MIB2_TCP_CONN) {
4877             if (item->group == MIB2_TCP) {
4878                 for (tp = (mib2_tcpConnEntry_t *)item->valp;
4879                     (char *)tp < (char *)item->valp + item->length;
4880                     /* LINTED: (note 1) */
4881                     tp = (mib2_tcpConnEntry_t *)((char *)tp +
4882                         tcpConnEntrySize)) {
4883                     aptr = v4a == NULL ? NULL : *v4a++;
4884                     print_hdr_once_v4 = tcp_report_item_v4(tp,
4885                         NULL, print_hdr_once_v4, aptr);

```

```

4886                 print_hdr_once_v4, aptr);
4887             } else if ((!Uflag) && item->group == MIB2_TCP6 &&
4888                 item->mib_id == MIB2_TCP6_CONN) {
4889             } else {
4890                 for (tp6 = (mib2_tcp6ConnEntry_t *)item->valp;
4891                     (char *)tp6 < (char *)item->valp + item->length;
4892                     /* LINTED: (note 1) */
4893                     tp6 = (mib2_tcp6ConnEntry_t *)((char *)tp6 +
4894                         tcp6ConnEntrySize)) {
4895                     aptr = v6a == NULL ? NULL : *v6a++;
4896                     print_hdr_once_v6 = tcp_report_item_v6(tp6,
4897                         NULL, print_hdr_once_v6, aptr);
4898                 }
4899             } else if ((Uflag) && item->group == MIB2_TCP &&
4900                 item->mib_id == EXPER_XPORT_PROC_INFO) {
4901                 for (tp = (mib2_tcpConnEntry_t *)item->valp;
4902                     (char *)tp < (char *)item->valp + item->length;
4903                     /* LINTED: (note 1) */
4904                     tp = (mib2_tcpConnEntry_t *)((char *)cpi +
4905                         cpi->cpi_tot_size)) {
4906                     aptr = v4a == NULL ? NULL : *v4a++;
4907                     /* LINTED: (note 1) */
4908                     cpi = (conn_pid_info_t *)((char *)tp +
4909                         tcpConnEntrySize);
4910                     print_hdr_once_v4 = tcp_report_item_v4(tp,
4911                         cpi, print_hdr_once_v4, aptr);
4912                 }
4913             } else if ((Uflag) && item->group == MIB2_TCP6 &&
4914                 item->mib_id == EXPER_XPORT_PROC_INFO) {
4915                 for (tp6 = (mib2_tcp6ConnEntry_t *)item->valp;
4916                     (char *)tp6 < (char *)item->valp + item->length;
4917                     /* LINTED: (note 1) */
4918                     tp6 = (mib2_tcp6ConnEntry_t *)((char *)cpi +
4919                         cpi->cpi_tot_size)) {
4920                     aptr = v6a == NULL ? NULL : *v6a++;
4921                     /* LINTED: (note 1) */
4922                     cpi = (conn_pid_info_t *)((char *)tp6 +
4923                         tcp6ConnEntrySize);
4924                     print_hdr_once_v6 = tcp_report_item_v6(tp6,
4925                         cpi, print_hdr_once_v6, aptr);
4926                 }
4927             }
4928         }
4929     }

4931     if (v4_attrs != NULL)
4932         free(v4_attrs);
4933     if (v6_attrs != NULL)
4934         free(v6_attrs);
4935 }

4937 static boolean_t
4938 tcp_report_item_v4(const mib2_tcpConnEntry_t *tp, conn_pid_info_t *cpi,
4939     boolean_t first, const mib2_transportMLPEntry_t *attr)
4940 tcp_report_item_v4(const mib2_tcpConnEntry_t *tp, boolean_t first,
4941     const mib2_transportMLPEntry_t *attr)
4942 {
4943     /*
4944     * lname and fname below are for the hostname as well as the portname
4945     * There is no limit on portname length so we assume MAXHOSTNAMELEN
4946     * as the limit
4947     */

```

```

4946 char lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
4947 char fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];

4950 #endif /* ! codereview */
4951 if (!(Aflag || tp->tcpConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
4952 return (first); /* Nothing to print */

4954 if (first) {
4955 (void) printf(v4compat ? tcp_hdr_v4_compat : tcp_hdr_v4);
4956 if (Uflag)
4957 (void) printf(Vflag ? tcp_hdr_v4_pid_verbose :
4958 tcp_hdr_v4_pid);
4959 else
4960 (void) printf(Vflag ? tcp_hdr_v4_verbose :
4961 tcp_hdr_v4_normal);
4305 (void) printf(Vflag ? tcp_hdr_v4_verbose : tcp_hdr_v4_normal);
4962 }

4964 if (!(Uflag) && Vflag) {
4308 if (Vflag) {
4965 (void) printf("%-20s\n%-20s %5u %08x %08x %5u %08x %08x "
4966 "%5u %5u %s\n",
4967 pr_ap(tp->tcpConnLocalAddress,
4968 tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
4969 pr_ap(tp->tcpConnRemAddress,
4970 tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
4971 tp->tcpConnEntryInfo.ce_swnd,
4972 tp->tcpConnEntryInfo.ce_snxt,
4973 tp->tcpConnEntryInfo.ce_suna,
4974 tp->tcpConnEntryInfo.ce_rwnd,
4975 tp->tcpConnEntryInfo.ce_rnxt,
4976 tp->tcpConnEntryInfo.ce_rack,
4977 tp->tcpConnEntryInfo.ce_rto,
4978 tp->tcpConnEntryInfo.ce_mss,
4979 mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
4980 } else if (!(Uflag) && !Vflag) {
4324 } else {
4981 int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
4982 (int)tp->tcpConnEntryInfo.ce_suna - 1;
4983 int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
4984 (int)tp->tcpConnEntryInfo.ce_rack;

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

```

```

5009 tp->tcpConnEntryInfo.ce_snxt,
5010 tp->tcpConnEntryInfo.ce_suna,
5011 tp->tcpConnEntryInfo.ce_rwnd,
5012 tp->tcpConnEntryInfo.ce_rnxt,
5013 tp->tcpConnEntryInfo.ce_rack,
5014 tp->tcpConnEntryInfo.ce_rto,
5015 tp->tcpConnEntryInfo.ce_mss,
5016 mitcp_state(tp->tcpConnEntryInfo.ce_state, attr),
5017 pinfo->pr_user, (int)*pids, pinfo->pr_sargs);
5018 i++; pids++;
5019 } while (i < cpi->cpi_pids_cnt);
5020 } else if (Uflag && !Vflag) {
5021 int sq = (int)tp->tcpConnEntryInfo.ce_snxt -
5022 (int)tp->tcpConnEntryInfo.ce_suna - 1;
5023 int rq = (int)tp->tcpConnEntryInfo.ce_rnxt -
5024 (int)tp->tcpConnEntryInfo.ce_rack;
5025 int i = 0;
5026 pid_t *pids = cpi->cpi_pids;
5027 proc_info_t *pinfo;
5028 do {
5029 pinfo = get_proc_info(*pids);
5030 (void) printf("%-20s %-20s %-8.8s %6u %-13.13s %7u "
5031 "%6d %7u %6d %s\n",
5032 pr_ap(tp->tcpConnLocalAddress,
5033 tp->tcpConnLocalPort, "tcp", lname, sizeof (lname)),
5034 pr_ap(tp->tcpConnRemAddress,
5035 tp->tcpConnRemPort, "tcp", fname, sizeof (fname)),
5036 pinfo->pr_user, (int)*pids, pinfo->pr_fname,
5037 tp->tcpConnEntryInfo.ce_swnd,
5038 (sq >= 0) ? sq : 0,
5039 tp->tcpConnEntryInfo.ce_rwnd,
5040 (rq >= 0) ? rq : 0,
5041 mitcp_state(tp->tcpConnEntryInfo.ce_state, attr));
5042 i++; pids++;
5043 } while (i < cpi->cpi_pids_cnt);
5044 #endif /* ! codereview */
5045 }

5047 print_transport_label(attr);

5049 return (B_FALSE);
5050 }

5052 static boolean_t
5053 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6, conn_pid_info_t *cpi,
5054 boolean_t first, const mib2_transportMLPEntry_t *attr)
5055 tcp_report_item_v6(const mib2_tcp6ConnEntry_t *tp6, boolean_t first,
5056 const mib2_transportMLPEntry_t *attr)
5057 {
5058 /*
5059 * lname and fname below are for the hostname as well as the portname
5060 * There is no limit on portname length so we assume MAXHOSTNAMELEN
5061 * as the limit
5062 */
5063 char lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5064 char fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5065 char ifname[LIFNAMSIZ + 1];
5066 char *ifnamep;

5066 if (!(Aflag || tp6->tcp6ConnEntryInfo.ce_state >= TCPS_ESTABLISHED))
5067 return (first); /* Nothing to print */

5069 if (first) {
5070 (void) printf(tcp_hdr_v6);
5071 if (Uflag)
5072 (void) printf(Vflag ? tcp_hdr_v6_pid_verbose :

```

```

5073         tcp_hdr_v6_pid);
5074     else
5075         (void) printf(Vflag ? tcp_hdr_v6_verbose :
5076             tcp_hdr_v6_normal);
4358     (void) printf(Vflag ? tcp_hdr_v6_verbose : tcp_hdr_v6_normal);
5077 }

5079 ifnamep = (tp6->tcp6ConnIfIndex != 0) ?
5080     if_indextoname(tp6->tcp6ConnIfIndex, ifname) : NULL;
5081 if (ifnamep == NULL)
5082     ifnamep = "";

5084 if ((!Uflag) && Vflag) {
4366     if (Vflag) {
5085         (void) printf("%-33s\n%-33s %5u %08x %08x %5u %08x %08x "
5086             "%5u %5u %-11s %s\n",
5087             pr_ap6(&tp6->tcp6ConnLocalAddress,
5088                 tp6->tcp6ConnLocalPort, "tcp", lname, sizeof (lname)),
5089             pr_ap6(&tp6->tcp6ConnRemAddress,
5090                 tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5091             tp6->tcp6ConnEntryInfo.ce_swnd,
5092             tp6->tcp6ConnEntryInfo.ce_snxt,
5093             tp6->tcp6ConnEntryInfo.ce_suna,
5094             tp6->tcp6ConnEntryInfo.ce_rwnd,
5095             tp6->tcp6ConnEntryInfo.ce_rnxt,
5096             tp6->tcp6ConnEntryInfo.ce_rack,
5097             tp6->tcp6ConnEntryInfo.ce_rto,
5098             tp6->tcp6ConnEntryInfo.ce_mss,
5099             mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5100             ifnamep);
5101     } else if ((!Uflag) && (!Vflag)) {
4383     } else {
5102         int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
5103             (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
5104         int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
5105             (int)tp6->tcp6ConnEntryInfo.ce_rack;

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

```

```

5136         tp6->tcp6ConnEntryInfo.ce_rnxt,
5137         tp6->tcp6ConnEntryInfo.ce_rack,
5138         tp6->tcp6ConnEntryInfo.ce_rto,
5139         tp6->tcp6ConnEntryInfo.ce_mss,
5140         mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5141         ifnamep, pinfo->pr_user, (int)*pids,
5142         pinfo->pr_psargs);
5143         i++; pids++;
5144     } while (i < cpi->cpi_pids_cnt);
5145 } else if (Uflag && (!Vflag)) {
5146     int sq = (int)tp6->tcp6ConnEntryInfo.ce_snxt -
5147         (int)tp6->tcp6ConnEntryInfo.ce_suna - 1;
5148     int rq = (int)tp6->tcp6ConnEntryInfo.ce_rnxt -
5149         (int)tp6->tcp6ConnEntryInfo.ce_rack;
5150     int i = 0;
5151     pid_t *pids = cpi->cpi_pids;
5152     proc_info_t *pinfo;
5153     do {
5154         pinfo = get_proc_info(*pids);
5155         (void) printf("%-33s %-33s %-8.8s %6u %-14.14s %7d "
5156             "%6u %7d %6d %-11s %s\n",
5157             pr_ap6(&tp6->tcp6ConnLocalAddress,
5158                 tp6->tcp6ConnLocalPort, "tcp", lname,
5159                 sizeof (lname)),
5160             pr_ap6(&tp6->tcp6ConnRemAddress,
5161                 tp6->tcp6ConnRemPort, "tcp", fname, sizeof (fname)),
5162             pinfo->pr_user, (int)*pids, pinfo->pr_fname,
5163             tp6->tcp6ConnEntryInfo.ce_swnd,
5164             (sq >= 0) ? sq : 0,
5165             tp6->tcp6ConnEntryInfo.ce_rwnd,
5166             (rq >= 0) ? rq : 0,
5167             mitcp_state(tp6->tcp6ConnEntryInfo.ce_state, attr),
5168             ifnamep);
5169         i++; pids++;
5170     } while (i < cpi->cpi_pids_cnt);
5171 #endif /* ! codereview */
5172     }

5174     print_transport_label(attr);

5176     return (B_FALSE);
5177 }

5179 /* ----- UDP_REPORT----- */

5181 static boolean_t udp_report_item_v4(const mib2_udpEntry_t *ude,
5182     conn_pid_info_t *cpi, boolean_t first,
5183     const mib2_transportMLPEntry_t *attr);
4400     boolean_t first, const mib2_transportMLPEntry_t *attr);
5184 static boolean_t udp_report_item_v6(const mib2_udp6Entry_t *ude6,
5185     conn_pid_info_t *cpi, boolean_t first,
5186     const mib2_transportMLPEntry_t *attr);
4402     boolean_t first, const mib2_transportMLPEntry_t *attr);

5188 static const char udp_hdr_v4[] =
5189 " Local Address Remote Address State\n"
5190 "-----\n";
5191 static const char udp_hdr_v4_pid[] =
5192 " Local Address Remote Address User Pid "
5193 " Command State\n"
5194 "-----"
5195 "-----\n";
5196 static const char udp_hdr_v4_pid_verbose[] =
5197 " Local Address Remote Address User Pid State "
5198 " Command\n"
5199 "-----"

```

```

5200 "-----\n";
5201 #endif /* ! codereview */

5203 static const char udp_hdr_v6[] =
5204 "  Local Address          Remote Address      "
5205 "  State                  If\n"
5206 "-----\n";
5207 "-----\n";
5208 static const char udp_hdr_v6_pid[] =
5209 "  Local Address          Remote Address      "
5210 "  User   Pid            Command          State   If\n"
5211 "-----\n";
5212 "-----\n";
5213 static const char udp_hdr_v6_pid_verbose[] =
5214 "  Local Address          Remote Address      "
5215 "  User   Pid            State   If          Command\n"
5216 "-----\n";
5217 "-----\n";

5219 #endif /* ! codereview */

5221 static void
5222 udp_report(const mib_item_t *item)
5223 {
5224     int                jtemp = 0;
5225     boolean_t          print_hdr_once_v4 = B_TRUE;
5226     boolean_t          print_hdr_once_v6 = B_TRUE;
5227     *ude;
5228     *ude6;
5229     **v4_attr, **v6_attr;
5230     **v4a, **v6a;
5231     *aptr;
5232     conn_pid_info_t    *cpi;
5233 #endif /* ! codereview */

5235     if (!protocol_selected(IPPROTO_UDP))
5236         return;

5238     /*
5239     * Preparation pass: the kernel returns separate entries for UDP
5240     * connection table entries and Multilevel Port attributes. We loop
5241     * through the attributes first and set up an array for each address
5242     * family.
5243     */
5244     v4_attr = family_selected(AF_INET) && RSECflag ?
5245         gather_attr(item, MIB2_UDP, MIB2_UDP_ENTRY, udpEntrySize) : NULL;
5246     v6_attr = family_selected(AF_INET6) && RSECflag ?
5247         gather_attr(item, MIB2_UDP6, MIB2_UDP6_ENTRY, udp6EntrySize) :
5248         NULL;

5250     v4a = v4_attr;
5251     v6a = v6_attr;
5252     /* 'for' loop 1: */
5253     for (; item; item = item->next_item) {
5254         if (Xflag) {
5255             (void) printf("\n--- Entry %d ---\n", ++jtemp);
5256             (void) printf("Group = %d, mib_id = %d, "
5257                 "length = %d, valp = 0x%p\n",
5258                 item->group, item->mib_id,
5259                 item->length, item->valp);
5260         }
5261         if (((item->group == MIB2_UDP &&
5262             item->mib_id == MIB2_UDP_ENTRY) ||
5263             (item->group == MIB2_UDP6 &&
5264             item->mib_id == MIB2_UDP6_ENTRY) ||
5265             (item->group == MIB2_UDP &&

```

```

5266         item->mib_id == EXPER_XPORT_PROC_INFO) ||
5267         (item->group == MIB2_UDP6 &&
5268         item->mib_id == EXPER_XPORT_PROC_INFO)))
5269         item->mib_id == MIB2_UDP6_ENTRY)))
5270         continue; /* 'for' loop 1 */

5271     if (item->group == MIB2_UDP && !family_selected(AF_INET))
5272         continue; /* 'for' loop 1 */
5273     else if (item->group == MIB2_UDP6 && !family_selected(AF_INET6))
5274         continue; /* 'for' loop 1 */

5276     /* xxx.xxx.xxx.xxx,pppp sss... */
5277     if ((!Uflag) && item->group == MIB2_UDP &&
5278         item->mib_id == MIB2_UDP_ENTRY) {
5279         if (item->group == MIB2_UDP) {
5280             for (ude = (mib2_udpEntry_t *)item->valp;
5281                 (char *)ude < (char *)item->valp + item->length;
5282                 /* LINTED: (note 1) */
5283                 ude = (mib2_udpEntry_t *)((char *)ude +
5284                     udpEntrySize)) {
5285                 aptr = v4a == NULL ? NULL : *v4a++;
5286                 print_hdr_once_v4 = udp_report_item_v4(ude,
5287                     NULL, print_hdr_once_v4, aptr);
5288             }
5289         } else if ((!Uflag) && item->group == MIB2_UDP6 &&
5290             item->mib_id == MIB2_UDP6_ENTRY) {
5291         } else {
5292             for (ude6 = (mib2_udp6Entry_t *)item->valp;
5293                 (char *)ude6 < (char *)item->valp + item->length;
5294                 /* LINTED: (note 1) */
5295                 ude6 = (mib2_udp6Entry_t *)((char *)ude6 +
5296                     udp6EntrySize)) {
5297                 aptr = v6a == NULL ? NULL : *v6a++;
5298                 print_hdr_once_v6 = udp_report_item_v6(ude6,
5299                     NULL, print_hdr_once_v6, aptr);
5300             }
5301         } else if ((Uflag) && item->group == MIB2_UDP &&
5302             item->mib_id == EXPER_XPORT_PROC_INFO) {
5303             for (ude = (mib2_udpEntry_t *)item->valp;
5304                 (char *)ude < (char *)item->valp + item->length;
5305                 /* LINTED: (note 1) */
5306                 ude = (mib2_udpEntry_t *)((char *)cpi +
5307                     cpi->cpi_tot_size)) {
5308                 aptr = v4a == NULL ? NULL : *v4a++;
5309                 /* LINTED: (note 1) */
5310                 cpi = (conn_pid_info_t *)((char *)ude +
5311                     udpEntrySize);
5312                 print_hdr_once_v4 = udp_report_item_v4(ude,
5313                     cpi, print_hdr_once_v4, aptr);
5314             }
5315         } else if ((Uflag) && item->group == MIB2_UDP6 &&
5316             item->mib_id == EXPER_XPORT_PROC_INFO) {
5317             for (ude6 = (mib2_udp6Entry_t *)item->valp;
5318                 (char *)ude6 < (char *)item->valp + item->length;
5319                 /* LINTED: (note 1) */
5320                 ude6 = (mib2_udp6Entry_t *)((char *)cpi +
5321                     cpi->cpi_tot_size)) {
5322                 aptr = v6a == NULL ? NULL : *v6a++;
5323                 /* LINTED: (note 1) */
5324                 cpi = (conn_pid_info_t *)((char *)ude6 +
5325                     udp6EntrySize);
5326                 print_hdr_once_v6 = udp_report_item_v6(ude6,
5327                     cpi, print_hdr_once_v6, aptr);
5328             }
5329         }
5330     }
5331     print_hdr_once_v4, aptr);
5332     print_hdr_once_v6, aptr);
5333 }

```

```

5327     }
5328     } /* 'for' loop 1 ends */
5329     (void) fflush(stdout);

5331     if (v4_attrs != NULL)
5332         free(v4_attrs);
5333     if (v6_attrs != NULL)
5334         free(v6_attrs);
5335 }

5337 static boolean_t
5338 udp_report_item_v4(const mib2_udpEntry_t *ude, conn_pid_info_t *cpi,
5339                  boolean_t first, const mib2_transportMLPEntry_t *attr)
5340 udp_report_item_v4(const mib2_udpEntry_t *ude, boolean_t first,
5341                  const mib2_transportMLPEntry_t *attr)
5342 {
5343     char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5344           /* hostname + portname */

5346     if (!(Aflag || ude->udpEntryInfo.ue_state >= MIB2_UDP_connected))
5347         return (first); /* Nothing to print */

5349     if (first) {
5350         (void) printf(v4compat ? "\nUDP\n" : "\nUDP: IPv4\n");

5352         if (Uflag)
5353             (void) printf(Vflag ? udp_hdr_v4_pid_verbose :
5354                          udp_hdr_v4_pid);
5355         else
5356             (void) printf(udp_hdr_v4_pid);
5357     #endif /* ! codereview */
5358     }

5360     (void) printf("%-20s %-20s ",
5361                  (void) printf("%-20s ",
5362                               pr_ap(ude->udpLocalAddress, ude->udpLocalPort, "udp",
5363                                     lname, sizeof (lname)),
5364                               lname, sizeof (lname)));
5365     (void) printf("%-20s %s\n",
5366                  ude->udpEntryInfo.ue_state == MIB2_UDP_connected ?
5367                  pr_ap(ude->udpEntryInfo.ue_RemoteAddress,
5368                        ude->udpEntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5369                  "");
5370     if (!Uflag) {
5371         (void) printf("%s\n",
5372                      "",
5373                      miudp_state(ude->udpEntryInfo.ue_state, attr));
5374     } else {
5375         int i = 0;
5376         pid_t *pids = cpi->cpi_pids;
5377         proc_info_t *pinfo;
5378         do {
5379             pinfo = get_proc_info(*pids);
5380             (void) printf("%-8.8s %6u ", pinfo->pr_user,
5381                          (int)*pids);
5382             if (Vflag) {
5383                 (void) printf("%-10.10s %s\n",
5384                               miudp_state(ude->udpEntryInfo.ue_state,
5385                                           attr,
5386                                           pinfo->pr_psargs);
5387             } else {
5388                 (void) printf("%-14.14s %s\n", pinfo->pr_fname,
5389                               miudp_state(ude->udpEntryInfo.ue_state,

```

```

5387         attr));
5388     }
5389     i++; pids++;
5390     } while (i < cpi->cpi_pids_cnt);
5391     }
5392 #endif /* ! codereview */

5394     print_transport_label(attr);

5396     return (first);
5397 }

5399 static boolean_t
5400 udp_report_item_v6(const mib2_udp6Entry_t *ude6, conn_pid_info_t *cpi,
5401                  boolean_t first, const mib2_transportMLPEntry_t *attr)
5402 udp_report_item_v6(const mib2_udp6Entry_t *ude6, boolean_t first,
5403                  const mib2_transportMLPEntry_t *attr)
5404 {
5405     char    lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5406           /* hostname + portname */
5407     char    ifname[LIFNAMSIZ + 1];
5408     const char *ifnamep;

5410     if (!(Aflag || ude6->udp6EntryInfo.ue_state >= MIB2_UDP_connected))
5411         return (first); /* Nothing to print */

5413     if (first) {
5414         (void) printf("\nUDP: IPv6\n");

5416         if (Uflag)
5417             (void) printf(Vflag ? udp_hdr_v6_pid_verbose :
5418                          udp_hdr_v6_pid);
5419         else
5420             (void) printf(udp_hdr_v6_pid);
5421     #endif /* ! codereview */
5422     }

5424     ifnamep = (ude6->udp6IfIndex != 0) ?
5425         if_indextoname(ude6->udp6IfIndex, ifname) : NULL;

5427     (void) printf("%-33s %-33s ",
5428                  (void) printf("%-33s ",
5429                               pr_ap6(ude6->udp6LocalAddress,
5430                                       ude6->udp6LocalPort, "udp", lname, sizeof (lname)),
5431                                       ude6->udp6LocalPort, "udp", lname, sizeof (lname)));
5432     (void) printf("%-33s %-10s %s\n",
5433                  ude6->udp6EntryInfo.ue_state == MIB2_UDP_connected ?
5434                  pr_ap6(ude6->udp6EntryInfo.ue_RemoteAddress,
5435                        ude6->udp6EntryInfo.ue_RemotePort, "udp", lname, sizeof (lname)) :
5436                  "");
5437     if (!Uflag) {
5438         (void) printf("%-10s %s\n",
5439                      "",
5440                      miudp_state(ude6->udp6EntryInfo.ue_state, attr,
5441                                  ifnamep == NULL ? "" : ifnamep);
5442     } else {
5443         int i = 0;
5444         pid_t *pids = cpi->cpi_pids;
5445         proc_info_t *pinfo;
5446         do {
5447             pinfo = get_proc_info(*pids);
5448             (void) printf("%-8.8s %6u ", pinfo->pr_user,

```

```

5447         if (vflag) {
5448             (void) printf("%-10.10s %-5.5s %s\n",
5449                 miudp_state(ude6->udp6EntryInfo.ue_state,
5450                     attr),
5451                 ifnamep == NULL ? "" : ifnamep,
5452                 pinfo->pr_psargs);
5453         } else {
5454             (void) printf("%-14.14s %-10.10s %s\n",
5455                 pinfo->pr_fname,
5456                 miudp_state(ude6->udp6EntryInfo.ue_state,
5457                     attr),
5458                 ifnamep == NULL ? "" : ifnamep);
5459         }
5460         i++; pids++;
5461     } while (i < cpi->cpi_pids_cnt);
5462 }
5463 #endif /* ! codereview */

5465     print_transport_label(attr);

5467     return (first);
5468 }

5470 /* ----- SCTP_REPORT----- */

5472 static const char sctp_hdr[] =
5473 "\nSCTP:";
5474 static const char sctp_hdr_normal[] =
5475 "    Local Address          Remote Address          "
5476 "Swind Send-Q Rwind Recv-Q StrsI/O  State\n"
5477 "-----"
5478 "-----";
5479 static const char sctp_hdr_pid[] =
5480 "    Local Address          Remote Address          "
5481 "Swind Send-Q Rwind Recv-Q StrsI/O  User  Pid  Command  State\n"
5482 "-----"
5483 "-----";
5484 static const char sctp_hdr_pid_verbose[] =
5485 "    Local Address          Remote Address          "
5486 "Swind Send-Q Rwind Recv-Q StrsI/O  User  Pid  State  Command\n"
5487 "-----"
5488 "-----";
5489 #endif /* ! codereview */

5491 static const char *
5492 nssctp_state(int state, const mib2_transportMLPEntry_t *attr)
5493 {
5494     static char sctpsbuf[50];
5495     const char *cp;

5497     switch (state) {
5498     case MIB2_SCTP_closed:
5499         cp = "CLOSED";
5500         break;
5501     case MIB2_SCTP_cookieWait:
5502         cp = "COOKIE_WAIT";
5503         break;
5504     case MIB2_SCTP_cookieEchoed:
5505         cp = "COOKIE_ECHOED";
5506         break;
5507     case MIB2_SCTP_established:
5508         cp = "ESTABLISHED";
5509         break;
5510     case MIB2_SCTP_shutdownPending:
5511         cp = "SHUTDOWN_PENDING";
5512         break;

```

```

5513     case MIB2_SCTP_shutdownSent:
5514         cp = "SHUTDOWN_SENT";
5515         break;
5516     case MIB2_SCTP_shutdownReceived:
5517         cp = "SHUTDOWN_RECEIVED";
5518         break;
5519     case MIB2_SCTP_shutdownAckSent:
5520         cp = "SHUTDOWN_ACK_SENT";
5521         break;
5522     case MIB2_SCTP_listen:
5523         cp = "LISTEN";
5524         break;
5525     default:
5526         (void) snprintf(sctpsbuf, sizeof (sctpsbuf),
5527             "UNKNOWN STATE(%d)", state);
5528         cp = sctpsbuf;
5529         break;
5530     }

5532     if (RSECFflag && attr != NULL && attr->tme_flags != 0) {
5533         if (cp != sctpsbuf) {
5534             (void) strncpy(sctpsbuf, cp, sizeof (sctpsbuf));
5535             cp = sctpsbuf;
5536         }
5537         if (attr->tme_flags & MIB2_TMEF_PRIVATE)
5538             (void) strcat(sctpsbuf, " P", sizeof (sctpsbuf));
5539         if (attr->tme_flags & MIB2_TMEF_SHARED)
5540             (void) strcat(sctpsbuf, " S", sizeof (sctpsbuf));
5541     }

5543     return (cp);
5544 }

5546 static const mib2_sctpConnRemoteEntry_t *
5547 sctp_getnext_rem(const mib_item_t **itemp,
5548     const mib2_sctpConnRemoteEntry_t *current, uint32_t associd)
5549 {
5550     const mib_item_t *item = *itemp;
5551     const mib2_sctpConnRemoteEntry_t *sre;

5553     for (; item != NULL; item = item->next_item, current = NULL) {
5554         if (!(item->group == MIB2_SCTP &&
5555             item->mib_id == MIB2_SCTP_CONN_REMOTE)) {
5556             continue;
5557         }

5559         if (current != NULL) {
5560             /* LINTED: (note 1) */
5561             sre = (const mib2_sctpConnRemoteEntry_t *)
5562                 ((const char *)current + sctpRemoteEntrySize);
5563         } else {
5564             sre = item->valp;
5565         }
5566         for (; (char *)sre < (char *)item->valp + item->length;
5567             /* LINTED: (note 1) */
5568             sre = (const mib2_sctpConnRemoteEntry_t *)
5569                 ((const char *)sre + sctpRemoteEntrySize)) {
5570             if (sre->sctpAssocId != associd) {
5571                 continue;
5572             }
5573             *itemp = item;
5574             return (sre);
5575         }
5576     }
5577     *itemp = NULL;
5578     return (NULL);

```

```

5579 }

5581 static const mib2_sctpConnLocalEntry_t *
5582 sctp_getnext_local(const mib_item_t **itemp,
5583                  const mib2_sctpConnLocalEntry_t *current, uint32_t associd)
5584 {
5585     const mib_item_t *item = *itemp;
5586     const mib2_sctpConnLocalEntry_t *sle;

5588     for (; item != NULL; item = item->next_item, current = NULL) {
5589         if (!(item->group == MIB2_SCTP &&
5590             item->mib_id == MIB2_SCTP_CONN_LOCAL)) {
5591             continue;
5592         }

5594         if (current != NULL) {
5595             /* LINTED: (note 1) */
5596             sle = (const mib2_sctpConnLocalEntry_t *)
5597                 ((const char *)current + sctpLocalEntrySize);
5598         } else {
5599             sle = item->valp;
5600         }
5601         for (; (char *)sle < (char *)item->valp + item->length;
5602             /* LINTED: (note 1) */
5603             sle = (const mib2_sctpConnLocalEntry_t *)
5604                 ((const char *)sle + sctpLocalEntrySize)) {
5605             if (sle->sctpAssocId != associd) {
5606                 continue;
5607             }
5608             *itemp = item;
5609             return (sle);
5610         }
5611     }
5612     *itemp = NULL;
5613     return (NULL);
5614 }

5616 static void
5617 sctp_pr_addr(int type, char *name, int namelen, const in6_addr_t *addr,
5618             int port)
5619 {
5620     ipaddr_t      v4addr;
5621     in6_addr_t    v6addr;

5623     /*
5624      * Address is either a v4 mapped or v6 addr. If
5625      * it's a v4 mapped, convert to v4 before
5626      * displaying.
5627      */
5628     switch (type) {
5629     case MIB2_SCTP_ADDR_V4:
5630         /* v4 */
5631         v6addr = *addr;

5633         IN6_V4MAPPED_TO_IPADDR(&v4addr, v6addr);
5634         if (port > 0) {
5635             (void) pr_ap(v4addr, port, "sctp", name, namelen);
5636         } else {
5637             (void) pr_addr(v4addr, name, namelen);
5638         }
5639         break;

5641     case MIB2_SCTP_ADDR_V6:
5642         /* v6 */
5643         if (port > 0) {
5644             (void) pr_ap6(addr, port, "sctp", name, namelen);

```

```

5645         } else {
5646             (void) pr_addr6(addr, name, namelen);
5647         }
5648         break;

5650     default:
5651         (void) snprintf(name, namelen, "<unknown addr type>");
5652         break;
5653     }
5654 }

5656 static boolean_t
5657 sctp_conn_report_item(const mib_item_t *head, conn_pid_info_t *cpi,
5658                     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,
5659                     const mib2_transportMLPEntry_t *attr)
5660 {
5661     char          lname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5662     char          fname[MAXHOSTNAMELEN + MAXHOSTNAMELEN + 1];
5663     const mib2_sctpConnRemoteEntry_t *sre = NULL;
5664     const mib2_sctpConnLocalEntry_t *sle = NULL;
5665     const mib_item_t *local = head;
5666     const mib_item_t *remote = head;
5667     uint32_t      id = sp->sctpAssocId;
5668     boolean_t     printfirst = B_TRUE;

5670     if (print_sctp_hdr == B_TRUE) {
5671         (void) puts(sctp_hdr);
5672         if (Uflag)
5673             (void) puts(Vflag? sctp_hdr_pid_verbose: sctp_hdr_pid);
5674         else
5675             (void) puts(sctp_hdr_normal);

5677         print_sctp_hdr = B_FALSE;
5678     }

5680 #endif /* ! codereview */
5681     sctp_pr_addr(sp->sctpAssocRemPrimAddrType, fname, sizeof (fname),
5682                &sp->sctpAssocRemPrimAddr, sp->sctpAssocRemPort);
5683     sctp_pr_addr(sp->sctpAssocRemPrimAddrType, lname, sizeof (lname),
5684                &sp->sctpAssocLocPrimAddr, sp->sctpAssocLocalPort);

5686     if (Uflag) {
5687         int i = 0;
5688         pid_t *pids = cpi->cpi_pids;
5689         proc_info_t *pinfo;
5690         do {
5691             pinfo = get_proc_info(*pids);
5692             (void) printf("%-31s %-31s %6d %6u %6d "
5693                 "%3d/%-3d %-8.8s %6u ",
5694                 lname, fname,
5695                 sp->sctpConnEntryInfo.ce_swnd,
5696                 sp->sctpConnEntryInfo.ce_sndq,
5697                 sp->sctpConnEntryInfo.ce_rwnd,
5698                 sp->sctpConnEntryInfo.ce_rcvq,
5699                 sp->sctpAssocInStreams,
5700                 sp->sctpAssocOutStreams,
5701                 pinfo->pr_user, (int)*pids);
5702             if (Vflag) {
5703                 (void) printf("%-11.11s %s\n",
5704                     nssctp_state(sp->sctpAssocState, attr),
5705                     pinfo->pr_psargs);
5706             } else {
5707                 (void) printf("%-14.14s %s\n",
5708                     pinfo->pr_fname,

```

```

5709             nssctp_state(sp->sctpAssocState, attr));
5710         }
5711         i++; pids++;
5712     } while (i < cpi->cpi_pids_cnt);

5714     } else {

5716 #endif /* ! codereview */
5717     (void) printf("%-31s %-31s %6u %6d %6u %6d %3d/%-3d %s\n",
5718         lname, fname,
5719         sp->sctpConnEntryInfo.ce_swnd,
5720         sp->sctpConnEntryInfo.ce_sendq,
5721         sp->sctpConnEntryInfo.ce_rwnd,
5722         sp->sctpConnEntryInfo.ce_rcvq,
5723         sp->sctpAssocInStreams, sp->sctpAssocOutStreams,
5724         nssctp_state(sp->sctpAssocState, attr));
5725     }
5726 #endif /* ! codereview */

5728     print_transport_label(attr);

5730     if (!Vflag) {
5731         return (print_sctp_hdr);
5732     }

5734     /* Print remote addresses/local addresses on following lines */
5735     while ((sre = sctp_getnext_rem(&remote, sre, id)) != NULL) {
5736         if (!IN6_ARE_ADDR_EQUAL(&sre->sctpAssocRemAddr,
5737             &sp->sctpAssocRemPrimAddr)) {
5738             if (printfirst == B_TRUE) {
5739                 (void) fputs("\t<Remote: ", stdout);
5740                 printfirst = B_FALSE;
5741             } else {
5742                 (void) fputs(", ", stdout);
5743             }
5744             sctp_pr_addr(sre->sctpAssocRemAddrType, fname,
5745                 sizeof (fname), &sre->sctpAssocRemAddr, -1);
5746             if (sre->sctpAssocRemAddrActive == MIB2_SCTP_ACTIVE) {
5747                 (void) fputs(fname, stdout);
5748             } else {
5749                 (void) printf("(%s)", fname);
5750             }
5751         }
5752     }
5753     if (printfirst == B_FALSE) {
5754         (void) puts(">");
5755         printfirst = B_TRUE;
5756     }
5757     while ((sle = sctp_getnext_local(&local, sle, id)) != NULL) {
5758         if (!IN6_ARE_ADDR_EQUAL(&sle->sctpAssocLocalAddr,
5759             &sp->sctpAssocLocPrimAddr)) {
5760             if (printfirst == B_TRUE) {
5761                 (void) fputs("\t<Local: ", stdout);
5762                 printfirst = B_FALSE;
5763             } else {
5764                 (void) fputs(", ", stdout);
5765             }
5766             sctp_pr_addr(sle->sctpAssocLocalAddrType, lname,
5767                 sizeof (lname), &sle->sctpAssocLocalAddr, -1);
5768             (void) fputs(lname, stdout);
5769         }
5770     }
5771     if (printfirst == B_FALSE) {
5772         (void) puts(">");
5773     }

```

```

5775     return (print_sctp_hdr);
5776 #endif /* ! codereview */
5777 }

5779 static void
5780 sctp_report(const mib_item_t *item)
5781 {
5782     const mib_item_t          *head;
5783     const mib2_sctpConnEntry_t *sp;
5784     boolean_t                 print_sctp_hdr_once = B_TRUE;
5785     boolean_t                 first = B_TRUE;
5786     mib2_transportMLPEntry_t **attrs, **aptr;
5787     mib2_transportMLPEntry_t *attr;
5788     conn_pid_info_t          *cpi;
5789 #endif /* ! codereview */

5790     /*
5791     * Preparation pass: the kernel returns separate entries for SCTP
5792     * connection table entries and Multilevel Port attributes. We loop
5793     * through the attributes first and set up an array for each address
5794     * family.
5795     */
5796     attrs = RSECflag ?
5797         gather_attrs(item, MIB2_SCTP, MIB2_SCTP_CONN, sctpEntrySize) :
5798         NULL;

5800     aptr = attrs;
5801     head = item;
5802     for (; item != NULL; item = item->next_item) {

5804         if (!(item->group == MIB2_SCTP &&
5805             item->mib_id == MIB2_SCTP_CONN) ||
5806             (item->group == MIB2_SCTP &&
5807             item->mib_id == EXPER_XPORT_PROC_INFO))
5808             continue;
5809         if (!(Uflag) && item->group == MIB2_SCTP &&
5810             item->mib_id == MIB2_SCTP_CONN) {
5811             #endif /* ! codereview */
5812             for (sp = item->valp;
5813                 (char *)sp < (char *)item->valp + item->length;
5814                 /* LINTED: (note 1) */
5815                 sp = (mib2_sctpConnEntry_t *)((char *)sp +
5816                     sctpEntrySize)) {
5817                 if (!(Aflag ||
5818                     sp->sctpAssocState >=
5819                     MIB2_SCTP_established))
5820                     continue;
5821                 sp = (mib2_sctpConnEntry_t *)((char *)sp + sctpEntrySize);
5822                 attr = aptr == NULL ? NULL : *aptr++;
5823                 print_sctp_hdr_once = sctp_conn_report_item(
5824                     head, NULL, print_sctp_hdr_once, sp, attr);
5825                 if (Aflag ||
5826                     sp->sctpAssocState >= MIB2_SCTP_established) {
5827                     if (first == B_TRUE) {
5828                         (void) puts(sctp_hdr);
5829                         (void) puts(sctp_hdr_normal);
5830                         first = B_FALSE;
5831                     }
5832                 } else if ((Uflag) && item->group == MIB2_SCTP &&
5833                     item->mib_id == EXPER_XPORT_PROC_INFO) {
5834                     for (sp = (mib2_sctpConnEntry_t *)item->valp;
5835                         (char *)sp < (char *)item->valp + item->length;

```

```

5830      /* LINTED: (note 1) */
5831      sp = (mib2_sctpConnEntry_t *)((char *)cpi +
5832      cpi->cpi_tot_size) {
5833          /* LINTED: (note 1) */
5834          cpi = (conn_pid_info_t *)((char *)sp +
5835          sctpEntrySize);
5836          if (!(Aflag ||
5837          sp->sctpAssocState >=
5838          MIB2_SCTP_established))
5839              continue;
5840          attr = aptr == NULL ? NULL : *aptr++;
5841          print_sctp_hdr_once = sctp_conn_report_item(
5842          head, cpi, print_sctp_hdr_once, sp, attr);
5843          sctp_conn_report_item(head, sp, attr);
5844      }
5845  }
5846  if (attrs != NULL)
5847      free(attrs);
5848  }

```

unchanged_portion_omitted

```

5868 static char *
5869 pktyscale(int n)
5889 pktyscale(n)
5890     int n;
5870 {
5871     static char buf[6];
5872     char t;
5873
5874     if (n < 1024) {
5875         t = ' ';
5876     } else if (n < 1024 * 1024) {
5877         t = 'k';
5878         n /= 1024;
5879     } else if (n < 1024 * 1024 * 1024) {
5880         t = 'm';
5881         n /= 1024 * 1024;
5882     } else {
5883         t = 'g';
5884         n /= 1024 * 1024 * 1024;
5885     }
5887     (void) snprintf(buf, sizeof (buf), "%4u%c", n, t);
5888     return (buf);
5889 }

```

unchanged_portion_omitted

```

6828 /*
6829  * Gets proc info in (proc_info_t) given pid. It doesn't return NULL.
6830  */
6831 proc_info_t *
6832 get_proc_info(pid_t pid)
6833 {
6834     static pid_t saved_pid = 0;
6835     static proc_info_t saved_proc_info;
6836     static proc_info_t unknown_proc_info = {"<unknown>", "", ""};
6837     static psinfo_t pinfo;
6838     char path[128];
6839     int fd;
6840
6841     /* hardcoded pid = 0 */
6842     if (pid == 0) {
6843         saved_proc_info.pr_user = "root";
6844         saved_proc_info.pr_fname = "sched";
6845         saved_proc_info.pr_psargs = "sched";

```

```

6846         saved_pid = 0;
6847         return (&saved_proc_info);
6848     }
6849
6850     if (pid == saved_pid)
6851         return (&saved_proc_info);
6852     if ((snprintf(path, 128, "/proc/%u/psinfo", (int)pid) > 0) &&
6853     ((fd = open(path, O_RDONLY)) != -1)) {
6854         if (read(fd, &pinfo, sizeof (pinfo)) == sizeof (pinfo)) {
6855             saved_proc_info.pr_user = get_username(pinfo.pr_uid);
6856             saved_proc_info.pr_fname = pinfo.pr_fname;
6857             saved_proc_info.pr_psargs = pinfo.pr_psargs;
6858             saved_pid = pid;
6859             (void) close(fd);
6860             return (&saved_proc_info);
6861         } else {
6862             (void) close(fd);
6863         }
6864     }
6865
6866     return (&unknown_proc_info);
6867 }
6868
6869 /*
6870  * Gets username given uid. It doesn't return NULL.
6871  */
6872 static char *
6873 get_username(uid_t u)
6874 {
6875     static uid_t saved_uid = UINT_MAX;
6876     static char saved_username[128];
6877     struct passwd *pw = NULL;
6878     if (u == UINT_MAX)
6879         return ("<unknown>");
6880     if (u == saved_uid && saved_username[0] != '\0')
6881         return (saved_username);
6882     setpwent();
6883     if ((pw = getpwuid(u)) != NULL)
6884         (void) strncpy(saved_username, pw->pw_name, 128);
6885     else
6886         (void) snprintf(saved_username, 128, "%u", u);
6887     saved_uid = u;
6888     return (saved_username);
6889 }
6890
6891 /*
6892  * #endif /* ! codereview */
6893  * print the usage line
6894  */
6895 static void
6896 usage(char *cmdname)
6897 {
6898     (void) fprintf(stderr, "usage: %s [-anuv] [-f address_family] "
6899     (void) fprintf(stderr, "usage: %s [-anv] [-f address_family] "
6900     "[-T d|u]\n", cmdname);
6901     (void) fprintf(stderr, "          %s [-n] [-f address_family] "
6902     "[-P protocol] [-T d|u] [-g | -p | -s [interval [count]]]\n",
6903     cmdname);
6904     (void) fprintf(stderr, "          %s -m [-v] [-T d|u] "
6905     "[interval [count]]\n", cmdname);
6906     (void) fprintf(stderr, "          %s -i [-I interface] [-an] "
6907     "[-f address_family] [-T d|u] [interval [count]]\n", cmdname);
6908     (void) fprintf(stderr, "          %s -r [-anv] "
6909     "[-f address_family|filter] [-T d|u]\n", cmdname);
6910     (void) fprintf(stderr, "          %s -M [-ns] [-f address_family] "
6911     "[-T d|u]\n", cmdname);

```

```

6911     (void) fprintf(stderr, "      %s -D [-I interface] "
6912     "[-f address_family] [-T d|u|n", cmdname);
6913     exit(EXIT_FAILURE);
6914 }

6916 /*
6917  * fatal: print error message to stderr and
6918  * call exit(errcode)
6919  */
6920 /*PRINTFLIKE2*/
6921 static void
6922 fatal(int errcode, char *format, ...)
6923 {
6924     va_list argp;

6926     if (format == NULL)
6927         return;

6929     va_start(argp, format);
6930     (void) vfprintf(stderr, format, argp);
6931     va_end(argp);

6933     exit(errcode);
6934 }

6937 /* -----UNIX Domain Sockets Report----- */

6940 #define NO_ADDR      "
6941 #define SO_PAIR     " (socketpair)

6943 static char      *typetname(t_scalar_t);
6944 static boolean_t uds_report_item(struct sockinfo *, boolean_t);

6947 static char uds_hdr[] = "\nActive UNIX domain sockets\n";

6949 static char uds_hdr_normal[] =
6950 " Type      Local Address
6951 " Remote Address\n"
6952 "-----"
6953 "-----\n";

6955 static char uds_hdr_pid[] =
6956 " Type      User      Pid      Command      "
6957 " Local Address
6958 " Remote Address\n"
6959 "-----"
6960 "-----"
6961 "-----\n";
6962 static char uds_hdr_pid_verbose[] =
6963 " Type      User      Pid      Local Address      "
6964 " Remote Address      Command\n"
6965 "-----"
6966 "-----\n";

6968 /*
6969  * Print a summary of connections related to unix protocols.
6970  */
6971 static void
6972 uds_report(kstat_ctl_t *kc)
6973 {
6974     int      i;
6975     kstat_t  *ksp;
6976     struct sockinfo *psi;

```

```

6977     boolean_t  print_uds_hdr_once = B_TRUE;

6979     if (kc == NULL) {
6980         fail(0, "uds_report: No kstat");
6981         exit(3);
6982     }

6984     if ((ksp = kstat_lookup(kc, "sockfs", 0, "sock_unix_list")) ==
6985         (kstat_t *)NULL) {
6986         fail(0, "kstat_data_lookup failed\n");
6987     }

6989     if (kstat_read(kc, ksp, NULL) == -1) {
6990         fail(0, "kstat_read failed for sock_unix_list\n");
6991     }

6993     if (ksp->ks_ndata == 0) {
6994         return;          /* no AF_UNIX sockets found */
6995     }

6997     /*
6998      * Having ks_data set with ks_data == NULL shouldn't happen;
6999      * If it does, the sockfs kstat is seriously broken.
7000      */
7001     if ((psi = ksp->ks_data) == NULL) {
7002         fail(0, "uds_report: no kstat data\n");
7003     }

7005     for (i = 0; i < ksp->ks_ndata; i++) {

7007         print_uds_hdr_once = uds_report_item(psi, print_uds_hdr_once);

7009         /* if si_size didn't get filled in, then we're done */
7010         if (psi->si_size == 0 ||
7011             !IS_P2ALIGNED(psi->si_size, sizeof (psi))) {
7012             break;
7013         }

7015         /* point to the next sockinfo in the array */
7016         /* LINTED: (note 1) */
7017         psi = (struct sockinfo *)(((char *)psi) + psi->si_size);
7018     }
7019 }

7021 static boolean_t
7022 uds_report_item(struct sockinfo *psi, boolean_t first)
7023 {
7024     int      i = 0;
7025     pid_t    *pids;
7026     proc_info_t *pinfo;
7027     char      *laddr, *raddr;

7029     if (first) {
7030         (void) printf("%s", uds_hdr);
7031         if (Uflag)
7032             (void) printf("%s", Vflag?uds_hdr_pid_verbose:
7033                 uds_hdr_pid);
7034     } else
7035         (void) printf("%s", uds_hdr_normal);

7037     first = B_FALSE;
7038 }

7040     pids = psi->si_pids;
7042     do {

```

```

7043     pinfo = get_proc_info(*pids);
7044     raddr = laddr = NO_ADDR;

7046     /* Try to fill laddr */
7047     if ((psi->si_state & SS_ISBOUND) &&
7048         strlen(psi->si_laddr_sun_path) != 0 &&
7049         psi->si_laddr_soa_len != 0) {
7050         if (psi->si_faddr_noxlate) {
7051             laddr = SO_PAIR;
7052         } else {
7053             if (psi->si_laddr_soa_len >
7054                 sizeof (psi->si_laddr_family))
7055                 laddr = psi->si_laddr_sun_path;
7056         }
7057     }

7059     /* Try to fill raddr */
7060     if ((psi->si_state & SS_ISCONNECTED) &&
7061         strlen(psi->si_faddr_sun_path) != 0 &&
7062         psi->si_faddr_soa_len != 0) {

7064         if (psi->si_faddr_noxlate) {
7065             raddr = SO_PAIR;
7066         } else {
7067             if (psi->si_faddr_soa_len >
7068                 sizeof (psi->si_faddr_family))
7069                 raddr = psi->si_faddr_sun_path;
7070         }
7071     }

7073     if (Uflag && Vflag) {
7074         (void) printf("%-10.10s %-8.8s %6u "
7075                     "%-39.39s %-39.39s %s\n",
7076                     typetname(psi->si_serv_type), pinfo->pr_user,
7077                     (int)*pids, laddr, raddr, pinfo->pr_psargs);
7078     } else if (Uflag && (!Vflag)) {
7079         (void) printf("%-10.10s %-8.8s %6u %-14.14s "
7080                     "%-39.39s %-39.39s\n",
7081                     typetname(psi->si_serv_type), pinfo->pr_user,
7082                     (int)*pids, pinfo->pr_fname, laddr, raddr);
7083     } else {
7084         (void) printf("%-10.10s %s %s\n",
7085                     typetname(psi->si_serv_type), laddr, raddr);
7086     }

7088     i++; pids++;
7089 } while (i < psi->si_pn_cnt);

7091     return (first);
7092 }

7094 static char *
7095 typetname(t_scalar_t type)
7096 {
7097     switch (type) {
7098     case T_CLTS:
7099         return ("dgram");

7101     case T_COTS:
7102         return ("stream");

7104     case T_COTS_ORD:
7105         return ("stream-ord");

7107     default:
7108         return ("");

```

```

7109     }
7110 #endif /* ! codereview */
7111 }

```

```

*****
48123 Wed Jul 13 01:32:35 2016
new/usr/src/cmd/perl/contrib/Sun/Solaris/Kstat/Kstat.xs
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2014 Racktop Systems.
25  */

27 /*
28  * Kstat.xs is a Perl XS (eXtension module) that makes the Solaris
29  * kstat(3KSTAT) facility available to Perl scripts. Kstat is a general-purpose
30  * mechanism for providing kernel statistics to users. The Solaris API is
31  * function-based (see the manpage for details), but for ease of use in Perl
32  * scripts this module presents the information as a nested hash data structure.
33  * It would be too inefficient to read every kstat in the system, so this module
34  * uses the Perl TIEHASH mechanism to implement a read-on-demand semantic, which
35  * only reads and updates kstats as and when they are actually accessed.
36  */

38 /*
39  * Ignored raw kstats.
40  *
41  * Some raw kstats are ignored by this module, these are listed below. The
42  * most common reason is that the kstats are stored as arrays and the ks_ndata
43  * and/or ks_data_size fields are invalid. In this case it is impossible to
44  * know how many records are in the array, so they can't be read.
45  *
46  * unix::sfmmu_percpu_stat
47  * This is stored as an array with one entry per cpu. Each element is of type
48  * struct sfmmu_percpu_stat. The ks_ndata and ks_data_size fields are bogus.
49  *
50  * ufs directio::UFS DirectIO Stats
51  * The structure definition used for these kstats (ufs_directio_kstats) is in a
52  * C file (uts/common/fs/ufs/ufs_directio.c) rather than a header file, so it
53  * isn't accessible.
54  *
55  * qlc::statistics
56  * This is a third-party driver for which we don't have source.
57  *
58  * mm::phys_installed
59  * This is stored as an array of uint64_t, with each pair of values being the
60  * (address, size) of a memory segment. The ks_ndata and ks_data_size fields
61  * are both zero.

```

```

62 *
63 * sockfs::sock_unix_list
64 * This is stored as an array with one entry per active socket. Each element
65 * is of type struct sockinfo. ks_ndata is the number of elements of that array
66 * and ks_data_size is the total size of the array.
67 *
68 * is of type struct k_sockinfo. The ks_ndata and ks_data_size fields are both
69 * zero.
70 *
71 * Note that the ks_ndata and ks_data_size of many non-array raw kstats are
72 * also incorrect. The relevant assertions are therefore commented out in the
73 * appropriate raw kstat read routines.
74 */

73 /* Kstat related includes */
74 #include <libgen.h>
75 #include <kstat.h>
76 #include <sys/var.h>
77 #include <sys/utsname.h>
78 #include <sys/sysinfo.h>
79 #include <sys/flock.h>
80 #include <sys/dnlc.h>
81 #include <nfs/nfs.h>
82 #include <nfs/nfs_clnt.h>

84 /* Ultra-specific kstat includes */
85 #ifdef __sparc
86 #include <vm/hat_sfmmu.h> /* from /usr/platform/sun4u/include */
87 #include <sys/simmstat.h> /* from /usr/platform/sun4u/include */
88 #include <sys/sysctrl.h> /* from /usr/platform/sun4u/include */
89 #include <sys/fhc.h> /* from /usr/include */
90 #endif

92 /*
93  * Solaris #defines SP, which conflicts with the perl definition of SP
94  * We don't need the Solaris one, so get rid of it to avoid warnings
95  */
96 #undef SP

98 /* Perl XS includes */
99 #include "EXTERN.h"
100 #include "perl.h"
101 #include "XSUB.h"

103 /* Debug macros */
104 #define DEBUG_ID "Sun::Solaris::Kstat"
105 #ifdef KSTAT_DEBUG
106 #define PERL_ASSERT(EXP) \
107     ((void)((EXP) || (croak("%s: assertion failed at %s:%d: %s", \
108     DEBUG_ID, __FILE__, __LINE__, #EXP), 0), 0))
109 #define PERL_ASSERTMSG(EXP, MSG) \
110     ((void)((EXP) || (croak(DEBUG_ID " : " MSG), 0), 0))
111 #else
112 #define PERL_ASSERT(EXP) ((void)0)
113 #define PERL_ASSERTMSG(EXP, MSG) ((void)0)
114 #endif

116 /* Macros for saving the contents of KSTAT_RAW structures */
117 #if defined(HAS_QUAD) && defined(USE_64_BIT_INT)
118 #define NEW_IV(V) \
119     (newSViv((IVTYPE) V))
120 #define NEW_UV(V) \
121     (newSVuv((UVTYPE) V))
122 #else
123 #define NEW_IV(V) \
124     (V >= IV_MIN && V <= IV_MAX ? newSViv((IVTYPE) V) : newSVnv((NVTYPE) V))
125 #endif

```

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

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

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

```

*****
23804 Wed Jul 13 01:32:35 2016
new/usr/src/man/man1m/netstat.1m
XXXX adding PID information to netstat output
*****
1  \' te
2  .\" Copyright (C) 2002, Sun Microsystems, Inc. All Rights Reserved
3  .\" Copyright 1989 AT&T
4  .\" Copyright (c) 1983 Regents of the University of California. All rights reser
5  .TH NETSTAT 1M "Jul 12, 2016"
6  .SH NAME
7  netstat \- show network status
8  .SH SYNOPSIS
9  .LP
10 .nf
11 \fBnetstat\fR [\fB-anuvR\fR] [\fB-f\fR \fIaddress_family\fR] [\fB-P\fR \fIprotoco
12 .fi
14 .LP
15 .nf
16 \fBnetstat\fR \fB-g\fR [\fB-nv\fR] [\fB-f\fR \fIaddress_family\fR]
17 .fi
19 .LP
20 .nf
21 \fBnetstat\fR \fB-p\fR [\fB-n\fR] [\fB-f\fR \fIaddress_family\fR]
22 .fi
24 .LP
25 .nf
26 \fBnetstat\fR \fB-s\fR [\fB-f\fR \fIaddress_family\fR] [\fB-P\fR \fIprotocol\fR]
27 [\fB-T\fR u | d ] [\fIinterval\fR [\fIcount\fR]]
28 .fi
30 .LP
31 .nf
32 \fBnetstat\fR \fB-m\fR [\fB-T\fR u | d ] [\fB-v\fR] [\fIinterval\fR [\fIcount\fR]
33 .fi
35 .LP
36 .nf
37 \fBnetstat\fR \fB-i\fR [\fB-I\fR \fIinterface\fR] [\fB-an\fR] [\fB-f\fR \fIaddre
38 [\fB-T\fR u | d ] [\fIinterval\fR [\fIcount\fR]]
39 .fi
41 .LP
42 .nf
43 \fBnetstat\fR \fB-r\fR [\fB-anvR\fR] [\fB-f\fR \fIaddress_family\fR | \fIfilter\
44 .fi
46 .LP
47 .nf
48 \fBnetstat\fR \fB-M\fR [\fB-ns\fR] [\fB-f\fR \fIaddress_family\fR]
49 .fi
51 .LP
52 .nf
53 \fBnetstat\fR \fB-D\fR [\fB-I\fR \fIinterface\fR] [\fB-f\fR \fIaddress_family\fR]
54 .fi
56 .SH DESCRIPTION
57 .LP
58 The \fBnetstat\fR command displays the contents of certain network-related data
59 structures in various formats, depending on the options you select.

```

```

60 .LP
61 The \fBnetstat\fR command has the several forms shown in the SYNOPSIS section,
62 above, listed as follows:
63 .RS +4
64 .TP
65 .ie t \(\bu
66 .el o
67 The first form of the command (with no required arguments) displays a list of
68 active sockets for each protocol.
69 .RE
70 .RS +4
71 .TP
72 .ie t \(\bu
73 .el o
74 The second, third, and fourth forms (\fB-g\fR, \fB-p\fR, and \fB-s\fR options)
75 display information from various network data structures.
76 .RE
77 .RS +4
78 .TP
79 .ie t \(\bu
80 .el o
81 The fifth form (\fB-m\fR option) displays STREAMS memory statistics.
82 .RE
83 .RS +4
84 .TP
85 .ie t \(\bu
86 .el o
87 The sixth form (\fB-i\fR option) shows the state of the interfaces.
88 .RE
89 .RS +4
90 .TP
91 .ie t \(\bu
92 .el o
93 The seventh form (\fB-r\fR option) displays the routing table.
94 .RE
95 .RS +4
96 .TP
97 .ie t \(\bu
98 .el o
99 The eighth form (\fB-M\fR option) displays the multicast routing table.
100 .RE
101 .RS +4
102 .TP
103 .ie t \(\bu
104 .el o
105 The ninth form (\fB-D\fR option) displays the state of \fBDHCP\fR on one or all
106 interfaces.
107 .RE
108 .LP
109 These forms are described in greater detail below.
110 .LP
111 With no arguments (the first form), \fBnetstat\fR displays connected sockets
112 for \fBPF_INET\fR, \fBPF_INET6\fR, and \fBPF_UNIX\fR, unless modified otherwise
113 by the \fB-f\fR option.
114 .SH OPTIONS
115 .ne 2
116 .na
117 \fB\fB-a\fR\fR
118 .ad
119 .sp .6
120 .RS 4n
121 Show the state of all sockets, all routing table entries, or all interfaces,
122 both physical and logical. Normally, listener sockets used by server processes
123 are not shown. Under most conditions, only interface, host, network, and
124 default routes are shown and only the status of physical interfaces is shown.
125 .RE

```

```

127 .sp
128 .ne 2
129 .na
130 \fB\fB-f\fR \fIaddress_family\fR\fR
131 .ad
132 .sp .6
133 .RS 4n
134 Limit all displays to those of the specified \fIaddress_family\fR. The value of
135 \fIaddress_family\fR can be one of the following:
136 .sp
137 .ne 2
138 .na
139 \fB\fBinet\fR\fR
140 .ad
141 .RS 9n
142 For the \fBAF_INET\fR address family showing IPv4 information.
143 .RE

145 .sp
146 .ne 2
147 .na
148 \fB\fBinet6\fR\fR
149 .ad
150 .RS 9n
151 For the \fBAF_INET6\fR address family showing IPv6 information.
152 .RE

154 .sp
155 .ne 2
156 .na
157 \fB\fBunix\fR\fR
158 .ad
159 .RS 9n
160 For the \fBAF_UNIX\fR address family.
161 .RE

163 .RE

165 .sp
166 .ne 2
167 .na
168 \fB\fB-f\fR \fIfilter\fR\fR
169 .ad
170 .sp .6
171 .RS 4n
172 With \fB-r\fR only, limit the display of routes to those matching the specified
173 filter. A filter rule consists of a \fIkeyword\fR:\fIvalue\fR pair. The known
174 keywords and the value syntax are:
175 .sp
176 .ne 2
177 .na
178 \fB\fBaf:\fR{\fBinet\fR|\fBinet6\fR|\fBunix\fR|\fInumber\fR}\fR
179 .ad
180 .sp .6
181 .RS 4n
182 Selects an address family. This is identical to \fB-f\fR \fIaddress_family\fR
183 and both syntaxes are supported.
184 .RE

186 .sp
187 .ne 2
188 .na
189 \fB\fBoutif\fR:{\fIname\fR|\fIifIndex\fR|\fBany\fR|\fBnone\fR}\fR
190 .ad
191 .sp .6

```

```

192 .RS 4n
193 Selects an output interface. You can specify the interface by name (such as
194 \fBhme0\fR) or by \fBifIndex\fR number (for example, \fB2\fR). If \fBany\fR is
195 used, the filter matches all routes having a specified interface (anything
196 other than null). If \fBnone\fR is used, the filter matches all routes having a
197 null interface. Note that you can view the index number (\fIifIndex\fR) for an
198 interface with the \fB-a\fR option of \fBifconfig\fR(1M).
199 .RE

201 .sp
202 .ne 2
203 .na
204 \fB\fBdst\fR:{\fIip-address\fR[/\fImask\fR]|\fBany\fR|\fBnone\fR}\fR
205 .ad
206 .sp .6
207 .RS 4n
208 Selects a destination IP address. If specified with a mask length, then any
209 routes with matching or longer (more specific) masks are selected. If \fBany\fR
210 is used, then all but addresses but 0 are selected. If \fBnone\fR is used, then
211 address 0 is selected.
212 .RE

214 .sp
215 .ne 2
216 .na
217 \fB\fBflags:\fR[\fB+ -\fR]?[\fBABDGHLSU\fR]\fB+\fR\fR
218 .ad
219 .sp .6
220 .RS 4n
221 Selects routes tagged with the specified flags. By default, the flags as
222 specified must be set in order to match. With a leading \fB+\fR, the flags
223 specified must be set but others are ignored. With a leading \fB-\fR, the flags
224 specified must not be set and others are permitted.
225 .RE

227 You can specify multiple instances of \fB-f\fR to specify multiple filters. For
228 example:
229 .sp
230 .in +2
231 .nf
232 % netstat -nr -f outif:hme0 -f outif:hme1 -f dst:10.0.0.0/8
233 .fi
234 .in -2
235 .sp

237 The preceding command displays routes within network 10.0.0.0/8, with mask
238 length 8 or greater, and an output interface of either \fBhme0\fR or
239 \fBhme1\fR, and excludes all other routes.
240 .RE

242 .sp
243 .ne 2
244 .na
245 \fB\fB-g\fR\fR
246 .ad
247 .sp .6
248 .RS 4n
249 Show the multicast group memberships for all interfaces. If the \fB-v\fR option
250 is included, source-specific membership information is also displayed. See
251 DISPLAYS, below.
252 .RE

254 .sp
255 .ne 2
256 .na
257 \fB\fB-i\fR\fR

```

```

258 .ad
259 .sp .6
260 .RS 4n
261 Show the state of the interfaces that are used for \fBIP\fR traffic. Normally
262 this shows statistics for the physical interfaces. When combined with the
263 \fB-a\fR option, this will also report information for the logical interfaces.
264 See \fBifconfig\fR(1M).
265 .RE

267 .sp
268 .ne 2
269 .na
270 \fB\fB-m\fR\fR
271 .ad
272 .sp .6
273 .RS 4n
274 Show the STREAMS memory statistics.
275 .RE

277 .sp
278 .ne 2
279 .na
280 \fB\fB-n\fR\fR
281 .ad
282 .sp .6
283 .RS 4n
284 Show network addresses as numbers. \fBnetstat\fR normally displays addresses as
285 symbols. This option may be used with any of the display formats.
286 .RE

288 .sp
289 .ne 2
290 .na
291 \fB\fB-p\fR\fR
292 .ad
293 .sp .6
294 .RS 4n
295 Show the net to media tables. See DISPLAYS, below.
296 .RE

298 .sp
299 .ne 2
300 .na
301 \fB\fB-r\fR\fR
302 .ad
303 .sp .6
304 .RS 4n
305 Show the routing tables. Normally, only interface, host, network, and default
306 routes are shown, but when this option is combined with the \fB-a\fR option,
307 all routes will be displayed, including cache. If you have not set up a
308 multicast route, \fB-ra\fR might not show any multicast routing entries,
309 although the kernel will derive such an entry if needed.
310 .RE

312 .sp
313 .ne 2
314 .na
315 \fB\fB-s\fR\fR
316 .ad
317 .sp .6
318 .RS 4n
319 Show per-protocol statistics. When used with the \fB-M\fR option, show
320 multicast routing statistics instead. When used with the \fB-a\fR option,
321 per-interface statistics will be displayed, when available, in addition to
322 statistics global to the system. See DISPLAYS, below.
323 .RE

```

```

325 .sp
326 .ne 2
327 .na
328 \fB\fB-T\fR \fB-u\fR | \fBd\fR\fR
329 .ad
330 .sp .6
331 .RS 4n
332 Display a time stamp.
333 .sp
334 Specify \fB-u\fR for a printed representation of the internal representation of
335 time. See \fBtime\fR(2). Specify \fBd\fR for standard date format. See
336 \fBdate\fR(1).
337 .RE

339 .sp
340 .ne 2
341 .na
342 \fB\fB-u\fR\fR
343 .ad
344 .sp .6
345 .RS 4n
346 When specified, for each network endpoint \fBnetstat\fR will print the list of
347 the processes currently have an open file descriptor pointing to that endpoint.
348 \fBnetstat\fR will list the username, process id, and the program for each
349 process in that list.
350 #endif /* ! codereview */
351 .RE

353 .sp
354 .ne 2
355 .na
356 \fB\fB-v\fR\fR
357 .ad
358 .sp .6
359 .RS 4n
360 Verbose. Show additional information for the sockets, STREAMS memory
361 statistics, routing table, and multicast group memberships.
362 .RE

364 .sp
365 .ne 2
366 .na
367 \fB\fB-I\fR \fB-iinterface\fR\fR
368 .ad
369 .sp .6
370 .RS 4n
371 Show the state of a particular interface. \fBinterface\fR can be any valid
372 interface such as \fBhme0\fR or \fBerio\fR. Normally, the status and statistics
373 for physical interfaces are displayed. When this option is combined with the
374 \fB-a\fR option, information for the logical interfaces is also reported.
375 .RE

377 .sp
378 .ne 2
379 .na
380 \fB\fB-M\fR\fR
381 .ad
382 .sp .6
383 .RS 4n
384 Show the multicast routing tables. When used with the \fB-s\fR option, show
385 multicast routing statistics instead.
386 .RE

388 .sp
389 .ne 2

```

```

390 .na
391 \fB\fB-P\fR \fIprotocol\fR\fR
392 .ad
393 .sp .6
394 .RS 4n
395 Limit display of statistics or state of all sockets to those applicable to
396 \fIprotocol\fR. The protocol can be one of \fBip\fR, \fBipv6\fR, \fBicmp\fR,
397 \fBicmpv6\fR, \fBicmp\fR, \fBicmpv6\fR, \fBbigmp\fR, \fBudp\fR, \fBtcp\fR,
398 \fBrawip\fR. \fBrawip\fR can also be specified as \fBraw\fR. The command
399 accepts protocol options only as all lowercase.
400 .RE

402 .sp
403 .ne 2
404 .na
405 \fB\fB-D\fR\fR
406 .ad
407 .sp .6
408 .RS 4n
409 Show the status of \fBDHCP\fR configured interfaces.
410 .RE

412 .sp
413 .ne 2
414 .na
415 \fB\fB-R\fR\fR
416 .ad
417 .sp .6
418 .RS 4n
419 This modifier displays extended security attributes for sockets and routing
420 table entries. The \fB-R\fR modifier is available only if the system is
421 configured with the Solaris Trusted Extensions feature.
422 .sp
423 With \fB-r\fR only, this option displays the routing entries' gateway security
424 attributes. See \fBroute\fR(1M) for more information on security attributes.
425 .sp
426 When displaying socket information using the first form of the command, this
427 option displays additional information for Multi-Level Port(MLP) sockets. This
428 includes:
429 .RS +4
430 .TP
431 .ie t \(\bu
432 .el o
433 The label for the peer if the socket is connected.
434 .RE
435 .RS +4
436 .TP
437 .ie t \(\bu
438 .el o
439 The following flags can be appended to the socket's "State" output:
440 .RS

442 .sp
443 .ne 2
444 .na
445 \fB\fB-BP\fR\fR
446 .ad
447 .RS 5n
448 The socket is a MLP on zone-private IP addresses.
449 .RE

451 .sp
452 .ne 2
453 .na
454 \fB\fB-S\fR\fR
455 .ad

```

```

456 .RS 5n
457 The socket is a MLP on IP addresses shared between zones.
458 .RE
459 .SH OPERANDS
460 .ne 2
461 .na
462 \fB\fIinterval\fR\fR
463 .ad
464 .RS 12n
465 Display statistics accumulated since last display every \fIinterval\fR seconds,
466 repeating forever, unless \fIcount\fR is specified. When invoked with
467 \fIinterval\fR, the first row of netstat output shows statistics accumulated
468 since last reboot.
469 .sp
470 The following options support \fIinterval\fR: \fB-i\fR, \fB-m\fR, \fB-s\fR and
471 \fB-Ms\fR. Some values are configuration parameters and are just redisplayed at
472 each interval.
473 .RE

475 .sp
476 .ne 2
477 .na
478 \fB\fIcount\fR\fR
479 .ad
480 .RS 12n
481 Display interface statistics the number of times specified by \fIcount\fR, at
482 the interval specified by \fIinterval\fR.
483 .RE

485 .SH DISPLAYS
486 .SS "Active Sockets (First Form)"
487 .LP
488 The display for each active socket shows the local and remote address, the send
489 and receive queue sizes (in bytes), the send and receive windows (in bytes),
490 and the internal state of the protocol.
491 .LP
492 The symbolic format normally used to display socket addresses is either:
493 .sp
494 .in +2
495 .nf
496 \fBhostname\fR.\fIport\fR
497 .fi
498 .in -2
499 .sp
500 when the name of the host is specified, or
501 .sp
502 .in +2
503 .nf
504 \fInetwork\fR.\fIport\fR
505 .fi
506 .in -2
507 .sp
508 if a socket address specifies a network but no specific host.
509 .LP
510 The numeric host address or network number associated with the socket is used
511 to look up the corresponding symbolic hostname or network name in the
512 \fIhosts\fR or \fInetworks\fR database.
513 .LP
514 If the network or hostname for an address is not known, or if the \fB-n\fR
515 option is specified, the numerical network address is shown. Unspecified, or
516 "wildcard", addresses and ports appear as an asterisk (\fB*\fR). For more
517 information regarding the Internet naming conventions, refer to \fBinet\fR(7P)
518 and \fBinet6\fR(7P).
519 .LP
520 For SCTP sockets, because an endpoint can be represented by multiple addresses,
521 the verbose option (\fB-v\fR) displays the list of all the local and remote

```

```

522 addresses.
523 .SS "\fITCP Sockets\fR"
524 .LP
525 The possible state values for \fBTCP\fR sockets are as follows:
526 .sp
527 .ne 2
528 .na
529 \fB\fBBOUND\fR\fR
530 .ad
531 .RS 16n
532 Bound, ready to connect or listen.
533 .RE

535 .sp
536 .ne 2
537 .na
538 \fB\fBCLOSED\fR\fR
539 .ad
540 .RS 16n
541 Closed. The socket is not being used.
542 .RE

544 .sp
545 .ne 2
546 .na
547 \fB\fBCLOSING\fR\fR
548 .ad
549 .RS 16n
550 Closed, then remote shutdown; awaiting acknowledgment.
551 .RE

553 .sp
554 .ne 2
555 .na
556 \fB\fBCLOSE_WAIT\fR\fR
557 .ad
558 .RS 16n
559 Remote shutdown; waiting for the socket to close.
560 .RE

562 .sp
563 .ne 2
564 .na
565 \fB\fBESTABLISHED\fR\fR
566 .ad
567 .RS 16n
568 Connection has been established.
569 .RE

571 .sp
572 .ne 2
573 .na
574 \fB\fBFIN_WAIT_1\fR\fR
575 .ad
576 .RS 16n
577 Socket closed; shutting down connection.
578 .RE

580 .sp
581 .ne 2
582 .na
583 \fB\fBFIN_WAIT_2\fR\fR
584 .ad
585 .RS 16n
586 Socket closed; waiting for shutdown from remote.
587 .RE

```

```

589 .sp
590 .ne 2
591 .na
592 \fB\fBIDLE\fR\fR
593 .ad
594 .RS 16n
595 Idle, opened but not bound.
596 .RE

598 .sp
599 .ne 2
600 .na
601 \fB\fBLAST_ACK\fR\fR
602 .ad
603 .RS 16n
604 Remote shutdown, then closed; awaiting acknowledgment.
605 .RE

607 .sp
608 .ne 2
609 .na
610 \fB\fBLISTEN\fR\fR
611 .ad
612 .RS 16n
613 Listening for incoming connections.
614 .RE

616 .sp
617 .ne 2
618 .na
619 \fB\fBSYN_RECEIVED\fR\fR
620 .ad
621 .RS 16n
622 Initial synchronization of the connection under way.
623 .RE

625 .sp
626 .ne 2
627 .na
628 \fB\fBSYN_SENT\fR\fR
629 .ad
630 .RS 16n
631 Actively trying to establish connection.
632 .RE

634 .sp
635 .ne 2
636 .na
637 \fB\fBTIME_WAIT\fR\fR
638 .ad
639 .RS 16n
640 Wait after close for remote shutdown retransmission.
641 .RE

643 .SS "\fISCTP Sockets\fR"
644 .LP
645 The possible state values for SCTP sockets are as follows:
646 .sp
647 .ne 2
648 .na
649 \fB\fBCLOSED\fR\fR
650 .ad
651 .RS 21n
652 Closed. The socket is not being used.
653 .RE

```

```

655 .sp
656 .ne 2
657 .na
658 \fb\FBLISTEN\FR\FR
659 .ad
660 .RS 21n
661 Listening for incoming associations.
662 .RE

664 .sp
665 .ne 2
666 .na
667 \fb\FBESTABLISHED\FR\FR
668 .ad
669 .RS 21n
670 Association has been established.
671 .RE

673 .sp
674 .ne 2
675 .na
676 \fb\FBCOOKIE_WAIT\FR\FR
677 .ad
678 .RS 21n
679 \fb\FINIT\FR has been sent to the peer, awaiting acknowledgment.
680 .RE

682 .sp
683 .ne 2
684 .na
685 \fb\FBCOOKIE_ECHOED\FR\FR
686 .ad
687 .RS 21n
688 State cookie from the INIT-ACK has been sent to the peer, awaiting
689 acknowledgment.
690 .RE

692 .sp
693 .ne 2
694 .na
695 \fb\FBSHUTDOWN_PENDING\FR\FR
696 .ad
697 .RS 21n
698 \fb\FBSHUTDOWN\FR has been received from the upper layer, awaiting acknowledgement
699 of all outstanding \fb\FBDATA\FR from the peer.
700 .RE

702 .sp
703 .ne 2
704 .na
705 \fb\FBSHUTDOWN_SENT\FR\FR
706 .ad
707 .RS 21n
708 All outstanding data has been acknowledged in the \fb\FBSHUTDOWN_SENT\FR state.
709 \fb\FBSHUTDOWN\FR has been sent to the peer, awaiting acknowledgment.
710 .RE

712 .sp
713 .ne 2
714 .na
715 \fb\FBSHUTDOWN_RECEIVED\FR\FR
716 .ad
717 .RS 21n
718 \fb\FBSHUTDOWN\FR has been received from the peer, awaiting acknowledgement of all
719 outstanding \fb\FBDATA\FR.

```

```

720 .RE

722 .sp
723 .ne 2
724 .na
725 \fb\FBSHUTDOWN_ACK_SENT\FR\FR
726 .ad
727 .RS 21n
728 All outstanding data has been acknowledged in the \fb\FBSHUTDOWN_RECEIVED\FR
729 state. \fb\FBSHUTDOWN_ACK\FR has been sent to the peer.
730 .RE

732 .SS "Network Data Structures (Second Through Fifth Forms)"
733 .LP
734 The form of the display depends upon which of the \fb-g\FR, \fb-m\FR, \fb-p\FR,
735 or \fb-s\FR options you select.
736 .sp
737 .ne 2
738 .na
739 \fb\FB-g\FR\FR
740 .ad
741 .RS 6n
742 Displays the list of multicast group membership.
743 .RE

745 .sp
746 .ne 2
747 .na
748 \fb\FB-m\FR\FR
749 .ad
750 .RS 6n
751 Displays the memory usage, for example, STREAMS mblks.
752 .RE

754 .sp
755 .ne 2
756 .na
757 \fb\FB-p\FR\FR
758 .ad
759 .RS 6n
760 Displays the net to media mapping table. For IPv4, the address resolution table
761 is displayed. See \fb\Barp\FR(lm). For IPv6, the neighbor cache is displayed.
762 .RE

764 .sp
765 .ne 2
766 .na
767 \fb\FB-s\FR\FR
768 .ad
769 .RS 6n
770 Displays the statistics for the various protocol layers.
771 .RE

773 .sp
774 .LP
775 The statistics use the MIB specified variables. The defined values for
776 \fb\FBipForwarding\FR are:
777 .sp
778 .ne 2
779 .na
780 \fb\FBforwarding(1)\FR\FR
781 .ad
782 .RS 21n
783 Acting as a gateway.
784 .RE

```

```

786 .sp
787 .ne 2
788 .na
789 \fB\fBnot-forwarding(2)\fR\fR
790 .ad
791 .RS 21n
792 Not acting as a gateway.
793 .RE

795 .sp
796 .LP
797 The IPv6 and ICMPv6 protocol layers maintain per-interface statistics. If the
798 \fB-a\fR option is specified with the \fB-s\fR option, then the per-interface
799 statistics as well as the total sums are displayed. Otherwise, just the sum of
800 the statistics are shown.
801 .LP
802 For the second, third, and fourth forms of the command, you must specify at
803 least \fB-g\fR, \fB-p\fR, or \fB-s\fR. You can specify any combination of these
804 options. You can also specify \fB-m\fR (the fifth form) with any set of the
805 \fB-g\fR, \fB-p\fR, and \fB-s\fR options. If you specify more than one of these
806 options, \fBnetstat\fR displays the information for each one of them.
807 .SS "Interface Status (Sixth Form)"
808 .LP
809 The interface status display lists information for all current interfaces, one
810 interface per line. If an interface is specified using the \fB-I\fR option, it
811 displays information for only the specified interface.
812 .LP
813 The list consists of the interface name, \fBmtu\fR (maximum transmission unit,
814 or maximum packet size)(see \fBifconfig\fR(1M)), the network to which the
815 interface is attached, addresses for each interface, and counter associated
816 with the interface. The counters show the number of input packets, input
817 errors, output packets, output errors, and collisions, respectively. For
818 Point-to-Point interfaces, the Net/Dest field is the name or address on the
819 other side of the link.
820 .LP
821 If the \fB-a\fR option is specified with either the \fB-i\fR option or the
822 \fB-I\fR option, then the output includes names of the physical interface(s),
823 counts for input packets and output packets for each logical interface, plus
824 additional information.
825 .LP
826 If the \fB-n\fR option is specified, the list displays the IP address instead
827 of the interface name.
828 .LP
829 If an optional \fB-iinterval\fR is specified, the output will be continually
830 displayed in \fBinterval\fR seconds until interrupted by the user or until
831 \fBicount\fR is reached. See OPERANDS.
832 .LP
833 The physical interface is specified using the \fB-I\fR option. When used with
834 the \fBinterval\fR operand, output for the \fB-I\fR option has the following
835 format:
836 .sp
837 .in +2
838 .nf
839 input      eri0          output          input          (Total)      output
840 packets  errs  packets  errs  colls  packets  errs  packets  errs  colls
841 227681    0    659471  1    502    261331  0    99597    1    502
842 10        0    0        0    0      10       0    0        0    0
843 8         0    0        0    0      8        0    0        0    0
844 10       0    2        0    0      10       0    2        0    0
845 .fi
846 .in -2
847 .sp
848 .LP
849 If the input interface is not specified, the first interface of address family
850 \fBbinet\fR or \fBbinet6\fR will be displayed.
851 .SS "Routing Table (Seventh Form)"

```

```

852 .LP
853 The routing table display lists the available routes and the status of each.
854 Each route consists of a destination host or network, and a gateway to use in
855 forwarding packets. The \fBiflags\fR column shows the status of the route. These
856 flags are as follows:
857 .sp
858 .ne 2
859 .na
860 \fB\fbu\fR\fR
861 .ad
862 .RS 5n
863 Indicates route is \fBup\fR.
864 .RE

866 .sp
867 .ne 2
868 .na
869 \fB\fbg\fR\fR
870 .ad
871 .RS 5n
872 Route is to a gateway.
873 .RE

875 .sp
876 .ne 2
877 .na
878 \fB\fbh\fR\fR
879 .ad
880 .RS 5n
881 Route is to a host and not a network.
882 .RE

884 .sp
885 .ne 2
886 .na
887 \fB\fbm\fR\fR
888 .ad
889 .RS 5n
890 Redundant route established with the \fB-multirt\fR option.
891 .RE

893 .sp
894 .ne 2
895 .na
896 \fB\fbS\fR\fR
897 .ad
898 .RS 5n
899 Route was established using the \fB-setsrc\fR option.
900 .RE

902 .sp
903 .ne 2
904 .na
905 \fB\fbD\fR\fR
906 .ad
907 .RS 5n
908 Route was created dynamically by a redirect.
909 .RE

911 .sp
912 .LP
913 If the \fB-a\fR option is specified, there will be routing entries with the
914 following flags:
915 .sp
916 .ne 2
917 .na

```

```

918 \fB\fBA\fR\fR
919 .ad
920 .RS 5n
921 Combined routing and address resolution entries.
922 .RE

924 .sp
925 .ne 2
926 .na
927 \fB\fBB\fR\fR
928 .ad
929 .RS 5n
930 Broadcast addresses.
931 .RE

933 .sp
934 .ne 2
935 .na
936 \fB\fBL\fR\fR
937 .ad
938 .RS 5n
939 Local addresses for the host.
940 .RE

942 .LP
943 Interface routes are created for each interface attached to the local host; the
944 gateway field for such entries shows the address of the outgoing interface.
945 .LP
946 The \fBuse\fR column displays the number of packets sent using a combined
947 routing and address resolution (\fBA\fR) or a broadcast (\fBB\fR) route. For a
948 local (\fBL\fR) route, this count is the number of packets received, and for
949 all other routes it is the number of times the routing entry has been used to
950 create a new combined route and address resolution entry.
951 .LP
952 The \fIinterface\fR entry indicates the network interface utilized for the
953 route.
954 .SS "Multicast Routing Tables (Eighth Form)"
955 .LP
956 The multicast routing table consists of the virtual interface table and the
957 actual routing table.
958 .SS "DHCP Interface Information (Ninth Form)"
959 .LP
960 The \fBDHCP\fR interface information consists of the interface name, its
961 current state, lease information, packet counts, and a list of flags.
962 .LP
963 The states correlate with the specifications set forth in \fIRFC 2131\fR.
964 .LP
965 Lease information includes:
966 .RS +4
967 .TP
968 .ie t \(\bu
969 .el o
970 when the lease began;
971 .RE
972 .RS +4
973 .TP
974 .ie t \(\bu
975 .el o
976 when lease renewal will begin; and
977 .RE
978 .RS +4
979 .TP
980 .ie t \(\bu
981 .el o
982 when the lease will expire.
983 .RE

```

```

984 .sp
985 .LP
986 The flags currently defined include:
987 .sp
988 .ne 2
989 .na
990 \fB\fBBOOTP\fR\fR
991 .ad
992 .RS 11n
993 The interface has a lease obtained through \fBBOOTP\fR (IPv4 only).
994 .RE

996 .sp
997 .ne 2
998 .na
999 \fB\fBBUSY\fR\fR
1000 .ad
1001 .RS 11n
1002 The interface is busy with a \fBDHCP\fR transaction.
1003 .RE

1005 .sp
1006 .ne 2
1007 .na
1008 \fB\fBPRIMARY\fR\fR
1009 .ad
1010 .RS 11n
1011 The interface is the primary interface. See \fBdhcpcinfo\fR(1) and
1012 \fBifconfig\fR(1M).
1013 .RE

1015 .sp
1016 .ne 2
1017 .na
1018 \fB\fBFAILED\fR\fR
1019 .ad
1020 .RS 11n
1021 The interface is in failure state and must be manually restarted.
1022 .RE

1024 .LP
1025 Packet counts are maintained for the number of packets sent, the number of
1026 packets received, and the number of lease offers declined by the \fBDHCP\fR
1027 client. All three counters are initialized to zero and then incremented while
1028 obtaining a lease. The counters are reset when the period of lease renewal
1029 begins for the interface. Thus, the counters represent either the number of
1030 packets sent, received, and declined while obtaining the current lease, or the
1031 number of packets sent, received, and declined while attempting to obtain a
1032 future lease.
1033 .SH FILES
1034 .ne 2
1035 .na
1036 \fB\fB/etc/default/inet_type\fR\fR
1037 .ad
1038 .RS 26n
1039 \fBDEFAULT_IP\fR setting
1040 .RE

1042 .SH SEE ALSO
1043 .LP
1044 \fBBarp\fR(1M), \fBdhcpcinfo\fR(1), \fBdhcpcagent\fR(1M), \fBifconfig\fR(1M),
1045 \fBlostat\fR(1M), \fBkstat\fR(1M), \fBmibiisa\fR(1M), \fBndp\fR(1M), \fBsavecore\fR(1M),
1046 \fBvmstat\fR(1M), \fBhosts\fR(4), \fBinet_type\fR(4), \fBnetworks\fR(4),
1047 \fBprotocols\fR(4), \fBservices\fR(4), \fBattributes\fR(5), \fBdhcpc\fR(5),
1048 \fBkstat\fR(7D), \fBinet\fR(7P), \fBinet6\fR(7P)
1049 .sp

```

1050 .LP
1051 Droms, R., \firfc 2131, Dynamic Host Configuration Protocol\fr, Network Working
1052 Group, March 1997.
1053 .sp
1054 .LP
1055 Droms, R. \firfc 3315, Dynamic Host Configuration Protocol for IPv6
1056 (DHCPv6)\fr. Cisco Systems. July 2003.
1057 .SH NOTES
1058 .LP
1059 When displaying interface information, \fbnetstat\fr honors the
1060 \fbDEFAULT_IP\fr setting in \fb/etc/default/inet_type\fr. If it is set to
1061 \fbIP_VERSION4\fr, then \fbnetstat\fr will omit information relating to IPv6
1062 interfaces, statistics, connections, routes and the like.
1063 .LP
1064 However, you can override the \fbDEFAULT_IP\fr setting in
1065 \fb/etc/default/inet_type\fr on the command-line. For example, if you have used
1066 the command-line to explicitly request IPv6 information by using the
1067 \fbinet6\fr address family or one of the IPv6 protocols, it will override the
1068 \fbDEFAULT_IP\fr setting.
1069 .LP
1070 If you need to examine network status information following a kernel crash, use
1071 the \fbmbd\fr(1) utility on the \fbsavecore\fr(1M) output.
1072 .LP
1073 The \fbnetstat\fr utility obtains TCP statistics from the system by opening
1074 \fb/dev/tcp\fr and issuing queries. Because of this, \fbnetstat\fr might
1075 display an extra, unused connection in \fbIDLE\fr state when reporting
1076 connection status.
1077 .LP
1078 Previous versions of \fbnetstat\fr had undocumented methods for reporting
1079 kernel statistics published using the \fbkstat\fr(7D) facility. This
1080 functionality has been removed. Use \fbkstat\fr(1M) instead.
1081 .LP
1082 \fbnetstat\fr restricts its output to information that is relevant to the zone
1083 in which \fbnetstat\fr runs. (This is true for both shared-IP and exclusive-IP
1084 zones.)

new/usr/src/pkg/manifests/system-header.mf

1

```
*****
90535 Wed Jul 13 01:32:36 2016
new/usr/src/pkg/manifests/system-header.mf
XXXX adding PID information to netstat output
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 #
23 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright (c) 2012 by Delphix. All rights reserved.
25 # Copyright 2013 Saso Kiselkov. All rights reserved.
26 # Copyright 2014 Garrett D'Amore <garrett@damore.org>
27 # Copyright 2016 Nexenta Systems, Inc.
28 #
29 #
30 set name=pkg.fmri value=pkg:/system/header@$(PKGVERS)
31 set name=pkg.description \
32   value="SunOS C/C++ header files for general development of software"
33 set name=pkg.summary value="SunOS Header Files"
34 set name=info.classification value=org.opensolaris.category.2008:System/Core
35 set name=variant.arch value=$(ARCH)
36 dir path=usr group=sys
37 dir path=usr/include
38 $(i386_ONLY)dir path=usr/include/$(ARCH64)
39 $(i386_ONLY)dir path=usr/include/$(ARCH64)/sys
40 dir path=usr/include/ads
41 dir path=usr/include/arpa
42 dir path=usr/include/asm
43 dir path=usr/include/ast
44 dir path=usr/include/bsm
45 dir path=usr/include/dat
46 dir path=usr/include/des
47 dir path=usr/include/gssapi
48 dir path=usr/include/hal
49 $(i386_ONLY)dir path=usr/include/ia32
50 $(i386_ONLY)dir path=usr/include/ia32/sys
51 dir path=usr/include/inet
52 dir path=usr/include/inet/kssl
53 dir path=usr/include/ipp
54 dir path=usr/include/ipp/ipgpc
55 dir path=usr/include/iso
56 dir path=usr/include/kerberosv5
57 dir path=usr/include/libpolkit
58 dir path=usr/include/net
59 dir path=usr/include/netinet
60 dir path=usr/include/nfs
61 dir path=usr/include/protocols
```

new/usr/src/pkg/manifests/system-header.mf

2

```
62 dir path=usr/include/rpc
63 dir path=usr/include/rpcsvc
64 dir path=usr/include/sasl
65 dir path=usr/include/scsi
66 dir path=usr/include/scsi/plugins
67 dir path=usr/include/scsi/plugins/ses
68 dir path=usr/include/scsi/plugins/ses/framework
69 dir path=usr/include/scsi/plugins/ses/vendor
70 dir path=usr/include/scsi/plugins/smp
71 dir path=usr/include/scsi/plugins/smp/engine
72 dir path=usr/include/scsi/plugins/smp/framework
73 dir path=usr/include/security
74 dir path=usr/include/sharefs
75 dir path=usr/include/sys
76 dir path=usr/include/sys/av
77 dir path=usr/include/sys/contract
78 dir path=usr/include/sys/crypto
79 dir path=usr/include/sys/dktp
80 dir path=usr/include/sys/fc4
81 dir path=usr/include/sys/fm
82 dir path=usr/include/sys/fm/cpu
83 dir path=usr/include/sys/fm/fs
84 dir path=usr/include/sys/fm/io
85 $(sparc_ONLY)dir path=usr/include/sys/fpu
86 dir path=usr/include/sys/fs
87 dir path=usr/include/sys/hotplug
88 dir path=usr/include/sys/hotplug/pci
89 dir path=usr/include/sys/ib
90 dir path=usr/include/sys/ib/adapters
91 dir path=usr/include/sys/ib/adapters/hermon
92 dir path=usr/include/sys/ib/adapters/tavor
93 dir path=usr/include/sys/ib/clients
94 dir path=usr/include/sys/ib/clients/ibd
95 dir path=usr/include/sys/ib/clients/of
96 dir path=usr/include/sys/ib/clients/of/rdma
97 dir path=usr/include/sys/ib/clients/of/sol_ofs
98 dir path=usr/include/sys/ib/clients/of/sol_ucma
99 dir path=usr/include/sys/ib/clients/of/sol_umad
100 dir path=usr/include/sys/ib/clients/of/sol_uverbs
101 dir path=usr/include/sys/ib/ibnex
102 dir path=usr/include/sys/ib/ibt1
103 dir path=usr/include/sys/ib/ibt1/impl
104 dir path=usr/include/sys/ib/mgt
105 dir path=usr/include/sys/ib/mgt/ibmf
106 dir path=usr/include/sys/iso
107 dir path=usr/include/sys/lvm
108 dir path=usr/include/sys/proc
109 dir path=usr/include/sys/rsm
110 $(i386_ONLY)dir path=usr/include/sys/sata group=sys
111 dir path=usr/include/sys/scsi
112 dir path=usr/include/sys/scsi/adapters
113 dir path=usr/include/sys/scsi/conf
114 dir path=usr/include/sys/scsi/generic
115 dir path=usr/include/sys/scsi/impl
116 dir path=usr/include/sys/scsi/targets
117 dir path=usr/include/sys/sysevent
118 dir path=usr/include/sys/tsol
119 dir path=usr/include/tsol
120 dir path=usr/include/uuid
121 $(sparc_ONLY)dir path=usr/include/v7
122 $(sparc_ONLY)dir path=usr/include/v7/sys
123 $(sparc_ONLY)dir path=usr/include/v9
124 $(sparc_ONLY)dir path=usr/include/v9/sys
125 dir path=usr/include/vm
126 dir path=usr/platform group=sys
127 $(sparc_ONLY)dir path=usr/platform/SUNW,A70 group=sys
```

```

128 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300 group=sys
129 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP2300/include
130 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010 group=sys
131 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-CP3010/include
132 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T12 group=sys
133 $(sparc_ONLY)dir path=usr/platform/SUNW,Netra-T4 group=sys
134 $(sparc_ONLY)dir path=usr/platform/SUNW,SPARC-Enterprise group=sys
135 $(sparc_ONLY)dir path=usr/platform/SUNW,Serverbladel1 group=sys
136 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-100 group=sys
137 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1000 group=sys
138 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-1500 group=sys
139 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Blade-2500 group=sys
140 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire group=sys
141 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-15000 group=sys
142 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-280R group=sys
143 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-480R group=sys
144 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-880 group=sys
145 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V215 group=sys
146 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V240 group=sys
147 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V250 group=sys
148 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V440 group=sys
149 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V445 group=sys
150 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V490 group=sys
151 $(sparc_ONLY)dir path=usr/platform/SUNW,Sun-Fire-V890 group=sys
152 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-2 group=sys
153 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-250 group=sys
154 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-4 group=sys
155 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise group=sys
156 $(sparc_ONLY)dir path=usr/platform/SUNW,Ultra-Enterprise-10000 group=sys
157 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40 group=sys
158 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60 group=sys
159 $(sparc_ONLY)dir path=usr/platform/SUNW,UltraSPARC-IIi-Netract group=sys
160 $(i386_ONLY)dir path=usr/platform/i86pc group=sys
161 $(i386_ONLY)dir path=usr/platform/i86pc/include
162 $(i386_ONLY)dir path=usr/platform/i86pc/include/sys
163 $(i386_ONLY)dir path=usr/platform/i86pc/include/vm
164 $(i386_ONLY)dir path=usr/platform/i86xpv group=sys
165 $(i386_ONLY)dir path=usr/platform/i86xpv/include
166 $(i386_ONLY)dir path=usr/platform/i86xpv/include/sys
167 $(i386_ONLY)dir path=usr/platform/i86xpv/include/vm
168 $(sparc_ONLY)dir path=usr/platform/sun4u group=sys
169 $(sparc_ONLY)dir path=usr/platform/sun4u/include
170 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys
171 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c
172 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/clients
173 $(sparc_ONLY)dir path=usr/platform/sun4u/include/sys/i2c/misc
174 $(sparc_ONLY)dir path=usr/platform/sun4u/include/vm
175 $(sparc_ONLY)dir path=usr/platform/sun4v group=sys
176 $(sparc_ONLY)dir path=usr/platform/sun4v/include
177 $(sparc_ONLY)dir path=usr/platform/sun4v/include/sys
178 $(sparc_ONLY)dir path=usr/platform/sun4v/include/vm
179 dir path=usr/share
180 dir path=usr/share/man
181 dir path=usr/share/man/man3head
182 dir path=usr/share/man/man4
183 dir path=usr/share/man/man5
184 dir path=usr/share/man/man7i
185 dir path=usr/share/src group=sys
186 dir path=usr/share/src/uts
187 $(i386_ONLY)dir path=usr/share/src/uts/i86pc
188 $(i386_ONLY)dir path=usr/share/src/uts/i86xpv
189 $(sparc_ONLY)dir path=usr/share/src/uts/sun4u
190 $(sparc_ONLY)dir path=usr/share/src/uts/sun4v
191 dir path=usr/xpg4
192 dir path=usr/xpg4/include
193 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/kdi_regs.h

```

```

194 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privmregs.h
195 $(i386_ONLY)file path=usr/include/$(ARCH64)/sys/privregs.h
196 file path=usr/include/ads/dsgetdc.h
197 file path=usr/include/aio.h
198 file path=usr/include/alloca.h
199 file path=usr/include/apprtrace.h
200 file path=usr/include/apprtrace_impl.h
201 file path=usr/include/ar.h
202 file path=usr/include/archives.h
203 file path=usr/include/arpa/ftp.h
204 file path=usr/include/arpa/inet.h
205 file path=usr/include/arpa/nameser.h
206 file path=usr/include/arpa/nameser_compat.h
207 file path=usr/include/arpa/telnet.h
208 file path=usr/include/arpa/tftp.h
209 $(i386_ONLY)file path=usr/include/asm/atomic.h
210 $(i386_ONLY)file path=usr/include/asm/bitmap.h
211 $(i386_ONLY)file path=usr/include/asm/byteorder.h
212 $(i386_ONLY)file path=usr/include/asm/clock.h
213 $(i386_ONLY)file path=usr/include/asm/cpu.h
214 $(i386_ONLY)file path=usr/include/asm/cpuvar.h
215 $(sparc_ONLY)file path=usr/include/asm/flush.h
216 $(i386_ONLY)file path=usr/include/asm/htable.h
217 $(i386_ONLY)file path=usr/include/asm/mmu.h
218 file path=usr/include/asm/sunddi.h
219 file path=usr/include/asm/thread.h
220 file path=usr/include/assert.h
221 file path=usr/include/ast/align.h
222 file path=usr/include/ast/ast.h
223 file path=usr/include/ast/ast_botch.h
224 file path=usr/include/ast/ast_ccode.h
225 file path=usr/include/ast/ast_common.h
226 file path=usr/include/ast/ast_dir.h
227 file path=usr/include/ast/ast_dirent.h
228 file path=usr/include/ast/ast_fcntl.h
229 file path=usr/include/ast/ast_float.h
230 file path=usr/include/ast/ast_fs.h
231 file path=usr/include/ast/ast_getopt.h
232 file path=usr/include/ast/ast_iconv.h
233 file path=usr/include/ast/ast_lib.h
234 file path=usr/include/ast/ast_limits.h
235 file path=usr/include/ast/ast_map.h
236 file path=usr/include/ast/ast_mmap.h
237 file path=usr/include/ast/ast_mode.h
238 file path=usr/include/ast/ast_namval.h
239 file path=usr/include/ast/ast_ndbm.h
240 file path=usr/include/ast/ast_nl_types.h
241 file path=usr/include/ast/ast_param.h
242 file path=usr/include/ast/ast_standards.h
243 file path=usr/include/ast/ast_std.h
244 file path=usr/include/ast/ast_stdio.h
245 file path=usr/include/ast/ast_sys.h
246 file path=usr/include/ast/ast_time.h
247 file path=usr/include/ast/ast_tty.h
248 file path=usr/include/ast/ast_version.h
249 file path=usr/include/ast/ast_vfork.h
250 file path=usr/include/ast/ast_wait.h
251 file path=usr/include/ast/ast_wchar.h
252 file path=usr/include/ast/ast_windows.h
253 file path=usr/include/ast/bytesex.h
254 file path=usr/include/ast/ccode.h
255 file path=usr/include/ast/cdt.h
256 file path=usr/include/ast/cmd.h
257 file path=usr/include/ast/cmdext.h
258 file path=usr/include/ast/debug.h
259 file path=usr/include/ast/dirent.h

```

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

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

```

392 $(i386_ONLY)file path=usr/include/ia32/sys/kdi_regs.h
393 $(i386_ONLY)file path=usr/include/ia32/sys/machtypes.h
394 $(i386_ONLY)file path=usr/include/ia32/sys/privmregs.h
395 $(i386_ONLY)file path=usr/include/ia32/sys/privregs.h
396 $(i386_ONLY)file path=usr/include/ia32/sys/psw.h
397 $(i386_ONLY)file path=usr/include/ia32/sys/pte.h
398 $(i386_ONLY)file path=usr/include/ia32/sys/reg.h
399 $(i386_ONLY)file path=usr/include/ia32/sys/stack.h
400 $(i386_ONLY)file path=usr/include/ia32/sys/trap.h
401 $(i386_ONLY)file path=usr/include/ia32/sys/traptrace.h
402 file path=usr/include/iconv.h
403 file path=usr/include/idmap.h
404 file path=usr/include/ieeefp.h
405 file path=usr/include/ifaddrs.h
406 file path=usr/include/inet/arp.h
407 file path=usr/include/inet/common.h
408 file path=usr/include/inet/ip.h
409 file path=usr/include/inet/ip6.h
410 file path=usr/include/inet/ip6_asp.h
411 file path=usr/include/inet/ip_arp.h
412 file path=usr/include/inet/ip_ftable.h
413 file path=usr/include/inet/ip_if.h
414 file path=usr/include/inet/ip_ire.h
415 file path=usr/include/inet/ip_multi.h
416 file path=usr/include/inet/ip_netinfo.h
417 file path=usr/include/inet/ip_rts.h
418 file path=usr/include/inet/ip_stack.h
419 file path=usr/include/inet/ipclassifier.h
420 file path=usr/include/inet/ipdrop.h
421 file path=usr/include/inet/ipnet.h
422 file path=usr/include/inet/ipp_common.h
423 file path=usr/include/inet/kssl/ksslapl.h
424 file path=usr/include/inet/led.h
425 file path=usr/include/inet/mi.h
426 file path=usr/include/inet/mib2.h
427 file path=usr/include/inet/nd.h
428 file path=usr/include/inet/optcom.h
429 file path=usr/include/inet/sctp_itf.h
430 file path=usr/include/inet/snmpcom.h
431 file path=usr/include/inet/tcp.h
432 file path=usr/include/inet/tcp_sack.h
433 file path=usr/include/inet/tcp_stack.h
434 file path=usr/include/inet/tcp_stats.h
435 file path=usr/include/inet/tunables.h
436 file path=usr/include/inet/wifi_ioctl.h
437 file path=usr/include/inttypes.h
438 file path=usr/include/ipmp.h
439 file path=usr/include/ipmp_admin.h
440 file path=usr/include/ipmp_mpathd.h
441 file path=usr/include/ipmp_query.h
442 file path=usr/include/ipp/ipgpc/ipgpc.h
443 file path=usr/include/ipp/ipp.h
444 file path=usr/include/ipp/ipp_config.h
445 file path=usr/include/ipp/ipp_impl.h
446 file path=usr/include/ipp/ippctl.h
447 file path=usr/include/iso/ctype_iso.h
448 file path=usr/include/iso/limits_iso.h
449 file path=usr/include/iso/locale_iso.h
450 file path=usr/include/iso/setjmp_iso.h
451 file path=usr/include/iso/signal_iso.h
452 file path=usr/include/iso/stdarg_c99.h
453 file path=usr/include/iso/stdarg_iso.h
454 file path=usr/include/iso/stddef_iso.h
455 file path=usr/include/iso/stdio_c99.h
456 file path=usr/include/iso/stdio_iso.h
457 file path=usr/include/iso/stdlib_c11.h

```

```

458 file path=usr/include/iso/stdlib_c99.h
459 file path=usr/include/iso/stdlib_iso.h
460 file path=usr/include/iso/string_iso.h
461 file path=usr/include/iso/time_iso.h
462 file path=usr/include/iso/wchar_c99.h
463 file path=usr/include/iso/wchar_iso.h
464 file path=usr/include/iso/wctype_iso.h
465 file path=usr/include/iso646.h
466 file path=usr/include/kerberos5/com_err.h
467 file path=usr/include/kerberos5/krb5.h
468 file path=usr/include/kerberos5/locate_plugin.h
469 file path=usr/include/kerberos5/mit-sipb-copyright.h
470 file path=usr/include/kerberos5/mit_copyright.h
471 file path=usr/include/Klpd.h
472 file path=usr/include/kmfapi.h
473 file path=usr/include/kmftypes.h
474 file path=usr/include/kstat.h
475 file path=usr/include/kvm.h
476 file path=usr/include/langinfo.h
477 file path=usr/include/lastlog.h
478 file path=usr/include/lber.h
479 file path=usr/include/ldap.h
480 file path=usr/include/libcontract.h
481 file path=usr/include/libctf.h
482 file path=usr/include/libdevice.h
483 file path=usr/include/libdevinfo.h
484 file path=usr/include/libdldm.h
485 file path=usr/include/libdlbridge.h
486 file path=usr/include/libdlib.h
487 file path=usr/include/libdlink.h
488 file path=usr/include/libdmpi.h
489 file path=usr/include/libdvlan.h
490 file path=usr/include/libelf.h
491 $(i386_ONLY)file path=usr/include/libfdisk.h
492 file path=usr/include/libfstyp.h
493 file path=usr/include/libfstyp_module.h
494 file path=usr/include/libgen.h
495 file path=usr/include/libgrubmgmt.h
496 file path=usr/include/libintl.h
497 file path=usr/include/libipmi.h
498 file path=usr/include/libipp.h
499 file path=usr/include/libnvpair.h
500 file path=usr/include/libnwm.h
501 file path=usr/include/libpolkit/libpolkit.h
502 file path=usr/include/libproc.h
503 file path=usr/include/librcm.h
504 file path=usr/include/libscf.h
505 file path=usr/include/libscf_priv.h
506 file path=usr/include/libshare.h
507 file path=usr/include/libsvm.h
508 file path=usr/include/libsysevent.h
509 file path=usr/include/libsysevent_impl.h
510 file path=usr/include/libtsnet.h
511 $(sparc_ONLY)file path=usr/include/libv12n.h
512 file path=usr/include/libw.h
513 file path=usr/include/libzfs.h
514 file path=usr/include/libzfs_core.h
515 file path=usr/include/libzoneinfo.h
516 file path=usr/include/limits.h
517 file path=usr/include/linenum.h
518 file path=usr/include/link.h
519 file path=usr/include/listen.h
520 file path=usr/include/locale.h
521 file path=usr/include/macros.h
522 file path=usr/include/maillock.h
523 file path=usr/include/malloc.h

```

524 file path=usr/include/md4.h
525 file path=usr/include/md5.h
526 file path=usr/include/mdiox.h
527 file path=usr/include/mdmn_changelog.h
528 file path=usr/include/memory.h
529 file path=usr/include/menu.h
530 file path=usr/include/meta.h
531 file path=usr/include/meta_basic.h
532 file path=usr/include/meta_runtime.h
533 file path=usr/include/metacl.h
534 file path=usr/include/metad.h
535 file path=usr/include/metadyn.h
536 file path=usr/include/metamed.h
537 file path=usr/include/metamhd.h
538 file path=usr/include/mhdx.h
539 file path=usr/include/mon.h
540 file path=usr/include/monetary.h
541 file path=usr/include/mp.h
542 file path=usr/include/mqueue.h
543 file path=usr/include/mtmalloc.h
544 file path=usr/include/nan.h
545 file path=usr/include/ndbm.h
546 file path=usr/include/ndpd.h
547 file path=usr/include/net/af.h
548 file path=usr/include/net/bridge.h
549 file path=usr/include/net/if.h
550 file path=usr/include/net/if_arp.h
551 file path=usr/include/net/if_dl.h
552 file path=usr/include/net/if_types.h
553 file path=usr/include/net/pfkeyv2.h
554 file path=usr/include/net/pfpolicy.h
555 file path=usr/include/net/ppp-comp.h
556 file path=usr/include/net/ppp-defs.h
557 file path=usr/include/net/pppio.h
558 file path=usr/include/net/radix.h
559 file path=usr/include/net/route.h
560 file path=usr/include/net/trill.h
561 file path=usr/include/net/vjcompress.h
562 file path=usr/include/netconfig.h
563 file path=usr/include/netdb.h
564 file path=usr/include/netdir.h
565 file path=usr/include/netinet/arp.h
566 file path=usr/include/netinet/dhcp.h
567 file path=usr/include/netinet/dhcp6.h
568 file path=usr/include/netinet/icmp6.h
569 file path=usr/include/netinet/icmp_var.h
570 file path=usr/include/netinet/if_ether.h
571 file path=usr/include/netinet/igmp.h
572 file path=usr/include/netinet/igmp_var.h
573 file path=usr/include/netinet/in.h
574 file path=usr/include/netinet/in_pcb.h
575 file path=usr/include/netinet/in_system.h
576 file path=usr/include/netinet/in_var.h
577 file path=usr/include/netinet/ip.h
578 file path=usr/include/netinet/ip6.h
579 file path=usr/include/netinet/ip_icmp.h
580 file path=usr/include/netinet/ip_mroute.h
581 file path=usr/include/netinet/ip_var.h
582 file path=usr/include/netinet/pim.h
583 file path=usr/include/netinet/sctp.h
584 file path=usr/include/netinet/tcp.h
585 file path=usr/include/netinet/tcp_debug.h
586 file path=usr/include/netinet/tcp_fsm.h
587 file path=usr/include/netinet/tcp_seq.h
588 file path=usr/include/netinet/tcp_timer.h
589 file path=usr/include/netinet/tcp_var.h

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

```

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

```

```

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

```

```

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

```

```

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

```

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

```
986 file path=usr/include/sys/fc4/fcal_transport.h
987 file path=usr/include/sys/fc4/fcio.h
988 file path=usr/include/sys/fc4/fcp.h
989 file path=usr/include/sys/fc4/linkapp.h
990 file path=usr/include/sys/fcntl.h
991 file path=usr/include/sys/fdbuffer.h
992 file path=usr/include/sys/fdio.h
993 $(sparc_ONLY)file path=usr/include/sys/fdreg.h
994 $(sparc_ONLY)file path=usr/include/sys/fdvar.h
995 file path=usr/include/sys/feature_tests.h
996 file path=usr/include/sys/fem.h
997 file path=usr/include/sys/file.h
998 file path=usr/include/sys/filio.h
999 file path=usr/include/sys/flock.h
1000 file path=usr/include/sys/flock_impl.h
1001 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/SPARC64-VI.h
1002 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-II.h
1003 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-III.h
1004 $(sparc_ONLY)file path=usr/include/sys/fm/cpu/UltraSPARC-T1.h
1005 file path=usr/include/sys/fm/fs/zfs.h
1006 file path=usr/include/sys/fm/io/ddi.h
1007 file path=usr/include/sys/fm/io/disk.h
1008 file path=usr/include/sys/fm/io/opl_mc_fm.h
1009 file path=usr/include/sys/fm/io/pci.h
1010 file path=usr/include/sys/fm/io/scsi.h
1011 file path=usr/include/sys/fm/io/sun4upci.h
1012 file path=usr/include/sys/fm/protocol.h
1013 file path=usr/include/sys/fm/util.h
1014 file path=usr/include/sys/fork.h
1015 $(i386_ONLY)file path=usr/include/sys/fp.h
1016 $(sparc_ONLY)file path=usr/include/sys/fpu/fpu_simulator.h
1017 $(sparc_ONLY)file path=usr/include/sys/fpu/fpysystem.h
1018 $(sparc_ONLY)file path=usr/include/sys/fpu/globals.h
1019 $(sparc_ONLY)file path=usr/include/sys/fpu/ieee.h
1020 file path=usr/include/sys/frame.h
1021 file path=usr/include/sys/fs/autofs.h
1022 file path=usr/include/sys/fs/decomp.h
1023 file path=usr/include/sys/fs/dv_node.h
1024 file path=usr/include/sys/fs/fifonode.h
1025 file path=usr/include/sys/fs/hsfs_isospec.h
1026 file path=usr/include/sys/fs/hsfs_node.h
1027 file path=usr/include/sys/fs/hsfs_rrip.h
1028 file path=usr/include/sys/fs/hsfs_spec.h
1029 file path=usr/include/sys/fs/hsfs_susp.h
1030 file path=usr/include/sys/fs/lofs_info.h
1031 file path=usr/include/sys/fs/lofs_node.h
1032 file path=usr/include/sys/fs/mntdata.h
1033 file path=usr/include/sys/fs/namenode.h
1034 file path=usr/include/sys/fs/pc_dir.h
1035 file path=usr/include/sys/fs/pc_fs.h
1036 file path=usr/include/sys/fs/pc_label.h
1037 file path=usr/include/sys/fs/pc_node.h
1038 file path=usr/include/sys/fs/pxfs_ki.h
1039 file path=usr/include/sys/fs/sdev_impl.h
1040 file path=usr/include/sys/fs/snnode.h
1041 file path=usr/include/sys/fs/swapnode.h
1042 file path=usr/include/sys/fs/tmp.h
1043 file path=usr/include/sys/fs/tmpnode.h
1044 file path=usr/include/sys/fs/udf_inode.h
1045 file path=usr/include/sys/fs/udf_volume.h
1046 file path=usr/include/sys/fs/ufs_acl.h
1047 file path=usr/include/sys/fs/ufs_bio.h
1048 file path=usr/include/sys/fs/ufs_filio.h
1049 file path=usr/include/sys/fs/ufs_fs.h
1050 file path=usr/include/sys/fs/ufs_fsdirent.h
1051 file path=usr/include/sys/fs/ufs_inode.h
```

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

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1580 file path=usr/include/sys/utsname.h
1581 file path=usr/include/sys/utssys.h
1582 file path=usr/include/sys/uuid.h
1583 file path=usr/include/sys/va_impl.h
1584 file path=usr/include/sys/va_list.h
1585 file path=usr/include/sys/var.h
1586 file path=usr/include/sys/varargs.h
1587 file path=usr/include/sys/vfs.h
1588 file path=usr/include/sys/vfs_opreg.h
1589 file path=usr/include/sys/vfstab.h
1590 file path=usr/include/sys/videodev2.h
1591 file path=usr/include/sys/visual_io.h
1592 file path=usr/include/sys/vm.h
1593 file path=usr/include/sys/vm_usage.h
1594 file path=usr/include/sys/vmem.h
1595 file path=usr/include/sys/vmem_impl.h
1596 file path=usr/include/sys/vmem_impl_user.h
1597 file path=usr/include/sys/vmparam.h
1598 file path=usr/include/sys/vmsystem.h
1599 file path=usr/include/sys/vnode.h
1600 file path=usr/include/sys/vt.h
1601 file path=usr/include/sys/vtdaemon.h
1602 file path=usr/include/sys/vtoc.h
1603 file path=usr/include/sys/vtrace.h
1604 file path=usr/include/sys/vuid_event.h
1605 file path=usr/include/sys/vuid_queue.h
1606 file path=usr/include/sys/vuid_state.h
1607 file path=usr/include/sys/vuid_store.h
1608 file path=usr/include/sys/vuid_wheel.h
1609 file path=usr/include/sys/wait.h
1610 file path=usr/include/sys/waitq.h
1611 file path=usr/include/sys/watchpoint.h
1612 $(i386_ONLY)file path=usr/include/sys/x86_archext.h
1613 $(i386_ONLY)file path=usr/include/sys/xen_errno.h
1614 file path=usr/include/sys/xti_inet.h
1615 file path=usr/include/sys/xti_osi.h
1616 file path=usr/include/sys/xti_xtiopt.h
1617 file path=usr/include/sys/zcons.h
1618 file path=usr/include/sys/zmod.h
1619 file path=usr/include/sys/zone.h
1620 $(sparc_ONLY)file path=usr/include/sys/zsdev.h
1621 file path=usr/include/syssexits.h
1622 file path=usr/include/syslog.h
1623 file path=usr/include/tar.h
1624 file path=usr/include/tcpd.h
1625 file path=usr/include/term.h
1626 file path=usr/include/termcap.h
1627 file path=usr/include/termio.h
1628 file path=usr/include/termios.h
1629 file path=usr/include/thread.h
1630 file path=usr/include/thread_db.h
1631 file path=usr/include/threads.h
1632 file path=usr/include/time.h
1633 file path=usr/include/tiuser.h
1634 file path=usr/include/tsol/label.h
1635 file path=usr/include/tzfile.h
1636 file path=usr/include/ucontext.h
1637 file path=usr/include/ucrd.h
1638 file path=usr/include/uid_stp.h
1639 file path=usr/include/ulimit.h
1640 file path=usr/include/umem.h
1641 file path=usr/include/umem_impl.h
1642 file path=usr/include/unctrl.h
1643 file path=usr/include/unistd.h
1644 file path=usr/include/user_attr.h
1645 file path=usr/include/userdefs.h

```

```

1646 file path=usr/include/ustat.h
1647 file path=usr/include/utility.h
1648 file path=usr/include/utime.h
1649 file path=usr/include/utmp.h
1650 file path=usr/include/utmpx.h
1651 file path=usr/include/uuid/uuid.h
1652 $(sparc_ONLY)file path=usr/include/v7/sys/machpcb.h
1653 $(sparc_ONLY)file path=usr/include/v7/sys/machtrap.h
1654 $(sparc_ONLY)file path=usr/include/v7/sys/mutex_impl.h
1655 $(sparc_ONLY)file path=usr/include/v7/sys/privregs.h
1656 $(sparc_ONLY)file path=usr/include/v7/sys/prom_isa.h
1657 $(sparc_ONLY)file path=usr/include/v7/sys/psr.h
1658 $(sparc_ONLY)file path=usr/include/v7/sys/traptrace.h
1659 $(sparc_ONLY)file path=usr/include/v9/sys/asi.h
1660 $(sparc_ONLY)file path=usr/include/v9/sys/machpcb.h
1661 $(sparc_ONLY)file path=usr/include/v9/sys/machtrap.h
1662 $(sparc_ONLY)file path=usr/include/v9/sys/membar.h
1663 $(sparc_ONLY)file path=usr/include/v9/sys/mutex_impl.h
1664 $(sparc_ONLY)file path=usr/include/v9/sys/privregs.h
1665 $(sparc_ONLY)file path=usr/include/v9/sys/prom_isa.h
1666 $(sparc_ONLY)file path=usr/include/v9/sys/psr_compat.h
1667 $(sparc_ONLY)file path=usr/include/v9/sys/vis_simulator.h
1668 file path=usr/include/valtools.h
1669 file path=usr/include/values.h
1670 file path=usr/include/varargs.h
1671 file path=usr/include/vm/anon.h
1672 file path=usr/include/vm/as.h
1673 file path=usr/include/vm/faultcode.h
1674 file path=usr/include/vm/hat.h
1675 file path=usr/include/vm/kpm.h
1676 file path=usr/include/vm/page.h
1677 file path=usr/include/vm/pvn.h
1678 file path=usr/include/vm/rm.h
1679 file path=usr/include/vm/seg.h
1680 file path=usr/include/vm/seg_dev.h
1681 file path=usr/include/vm/seg_enum.h
1682 file path=usr/include/vm/seg_kmem.h
1683 file path=usr/include/vm/seg_kp.h
1684 file path=usr/include/vm/seg_kpm.h
1685 file path=usr/include/vm/seg_map.h
1686 file path=usr/include/vm/seg_spt.h
1687 file path=usr/include/vm/seg_vn.h
1688 file path=usr/include/vm/vpage.h
1689 file path=usr/include/vm/vpm.h
1690 file path=usr/include/volmgt.h
1691 file path=usr/include/wait.h
1692 file path=usr/include/wchar.h
1693 file path=usr/include/wchar_impl.h
1694 file path=usr/include/wctype.h
1695 file path=usr/include/widec.h
1696 file path=usr/include/wordep.h
1697 file path=usr/include/xlocale.h
1698 file path=usr/include/xti.h
1699 file path=usr/include/xti_inet.h
1700 file path=usr/include/zone.h
1701 file path=usr/include/zonestat.h
1702 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/acpidev.h
1703 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/amd_iommu.h
1704 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/asm_misc.h
1705 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/clock.h
1706 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/cram.h
1707 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/ddi_subrdefs.h
1708 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/debug_info.h
1709 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/fastboot.h
1710 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/mach_mmu.h
1711 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machclock.h

```

```

1712 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machcpuvar.h
1713 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machparam.h
1714 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machprivregs.h
1715 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machsystem.h
1716 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/machthread.h
1717 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/memmode.h
1718 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/pc_mmu.h
1719 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm.h
1720 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_defs.h
1721 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_modctl.h
1722 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/psm_types.h
1723 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/rm_platter.h
1724 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/sbd_ioctl.h
1725 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/smp_impldefs.h
1726 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/vm_machparam.h
1727 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/x_call.h
1728 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/xc_levels.h
1729 $(i386_ONLY)file path=usr/platform/i86pc/include/sys/xsvc.h
1730 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_i86.h
1731 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hat_pte.h
1732 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/hment.h
1733 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/htable.h
1734 $(i386_ONLY)file path=usr/platform/i86pc/include/vm/kboot_mmu.h
1735 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/balloon.h
1736 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/machprivregs.h
1737 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xen_mmu.h
1738 $(i386_ONLY)file path=usr/platform/i86xpv/include/sys/xpv_impl.h
1739 $(i386_ONLY)file path=usr/platform/i86xpv/include/vm/seg_mf.h
1740 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ac.h
1741 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/async.h
1742 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cheetahregs.h
1743 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cherrystone.h
1744 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/clock.h
1745 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cmp.h
1746 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpc_ultra.h
1747 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpr_impl.h
1748 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_impl.h
1749 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cpu_sgnblk_defs.h
1750 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/cvc.h
1751 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/daktari.h
1752 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ddi_subrdefs.h
1753 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/dvma.h
1754 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ecc_kstat.h
1755 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/EEPROM.h
1756 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl.h
1757 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_gen.h
1758 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue250.h
1759 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/envctrl_ue450.h
1760 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/environ.h
1761 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/errclassify.h
1762 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/fhc.h
1763 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/gpio_87317.h
1764 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/hpc3130_events.h
1765 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/hpc3130.h
1766 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/i2c_client.h
1767 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/lm75.h
1768 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/max1617.h
1769 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/pcf8591.h
1770 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/clients/ssc050.h
1771 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/i2c/misc/i2c_svc.h
1772 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/idprom.h
1773 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intr.h
1774 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/intreg.h
1775 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iocache.h
1776 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/iommu.h
1777 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/ivintr.h

```

```

1778 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/lom_io.h
1779 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machasi.h
1780 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machclock.h
1781 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machcpuvar.h
1782 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machparam.h
1783 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machsystem.h
1784 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/machthread.h
1785 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mem_cache.h
1786 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memlist_plat.h
1787 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/memmode.h
1788 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/mmu.h
1789 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/nexusdebug.h
1790 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_hwdesc.h
1791 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/opl_module.h
1792 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_debug.h
1793 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/prom_plat.h
1794 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/pte.h
1795 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sbd_ioctl.h
1796 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scb.h
1797 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/scsb_led.h
1798 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/simstat.h
1799 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/splitregs.h
1800 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sram.h
1801 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/starfire.h
1802 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sun4asi.h
1803 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysctrl.h
1804 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysioerr.h
1805 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/sysiosbus.h
1806 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/tod.h
1807 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/todmostek.h
1808 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/trapstat.h
1809 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/traptrace.h
1810 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vis.h
1811 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/vm_machparam.h
1812 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/x_call.h
1813 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/xc_impl.h
1814 $(sparc_ONLY)file path=usr/platform/sun4u/include/sys/zsmach.h
1815 $(sparc_ONLY)file path=usr/platform/sun4u/include/vm/hat_smmu.h
1816 $(sparc_ONLY)file path=usr/platform/sun4u/include/vm/mach_smmu.h
1817 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/clock.h
1818 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cmp.h
1819 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpc_ultra.h
1820 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/cpu_sgnblk_defs.h
1821 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ddi_subrdefs.h
1822 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_pri.h
1823 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ds_smp.h
1824 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/dvma.h
1825 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/EEPROM.h
1826 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/fcode.h
1827 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hsvc.h
1828 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/hypervisor_api.h
1829 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/idprom.h
1830 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intr.h
1831 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/intreg.h
1832 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ivintr.h
1833 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machasi.h
1834 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machclock.h
1835 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machcpuvar.h
1836 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machintreg.h
1837 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machparam.h
1838 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machsystem.h
1839 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/machthread.h
1840 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memlist_plat.h
1841 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/memmode.h
1842 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/mmu.h
1843 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/nexusdebug.h

```

```

1844 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagaraasi.h
1845 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/niagararegs.h
1846 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/ntwdt.h
1847 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pri.h
1848 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_debug.h
1849 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/prom_plat.h
1850 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/pte.h
1851 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/qcn.h
1852 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/scb.h
1853 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/soft_state.h
1854 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/sun4asi.h
1855 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/tod.h
1856 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/trapstat.h
1857 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/traptrace.h
1858 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vis.h
1859 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/vm_machparam.h
1860 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/x_call.h
1861 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/xc_impl.h
1862 $(sparc_ONLY)file path=usr/platform/sun4v/include/sys/zsmach.h
1863 $(sparc_ONLY)file path=usr/platform/sun4v/include/vm/hat_sfmmu.h
1864 $(sparc_ONLY)file path=usr/platform/sun4v/include/vm/mach_sfmmu.h
1865 file path=usr/share/man/man3head/acct.h.3head
1866 file path=usr/share/man/man3head/aio.h.3head
1867 file path=usr/share/man/man3head/ar.h.3head
1868 file path=usr/share/man/man3head/archives.h.3head
1869 file path=usr/share/man/man3head/assert.h.3head
1870 file path=usr/share/man/man3head/complex.h.3head
1871 file path=usr/share/man/man3head/cpio.h.3head
1872 file path=usr/share/man/man3head/dirent.h.3head
1873 file path=usr/share/man/man3head/ndian.h.3head
1874 file path=usr/share/man/man3head/errno.h.3head
1875 file path=usr/share/man/man3head/fcntl.h.3head
1876 file path=usr/share/man/man3head/fenv.h.3head
1877 file path=usr/share/man/man3head/float.h.3head
1878 file path=usr/share/man/man3head/floatingpoint.h.3head
1879 file path=usr/share/man/man3head/fmtmsg.h.3head
1880 file path=usr/share/man/man3head/fnmatch.h.3head
1881 file path=usr/share/man/man3head/ftw.h.3head
1882 file path=usr/share/man/man3head/glob.h.3head
1883 file path=usr/share/man/man3head/grp.h.3head
1884 file path=usr/share/man/man3head/iconv.h.3head
1885 file path=usr/share/man/man3head/if.h.3head
1886 file path=usr/share/man/man3head/in.h.3head
1887 file path=usr/share/man/man3head/inet.h.3head
1888 file path=usr/share/man/man3head/inttypes.h.3head
1889 file path=usr/share/man/man3head/ipc.h.3head
1890 file path=usr/share/man/man3head/iso646.h.3head
1891 file path=usr/share/man/man3head/langinfo.h.3head
1892 file path=usr/share/man/man3head/libgen.h.3head
1893 file path=usr/share/man/man3head/libintl.h.3head
1894 file path=usr/share/man/man3head/limits.h.3head
1895 file path=usr/share/man/man3head/locale.h.3head
1896 file path=usr/share/man/man3head/math.h.3head
1897 file path=usr/share/man/man3head/mman.h.3head
1898 file path=usr/share/man/man3head/monetary.h.3head
1899 file path=usr/share/man/man3head/mqueue.h.3head
1900 file path=usr/share/man/man3head/msg.h.3head
1901 file path=usr/share/man/man3head/ndbm.h.3head
1902 file path=usr/share/man/man3head/netdb.h.3head
1903 file path=usr/share/man/man3head/nl_types.h.3head
1904 file path=usr/share/man/man3head/poll.h.3head
1905 file path=usr/share/man/man3head/pthread.h.3head
1906 file path=usr/share/man/man3head/pwd.h.3head
1907 file path=usr/share/man/man3head/regex.h.3head
1908 file path=usr/share/man/man3head/resource.h.3head
1909 file path=usr/share/man/man3head/sched.h.3head

```

```

1910 file path=usr/share/man/man3head/search.h.3head
1911 file path=usr/share/man/man3head/select.h.3head
1912 file path=usr/share/man/man3head/sem.h.3head
1913 file path=usr/share/man/man3head/semaphore.h.3head
1914 file path=usr/share/man/man3head/setjmp.h.3head
1915 file path=usr/share/man/man3head/shm.h.3head
1916 file path=usr/share/man/man3head/signinfo.h.3head
1917 file path=usr/share/man/man3head/signal.h.3head
1918 file path=usr/share/man/man3head/socket.h.3head
1919 file path=usr/share/man/man3head/spawn.h.3head
1920 file path=usr/share/man/man3head/stat.h.3head
1921 file path=usr/share/man/man3head/statvfs.h.3head
1922 file path=usr/share/man/man3head/stdbool.h.3head
1923 file path=usr/share/man/man3head/stddef.h.3head
1924 file path=usr/share/man/man3head/stdint.h.3head
1925 file path=usr/share/man/man3head/stdio.h.3head
1926 file path=usr/share/man/man3head/stdlib.h.3head
1927 file path=usr/share/man/man3head/string.h.3head
1928 file path=usr/share/man/man3head/strings.h.3head
1929 file path=usr/share/man/man3head/stropts.h.3head
1930 file path=usr/share/man/man3head/syslog.h.3head
1931 file path=usr/share/man/man3head/tar.h.3head
1932 file path=usr/share/man/man3head/tcp.h.3head
1933 file path=usr/share/man/man3head/termios.h.3head
1934 file path=usr/share/man/man3head/tgmach.h.3head
1935 file path=usr/share/man/man3head/time.h.3head
1936 file path=usr/share/man/man3head/timeb.h.3head
1937 file path=usr/share/man/man3head/times.h.3head
1938 file path=usr/share/man/man3head/types.h.3head
1939 file path=usr/share/man/man3head/types32.h.3head
1940 file path=usr/share/man/man3head/ucontext.h.3head
1941 file path=usr/share/man/man3head/uio.h.3head
1942 file path=usr/share/man/man3head/ulimit.h.3head
1943 file path=usr/share/man/man3head/un.h.3head
1944 file path=usr/share/man/man3head/unistd.h.3head
1945 file path=usr/share/man/man3head/utime.h.3head
1946 file path=usr/share/man/man3head/utmpx.h.3head
1947 file path=usr/share/man/man3head/utname.h.3head
1948 file path=usr/share/man/man3head/values.h.3head
1949 file path=usr/share/man/man3head/wait.h.3head
1950 file path=usr/share/man/man3head/wchar.h.3head
1951 file path=usr/share/man/man3head/wctype.h.3head
1952 file path=usr/share/man/man3head/wordep.h.3head
1953 file path=usr/share/man/man3head/xlocale.h.3head
1954 file path=usr/share/man/man4/note.4
1955 file path=usr/share/man/man5/prof.5
1956 file path=usr/share/man/man7i/cdio.7i
1957 file path=usr/share/man/man7i/dkio.7i
1958 file path=usr/share/man/man7i/fbio.7i
1959 file path=usr/share/man/man7i/fdio.7i
1960 file path=usr/share/man/man7i/hdio.7i
1961 file path=usr/share/man/man7i/iec61883.7i
1962 file path=usr/share/man/man7i/mhd.7i
1963 file path=usr/share/man/man7i/mtio.7i
1964 file path=usr/share/man/man7i/prnio.7i
1965 file path=usr/share/man/man7i/quotactl.7i
1966 file path=usr/share/man/man7i/sesio.7i
1967 file path=usr/share/man/man7i/sockio.7i
1968 file path=usr/share/man/man7i/streamio.7i
1969 file path=usr/share/man/man7i/termio.7i
1970 file path=usr/share/man/man7i/termiox.7i
1971 file path=usr/share/man/man7i/uscio.7i
1972 file path=usr/share/man/man7i/visual_io.7i
1973 file path=usr/share/man/man7i/vt.7i
1974 file path=usr/xpg4/include/curses.h
1975 file path=usr/xpg4/include/term.h

```

```

1976 file path=usr/xpg4/include/unctrl.h
1977 legacy pkg=SUNWhea \
1978 desc="SunOS C/C++ header files for general development of software" \
1979 name="SunOS Header Files"
1980 license cr_Sun license=cr_Sun
1981 license lic_CDDL license=lic_CDDL
1982 license license_in_headers license=license_in_headers
1983 license usr/src/lib/pkcs11/include/THIRDPARTYLICENSE \
1984 license=usr/src/lib/pkcs11/include/THIRDPARTYLICENSE
1985 link path=usr/include/iso/assert_iso.h target=../assert.h
1986 link path=usr/include/iso/errno_iso.h target=../errno.h
1987 link path=usr/include/iso/float_iso.h target=../float.h
1988 link path=usr/include/iso/iso646_iso.h target=../iso646.h
1989 $(sparc_ONLY)link path=usr/platform/SUNW,A70/include target=../sun4u/include
1990 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T12/include \
1991 target=../sun4u/include
1992 $(sparc_ONLY)link path=usr/platform/SUNW,Netra-T4/include \
1993 target=../sun4u/include
1994 $(sparc_ONLY)link path=usr/platform/SUNW,SPARC-Enterprise/include \
1995 target=../sun4u/include
1996 $(sparc_ONLY)link path=usr/platform/SUNW,Serverbladel1/include \
1997 target=../sun4u/include
1998 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-100/include \
1999 target=../sun4u/include
2000 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1000/include \
2001 target=../sun4u/include
2002 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-1500/include \
2003 target=../sun4u/include
2004 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Blade-2500/include \
2005 target=../sun4u/include
2006 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-15000/include \
2007 target=../sun4u/include
2008 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-280R/include \
2009 target=../sun4u/include
2010 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-480R/include \
2011 target=../sun4u/include
2012 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-880/include \
2013 target=../sun4u/include
2014 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V215/include \
2015 target=../sun4u/include
2016 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V240/include \
2017 target=../sun4u/include
2018 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V250/include \
2019 target=../sun4u/include
2020 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V440/include \
2021 target=../sun4u/include
2022 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V445/include \
2023 target=../sun4u/include
2024 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V490/include \
2025 target=../sun4u/include
2026 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire-V890/include \
2027 target=../sun4u/include
2028 $(sparc_ONLY)link path=usr/platform/SUNW,Sun-Fire/include \
2029 target=../sun4u/include
2030 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-2/include \
2031 target=../sun4u/include
2032 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-250/include \
2033 target=../sun4u/include
2034 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-4/include \
2035 target=../sun4u/include
2036 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise-10000/include \
2037 target=../sun4u/include
2038 $(sparc_ONLY)link path=usr/platform/SUNW,Ultra-Enterprise/include \
2039 target=../sun4u/include
2040 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-40/include \
2041 target=../sun4u/include

```

```

2042 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIe-NetraCT-60/include \
2043 target=../sun4u/include
2044 $(sparc_ONLY)link path=usr/platform/SUNW,UltraSPARC-IIi-Netract/include \
2045 target=../sun4u/include
2046 link path=usr/share/man/man3head/acct.3head target=acct.h.3head
2047 link path=usr/share/man/man3head/aio.3head target=aio.h.3head
2048 link path=usr/share/man/man3head/ar.3head target=ar.h.3head
2049 link path=usr/share/man/man3head/archives.3head target=archives.h.3head
2050 link path=usr/share/man/man3head/assert.3head target=assert.h.3head
2051 link path=usr/share/man/man3head/complex.3head target=complex.h.3head
2052 link path=usr/share/man/man3head/cpio.3head target=cpio.h.3head
2053 link path=usr/share/man/man3head/dirent.3head target=dirent.h.3head
2054 link path=usr/share/man/man3head/errno.3head target=errno.h.3head
2055 link path=usr/share/man/man3head/fcntl.3head target=fcntl.h.3head
2056 link path=usr/share/man/man3head/fenv.3head target=fenv.h.3head
2057 link path=usr/share/man/man3head/float.3head target=float.h.3head
2058 link path=usr/share/man/man3head/floatingpoint.3head \
2059 target=floatingpoint.h.3head
2060 link path=usr/share/man/man3head/fmtmsg.3head target=fmtmsg.h.3head
2061 link path=usr/share/man/man3head/fnmatch.3head target=fnmatch.h.3head
2062 link path=usr/share/man/man3head/ftw.3head target=ftw.h.3head
2063 link path=usr/share/man/man3head/glob.3head target=glob.h.3head
2064 link path=usr/share/man/man3head/grp.3head target=grp.h.3head
2065 link path=usr/share/man/man3head/iconv.3head target=iconv.h.3head
2066 link path=usr/share/man/man3head/if.3head target=if.h.3head
2067 link path=usr/share/man/man3head/in.3head target=in.h.3head
2068 link path=usr/share/man/man3head/inet.3head target=inet.h.3head
2069 link path=usr/share/man/man3head/inttypes.3head target=inttypes.h.3head
2070 link path=usr/share/man/man3head/ipc.3head target=ipc.h.3head
2071 link path=usr/share/man/man3head/iso646.3head target=iso646.h.3head
2072 link path=usr/share/man/man3head/langinfo.3head target=langinfo.h.3head
2073 link path=usr/share/man/man3head/libgen.3head target=libgen.h.3head
2074 link path=usr/share/man/man3head/libintl.3head target=libintl.h.3head
2075 link path=usr/share/man/man3head/limits.3head target=limits.h.3head
2076 link path=usr/share/man/man3head/locale.3head target=locale.h.3head
2077 link path=usr/share/man/man3head/math.3head target=math.h.3head
2078 link path=usr/share/man/man3head/mman.3head target=mman.h.3head
2079 link path=usr/share/man/man3head/monetary.3head target=monetary.h.3head
2080 link path=usr/share/man/man3head/mqueue.3head target=mqueue.h.3head
2081 link path=usr/share/man/man3head/msg.3head target=msg.h.3head
2082 link path=usr/share/man/man3head/ndbm.3head target=ndbm.h.3head
2083 link path=usr/share/man/man3head/netdb.3head target=netdb.h.3head
2084 link path=usr/share/man/man3head/nl_types.3head target=nl_types.h.3head
2085 link path=usr/share/man/man3head/poll.3head target=poll.h.3head
2086 link path=usr/share/man/man3head/pthread.3head target=pthread.h.3head
2087 link path=usr/share/man/man3head/pwd.3head target=pwd.h.3head
2088 link path=usr/share/man/man3head/regex.3head target=regex.h.3head
2089 link path=usr/share/man/man3head/resource.3head target=resource.h.3head
2090 link path=usr/share/man/man3head/sched.3head target=sched.h.3head
2091 link path=usr/share/man/man3head/search.3head target=search.h.3head
2092 link path=usr/share/man/man3head/select.3head target=select.h.3head
2093 link path=usr/share/man/man3head/sem.3head target=sem.h.3head
2094 link path=usr/share/man/man3head/semaphore.3head target=semaphore.h.3head
2095 link path=usr/share/man/man3head/setjmp.3head target=setjmp.h.3head
2096 link path=usr/share/man/man3head/shm.3head target=shm.h.3head
2097 link path=usr/share/man/man3head/siginfo.3head target=siginfo.h.3head
2098 link path=usr/share/man/man3head/signal.3head target=signal.h.3head
2099 link path=usr/share/man/man3head/socket.3head target=socket.h.3head
2100 link path=usr/share/man/man3head/spawn.3head target=spawn.h.3head
2101 link path=usr/share/man/man3head/stat.3head target=stat.h.3head
2102 link path=usr/share/man/man3head/statvfs.3head target=statvfs.h.3head
2103 link path=usr/share/man/man3head/stdbool.3head target=stdbool.h.3head
2104 link path=usr/share/man/man3head/stddef.3head target=stddef.h.3head
2105 link path=usr/share/man/man3head/stdint.3head target=stdint.h.3head
2106 link path=usr/share/man/man3head/stdio.3head target=stdio.h.3head
2107 link path=usr/share/man/man3head/stdlib.3head target=stdlib.h.3head

```

```
2108 link path=usr/share/man/man3head/string.3head target=string.h.3head
2109 link path=usr/share/man/man3head/strings.3head target=strings.h.3head
2110 link path=usr/share/man/man3head/stropts.3head target=stropts.h.3head
2111 link path=usr/share/man/man3head/syslog.3head target=syslog.h.3head
2112 link path=usr/share/man/man3head/tar.3head target=tar.h.3head
2113 link path=usr/share/man/man3head/tcp.3head target=tcp.h.3head
2114 link path=usr/share/man/man3head/termios.3head target=termios.h.3head
2115 link path=usr/share/man/man3head/tgmath.3head target=tgmath.h.3head
2116 link path=usr/share/man/man3head/time.3head target=time.h.3head
2117 link path=usr/share/man/man3head/timeb.3head target=timeb.h.3head
2118 link path=usr/share/man/man3head/times.3head target=times.h.3head
2119 link path=usr/share/man/man3head/types.3head target=types.h.3head
2120 link path=usr/share/man/man3head/types32.3head target=types32.h.3head
2121 link path=usr/share/man/man3head/ucontext.3head target=ucontext.h.3head
2122 link path=usr/share/man/man3head/uio.3head target=uio.h.3head
2123 link path=usr/share/man/man3head/ulimit.3head target=ulimit.h.3head
2124 link path=usr/share/man/man3head/un.3head target=un.h.3head
2125 link path=usr/share/man/man3head/unistd.3head target=unistd.h.3head
2126 link path=usr/share/man/man3head/utime.3head target=utime.h.3head
2127 link path=usr/share/man/man3head/utmpx.3head target=utmpx.h.3head
2128 link path=usr/share/man/man3head/utsname.3head target=utsname.h.3head
2129 link path=usr/share/man/man3head/values.3head target=values.h.3head
2130 link path=usr/share/man/man3head/wait.3head target=wait.h.3head
2131 link path=usr/share/man/man3head/wchar.3head target=wchar.h.3head
2132 link path=usr/share/man/man3head/wctype.3head target=wctype.h.3head
2133 link path=usr/share/man/man3head/wordexp.3head target=wordexp.h.3head
2134 link path=usr/share/man/man3head/xlocale.3head target=xlocale.h.3head
2135 $(i386_ONLY)link path=usr/share/src/uts/i86pc/sys \
2136     target=../../../../platform/i86pc/include/sys
2137 $(i386_ONLY)link path=usr/share/src/uts/i86pc/vm \
2138     target=../../../../platform/i86pc/include/vm
2139 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/sys \
2140     target=../../../../platform/i86xpv/include/sys
2141 $(i386_ONLY)link path=usr/share/src/uts/i86xpv/vm \
2142     target=../../../../platform/i86xpv/include/vm
2143 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/sys \
2144     target=../../../../platform/sun4u/include/sys
2145 $(sparc_ONLY)link path=usr/share/src/uts/sun4u/vm \
2146     target=../../../../platform/sun4u/include/vm
2147 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/sys \
2148     target=../../../../platform/sun4v/include/sys
2149 $(sparc_ONLY)link path=usr/share/src/uts/sun4v/vm \
2150     target=../../../../platform/sun4v/include/vm
```

```

*****
46916 Wed Jul 13 01:32:36 2016
new/usr/src/uts/common/Makefile.files
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 #
23 # Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright (c) 2011, 2014 by Delphix. All rights reserved.
25 # Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
26 # Copyright 2015 Nexenta Systems, Inc. All rights reserved.
27 # Copyright 2016 Garrett D'Amore <garrett@damore.org>
28 # Copyright 2016 Joyent, Inc.
29 #
30 #
31 #
32 # This Makefile defines all file modules for the directory uts/common
33 # and its children. These are the source files which may be considered
34 # common to all SunOS systems.
35 #
36 i386_CORE_OBJS += \
37     atomic.o      \
38     avintr.o      \
39     pic.o
40 #
41 sparc_CORE_OBJS +=
42 #
43 COMMON_CORE_OBJS +=
44     beep.o        \
45     bitset.o      \
46     bp_map.o      \
47     brand.o       \
48     cpucaps.o     \
49     cmt.o         \
50     cmt_policy.o  \
51     cpu.o         \
52     cpu_event.o   \
53     cpu_intr.o    \
54     cpu_pm.o      \
55     cpupart.o     \
56     cap_util.o    \
57     disp.o        \
58     group.o       \
59     kstat_fr.o    \
60     iscsiboot_prop.o \
61     lgrp.o

```

```

62     lgrp_topo.o   \
63     mmapobj.o    \
64     mutex.o      \
65     page_lock.o  \
66     page_retire.o \
67     panic.o      \
68     param.o      \
69     pg.o         \
70     pghw.o       \
71     putnext.o    \
72     rctl_proc.o  \
73     rwlock.o     \
74     seg_kmem.o   \
75     softint.o    \
76     string.o     \
77     strtol.o     \
78     strtoul.o    \
79     strtoll.o    \
80     strtoull.o   \
81     thread_intr.o \
82     vm_page.o    \
83     vm_pagelist.o \
84     zlib_obj.o   \
85     clock_tick.o
86 #
87 CORE_OBJS += $(COMMON_CORE_OBJS) $(MACH)_CORE_OBJS
88 #
89 ZLIB_OBJS = zutil.o zmod.o zmod_subr.o \
90     Adler32.o crc32.o deflate.o inflate.o \
91     inflate.o infrees.o trees.o
92 #
93 GENUNIX_OBJS += \
94     access.o      \
95     acl.o         \
96     acl_common.o \
97     adjtime.o    \
98     alarm.o      \
99     aio_subr.o   \
100    auditsys.o    \
101    audit_core.o  \
102    audit_zone.o  \
103    audit_memory.o \
104    autoconf.o    \
105    avl.o         \
106    bdev_dsort.o  \
107    bio.o         \
108    bitmap.o     \
109    blabel.o     \
110    brandsys.o   \
111    bz2blocksort.o \
112    bz2compress.o \
113    bz2decompress.o \
114    bz2randtable.o \
115    bz2bzlib.o   \
116    bz2crctable.o \
117    bz2huffman.o \
118    callb.o      \
119    callout.o    \
120    chdir.o      \
121    chmod.o     \
122    chown.o     \
123    cladm.o     \
124    class.o     \
125    clock.o     \
126    clock_highres.o \
127    clock_realtime.o

```

new/usr/src/uts/common/Makefile.files

```

128         close.o          \|
129         compress.o       \|
130         condvar.o        \|
131         conf.o           \|
132         console.o        \|
133         contract.o       \|
134         copyops.o        \|
135         core.o           \|
136         corectl.o        \|
137         cred.o           \|
138         cs_stubs.o        \|
139         dacf.o           \|
140         dacf_clnt.o      \|
141         damap.o \        \|
142         cyclic.o         \|
143         ddi.o            \|
144         ddifm.o          \|
145         ddi_hp_impl.o    \|
146         ddi_hp_ndi.o     \|
147         ddi_intr.o       \|
148         ddi_intr_impl.o  \|
149         ddi_intr_irm.o   \|
150         ddi_nodeid.o     \|
151         ddi_periodic.o   \|
152         devcfg.o         \|
153         devcache.o       \|
154         device.o         \|
155         devid.o          \|
156         devid_cache.o    \|
157         devid_scsi.o     \|
158         devid_smp.o      \|
159         devpolicy.o      \|
160         disp_lock.o      \|
161         dnlc.o           \|
162         driver.o         \|
163         dumpsubr.o       \|
164         driver_lyr.o     \|
165         dtrace_subr.o    \|
166         errorq.o         \|
167         etheraddr.o     \|
168         evchannels.o     \|
169         exact.o          \|
170         exact_core.o     \|
171         exec.o           \|
172         exit.o           \|
173         fbio.o           \|
174         fcntl.o          \|
175         fdbuffer.o       \|
176         fdsync.o         \|
177         fem.o            \|
178         ffs.o            \|
179         fio.o            \|
180         flock.o          \|
181         fm.o             \|
182         fork.o           \|
183         vpm.o            \|
184         fs_reparse.o     \|
185         fs_subr.o        \|
186         fsflush.o        \|
187         ftrace.o         \|
188         getcwd.o         \|
189         getdents.o       \|
190         getloadavg.o     \|
191         getpagesizes.o   \|
192         getpid.o         \|
193         gfs.o            \|

```

3

new/usr/src/uts/common/Makefile.files

```

194         rusagesys.o     \|
195         gid.o            \|
196         groups.o        \|
197         grow.o           \|
198         hat_refmod.o    \|
199         id32.o           \|
200         id_space.o       \|
201         inet_ntop.o      \|
202         instance.o       \|
203         ioctl.o          \|
204         ip_cksum.o       \|
205         issetugid.o     \|
206         ippconf.o        \|
207         kcp.c.o          \|
208         kdi.o            \|
209         kiconv.o         \|
210         klpd.o           \|
211         kmem.o           \|
212         ksyms_snapshot.o \|
213         l_strplumb.o     \|
214         labelsys.o       \|
215         link.o           \|
216         list.o           \|
217         lockstat_subr.o  \|
218         log_sysevent.o   \|
219         logsubr.o        \|
220         lookup.o         \|
221         lseek.o          \|
222         ltos.o           \|
223         lwp.o            \|
224         lwp_create.o     \|
225         lwp_info.o       \|
226         lwp_self.o       \|
227         lwp_sobj.o       \|
228         lwp_timer.o      \|
229         lwpsys.o         \|
230         main.o           \|
231         mmapobjsys.o     \|
232         memcntl.o        \|
233         memstr.o         \|
234         mgrpsys.o        \|
235         mkdir.o          \|
236         mknod.o          \|
237         mount.o          \|
238         move.o           \|
239         msacct.o         \|
240         multidata.o      \|
241         nbmlck.o         \|
242         ndifm.o          \|
243         nice.o           \|
244         netstack.o       \|
245         ntptime.o        \|
246         nvpair.o         \|
247         nvpair_alloc_system.o \|
248         nvpair_alloc_fixed.o \|
249         fnvpair.o        \|
250         octet.o          \|
251         open.o           \|
252         p_online.o       \|
253         pathconf.o       \|
254         pathname.o       \|
255         pause.o          \|
256         serializer.o     \|
257         pci_intr_lib.o   \|
258         pci_cap.o        \|
259         pcifm.o          \|

```

4

```

260         pgrp.o          \
261         pgrpso.o        \
262         pid.o           \
263         pidnode.o      \
264 #endif /* ! codereview */
265         pkp_hash.o     \
266         policy.o       \
267         poll.o         \
268         pool.o         \
269         pool_pset.o    \
270         port_subr.o    \
271         ppriv.o        \
272         printf.o       \
273         prioctl.o      \
274         priv.o         \
275         priv_const.o   \
276         proc.o         \
277         procset.o      \
278         processor_bind.o \
279         processor_info.o \
280         profil.o       \
281         project.o      \
282         qsort.o        \
283         getrandom.o    \
284         rctl.o         \
285         rctlsys.o     \
286         readlink.o    \
287         refstr.o       \
288         rename.o       \
289         resolvepath.o  \
290         retire_store.o \
291         process.o      \
292         rlimit.o       \
293         rmap.o         \
294         rw.o           \
295         rwstlock.o    \
296         sad_conf.o    \
297         sid.o          \
298         sidsys.o      \
299         sched.o        \
300         schedctl.o    \
301         sctp_crc32.o  \
302         seg_dev.o      \
303         seg_kp.o       \
304         seg_kpm.o      \
305         seg_map.o     \
306         seg_vn.o       \
307         seg_spt.o     \
308         semaphore.o   \
309         sendfile.o    \
310         session.o     \
311         share.o       \
312         shuttle.o     \
313         sig.o          \
314         sigaction.o   \
315         sigaltstack.o \
316         signotify.o   \
317         sigpending.o  \
318         sigprocmask.o \
319         sigqueue.o    \
320         sigsendset.o  \
321         sigsuspend.o  \
322         sigtimedwait.o \
323         sleepq.o      \
324         sock_conf.o   \
325         space.o       \

```

```

326         sscanf.o      \
327         stat.o        \
328         statfs.o      \
329         statvfs.o     \
330         stol.o        \
331         str_conf.o    \
332         strcalls.o    \
333         stream.o      \
334         streamio.o    \
335         strext.o      \
336         strsubr.o     \
337         strsun.o      \
338         subr.o        \
339         sunddi.o      \
340         sunmdi.o      \
341         sunndi.o      \
342         sunpci.o      \
343         sunpm.o       \
344         sundlpi.o     \
345         suntpi.o      \
346         swap_subr.o   \
347         swap_vnops.o  \
348         symlink.o     \
349         sync.o        \
350         sysclass.o    \
351         sysconfig.o   \
352         sysent.o      \
353         sysfs.o       \
354         systeminfo.o  \
355         task.o        \
356         taskq.o       \
357         tasksys.o     \
358         time.o        \
359         timer.o       \
360         times.o       \
361         timers.o      \
362         thread.o      \
363         tlabel.o      \
364         tnf_res.o     \
365         turnstile.o   \
366         tty_common.o  \
367         u8_textprep.o \
368         uadmin.o      \
369         uconv.o       \
370         ucredsys.o    \
371         uid.o         \
372         umask.o       \
373         umount.o      \
374         uname.o       \
375         unix_bb.o     \
376         unlink.o     \
377         urw.o         \
378         utime.o       \
379         utssys.o      \
380         uucopy.o      \
381         vfs.o         \
382         vfs_conf.o    \
383         vmem.o        \
384         vm_anon.o     \
385         vm_as.o       \
386         vm_meter.o    \
387         vm_pageout.o  \
388         vm_pvn.o      \
389         vm_rm.o       \
390         vm_seg.o      \
391         vm_subr.o     \

```

```

392         vm_swap.o      \
393         vm_usage.o     \
394         vnode.o        \
395         vuid_queue.o   \
396         vuid_store.o   \
397         waitq.o        \
398         watchpoint.o   \
399         yield.o        \
400         scsi_confdata.o \
401         xattr.o        \
402         xattr_common.o \
403         xdr_mblk.o     \
404         xdr_mem.o      \
405         xdr.o          \
406         xdr_array.o   \
407         xdr_refer.o   \
408         zone.o

410 #
411 #     Stubs for the stand-alone linker/loader
412 #
413 sparc_GENSTUBS_OBJS = \
414     kobj_stubs.o

416 i386_GENSTUBS_OBJS =

418 COMMON_GENSTUBS_OBJS =

420 GENSTUBS_OBJS += $(COMMON_GENSTUBS_OBJS) $($ (MACH)_GENSTUBS_OBJS)

422 #
423 #     DTrace and DTrace Providers
424 #
425 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

427 SDT_OBJS += sdt_subr.o

429 PROFILE_OBJS += profile.o

431 SYSTRACE_OBJS += systrace.o

433 LOCKSTAT_OBJS += lockstat.o

435 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

437 DCPC_OBJS += dcpc.o

439 #
440 #     Driver (pseudo-driver) Modules
441 #
442 IPP_OBJS += ippctl.o

444 AUDIO_OBJS += audio_client.o audio_ddi.o audio_engine.o \
445     audiofltdata.o audio_format.o audio_ctrl.o \
446     audio_grc3.o audio_output.o audio_input.o \
447     audio_oss.o audio_sun.o

449 AUDIOEMU10K_OBJS += audioemu10k.o

451 AUDIOENS_OBJS += audioens.o

453 AUDIOVIA823X_OBJS += audiovia823x.o

455 AUDIOVIA97_OBJS += audiovia97.o

457 AUDIO1575_OBJS += audio1575.o

```

```

459 AUDIO810_OBJS += audio810.o

461 AUDIOCMI_OBJS += audiocmi.o

463 AUDIOCMIHD_OBJS += audiocmihd.o

465 AUDIOHD_OBJS += audiohd.o

467 AUDIOIXP_OBJS += audioixp.o

469 AUDIOLS_OBJS += audiols.o

471 AUDIOP16X_OBJS += audiop16x.o

473 AUDIOPCI_OBJS += audiopci.o

475 AUDIOSOLO_OBJS += audiosolo.o

477 AUDIOTS_OBJS += audiots.o

479 AC97_OBJS += ac97.o ac97_ad.o ac97_alc.o ac97_cmi.o

481 BLKDEV_OBJS += blkdev.o

483 CARDBUS_OBJS += cardbus.o cardbus_hp.o cardbus_cfg.o

485 CONSKBD_OBJS += conskbd.o

487 CONSMS_OBJS += consms.o

489 OLDPPTY_OBJS += tty_ptyconf.o

491 PTC_OBJS += tty_pty.o

493 PDSL_OBJS += tty_pts.o

495 PTM_OBJS += ptm.o

497 MII_OBJS += mii.o mii_cicada.o mii_natsemi.o mii_intel.o mii_qualsemi.o \
498     mii_marvell.o mii_realtek.o mii_other.o

500 PTS_OBJS += pts.o

502 PTY_OBJS += ptms_conf.o

504 SAD_OBJS += sad.o

506 MD4_OBJS += md4.o md4_mod.o

508 MD5_OBJS += md5.o md5_mod.o

510 SHA1_OBJS += sha1.o sha1_mod.o

512 SHA2_OBJS += sha2.o sha2_mod.o

514 SKEIN_OBJS += skein.o skein_block.o skein_iv.o skein_mod.o

516 EDONR_OBJS += edonr.o edonr_mod.o

518 IPGPC_OBJS += classifierddi.o classifier.o filters.o trie.o table.o \
519     ba_table.o

521 DSCPMK_OBJS += dscpmk.o dscpmkddi.o

523 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o

```

```

525 FLOWACCT_OBJS +=      flowacctddi.o flowacct.o
527 TOKENMT_OBJS += tokenmt.o tokenmtddi.o
529 TSWTCL_OBJS += tswtcl.o tswtclddi.o
531 ARP_OBJS += arpddi.o
533 ICMP_OBJS += icmpddi.o
535 ICMP6_OBJS += icmp6ddi.o
537 RTS_OBJS += rtsddi.o
539 IP_ICMP_OBJS = icmp.o icmp_opt_data.o
540 IP_RTS_OBJS = rts.o rts_opt_data.o
541 IP_TCP_OBJS = tcp.o tcp_fusion.o tcp_opt_data.o tcp_sack.o tcp_stats.o \
542 tcp_misc.o tcp_timers.o tcp_time_wait.o tcp_tpi.o tcp_output.o \
543 tcp_input.o tcp_socket.o tcp_bind.o tcp_cluster.o tcp_tunables.o
544 IP_UDP_OBJS = udp.o udp_opt_data.o udp_tunables.o udp_stats.o
545 IP_SCTP_OBJS = sctp.o sctp_opt_data.o sctp_output.o \
546 sctp_init.o sctp_input.o sctp_cookie.o \
547 sctp_conn.o sctp_error.o sctp_snmp.o \
548 sctp_tunables.o sctp_shutdown.o sctp_common.o \
549 sctp_timer.o sctp_heartbeat.o sctp_hash.o \
550 sctp_bind.o sctp_notify.o sctp_asconf.o \
551 sctp_addr.o tn_ipopt.o tnet.o ip_netinfo.o \
552 sctp_misc.o
553 IP_ILB_OBJS = ilb.o ilb_nat.o ilb_conn.o ilb_alg_hash.o ilb_alg_rr.o
555 IP_OBJS += igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
556 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
557 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
558 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
559 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_queue.o \
560 queue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
561 ip_helper_stream.o ip_tunables.o \
562 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
563 conn_opt.o ip_attr.o ip_dce.o \
564 $(IP_ICMP_OBJS) \
565 $(IP_RTS_OBJS) \
566 $(IP_TCP_OBJS) \
567 $(IP_UDP_OBJS) \
568 $(IP_SCTP_OBJS) \
569 $(IP_ILB_OBJS)
571 IP6_OBJS += ip6ddi.o
573 HOOK_OBJS += hook.o
575 NETI_OBJS += neti_impl.o neti_mod.o neti_stack.o
577 KEYSOCK_OBJS += keysockddi.o keysock.o keysock_opt_data.o
579 IPNET_OBJS += ipnet.o ipnet_bpf.o
581 SPDSOCK_OBJS += spdsockddi.o spdsock.o spdsock_opt_data.o
583 IPSECESP_OBJS += ipsecespddi.o ipsecesp.o
585 IPSECAH_OBJS += ipsecahddi.o ipsecah.o sadb.o
587 SPPP_OBJS += sPPP.o sPPP_dlpi.o sPPP_mod.o s_common.o
589 SPPPTUN_OBJS += sppptun.o sppptun_mod.o

```

```

591 SPPASYN_OBJS += spppasyn.o spppasyn_mod.o
593 SPPPCOMP_OBJS += sppppcomp.o sppppcomp_mod.o deflate.o bsd-comp.o vjcompress.o \
594 zlib.o
596 TCP_OBJS += tcpddi.o
598 TCP6_OBJS += tcp6ddi.o
600 NCA_OBJS += ncaddi.o
602 SDP SOCK_MOD_OBJS += sockmod_sdp.o socksdp.o socksdpsubr.o
604 SCTP SOCK_MOD_OBJS += sockmod_sctp.o sockstcp.o sockstcpsubr.o
606 PFP SOCK_MOD_OBJS += sockmod_pfp.o
608 RDS SOCK_MOD_OBJS += sockmod_rds.o
610 RDS_OBJS += rdsddi.o rdssubr.o rds_opt.o rds_ioctl.o
612 RDSIB_OBJS += rdsib.o rdsib_ib.o rdsib_cm.o rdsib_ep.o rdsib_buf.o \
613 rdsib_debug.o rdsib_sc.o
615 RDSV3_OBJS += af_rds.o rdsv3_ddi.o bind.o loop.o threads.o connection.o \
616 transport.o cong.o sysctl.o message.o rds_recv.o send.o \
617 stats.o info.o page.o rdma_transport.o ib_ring.o ib_rdma.o \
618 ib_recv.o ib.o ib_send.o ib_sysctl.o ib_stats.o ib_cm.o \
619 rdsv3_sc.o rdsv3_debug.o rdsv3_impl.o rdma.o rdsv3_af_thr.o
621 ISER_OBJS += iser.o iser_cm.o iser_cq.o iser_ib.o iser_idm.o \
622 iser_resource.o iser_xfer.o
624 UDP_OBJS += udpddi.o
626 UDP6_OBJS += udp6ddi.o
628 SY_OBJS += gentyty.o
630 TCO_OBJS += ticots.o
632 TCOO_OBJS += ticotsord.o
634 TCL_OBJS += ticlts.o
636 TL_OBJS += tl.o
638 DUMP_OBJS += dump.o
640 BPF_OBJS += bpf.o bpf_filter.o bpf_mod.o bpf_dlt.o bpf_mac.o
642 CLONE_OBJS += clone.o
644 CN_OBJS += cons.o
646 DLD_OBJS += dld_drv.o dld_proto.o dld_str.o dld_flow.o
648 DLS_OBJS += dls.o dls_link.o dls_mod.o dls_stat.o dls_mgmt.o
650 GLD_OBJS += gld.o gldutil.o
652 MAC_OBJS += mac.o mac_bcast.o mac_client.o mac_datapath_setup.o mac_flow.o
653 mac_hio.o mac_mod.o mac_ndd.o mac_provider.o mac_sched.o \
654 mac_protect.o mac_soft_ring.o mac_stat.o mac_util.o

```

```

656 MAC_6TO4_OBJS +=      mac_6to4.o
658 MAC_ETHER_OBJS +=    mac_ether.o
660 MAC_IPV4_OBJS +=     mac_ipv4.o
662 MAC_IPV6_OBJS +=     mac_ipv6.o
664 MAC_WIFI_OBJS +=     mac_wifi.o
666 MAC_IB_OBJS +=       mac_ib.o
668 IPTUN_OBJS +=        iptun_dev.o iptun_ctl.o iptun.o
670 AGGR_OBJS +=          aggr_dev.o aggr_ctl.o aggr_grp.o aggr_port.o \
671                        aggr_send.o aggr_recv.o aggr_lacp.o
673 SOFTMAC_OBJS +=       softmac_main.o softmac_ctl.o softmac_capab.o \
674                        softmac_dev.o softmac_stat.o softmac_pkt.o softmac_fp.o
676 NET80211_OBJS +=      net80211.o net80211_proto.o net80211_input.o \
677                        net80211_output.o net80211_node.o net80211_crypto.o \
678                        net80211_crypto_none.o net80211_crypto_wep.o net80211_ioctl.o \
679                        net80211_crypto_tkip.o net80211_crypto_ccmp.o \
680                        net80211_ht.o
682 VNIC_OBJS +=         vnic_ctl.o vnic_dev.o
684 SIMNET_OBJS +=       simnet.o
686 IB_OBJS +=           ibnex.o ibnex_ioctl.o ibnex_hca.o
688 IBCM_OBJS +=         ibcm_impl.o ibcm_sm.o ibcm_ti.o ibcm_utils.o ibcm_path.o \
689                        ibcm_arp.o ibcm_arp_link.o
691 IBDM_OBJS +=         ibdm.o
693 IBDMA_OBJS +=        ibdma.o
695 IBMF_OBJS +=         ibmf.o ibmf_impl.o ibmf_dr.o ibmf_wqe.o ibmf_ud_dest.o ibmf_mod.
696                        ibmf_send.o ibmf_recv.o ibmf_handlers.o ibmf_trans.o \
697                        ibmf_timers.o ibmf_msg.o ibmf_utils.o ibmf_rmpp.o \
698                        ibmf_saa.o ibmf_saa_impl.o ibmf_saa_utils.o ibmf_saa_events.o
700 IBTL_OBJS +=         ibtl_impl.o ibtl_util.o ibtl_mem.o ibtl_handlers.o ibtl_qp.o \
701                        ibtl_cq.o ibtl_wr.o ibtl_hca.o ibtl_chan.o ibtl_cm.o \
702                        ibtl_mcg.o ibtl_ibnex.o ibtl_srq.o ibtl_part.o
704 TAVOR_OBJS +=        tavor.o tavor_agents.o tavor_cfg.o tavor_ci.o tavor_cmd.o \
705                        tavor_cq.o tavor_event.o tavor_ioctl.o tavor_misc.o \
706                        tavor_mr.o tavor_qp.o tavor_qpmod.o tavor_rsrc.o \
707                        tavor_srq.o tavor_stats.o tavor_umap.o tavor_wr.o
709 HERMON_OBJS +=       hermon.o hermon_agents.o hermon_cfg.o hermon_ci.o hermon_cmd.o \
710                        hermon_cq.o hermon_event.o hermon_ioctl.o hermon_misc.o \
711                        hermon_mr.o hermon_qp.o hermon_qpmod.o hermon_rsrc.o \
712                        hermon_srq.o hermon_stats.o hermon_umap.o hermon_wr.o \
713                        hermon_fcoib.o hermon_fm.o
715 DAPLT_OBJS +=        daplt.o
717 SOL_OFS_OBJS +=      sol_cma.o sol_ib_cma.o sol_uobj.o \
718                        sol_ofs_debug_util.o sol_ofs_gen_util.o \
719                        sol_kverbs.o
721 SOL_UCMA_OBJS +=     sol_ucma.o

```

```

723 SOL_UVERBS_OBJS +=    sol_uverbs.o sol_uverbs_comp.o sol_uverbs_event.o \
724                        sol_uverbs_hca.o sol_uverbs_qp.o
726 SOL_UMAD_OBJS +=      sol_umad.o
728 KSTAT_OBJS +=        kstat.o
730 KSYMS_OBJS +=        ksyms.o
732 INSTANCE_OBJS +=     inst_sync.o
734 IWSCN_OBJS +=        iwscons.o
736 LOFI_OBJS +=         lofi.o LzmaDec.o
738 FSSNAP_OBJS +=       fssnap.o
740 FSSNAPIF_OBJS +=     fssnap_if.o
742 MM_OBJS +=           mem.o
744 PHYSMEM_OBJS +=     physmem.o
746 OPTIONS_OBJS +=     options.o
748 WINLOCK_OBJS +=     winlockio.o
750 PM_OBJS +=           pm.o
751 SRN_OBJS +=          srn.o
753 PSEUDO_OBJS +=      pseudonex.o
755 RAMDISK_OBJS +=     ramdisk.o
757 LLC1_OBJS +=        llc1.o
759 USBKBM_OBJS +=      usbkbm.o
761 USBWCM_OBJS +=      usbwcm.o
763 BOFI_OBJS +=        bofi.o
765 HID_OBJS +=         hid.o
767 USBSKEL_OBJS +=     usbskel.o
769 USBVC_OBJS +=       usbvc.o usbvc_v412.o
771 HIDPARSER_OBJS +=   hidparser.o
773 USB_AC_OBJS +=      usb_ac.o
775 USB_AS_OBJS +=      usb_as.o
777 USB_AH_OBJS +=      usb_ah.o
779 USBMS_OBJS +=       usbms.o
781 USBPRN_OBJS +=      usbprn.o
783 UGEN_OBJS +=        ugen.o
785 USBSER_OBJS +=      usbser.o usbser_rseq.o
787 USBSACM_OBJS +=     usbzacm.o

```

```

789 USBSER_KEYSPAN_OBJS += usbser_keyspan.o keyspan_dsd.o keyspan_pipe.o
791 USBS49_FW_OBJS += keyspan_49fw.o
793 USBSPRL_OBJS += usbser_pl2303.o pl2303_dsd.o
795 USBFTDI_OBJS += usbser_uftdi.o uftdi_dsd.o
797 USBECM_OBJS += usbecm.o
799 WC_OBJS += wscons.o vcons.o
801 VCONS_CONF_OBJS += vcons_conf.o
803 SCSI_OBJS +=      scsi_capabilities.o scsi_confsubr.o scsi_control.o \
804                 scsi_data.o scsi_fm.o scsi_hba.o scsi_reset_notify.o \
805                 scsi_resource.o scsi_subr.o scsi_transport.o scsi_watch.o \
806                 smp_transport.o
808 SCSI_VHCI_OBJS +=      scsi_vhci.o mpapi_impl.o scsi_vhci_tpgs.o
810 SCSI_VHCI_F_SYM_OBJS +=      sym.o
812 SCSI_VHCI_F_TPGS_OBJS +=      tpgs.o
814 SCSI_VHCI_F_ASYM_SUN_OBJS +=  asym_sun.o
816 SCSI_VHCI_F_SYM_HDS_OBJS +=   sym_hds.o
818 SCSI_VHCI_F_TAPE_OBJS +=     tape.o
820 SCSI_VHCI_F_TPGS_TAPE_OBJS += tpgs_tape.o
822 SGEN_OBJS +=      sgen.o
824 SMP_OBJS +=      smp.o
826 SATA_OBJS +=     sata.o
828 USBA_OBJS +=      hcidi.o usba.o usbai.o hubdi.o parser.o genconsole.o \
829                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
830                 usba_devdb.o usba10_calls.o usba_uugen.o
832 USBA10_OBJS +=   usba10.o
834 RSM_OBJS +=      rsm.o rsmka_pathmanager.o rsmka_util.o
836 RSMOPS_OBJS +=   rsmops.o
838 S1394_OBJS +=     t1394.o t1394_errmsg.o s1394.o s1394_addr.o s1394_async.o \
839                 s1394_bus_reset.o s1394_cmp.o s1394_csr.o s1394_dev_disc.o \
840                 s1394_fa.o s1394_fcp.o \
841                 s1394_hotplug.o s1394_isoch.o s1394_misc.o h1394.o nx1394.o
843 HCI1394_OBJS +=   hcil1394.o hcil1394_async.o hcil1394_attach.o hcil1394_buf.o \
844                 hcil1394_csr.o hcil1394_detach.o hcil1394_extern.o \
845                 hcil1394_ioctl.o hcil1394_isoch.o hcil1394_isr.o \
846                 hcil1394_ixl_comp.o hcil1394_ixl_isr.o hcil1394_ixl_misc.o \
847                 hcil1394_ixl_update.o hcil1394_misc.o hcil1394_ohci.o \
848                 hcil1394_q.o hcil1394_s1394if.o hcil1394_tlabel.o \
849                 hcil1394_tlist.o hcil1394_vendor.o
851 AV1394_OBJS +=    av1394.o av1394_as.o av1394_async.o av1394_cfgrom.o \
852                 av1394_cmp.o av1394_fcp.o av1394_isoch.o av1394_isoch_chan.o \
853                 av1394_isoch_recv.o av1394_isoch_xmit.o av1394_list.o \

```

```

854                 av1394_queue.o
856 DCAM1394_OBJS +=  dcam.o dcam_frame.o dcam_param.o dcam_reg.o \
857                 dcam_ring_buff.o
859 SCSA1394_OBJS +=  hba.o sbp2_driver.o sbp2_bus.o
861 SBP2_OBJS +=      cfgrom.o sbp2.o
863 PMODEM_OBJS +=    pmodem.o pmodem_cis.o cis.o cis_callout.o cis_handlers.o cis_para
865 DSW_OBJS +=       dsw.o dsw_dev.o ii_tree.o
867 NCALL_OBJS +=     ncall.o \
868                 ncall_stub.o
870 RDC_OBJS +=        rdc.o \
871                 rdc_dev.o \
872                 rdc_io.o \
873                 rdc_clnt.o \
874                 rdc_prot_xdr.o \
875                 rdc_svc.o \
876                 rdc_bitmap.o \
877                 rdc_health.o \
878                 rdc_subr.o \
879                 rdc_diskq.o
881 RDCSRV_OBJS +=    rdcsrv.o
883 RDCSTUB_OBJS +=   rdc_stub.o
885 SDBC_OBJS +=      sd_bcache.o \
886                 sd_bio.o \
887                 sd_conf.o \
888                 sd_ft.o \
889                 sd_hash.o \
890                 sd_io.o \
891                 sd_misc.o \
892                 sd_pcu.o \
893                 sd_tdaemon.o \
894                 sd_trace.o \
895                 sd_iob_impl0.o \
896                 sd_iob_impl1.o \
897                 sd_iob_impl2.o \
898                 sd_iob_impl3.o \
899                 sd_iob_impl4.o \
900                 sd_iob_impl5.o \
901                 sd_iob_impl6.o \
902                 sd_iob_impl7.o \
903                 safestore.o \
904                 safestore_ram.o
906 NSCTL_OBJS +=     nsctl.o \
907                 nsc_cache.o \
908                 nsc_disk.o \
909                 nsc_dev.o \
910                 nsc_freeze.o \
911                 nsc_gen.o \
912                 nsc_mem.o \
913                 nsc_ncallio.o \
914                 nsc_power.o \
915                 nsc_resv.o \
916                 nsc_rmspin.o \
917                 nsc_solaris.o \
918                 nsc_trap.o \
919                 nsc_list.o

```

```

920 UNISTAT_OBJS += spuni.o \
921                spcs_s_k.o

923 NSKERN_OBJS += nsc_ddi.o \
924                nsc_proc.o \
925                nsc_raw.o \
926                nsc_thread.o \
927                nskernd.o

929 SV_OBJS +=      sv.o

931 PMCS_OBJS += pmcs_attach.o pmcs_ds.o pmcs_intr.o pmcs_nvram.o pmcs_sata.o \
932                pmcs_scsa.o pmcs_smhba.o pmcs_subr.o pmcs_fwlog.o

934 PMCS8001FW_C_OBJS += pmcs_fw_hdr.o
935 PMCS8001FW_OBJS += $(PMCS8001FW_C_OBJS) SPCBoot.o ila.o firmware.o

937 #
938 #      Build up defines and paths.

940 ST_OBJS +=      st.o      st_conf.o

942 EMLXS_OBJS +=   emlxs_clock.o emlxs_dfc.o emlxs_dhchap.o emlxs_diag.o \
943                 emlxs_download.o emlxs_dump.o emlxs_eis.o emlxs_event.o \
944                 emlxs_fcf.o emlxs_fcp.o emlxs_fct.o emlxs_hba.o emlxs_ip.o \
945                 emlxs_mbox.o emlxs_mem.o emlxs_msg.o emlxs_node.o \
946                 emlxs_pkt.o emlxs_sli3.o emlxs_sli4.o emlxs_solaris.o \
947                 emlxs_thread.o

949 EMLXS_FW_OBJS +=      emlxs_fw.o

951 OCE_OBJS +=      oce_buf.o oce_fm.o oce_gld.o oce_hw.o oce_intr.o oce_main.o \
952                 oce_mbx.o oce_mq.o oce_queue.o oce_rx.o oce_stat.o oce_tx.o \
953                 oce_utils.o

955 FCT_OBJS +=      discovery.o fct.o

957 QLT_OBJS +=      2400.o 2500.o 8100.o qlt.o qlt_dma.o

959 SRPT_OBJS +=      srpt_mod.o srpt_ch.o srpt_cm.o srpt_ioc.o srpt_stp.o

961 FCOE_OBJS +=      fcoe.o fcoe_eth.o fcoe_fc.o

963 FCOET_OBJS +=     fcoet.o fcoet_eth.o fcoet_fc.o

965 FCOEI_OBJS +=     fcoei.o fcoei_eth.o fcoei_lv.o

967 ISCSIT_SHARED_OBJS += \
968                 iscsit_common.o

970 ISCSIT_OBJS +=    $(ISCSIT_SHARED_OBJS) \
971                 iscsit.o iscsit_tgt.o iscsit_sess.o iscsit_login.o \
972                 iscsit_text.o iscsit_isns.o iscsit_radiusauth.o \
973                 iscsit_radiuspacket.o iscsit_auth.o iscsit_authclient.o

975 PPPT_OBJS +=     alua_ic_if.o pppt.o pppt_msg.o pppt_tgt.o

977 STMF_OBJS +=     lun_map.o stmf.o

979 STMF_SBD_OBJS +=  sbd.o sbd_scsi.o sbd_pgr.o sbd_zvol.o

981 SYMSG_OBJS +=     sysmsg.o

983 SES_OBJS +=       ses.o ses_sen.o ses_safte.o ses_ses.o

985 TNF_OBJS +=       tnf_buf.o      tnf_trace.o      tnf_writer.o      trace_init.o \

```

```

986                trace_funcs.o  tnf_probe.o      tnf.o

988 LOGINDMUX_OBJS += logindmux.o

990 DEVINFO_OBJS +=  devinfo.o

992 DEVPOLL_OBJS +=  devpoll.o

994 DEVPOOL_OBJS +=  devpool.o

996 EVENTFD_OBJS +=  eventfd.o

998 SIGNALFD_OBJS +=  signalfd.o

1000 I8042_OBJS +=     i8042.o

1002 KB8042_OBJS +=    \
1003                 at_keyprocess.o \
1004                 kb8042.o \
1005                 kb8042_keytables.o

1007 MOUSE8042_OBJS += mouse8042.o

1009 FDC_OBJS +=       fdc.o

1011 ASY_OBJS +=       asy.o

1013 ECPP_OBJS +=      ecpp.o

1015 VUIDM3P_OBJS +=   vuidmice.o vuidm3p.o

1017 VUIDM4P_OBJS +=   vuidmice.o vuidm4p.o

1019 VUIDM5P_OBJS +=   vuidmice.o vuidm5p.o

1021 VUIDPS2_OBJS +=   vuidmice.o vuidps2.o

1023 HPCSV_C_OBJS +=   hpcsvc.o

1025 PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehpc.o pcishpc.o pcie_pwr.o p

1027 PCIHPNEXUS_OBJS += pcihp.o

1029 OPENEEPR_OBJS += openprom.o

1031 RANDOM_OBJS +=    random.o

1033 PSHOT_OBJS +=     pshot.o

1035 GEN_DRV_OBJS +=   gen_drv.o

1037 TCLIENT_OBJS +=   tclient.o

1039 TIMERFD_OBJS +=   timerfd.o

1041 TPHCI_OBJS +=     tphci.o

1043 TVHCI_OBJS +=     tvhci.o

1045 EMUL64_OBJS +=    emul64.o emul64_bsd.o

1047 FCP_OBJS +=       fcp.o

1049 FCIP_OBJS +=       fcip.o

1051 FCSM_OBJS +=       fcsm.o

```

```

1053 FCTL_OBJS += fctl.o
1055 FP_OBJS += fp.o
1057 QLC_OBJS += ql_api.o ql_debug.o ql_hba_fru.o ql_init.o ql_iocb.o ql_ioctl.o \
1058 ql_isr.o ql_mbx.o ql_nx.o ql_xioctl.o ql_fw_table.o
1060 QLC_FW_2200_OBJS += ql_fw_2200.o
1062 QLC_FW_2300_OBJS += ql_fw_2300.o
1064 QLC_FW_2400_OBJS += ql_fw_2400.o
1066 QLC_FW_2500_OBJS += ql_fw_2500.o
1068 QLC_FW_6322_OBJS += ql_fw_6322.o
1070 QLC_FW_8100_OBJS += ql_fw_8100.o
1072 QLGE_OBJS += qlge.o qlge_dbg.o qlge_flash.o qlge_fm.o qlge_gld.o qlge_mpi.o
1074 ZCONS_OBJS += zcons.o
1076 NV_SATA_OBJS += nv_sata.o
1078 SI3124_OBJS += si3124.o
1080 AHCI_OBJS += ahci.o
1082 PCIIDE_OBJS += pci-ide.o
1084 PCEPP_OBJS += pcepp.o
1086 CPC_OBJS += cpc.o
1088 CPUID_OBJS += cpuid_drv.o
1090 SYSEVENT_OBJS += sysevent.o
1092 BL_OBJS += bl.o
1094 DRM_OBJS += drm_sunmod.o drm_kstat.o drm_agpsupport.o \
1095 drm_auth.o drm_bufs.o drm_context.o drm_dma.o \
1096 drm_drawable.o drm_drv.o drm_fops.o drm_ioctl.o drm_irq.o \
1097 drm_lock.o drm_memory.o drm_msg.o drm_pci.o drm_scatter.o \
1098 drm_cache.o drm_gem.o drm_mm.o ati_pcigart.o
1100 FM_OBJS += devfm.o devfm_machdep.o
1102 RTLS_OBJS += rtls.o
1104 #
1105 #           exec modules
1106 #
1107 AOUTEXEC_OBJS +=aout.o
1109 ELFEXEC_OBJS += elf.o elf_notes.o old_notes.o
1111 INTPEXEC_OBJS +=intp.o
1113 SHBINEXEC_OBJS +=shbin.o
1115 JAVAEXEC_OBJS +=java.o
1117 #

```

```

1118 #           file system modules
1119 #
1120 AUTOFS_OBJS += auto_vfsops.o auto_vnops.o auto_subr.o auto_xdr.o auto_sys.o
1122 DCFS_OBJS += dc_vnops.o
1124 DEVFS_OBJS += devfs_subr.o devfs_vfsops.o devfs_vnops.o
1126 DEV_OBJS += sdev_subr.o sdev_vfsops.o sdev_vnops.o \
1127 sdev_ptsops.o sdev_zvolops.o sdev_comm.o \
1128 sdev_profile.o sdev_ncache.o sdev_netops.o \
1129 sdev_ipnetops.o \
1130 sdev_vtops.o
1132 CTFS_OBJS += ctfs_all.o ctfs_cdir.o ctfs_ctl.o ctfs_event.o \
1133 ctfs_latest.o ctfs_root.o ctfs_sym.o ctfs_tdir.o ctfs_tmpl.o
1135 OBJFS_OBJS += objfs_vfs.o objfs_root.o objfs_common.o \
1136 objfs_odir.o objfs_data.o
1138 FDFS_OBJS += fdops.o
1140 FIFO_OBJS += fifosubr.o fifovnops.o
1142 PIPE_OBJS += pipe.o
1144 HSFS_OBJS += hsfs_node.o hsfs_subr.o hsfs_vfsops.o hsfs_vnops.o \
1145 hsfs_susp.o hsfs_rrip.o hsfs_susp_subr.o
1147 LOFS_OBJS += lofs_subr.o lofs_vfsops.o lofs_vnops.o
1149 NAMEFS_OBJS += namevfs.o namevno.o
1151 NFS_OBJS += nfs_client.o nfs_common.o nfs_dump.o \
1152 nfs_subr.o nfs_vfsops.o nfs_vnops.o \
1153 nfs_xdr.o nfs_sys.o nfs_strerror.o \
1154 nfs3_vfsops.o nfs3_vnops.o nfs3_xdr.o \
1155 nfs_acl_vnops.o nfs_acl_xdr.o nfs4_vfsops.o \
1156 nfs4_vnops.o nfs4_xdr.o nfs4_idmap.o \
1157 nfs4_shadow.o nfs4_subr.o \
1158 nfs4_attr.o nfs4_rnode.o nfs4_client.o \
1159 nfs4_acache.o nfs4_common.o nfs4_client_state.o \
1160 nfs4_callback.o nfs4_recovery.o nfs4_client_secinfo.o \
1161 nfs4_client_debug.o nfs_stats.o \
1162 nfs4_acl.o nfs4_stub_vnops.o nfs_cmd.o
1164 NFSSRV_OBJS += nfs_server.o nfs_srv.o nfs3_srv.o \
1165 nfs_acl_srv.o nfs_auth.o nfs_auth_xdr.o \
1166 nfs_export.o nfs_log.o nfs_log_xdr.o \
1167 nfs4_srv.o nfs4_state.o nfs4_srv_attr.o \
1168 nfs4_srv_ns.o nfs4_db.o nfs4_srv_deleg.o \
1169 nfs4_deleg_ops.o nfs4_srv_readdir.o nfs4_dispatch.o
1171 SMBSRV_SHARED_OBJS += \
1172 smb_door_legacy.o \
1173 smb_inet.o \
1174 smb_match.o \
1175 smb_msgbuf.o \
1176 smb_native.o \
1177 smb_netbios_util.o \
1178 smb_oem.o \
1179 smb_sid.o \
1180 smb_status2winerr.o \
1181 smb_string.o \
1182 smb_token.o \
1183 smb_token_xdr.o \

```

```

1184         smb_utf8.o \
1185         smb_xdr.o

1187 # See also: $SRC/lib/smb/rv/libfksmb/rv/Makefile.com
1188 SMBSRV_OBJS += $(SMBSRV_SHARED_OBJS) \
1189         smb_acl.o \
1190         smb_alloc.o \
1191         smb_authenticate.o \
1192         smb_close.o \
1193         smb_cmh_rename.o \
1194         smb_cmh_setfile.o \
1195         smb_common_open.o \
1196         smb_common_transact.o \
1197         smb_create.o \
1198         smb_cred.o \
1199         smb_delete.o \
1200         smb_dfs.o \
1201         smb_directory.o \
1202         smb_dispatch.o \
1203         smb_echo.o \
1204         smb_errno.o \
1205         smb_fem.o \
1206         smb_find.o \
1207         smb_flush.o \
1208         smb_fsinfo.o \
1209         smb_fsops.o \
1210         smb_idmap.o \
1211         smb_init.o \
1212         smb_kdoor.o \
1213         smb_kshare.o \
1214         smb_kutil.o \
1215         smb_lock.o \
1216         smb_lock_byte_range.o \
1217         smb_locking_andx.o \
1218         smb_logoff_andx.o \
1219         smb_mangle_name.o \
1220         smb_mbuf_marshall.o \
1221         smb_mbuf_util.o \
1222         smb_negotiate.o \
1223         smb_net.o \
1224         smb_node.o \
1225         smb_notify.o \
1226         smb_nt_cancel.o \
1227         smb_nt_create_andx.o \
1228         smb_nt_transact_create.o \
1229         smb_nt_transact_ioctl.o \
1230         smb_nt_transact_notify_change.o \
1231         smb_nt_transact_quota.o \
1232         smb_nt_transact_security.o \
1233         smb_odir.o \
1234         smb_ofile.o \
1235         smb_open_andx.o \
1236         smb_opipe.o \
1237         smb_oplock.o \
1238         smb_pathname.o \
1239         smb_print.o \
1240         smb_process_exit.o \
1241         smb_query_fileinfo.o \
1242         smb_quota.o \
1243         smb_read.o \
1244         smb_rename.o \
1245         smb_sd.o \
1246         smb_seek.o \
1247         smb_server.o \
1248         smb_session.o \
1249         smb_session_setup_andx.o

```

```

1250         smb_set_fileinfo.o \
1251         smb_sign_kcf.o \
1252         smb_signing.o \
1253         smb_thread.o \
1254         smb_tree.o \
1255         smb_trans2_create_directory.o \
1256         smb_trans2_dfs.o \
1257         smb_trans2_find.o \
1258         smb_tree_connect.o \
1259         smb_unlock_byte_range.o \
1260         smb_user.o \
1261         smb_vfs.o \
1262         smb_vops.o \
1263         smb_vss.o \
1264         smb_write.o \
1265         \
1266         smb2_dispatch.o \
1267         smb2_cancel.o \
1268         smb2_change_notify.o \
1269         smb2_close.o \
1270         smb2_create.o \
1271         smb2_echo.o \
1272         smb2_flush.o \
1273         smb2_ioctl.o \
1274         smb2_lock.o \
1275         smb2_logoff.o \
1276         smb2_negotiate.o \
1277         smb2_ofile.o \
1278         smb2_oplock.o \
1279         smb2_qinfo_file.o \
1280         smb2_qinfo_fs.o \
1281         smb2_qinfo_sec.o \
1282         smb2_qinfo_quota.o \
1283         smb2_query_dir.o \
1284         smb2_query_info.o \
1285         smb2_read.o \
1286         smb2_session_setup.o \
1287         smb2_set_info.o \
1288         smb2_setinfo_file.o \
1289         smb2_setinfo_fs.o \
1290         smb2_setinfo_quota.o \
1291         smb2_setinfo_sec.o \
1292         smb2_signing.o \
1293         smb2_tree_connect.o \
1294         smb2_tree_disconn.o \
1295         smb2_write.o

1297 PCFS_OBJS += pc_alloc.o pc_dir.o pc_node.o pc_subr.o \
1298         pc_vfsops.o pc_vnops.o

1300 PROC_OBJS += prcontrol.o priotcl.o prsubr.o prusr.o \
1301         prvfsops.o prvnops.o

1303 MNTFS_OBJS += mntvfsops.o mntvnops.o

1305 SHAREFS_OBJS += sharetab.o sharefs_vfsops.o sharefs_vnops.o

1307 SPEC_OBJS += specsusr.o specvfsops.o specvnops.o

1309 SOCK_OBJS += socksusr.o sockvfsops.o sockparams.o \
1310         socksyscalls.o socktpi.o sockstr.o \
1311         sockcommon_vnops.o sockcommon_subr.o \
1312         sockcommon_sops.o sockcommon.o \
1313         sock_notsupp.o socknotify.o \
1314         nl7c.o nl7curi.o nl7chttp.o nl7clogd.o \
1315         nl7cnca.o sodirect.o sockfilter.o

```

```

1317 TMPFS_OBJS += tmp_dir.o tmp_subr.o tmp_tnode.o tmp_vfsops.o \
1318 tmp_vnops.o
1320 UDFS_OBJS += udf_alloc.o udf_bmap.o udf_dir.o \
1321 udf_inode.o udf_subr.o udf_vfsops.o \
1322 udf_vnops.o
1324 UFS_OBJS += ufs_alloc.o ufs_bmap.o ufs_dir.o ufs_xattr.o \
1325 ufs_inode.o ufs_subr.o ufs_tables.o ufs_vfsops.o \
1326 ufs_vnops.o quota.o quotacalls.o quota_ufs.o \
1327 ufs_filio.o ufs_lockfs.o ufs_thread.o ufs_trans.o \
1328 ufs_acl.o ufs_panic.o ufs_directio.o ufs_log.o \
1329 ufs_extvnops.o lufs.o lufs_snap.o lufs_thread.o \
1330 lufs_log.o lufs_map.o lufs_top.o lufs_debug.o
1331 VSCAN_OBJS += vscan_drv.o vscan_svc.o vscan_door.o
1333 NSMB_OBJS += smb_conn.o smb_dev.o smb_iod.o smb_pass.o \
1334 smb_rq.o smb_sign.o smb_smb.o smb_subrs.o \
1335 smb_time.o smb_tran.o smb_trantcp.o smb_usr.o \
1336 subr_mchain.o
1338 SMBFS_COMMON_OBJS += smbfs_ntacl.o
1339 SMBFS_OBJS += smbfs_vfsops.o smbfs_vnops.o smbfs_node.o \
1340 smbfs_acl.o smbfs_client.o smbfs_smb.o \
1341 smbfs_subr.o smbfs_subr2.o \
1342 smbfs_rwlock.o smbfs_xattr.o \
1343 $(SMBFS_COMMON_OBJS)
1345 BOOTFS_OBJS += bootfs_construct.o bootfs_vfsops.o bootfs_vnops.o
1347 #
1348 # LVM modules
1349 #
1350 MD_OBJS += md.o md_error.o md_ioctl.o md_mddb.o md_names.o \
1351 md_med.o md_rename.o md_subr.o
1353 MD_COMMON_OBJS = md_convert.o md_crc.o md_revchk.o
1355 MD_DERIVED_OBJS = metamed_xdr.o meta_basic_xdr.o
1357 SOFTPART_OBJS += sp.o sp_ioctl.o
1359 STRIPE_OBJS += stripe.o stripe_ioctl.o
1361 HOTSPARES_OBJS += hotspares.o
1363 RAID_OBJS += raid.o raid_ioctl.o raid_replay.o raid_resync.o raid_hotspare.o
1365 MIRROR_OBJS += mirror.o mirror_ioctl.o mirror_resync.o
1367 NOTIFY_OBJS += md_notify.o
1369 TRANS_OBJS += mdtrans.o trans_ioctl.o trans_log.o
1371 ZFS_COMMON_OBJS += \
1372 arc.o \
1373 blkptr.o \
1374 bplist.o \
1375 bpobj.o \
1376 bptree.o \
1377 bqueue.o \
1378 dbuf.o \
1379 ddt.o \
1380 ddt_zap.o \
1381 dmuf.o

```

```

1382 dmuf.o \
1383 dmuf_send.o \
1384 dmuf_object.o \
1385 dmuf_objset.o \
1386 dmuf_traverse.o \
1387 dmuf_tx.o \
1388 dnode.o \
1389 dnode_sync.o \
1390 dsl_bookmark.o \
1391 dsl_dir.o \
1392 dsl_dataset.o \
1393 dsl_deadlist.o \
1394 dsl_destroy.o \
1395 dsl_pool.o \
1396 dsl_synctask.o \
1397 dsl_userhold.o \
1398 dmuf_zfetch.o \
1399 dsl_deleg.o \
1400 dsl_prop.o \
1401 dsl_scan.o \
1402 zfeature.o \
1403 gzip.o \
1404 lz4.o \
1405 lzjb.o \
1406 metaslab.o \
1407 multilist.o \
1408 range_tree.o \
1409 refcount.o \
1410 rrwlock.o \
1411 sa.o \
1412 sha256.o \
1413 edonr_zfs.o \
1414 skein_zfs.o \
1415 spa.o \
1416 spa_config.o \
1417 spa_errlog.o \
1418 spa_history.o \
1419 spa_misc.o \
1420 space_map.o \
1421 space_reftree.o \
1422 txg.o \
1423 uberblock.o \
1424 unique.o \
1425 vdev.o \
1426 vdev_cache.o \
1427 vdev_file.o \
1428 vdev_label.o \
1429 vdev_mirror.o \
1430 vdev_missing.o \
1431 vdev_queue.o \
1432 vdev_raidz.o \
1433 vdev_root.o \
1434 zap.o \
1435 zap_leaf.o \
1436 zap_micro.o \
1437 zfs_byteswap.o \
1438 zfs_debug.o \
1439 zfs_fm.o \
1440 zfs_fuid.o \
1441 zfs_sa.o \
1442 zfs_znode.o \
1443 zil.o \
1444 zio.o \
1445 zio_checksum.o \
1446 zio_compress.o \
1447 zio_inject.o \

```

```

1448     zle.o \
1449     zrlock.o \

1451 ZFS_SHARED_OBJS += \
1452     zfeature_common.o \
1453     zfs_comutil.o \
1454     zfs_deleg.o \
1455     zfs_fletcher.o \
1456     zfs_namecheck.o \
1457     zfs_prop.o \
1458     zpool_prop.o \
1459     zprop_common.o \

1461 ZFS_OBJS += \
1462     $(ZFS_COMMON_OBJS) \
1463     $(ZFS_SHARED_OBJS) \
1464     vdev_disk.o \
1465     zfs_acl.o \
1466     zfs_ctldir.o \
1467     zfs_dir.o \
1468     zfs_ioctl.o \
1469     zfs_log.o \
1470     zfs_onexit.o \
1471     zfs_replay.o \
1472     zfs_rlock.o \
1473     zfs_vfsops.o \
1474     zfs_vnops.o \
1475     zvol.o \

1477 ZUT_OBJS += \
1478     zut.o \

1480 #
1481 #           streams modules
1482 #
1483 BUFMOD_OBJS +=     bufmod.o

1485 CONNLD_OBJS +=     connld.o

1487 DEDUMP_OBJS +=     dedump.o

1489 DRCOMPAT_OBJS +=     drcompat.o

1491 LDLINUX_OBJS +=     ldlinux.o

1493 LDTERM_OBJS +=     ldterm.o uwidth.o

1495 PKKT_OBJS +=     pckt.o

1497 PFMOD_OBJS +=     pfmod.o

1499 PTEM_OBJS +=     ptem.o

1501 REDIRMOD_OBJS +=     strredirm.o

1503 TIMOD_OBJS +=     timod.o

1505 TIRDWR_OBJS +=     tirdwr.o

1507 TTCOMPAT_OBJS +=     ttcompat.o

1509 LOG_OBJS +=     log.o

1511 PIPEMOD_OBJS +=     pipemod.o

1513 RPCMOD_OBJS +=     rpcmod.o     clnt_cots.o     clnt_clts.o \

```

```

1514     clnt_gen.o     clnt_perr.o     mt_rpcinit.o     rpc_calmsg.o \
1515     rpc_prot.o     rpc_sztypes.o     rpc_subr.o     rpch_prot.o \
1516     svc.o     svc_clts.o     svc_gen.o     svc_cots.o \
1517     rpcsys.o     xdr_sizeof.o     clnt_rdma.o     svc_rdma.o \
1518     xdr_rdma.o     rdma_subr.o     xdrdma_sizeof.o \

1520 KLMMOD_OBJS +=     klmmod.o \
1521     nlm_impl.o \
1522     nlm_rpc_handle.o \
1523     nlm_dispatch.o \
1524     nlm_rpc_svc.o \
1525     nlm_client.o \
1526     nlm_service.o \
1527     nlm_prot_clnt.o \
1528     nlm_prot_xdr.o \
1529     nlm_rpc_clnt.o \
1530     nsm_addr_clnt.o \
1531     nsm_addr_xdr.o \
1532     sm_inter_clnt.o \
1533     sm_inter_xdr.o \

1535 KLMOPS_OBJS +=     klmops.o

1537 TLIMOD_OBJS +=     tlimod.o     t_kalloc.o     t_kbind.o     t_kclose.o \
1538     t_kconnect.o     t_kfree.o     t_kgtstate.o     t_kopen.o \
1539     t_krcvudat.o     t_ksndudat.o     t_kspoll.o     t_kunbind.o \
1540     t_kutil.o \

1542 RLMOD_OBJS +=     rlmod.o

1544 TELMOD_OBJS +=     telmod.o

1546 CRYPTMOD_OBJS +=     cryptmod.o

1548 KB_OBJS +=     kbd.o     keytables.o

1550 #
1551 #           ID mapping module
1552 #
1553 IDMAP_OBJS +=     idmap_mod.o     idmap_kapi.o     idmap_xdr.o     idmap_cache.o

1555 #
1556 #           scheduling class modules
1557 #
1558 SDC_OBJS +=     sysdc.o

1560 RT_OBJS +=     rt.o
1561 RT_DPTBL_OBJS +=     rt_dptbl.o

1563 TS_OBJS +=     ts.o
1564 TS_DPTBL_OBJS +=     ts_dptbl.o

1566 IA_OBJS +=     ia.o

1568 FSS_OBJS +=     fss.o

1570 FX_OBJS +=     fx.o
1571 FX_DPTBL_OBJS +=     fx_dptbl.o

1573 #
1574 #           Inter-Process Communication (IPC) modules
1575 #
1576 IPC_OBJS +=     ipc.o

1578 IPCMSG_OBJS +=     msg.o

```

```

1580 IPCSEM_OBJS += sem.o
1582 IPCSHM_OBJS += shm.o

1584 #
1585 #             bignum module
1586 #
1587 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o

1589 BIGNUM_OBJS += $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)

1591 #
1592 #             kernel cryptographic framework
1593 #
1594 KCF_OBJS += kcf.o kcf_callprov.o kcf_cbufcall.o kcf_cipher.o kcf_crypto.o \
1595 kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1596 kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1597 kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1598 kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1599 kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1600 fips_random.o

1602 CRYPTOADM_OBJS += cryptoadm.o

1604 CRYPTO_OBJS += crypto.o

1606 DPROV_OBJS += dprov.o

1608 DCA_OBJS += dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1609 dca_rsa.o

1611 AESPROV_OBJS += aes.o aes_impl.o aes_modes.o

1613 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o

1615 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o

1617 ECCPROV_OBJS += ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1618 ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1619 ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1620 ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1621 mpi.o mplogic.o mpmontg.o mprime.o oid.o \
1622 secitem.o ec2_test.o ecp_test.o

1624 RSAPROV_OBJS += rsa.o rsa_impl.o pkcs1.o

1626 SWRANDPROV_OBJS += swrand.o

1628 #
1629 #             kernel SSL
1630 #
1631 KSSL_OBJS += kssl.o ksslioc1.o

1633 KSSL_SOCKETFIL_MOD_OBJS += ksslfilter.o ksslapi.o ksslrec.o

1635 #
1636 #             misc. modules
1637 #

1639 C2AUDIT_OBJS += adr.o audit.o audit_event.o audit_io.o \
1640 audit_path.o audit_start.o audit_syscalls.o audit_token.o \
1641 audit_mem.o

1643 PCIC_OBJS += pcic.o

1645 RPCSEC_OBJS += secmod.o sec_clnt.o sec_svc.o sec_gen.o \

```

```

1646 auth_des.o auth_kern.o auth_none.o auth_loopb.o \
1647 authdesprt.o authdesubr.o authu_prot.o \
1648 key_call.o key_prot.o svc_authu.o svcauthdes.o

1650 RPCSEC_GSS_OBJS += rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
1651 rpcsec_gss_utils.o svc_rpcsec_gss.o

1653 CONSCONFIG_OBJS += consconfig.o

1655 CONSCONFIG_DACF_OBJS += consconfig_dacf.o consplat.o

1657 TEM_OBJS += tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1659 KBTRANS_OBJS += \
1660 kbtrans.o \
1661 kbtrans_keytables.o \
1662 kbtrans_polled.o \
1663 kbtrans_streams.o \
1664 usb_keytables.o

1666 KGSSD_OBJS += gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1667 gss_display_name.o gss_release_name.o gss_import_name.o \
1668 gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1670 KGSSD_DERIVED_OBJS = gssd_xdr.o

1672 KGSS_DUMMY_OBJS += dmec.o

1674 KSOCKET_OBJS += ksocket.o ksocket_mod.o

1676 CRYPTO= cksmtypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1677 nfold.o verify_checksum.o prng.o block_size.o make_checksum.o \
1678 checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1680 # crypto/des
1681 CRYPTO_DES= fcbc.o fcksum.o fparity.o weak_key.o d3cbc.o efcrypto.o

1683 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1685 CRYPTO_ARCFOUR= k5_arcfour.o

1687 # crypto/enc_provider
1688 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o

1690 # crypto/hash_provider
1691 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_kshal.o

1693 # crypto/keyhash_provider
1694 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1696 # crypto/crc32
1697 CRYPTO_CRC32= crc32.o

1699 # crypto/old
1700 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1702 # crypto/raw
1703 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1705 K5_KRB= kfree.o copy_key.o \
1706 parse.o init_ctx.o \
1707 ser_adata.o ser_addr.o \
1708 ser_auth.o ser_cksum.o \
1709 ser_key.o ser_princ.o \
1710 serialize.o unparse.o \
1711 ser_actx.o

```

```

1713 K5_OS=   timeofday.o toffset.o \
1714         init_os_ctx.o c_uptime.o

1716 SEAL=   seal.o unseal.o

1718 MECH=   delete_sec_context.o \
1719         import_sec_context.o \
1720         gssapi_krb5.o \
1721         k5seal.o k5unseal.o k5sealv3.o \
1722         ser_sctx.o \
1723         sign.o \
1724         util_crypt.o \
1725         util_validate.o util_ordering.o \
1726         util_seqnum.o util_set.o util_seed.o \
1727         wrap_size_limit.o verify.o

1731 MECH_GEN= util_token.o

1734 KGSS_KRB5_OBJS += krb5mech.o \
1735                 $(MECH) $(SEAL) $(MECH_GEN) \
1736                 $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1737                 $(CRYPTO_ENC) $(CRYPTO_HASH) \
1738                 $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1739                 $(CRYPTO_OLD) \
1740                 $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1742 DES_OBJS +=   des_crypt.o des_impl.o des_ks.o des_soft.o

1744 DLBOOT_OBJS += bootparam_xdr.o nfs_dlinet.o scan.o

1746 KRTLD_OBJS +=  kobj_bootflags.o getoptstr.o \
1747                 kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1749 MOD_OBJS +=    modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1751 STRPLUMB_OBJS += strplumb.o

1753 CPR_OBJS +=    cpr_driver.o cpr_dump.o \
1754                 cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1755                 cpr_uthread.o

1757 PROF_OBJS +=  prf.o

1759 SE_OBJS +=    se_driver.o

1761 SYSACCT_OBJS += acct.o

1763 ACCTCTL_OBJS += acctctl.o

1765 EXACCTSYS_OBJS += exacctsys.o

1767 KAIO_OBJS +=  aio.o

1769 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o

1771 BUSRA_OBJS += busra.o

1773 PCS_OBJS +=  pcs.o

1775 PSET_OBJS +=  pset.o

1777 OHCI_OBJS +=  ohci.o ohci_hub.o ohci_polled.o

```

```

1779 UHCI_OBJS +=  uhci.o uhciutil.o uhci_tgt.o uhcihub.o uhci_polled.o

1781 EHCI_OBJS +=  ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o

1783 HUBD_OBJS +=  hubd.o

1785 USB_MID_OBJS += usb_mid.o

1787 USB_IA_OBJS += usb_ia.o

1789 SCESA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o

1791 IPF_OBJS +=   ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1792             ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1793             ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o

1795 IPD_OBJS +=   ipd.o

1797 IBD_OBJS +=   ibd.o ibd_cm.o

1799 EIBNX_OBJS +=  enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1800             enx_misc.o enx_q.o enx_ctl.o

1802 EOIB_OBJS +=   eib_adm.o eib_chan.o eib_cmn.o eib_ctl.o eib_data.o \
1803             eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1804             eib_rsrc.o eib_svc.o eib_vnic.o

1806 DLPSTUB_OBJS += dlpstub.o

1808 SDP_OBJS +=    sdpddi.o

1810 TRILL_OBJS +=  trill.o

1812 CTF_OBJS +=    ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1813             ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o

1815 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o

1817 RPCIB_OBJS +=  rpcib.o

1819 KMDB_OBJS +=   kdrv.o

1821 AFE_OBJS +=    afe.o

1823 BGE_OBJS +=    bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1824             bge_atomic.o bge_mii.o bge_send.o bge_rcv2.o bge_mii_5906.o

1826 DMFE_OBJS +=   dmfe_log.o dmfe_main.o dmfe_mii.o

1828 EFE_OBJS +=    efe.o

1830 ELXL_OBJS +=   elxl.o

1832 HME_OBJS +=    hme.o

1834 IXGB_OBJS +=   ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1835             ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o

1837 NGE_OBJS +=    nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1838             nge_log.o nge_rx.o nge_tx.o nge_xmii.o

1840 PCN_OBJS +=    pcn.o

1842 RGE_OBJS +=    rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rxtx.o

```

```

1844 URTW_OBJS += urtw.o
1846 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_ani.o arn_phy.o arn_
1847         arn_main.o arn_rcv.o arn_xmit.o arn_rc.o
1849 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o
1851 ATU_OBJS += atu.o
1853 IPW_OBJS += ipw2100_hw.o ipw2100.o
1855 IWI_OBJS += ipw2200_hw.o ipw2200.o
1857 IWH_OBJS += iwh.o
1859 IWK_OBJS += iwk2.o
1861 IWP_OBJS += iwp.o
1863 MWL_OBJS += mwl.o
1865 MWLFW_OBJS += mwlfw_mode.o
1867 WPI_OBJS += wpi.o
1869 RAL_OBJS += rt2560.o ral_rate.o
1871 RUM_OBJS += rum.o
1873 RWD_OBJS += rt2661.o
1875 RWN_OBJS += rt2860.o
1877 UATH_OBJS += uath.o
1879 UATHFW_OBJS += uathfw_mod.o
1881 URAL_OBJS += ural.o
1883 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o
1885 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o
1887 MXFE_OBJS += mxfe.o
1889 MPTSAS_OBJS += mptsas.o mptsas_hash.o mptsas_impl.o mptsas_init.o \
1890         mptsas_raid.o mptsas_smhba.o
1892 SFE_OBJS += sfe.o sfe_util.o
1894 BFE_OBJS += bfe.o
1896 BRIDGE_OBJS += bridge.o
1898 IDM_SHARED_OBJS += base64.o
1900 IDM_OBJS += $(IDM_SHARED_OBJS) \
1901         idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o
1903 VR_OBJS += vr.o
1905 ATGE_OBJS += atge_main.o atge_lle.o atge_mii.o atge_ll.o atge_llc.o
1907 YGE_OBJS = yge.o
1909 SKD_OBJS = skd.o

```

```

1911 NVME_OBJS = nvme.o
1913 #
1914 #     Build up defines and paths.
1915 #
1916 LINT_DEFS     += -Dunix
1918 #
1919 #     This duality can be removed when the native and target compilers
1920 #     are the same (or at least recognize the same command line syntax!)
1921 #     It is a bug in the current compilation system that the assembler
1922 #     can't process the -Y I, flag.
1923 #
1924 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1925 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1926 INCLUDE_PATH   += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1928 PCIEB_OBJS += pcieb.o
1930 #     Chelsio N110 10G NIC driver module
1931 #
1932 CH_OBJS = ch.o glue.o pe.o sge.o
1934 CH_COM_OBJS = ch_mac.o ch_subr.o cspi.o espi.o ixfl1010.o mc3.o mc4.o mc5.o \
1935         mv88e1xxx.o mv88x201x.o my3126.o pm3393.o tp.o ulp.o \
1936         vsc7321.o vsc7326.o xpak.o
1938 #
1939 #     Chelsio Terminator 4 10G NIC nexus driver module
1940 #
1941 CXGBE_FW_OBJS = t4_fw.o t4_cfg.o
1942 CXGBE_COM_OBJS = t4_hw.o common.o
1943 CXGBE_NEX_OBJS = t4_nexus.o t4_sge.o t4_mac.o t4_ioctl.o shared.o \
1944         t4_l2t.o adapter.o osdep.o
1946 #
1947 #     Chelsio Terminator 4 10G NIC driver module
1948 #
1949 CXGBE_OBJS = cxgbe.o
1951 #
1952 #     PCI strings file
1953 #
1954 PCI_STRING_OBJS = pci_strings.o
1956 NET_DACF_OBJS += net_dacf.o
1958 #
1959 #     Xframe 10G NIC driver module
1960 #
1961 XGE_OBJS = xge.o xgell.o
1963 XGE_HAL_OBJS = xgehal-channel.o xgehal-fifo.o xgehal-ring.o xgehal-config.o \
1964         xgehal-driver.o xgehal-mm.o xgehal-stats.o xgehal-device.o \
1965         xge-queue.o xgehal-mgmt.o xgehal-mgmtaux.o
1967 #
1968 #     e1000/igb common objs
1969 #
1970 #     Historically e1000g and igb had separate copies of all of the common
1971 #     code. At this time while they are now sharing the same copy of it, they
1972 #     are building it into their own modules which is due to the differences
1973 #     in the osdep and debug portions of their code.
1974 #
1975 E1000API_OBJS += e1000_80003es21an.o e1000_82540.o e1000_82541.o e1000_82542.o \

```

```

1976         e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
1977         e1000_mac.o e1000_manage.o e1000_nvmm.o e1000_phy.o \
1978         e1000_82575.o e1000_i210.o e1000_mbx.o e1000_vf.o

1980 #
1981 #         e1000g module
1982 #
1983 E1000G_OBJS += e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1984               e1000g_tx.o e1000g_rx.o e1000g_stat.o \
1985               e1000g_osdep.o e1000g_workarounds.o
1986

1988 #
1989 #         Intel 82575 1G NIC driver module
1990 #
1991 IGB_OBJS =    igb_buf.o igb_debug.o igb_gld.o igb_log.o igb_main.o \
1992               igb_rx.o igb_stat.o igb_tx.o igb_osdep.o

1994 #
1995 #         Intel Pro/100 NIC driver module
1996 #
1997 IPRB_OBJS =   iprb.o

1999 #
2000 #         Intel 10GbE PCIE NIC driver module
2001 #
2002 IXGBE_OBJS =  ixgbe_82598.o ixgbe_82599.o ixgbe_api.o      \
2003               ixgbe_common.o ixgbe_phy.o                  \
2004               ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o        \
2005               ixgbe_log.o ixgbe_main.o                     \
2006               ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o        \
2007               ixgbe_tx.o ixgbe_x540.o ixgbe_mbx.o

2009 #
2010 #         Intel 40GbE PCIe NIC driver module
2011 #

2013 # illumos-written ones.
2014 I40E_OBJS =   i40e_main.o i40e_osdep.o i40e_intr.o i40e_transceiver.o \
2015               i40e_stats.o i40e_gld.o
2016 # Intel-written ones.
2017 I40E_INTC_OBJS = i40e_adminq.o i40e_common.o i40e_hmc.o i40e_lan_hmc.o \
2018                 i40e_nvmm.o

2020 #
2021 #         Solarflare 1/10/40GbE NIC driver module
2022 #
2023 #         NB: The illumos specific sources are listed first, with the
2024 #         common (OS-independent) sources afterwards.
2025 #
2026 SFXGE_OBJS =  sfxge_err.o sfxge_ev.o sfxge_hash.o sfxge_intr.o sfxge_mac.o \
2027               sfxge_gld_v3.o sfxge_mon.o sfxge_phy.o \
2028               sfxge_sram.o sfxge_bar.o sfxge_pci.o sfxge_nvram.o \
2029               sfxge_rx.o sfxge_tcp.o sfxge_tx.o sfxge_mcdi.o sfxge_vpd.o \
2030               sfxge.o sfxge_dma.o
2031 SFXGE_SF_OBJS = efx_bootcfg.o efx_crc32.o efx_ev.o efx_filter.o \
2032                 efx_hash.o efx_intr.o efx_mac.o efx_mcdi.o efx_mon.o \
2033                 efx_nic.o efx_nvram.o efx_phy.o efx_port.o efx_rx.o \
2034                 efx_sram.o efx_tx.o efx_vpd.o efx_wol.o mcdi_mon.o \
2035                 siena_mac.o siena_mcdi.o siena_nic.o siena_nvram.o \
2036                 siena_phy.o siena_sram.o siena_vpd.o \
2037                 ef10_ev.o ef10_filter.o ef10_intr.o ef10_mac.o ef10_mcdi.o \
2038                 ef10_nic.o ef10_nvram.o ef10_phy.o ef10_rx.o ef10_tx.o \
2039                 ef10_vpd.o hunt_nic.o hunt_phy.o

2041 #

```

```

2042 #         NIU 10G/1G driver module
2043 #
2044 NXGE_OBJS =   nxge_mac.o nxge_ipp.o nxge_rxdma.o      \
2045               nxge_txdma.o nxge_txc.o nxge_main.o    \
2046               nxge_hw.o nxge_fzc.o nxge_virtual.o   \
2047               nxge_send.o nxge_classify.o nxge_fflp.o \
2048               nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o \
2049               nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o \
2050               nxge_hio.o nxge_hio_guest.o nxge_intr.o

2052 NXGE_NPI_OBJS = \
2053               napi.o napi_mac.o napi_ipp.o           \
2054               napi_txdma.o napi_rxdma.o napi_txc.o   \
2055               napi_zcp.o napi_espc.o napi_fflp.o     \
2056               napi_vir.o

2058 NXGE_HCALL_OBJS = \
2059               nxge_hcall.o

2061 #
2062 #         Virtio modules
2063 #

2065 #         Virtio core
2066 VIRTIO_OBJS = virtio.o

2068 #         Virtio block driver
2069 VIOBLK_OBJS = vioblk.o

2071 #         Virtio network driver
2072 VIOIF_OBJS = vioif.o

2074 #
2075 #         kiconv modules
2076 #
2077 KICONV_EMEA_OBJS += kiconv_emea.o

2079 KICONV_JA_OBJS += kiconv_ja.o

2081 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o

2083 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o

2085 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o

2087 #
2088 #         AAC module
2089 #
2090 AAC_OBJS = aac.o aac_ioctl.o

2092 #
2093 #         sdcard modules
2094 #
2095 SDA_OBJS =    sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o
2096 SDHOST_OBJS = sdhost.o

2098 #
2099 #         hxge 10G driver module
2100 #
2101 HXGE_OBJS =   hxge_main.o hxge_vmac.o hxge_send.o      \
2102               hxge_txdma.o hxge_rxdma.o hxge_virtual.o \
2103               hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o \
2104               hxge_ndd.o hxge_pfc.o                     \
2105               hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o  \
2106               hpi_vir.o hpi_pfc.o

```

```

2108 #
2109 #     MEGARAID_SAS module
2110 #
2111 MEGA_SAS_OBJS = megaraid_sas.o

2113 #
2114 #     MR_SAS module
2115 #
2116 MR_SAS_OBJS = ld_pd_map.o mr_sas.o mr_sas_tbolt.o mr_sas_list.o

2118 #
2119 #     CPQARY3 module
2120 #
2121 CPQARY3_OBJS = cpqary3.o cpqary3_noe.o cpqary3_talk2ctrl.o \
2122 cpqary3_isr.o cpqary3_transport.o cpqary3_mem.o \
2123 cpqary3_scsi.o cpqary3_util.o cpqary3_ioctl.o \
2124 cpqary3_bd.o

2126 #
2127 #     ISCSI_INITIATOR module
2128 #
2129 ISCSI_INITIATOR_OBJS = chap.o iscsi_io.o iscsi_thread.o \
2130 iscsi_ioctl.o iscsid.o iscsi.o \
2131 iscsi_login.o isns_client.o iscsiAuthClient.o \
2132 iscsi_lun.o iscsiAuthClientGlue.o \
2133 iscsi_net.o nvfile.o iscsi_cmd.o \
2134 iscsi_queue.o persistent.o iscsi_conn.o \
2135 iscsi_sess.o radius_auth.o iscsi_crc.o \
2136 iscsi_stats.o radius_packet.o iscsi_doorclt.o \
2137 iscsi_targetparam.o utils.o kifconf.o

2139 #
2140 #     ntxn 10Gb/1Gb NIC driver module
2141 #
2142 NTXN_OBJS = unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
2143 unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

2145 #
2146 #     Myricom 10Gb NIC driver module
2147 #
2148 MYRI10GE_OBJS = myril0ge.o myril0ge_lro.o

2150 #
2151 #     nulldriver module
2152 #
2152 NULLDRIVER_OBJS = nulldriver.o

2154 TPM_OBJS = tpm.o tpm_hcall.o

2156 #
2157 #     BNXE objects
2158 #
2159 BNXE_OBJS += bnxe_cfg.o \
2160 bnxe_fcoe.o \
2161 bnxe_debug.o \
2162 bnxe_gld.o \
2163 bnxe_hw.o \
2164 bnxe_intr.o \
2165 bnxe_kstat.o \
2166 bnxe_lock.o \
2167 bnxe_main.o \
2168 bnxe_mm.o \
2169 bnxe_mm_l4.o \
2170 bnxe_mm_l5.o \
2171 bnxe_rr.o \
2172 bnxe_rx.o \
2173 bnxe_timer.o

```

```

2174 bnxe_tx.o \
2175 bnxe_workq.o \
2176 bnxe_clc.o \
2177 ecore_sp_verbs.o \
2178 bnxe_context.o \
2179 57710_init_values.o \
2180 57711_init_values.o \
2181 57712_init_values.o \
2182 bnxe_fw_funcs.o \
2183 bnxe_hw_debug.o \
2184 lm_l4fp.o \
2185 lm_l4rx.o \
2186 lm_l4sp.o \
2187 lm_l4tx.o \
2188 lm_l5.o \
2189 lm_l5sp.o \
2190 lm_dcbx.o \
2191 lm_devinfo.o \
2192 lm_dmae.o \
2193 lm_er.o \
2194 lm_hw_access.o \
2195 lm_hw_attn.o \
2196 lm_hw_init_reset.o \
2197 lm_main.o \
2198 lm_mcp.o \
2199 lm_niv.o \
2200 lm_nvram.o \
2201 lm_phy.o \
2202 lm_power.o \
2203 lm_recv.o \
2204 lm_resc.o \
2205 lm_sb.o \
2206 lm_send.o \
2207 lm_sp.o \
2208 lm_dcbx_mp.o \
2209 lm_sp_req_mgr.o \
2210 lm_stats.o \
2211 lm_util.o

```

```

*****
87211 Wed Jul 13 01:32:36 2016
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 = ufallloc(0)) == -1)
1783         return (-1);
1784     setf(fd, fp);
1785     dp->d_data.d_desc.d_descriptor = fd;

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

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

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

1823     disp_lock_t *tlp;
1824     cpu_t *cp;

1826     ASSERT(MUTEX_HELD(&door_knob));

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

1958 /*

```

```

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

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

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

1982      if (fpp)
1983          *fpp = fp;

1985      return (VTOD(vp));
1986 }

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

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

2009      mutex_enter(&door_knob);
2010      for (;;) {
2011          if (DOOR_T_HELD(ct))
2012              cv_wait(&ct->d_cv, &door_knob);
2013          else if (DOOR_T_HELD(st))
2014              cv_wait(&st->d_cv, &door_knob);
2015          else
2016              break;
2017          /* neither flag is set */
2018      }
2019      curthread->t_door = NULL;
2020      if ((dp = st->d_active) != NULL) {
2021          kthread_t *t = st->d_caller;
2022          proc_t *p = curproc;

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

```

```

2025     door_list_delete(dp);
2026     dp->door_target = NULL;
2027     dp->door_flags |= DOOR_REVOKED;
2028     if (dp->door_flags & DOOR_PRIVATE)
2029         cv_broadcast(&dp->door_servers.dp_cv);
2030     else
2031         cv_broadcast(&p->p_server_threads.dp_cv);
2032 }
2033
2034     if (t != NULL) {
2035         /*
2036          * Let the caller know we are gone
2037          */
2038         DOOR_CLIENT(t->t_door)->d_error = DOOR_EXIT;
2039         thread_lock(t);
2040         if (t->t_state == TS_SLEEP &&
2041             SOBJ_TYPE(t->t_sobj_ops) == SOBJ_SHUTTLE)
2042             setrun_locked(t);
2043         thread_unlock(t);
2044     }
2045 }
2046 mutex_exit(&door_knob);
2047 if (st->d_pool)
2048     door_unbind_thread(st->d_pool); /* Implicit door_unbind */
2049 kmem_free(dt, sizeof (door_data_t));
2050 }
2051
2052 /*
2053 * Set DOOR_REVOKED for all doors of the current process. This is called
2054 * on exit before all lwp's are being terminated so that door calls will
2055 * return with an error.
2056 */
2057 void
2058 door_revoke_all()
2059 {
2060     door_node_t *dp;
2061     proc_t *p = ttoproc(curthread);
2062
2063     mutex_enter(&door_knob);
2064     for (dp = p->p_door_list; dp != NULL; dp = dp->door_list) {
2065         ASSERT(dp->door_target == p);
2066         dp->door_flags |= DOOR_REVOKED;
2067         if (dp->door_flags & DOOR_PRIVATE)
2068             cv_broadcast(&dp->door_servers.dp_cv);
2069     }
2070     cv_broadcast(&p->p_server_threads.dp_cv);
2071     mutex_exit(&door_knob);
2072 }
2073
2074 /*
2075 * The process is exiting, and all doors it created need to be revoked.
2076 */
2077 void
2078 door_exit(void)
2079 {
2080     door_node_t *dp;
2081     proc_t *p = ttoproc(curthread);
2082
2083     ASSERT(p->p_lwpcnt == 1);
2084     /*
2085      * Walk the list of active doors created by this process and
2086      * revoke them all.
2087      */
2088     mutex_enter(&door_knob);
2089     for (dp = p->p_door_list; dp != NULL; dp = dp->door_list) {
2090         dp->door_target = NULL;

```

```

2091         dp->door_flags |= DOOR_REVOKED;
2092         if (dp->door_flags & DOOR_PRIVATE)
2093             cv_broadcast(&dp->door_servers.dp_cv);
2094     }
2095     cv_broadcast(&p->p_server_threads.dp_cv);
2096     /* Clear the list */
2097     p->p_door_list = NULL;
2098
2099     /* Clean up the unref list */
2100     while ((dp = p->p_unref_list) != NULL) {
2101         p->p_unref_list = dp->door_ulist;
2102         dp->door_ulist = NULL;
2103         mutex_exit(&door_knob);
2104         VN_RELE(DTOV(dp));
2105         mutex_enter(&door_knob);
2106     }
2107     mutex_exit(&door_knob);
2108 }
2109
2110 void
2111 door_fork(kthread_t *parent, kthread_t *child)
2112 {
2113     door_data_t *pt = parent->t_door;
2114     door_server_t *st = DOOR_SERVER(pt);
2115     door_data_t *dt;
2116
2117     ASSERT(MUTEX_NOT_HELD(&door_knob));
2118     if (pt != NULL && (st->d_pool != NULL || st->d_invbound)) {
2119         /* parent thread is bound to a door */
2120         dt = child->t_door =
2121             kmem_zalloc(sizeof (door_data_t), KM_SLEEP);
2122         DOOR_SERVER(dt)->d_invbound = 1;
2123     }
2124 }
2125
2126 /* Deliver queued unrefs to appropriate door server.
2127 */
2128 static int
2129 door_unref(void)
2130 {
2131     door_node_t *dp;
2132     static door_arg_t unref_args = { DOOR_UNREF_DATA, 0, 0, 0, 0, 0 };
2133     proc_t *p = ttoproc(curthread);
2134
2135     /* make sure there's only one unref thread per process */
2136     mutex_enter(&door_knob);
2137     if (p->p_unref_thread) {
2138         mutex_exit(&door_knob);
2139         return (set_errno(EALREADY));
2140     }
2141     p->p_unref_thread = 1;
2142     mutex_exit(&door_knob);
2143
2144     (void) door_my_data(1); /* create info, if necessary */
2145
2146     for (;;) {
2147         mutex_enter(&door_knob);
2148
2149         /* Grab a queued request */

```

```

2157     while ((dp = p->p_unref_list) == NULL) {
2158         if (!cv_wait_sig(&p->p_unref_cv, &door_knob)) {
2159             /*
2160              * Interrupted.
2161              * Return so we can finish forkall() or exit().
2162              */
2163             p->p_unref_thread = 0;
2164             mutex_exit(&door_knob);
2165             return (set_errno(EINTR));
2166         }
2167     }
2168     p->p_unref_list = dp->door_ulist;
2169     dp->door_ulist = NULL;
2170     dp->door_flags |= DOOR_UNREF_ACTIVE;
2171     mutex_exit(&door_knob);
2172
2173     (void) door_upcall(DTOV(dp), &unref_args, NULL, SIZE_MAX, 0);
2174
2175     if (unref_args.rbuf != 0) {
2176         kmem_free(unref_args.rbuf, unref_args.rsize);
2177         unref_args.rbuf = NULL;
2178         unref_args.rsize = 0;
2179     }
2180
2181     mutex_enter(&door_knob);
2182     ASSERT(dp->door_flags & DOOR_UNREF_ACTIVE);
2183     dp->door_flags &= ~DOOR_UNREF_ACTIVE;
2184     mutex_exit(&door_knob);
2185     VN_RELE(DTOV(dp));
2186 }
2187
2188
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;
2200
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);
2209
2210     (void) door_my_data(1); /* make sure we have a door_data_t */
2211
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;

```

```

2223         dp->door_ulist = NULL;
2224         dp->door_flags |= DOOR_UNREF_ACTIVE;
2225         mutex_exit(&door_knob);
2226
2227         (*(dp->door_pc))(dp->door_data, &unref_args, NULL, NULL, NULL);
2228
2229         mutex_enter(&door_knob);
2230         ASSERT(dp->door_flags & DOOR_UNREF_ACTIVE);
2231         dp->door_flags &= ~DOOR_UNREF_ACTIVE;
2232         mutex_exit(&door_knob);
2233         VN_RELE(DTOV(dp));
2234     }
2235 }
2236
2237
2238 /*
2239  * Queue an unref invocation for processing for the current process
2240  * The door may or may not be revoked at this point.
2241  */
2242 void
2243 door_deliver_unref(door_node_t *d)
2244 {
2245     struct proc *server = d->door_target;
2246
2247     ASSERT(MUTEX_HELD(&door_knob));
2248     ASSERT(d->door_active == 0);
2249
2250     if (server == NULL)
2251         return;
2252     /*
2253      * Create a lwp to deliver unref calls if one isn't already running.
2254      *
2255      * A separate thread is used to deliver unrefs since the current
2256      * thread may be holding resources (e.g. locks) in user land that
2257      * may be needed by the unref processing. This would cause a
2258      * deadlock.
2259      */
2260     if (d->door_flags & DOOR_UNREF_MULTTI) {
2261         /* multiple unrefs */
2262         d->door_flags &= ~DOOR_DELAY;
2263     } else {
2264         /* Only 1 unref per door */
2265         d->door_flags &= ~(DOOR_UNREF|DOOR_DELAY);
2266     }
2267     mutex_exit(&door_knob);
2268
2269     /*
2270      * Need to bump the vnode count before putting the door on the
2271      * list so it doesn't get prematurely released by door_unref.
2272      */
2273     VN_HOLD(DTOV(d));
2274
2275     mutex_enter(&door_knob);
2276     /* is this door already on the unref list? */
2277     if (d->door_flags & DOOR_UNREF_MULTTI) {
2278         door_node_t *dp;
2279         for (dp = server->p_unref_list; dp != NULL;
2280              dp = dp->door_ulist) {
2281             if (d == dp) {
2282                 /* already there, don't need to add another */
2283                 mutex_exit(&door_knob);
2284                 VN_RELE(DTOV(d));
2285                 mutex_enter(&door_knob);
2286                 return;
2287             }
2288         }

```

```

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

```

```

2421     }
2422 }
2423 out:
2424     ct->d_overflow = 1;
2425     ct->d_args.rbuf = addr;
2426     ct->d_args.rsize = rlen;
2427     return (0);
2428 }
2429
2430 /*
2431  * Transfer arguments from the client to the server.
2432  */
2433 static int
2434 door_args(kthread_t *server, int is_private)
2435 {
2436     door_server_t *st = DOOR_SERVER(server->t_door);
2437     door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2438     uint_t ndid;
2439     size_t dsize;
2440     int error;
2441
2442     ASSERT(DOOR_T_HELD(st));
2443     ASSERT(MUTEX_NOT_HELD(&door_knob));
2444
2445     ndid = ct->d_args.desc_num;
2446     if (ndid > door_max_desc)
2447         return (E2BIG);
2448
2449     /*
2450      * Get the stack layout, and fail now if it won't fit.
2451      */
2452     error = door_layout(server, ct->d_args.data_size, ndid, is_private);
2453     if (error != 0)
2454         return (error);
2455
2456     dsize = ndid * sizeof (door_desc_t);
2457     if (ct->d_args.data_size != 0) {
2458         if (ct->d_args.data_size <= door_max_arg) {
2459             /*
2460              * Use a 2 copy method for small amounts of data
2461              * 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;
2482
2483             /*
2484              * Use a 1 copy method
2485              */

```

```

2487         as = ttoproc(server)->p_as;
2488         src = ct->d_args.data_ptr;
2489
2490         dest = st->d_layout.dl_datap;
2491         base = (uintptr_t)dest;
2492
2493         /*
2494          * Copy data directly into server. We proceed
2495          * downward from the top of the stack, to mimic
2496          * normal stack usage. This allows the guard page
2497          * to stop us before we corrupt anything.
2498          */
2499         while (len != 0) {
2500             uintptr_t start;
2501             uintptr_t end;
2502             uintptr_t offset;
2503             size_t amount;
2504
2505             /*
2506              * Locate the next part to copy.
2507              */
2508             end = base + len;
2509             start = P2ALIGN(end - 1, PAGE_SIZE);
2510
2511             /*
2512              * if we are on the final (first) page, fix
2513              * up the start position.
2514              */
2515             if (P2ALIGN(base, PAGE_SIZE) == start)
2516                 start = base;
2517
2518             offset = start - base; /* the copy offset */
2519             amount = end - start; /* # bytes to copy */
2520
2521             ASSERT(amount > 0 && amount <= len &&
2522                 amount <= PAGE_SIZE);
2523
2524             error = door_copy(as, src + offset,
2525                 dest + offset, amount);
2526             if (error != 0)
2527                 return (error);
2528             len -= amount;
2529         }
2530     }
2531 }
2532 /*
2533  * Copyin the door args and translate them into files
2534  */
2535 if (ndid != 0) {
2536     door_desc_t      *didpp;
2537     door_desc_t      *start;
2538     struct file      **fpp;
2539
2540     start = didpp = kmem_alloc(dsize, KM_SLEEP);
2541
2542     if (copyin_nowatch(ct->d_args.desc_ptr, didpp, dsize)) {
2543         kmem_free(start, dsize);
2544         return (EFAULT);
2545     }
2546     ct->d_fpp_size = ndid * sizeof (struct file *);
2547     ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2548     fpp = ct->d_fpp;
2549     while (ndid--) {
2550         struct file *fp;
2551         int fd = didpp->d_data.d_desc.d_descriptor;

```

```

2553     /* We only understand file descriptors as passed objs */
2554     if (!(didpp->d_attributes & DOOR_DESCRIPTOR) ||
2555         (fp = getf(fd)) == NULL) {
2556         /* close translated references */
2557         door_fd_close(ct->d_fpp, fpp - ct->d_fpp);
2558         /* close untranslated references */
2559         door_fd_rele(didpp, ndid + 1, 0);
2560         kmem_free(start, dsize);
2561         kmem_free(ct->d_fpp, ct->d_fpp_size);
2562         ct->d_fpp = NULL;
2563         ct->d_fpp_size = 0;
2564         return (EINVAL);
2565     }
2566     /* Hold the fp */
2567     mutex_enter(&fp->f_tlock);
2568     fp->f_count++;
2569     mutex_exit(&fp->f_tlock);
2571
2572     *fpp = fp;
2573     releasef(fd);
2574
2575     if (didpp->d_attributes & DOOR_RELEASE) {
2576         /* release passed reference */
2577         (void) closeandsetf(fd, NULL);
2578     }
2579     fpp++; didpp++;
2580     kmem_free(start, dsize);
2581 }
2582 return (0);
2583 }
2584 }
2586 /*
2587  * Transfer arguments from a user client to a kernel server. This copies in
2588  * descriptors and translates them into door handles. It doesn't touch the
2589  * other data, letting the kernel server deal with that (to avoid needing
2590  * to copy the data twice).
2591  */
2592 static int
2593 door_translate_in(void)
2594 {
2595     door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2596     uint_t ndid;
2598     ASSERT(MUTEX_NOT_HELD(&door_knob));
2599     ndid = ct->d_args.desc_num;
2600     if (ndid > door_max_desc)
2601         return (E2BIG);
2602     /*
2603      * Copyin the door args and translate them into door handles.
2604      */
2605     if (ndid != 0) {
2606         door_desc_t *didpp;
2607         door_desc_t *start;
2608         size_t dsize = ndid * sizeof (door_desc_t);
2609         struct file *fp;
2611         start = didpp = kmem_alloc(dsize, KM_SLEEP);
2613         if (copyin_nowatch(ct->d_args.desc_ptr, didpp, dsize) {
2614             kmem_free(start, dsize);
2615             return (EFAULT);
2616         }
2617         while (ndid--) {
2618             vnode_t *vp;

```

```

2619         int fd = didpp->d_data.d_desc.d_descriptor;
2621         /*
2622          * We only understand file descriptors as passed objs
2623          */
2624         if ((didpp->d_attributes & DOOR_DESCRIPTOR) &&
2625             (fp = getf(fd)) != NULL) {
2626             didpp->d_data.d_handle = FTODH(fp);
2627             /* Hold the door */
2628             door_ki_hold(didpp->d_data.d_handle);
2630             releasef(fd);
2632             if (didpp->d_attributes & DOOR_RELEASE) {
2633                 /* release passed reference */
2634                 (void) closeandsetf(fd, NULL);
2635             }
2637             if (VOP_REALVP(fp->f_vnode, &vp, NULL))
2638                 vp = fp->f_vnode;
2640             /* Set attributes */
2641             didpp->d_attributes = DOOR_HANDLE |
2642                 (VTOD(vp)->door_flags & DOOR_ATTR_MASK);
2643         } else {
2644             /* close translated references */
2645             door_fd_close(start, didpp - start);
2646             /* close untranslated references */
2647             door_fd_rele(didpp, ndid + 1, 0);
2648             kmem_free(start, dsize);
2649             return (EINVAL);
2650         }
2651         didpp++;
2652     }
2653     ct->d_args.desc_ptr = start;
2654 }
2655 return (0);
2656 }
2658 /*
2659  * Translate door arguments from kernel to user. This copies the passed
2660  * door handles. It doesn't touch other data. It is used by door_upcall,
2661  * and for data returned by a door_call to a kernel server.
2662  */
2663 static int
2664 door_translate_out(void)
2665 {
2666     door_client_t *ct = DOOR_CLIENT(curthread->t_door);
2667     uint_t ndid;
2669     ASSERT(MUTEX_NOT_HELD(&door_knob));
2670     ndid = ct->d_args.desc_num;
2671     if (ndid > door_max_desc) {
2672         door_fd_rele(ct->d_args.desc_ptr, ndid, 1);
2673         return (E2BIG);
2674     }
2675     /*
2676      * Translate the door args into files
2677      */
2678     if (ndid != 0) {
2679         door_desc_t *didpp = ct->d_args.desc_ptr;
2680         struct file **fpp;
2682         ct->d_fpp_size = ndid * sizeof (struct file *);
2683         fpp = ct->d_fpp = kmem_alloc(ct->d_fpp_size, KM_SLEEP);
2684         while (ndid--) {

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

```

```

2817         int      error;
2819         off = (uintptr_t)dest & PAGEOFFSET;
2820         if (off)
2821             max = PAGE_SIZE - off;
2822         else
2823             max = PAGE_SIZE;
2824         amount = len > max ? max : len;
2825         error = door_copy(as, src, dest, amount);
2826         if (error != 0)
2827             return (error);
2828         dest += amount;
2829         src += amount;
2830         len -= amount;
2831     }
2832 }
2833
2835 /*
2836  * Copyin the returned door ids and translate them into door_node_t
2837  */
2838 if (desc_num != 0) {
2839     door_desc_t *start;
2840     door_desc_t *didpp;
2841     struct file **fpp;
2842     size_t fpp_size;
2843     uint_t i;
2844
2845     /* First, check if we would overflow client */
2846     if (!ufcanalloc(ttoproc.caller), desc_num)
2847         return (EMFILE);
2848
2849     start = didpp = kmem_alloc(dsize, KM_SLEEP);
2850     if (copyin_nowatch(desc_ptr, didpp, dsize)) {
2851         kmem_free(start, dsize);
2852         return (EFAULT);
2853     }
2854     fpp_size = desc_num * sizeof (struct file *);
2855     if (fpp_size > ct->d_fpp_size) {
2856         /* make more space */
2857         if (ct->d_fpp_size)
2858             kmem_free(ct->d_fpp, ct->d_fpp_size);
2859         ct->d_fpp_size = fpp_size;
2860         ct->d_fpp = kmem_alloc(fpp_size, KM_SLEEP);
2861     }
2862     fpp = ct->d_fpp;
2863
2864     for (i = 0; i < desc_num; i++) {
2865         struct file *fp;
2866         int fd = didpp->d_data.d_desc.d_descriptor;
2867
2868         /* Only understand file descriptor results */
2869         if (!(didpp->d_attributes & DOOR_DESCRIPTOR) ||
2870             (fp = getf(fd)) == NULL) {
2871             /* close translated references */
2872             door_fp_close(ct->d_fpp, fpp - ct->d_fpp);
2873             /* close untranslated references */
2874             door_fd_rele(didpp, desc_num - i, 0);
2875             kmem_free(start, dsize);
2876             return (EINVAL);
2877         }
2878
2879         mutex_enter(&fp->f_tlock);
2880         fp->f_count++;
2881         mutex_exit(&fp->f_tlock);

```

```

2883         *fpp = fp;
2884         releasef(fd);
2885
2886         if (didpp->d_attributes & DOOR_RELEASE) {
2887             /* release passed reference */
2888             (void) closeandsetf(fd, NULL);
2889         }
2890
2891         fpp++; didpp++;
2892     }
2893     kmem_free(start, dsize);
2894 }
2895     return (0);
2896 }
2897
2898 /*
2899  * Close all the descriptors.
2900  */
2901 static void
2902 door_fd_close(door_desc_t *d, uint_t n)
2903 {
2904     uint_t i;
2905
2906     ASSERT(MUTEX_NOT_HELD(&door_knob));
2907     for (i = 0; i < n; i++) {
2908         if (d->d_attributes & DOOR_DESCRIPTOR) {
2909             (void) closeandsetf(
2910                 d->d_data.d_desc.d_descriptor, NULL);
2911         } else if (d->d_attributes & DOOR_HANDLE) {
2912             door_ki_rele(d->d_data.d_handle);
2913         }
2914         d++;
2915     }
2916 }
2917
2918 /*
2919  * Close descriptors that have the DOOR_RELEASE attribute set.
2920  */
2921 void
2922 door_fd_rele(door_desc_t *d, uint_t n, int from_kernel)
2923 {
2924     uint_t i;
2925
2926     ASSERT(MUTEX_NOT_HELD(&door_knob));
2927     for (i = 0; i < n; i++) {
2928         if (d->d_attributes & DOOR_RELEASE) {
2929             if (d->d_attributes & DOOR_DESCRIPTOR) {
2930                 (void) closeandsetf(
2931                     d->d_data.d_desc.d_descriptor, NULL);
2932             } else if (d->d_attributes & DOOR_HANDLE) {
2933                 door_ki_rele(d->d_data.d_handle);
2934             }
2935         }
2936         d++;
2937     }
2938 }
2939
2940 /*
2941  * Copy descriptors into the kernel so we can release any marked
2942  * DOOR_RELEASE.
2943  */
2944 void
2945 door_release_fds(door_desc_t *desc_ptr, uint_t ndesc)
2946 {
2947     size_t dsize;

```

```

2949     door_desc_t *didpp;
2950     uint_t desc_num;

2952     ASSERT(MUTEX_NOT_HELD(&door_knob));
2953     ASSERT(ndesc != 0);

2955     desc_num = MIN(ndesc, door_max_desc);

2957     dsize = desc_num * sizeof (door_desc_t);
2958     didpp = kmem_alloc(dsize, KM_SLEEP);

2960     while (ndesc > 0) {
2961         uint_t count = MIN(ndesc, desc_num);

2963         if (copyin_nowatch(desc_ptr, didpp,
2964             count * sizeof (door_desc_t)) {
2965             kmem_free(didpp, dsize);
2966             return (EFAULT);
2967         }
2968         door_fd_rele(didpp, count, 0);

2970         ndesc -= count;
2971         desc_ptr += count;
2972     }
2973     kmem_free(didpp, dsize);
2974     return (0);
2975 }

2977 /*
2978  * Decrement ref count on all the files passed
2979  */
2980 static void
2981 door_fp_close(struct file **fp, uint_t n)
2982 {
2983     uint_t i;

2985     ASSERT(MUTEX_NOT_HELD(&door_knob));

2987     for (i = 0; i < n; i++)
2988         (void) closef(fp[i]);
2989 }

2991 /*
2992  * Copy data from 'src' in current address space to 'dest' in 'as' for 'len'
2993  * bytes.
2994  *
2995  * Performs this using 1 mapin and 1 copy operation.
2996  *
2997  * We really should do more than 1 page at a time to improve
2998  * performance, but for now this is treated as an anomalous condition.
2999  */
3000 static int
3001 door_copy(struct as *as, caddr_t src, caddr_t dest, uint_t len)
3002 {
3003     caddr_t kaddr;
3004     caddr_t rdest;
3005     uint_t off;
3006     page_t **pplist;
3007     page_t *pp = NULL;
3008     int error = 0;

3010     ASSERT(len <= PAGESIZE);
3011     off = (uintptr_t)dest & PAGEOFFSET; /* offset within the page */
3012     rdest = (caddr_t)((uintptr_t)dest &
3013         (uintptr_t)PAGEMASK); /* Page boundary */
3014     ASSERT(off + len <= PAGESIZE);

```

```

3016     /*
3017      * Lock down destination page.
3018      */
3019     if (as_pagelock(as, &pplist, rdest, PAGESIZE, S_WRITE))
3020         return (E2BIG);
3021     /*
3022      * Check if we have a shadow page list from as_pagelock. If not,
3023      * we took the slow path and have to find our page struct the hard
3024      * way.
3025      */
3026     if (pplist == NULL) {
3027         pfn_t pfnnum;

3029         /* MMU mapping is already locked down */
3030         AS_LOCK_ENTER(as, RW_READER);
3031         pfnnum = hat_getpfnnum(as->a_hat, rdest);
3032         AS_LOCK_EXIT(as);

3034         /*
3035          * TODO: The pfn step should not be necessary - need
3036          * a hat_getpp() function.
3037          */
3038         if (pf_is_memory(pfnnum)) {
3039             pp = page_numtopp_nolock(pfnnum);
3040             ASSERT(pp == NULL || PAGE_LOCKED(pp));
3041         } else
3042             pp = NULL;
3043         if (pp == NULL) {
3044             as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3045             return (E2BIG);
3046         }
3047     } else {
3048         pp = *pplist;
3049     }
3050     /*
3051      * Map destination page into kernel address
3052      */
3053     if (kpm_enable)
3054         kaddr = (caddr_t)hat_kpm_mapin(pp, (struct kpme *)NULL);
3055     else
3056         kaddr = (caddr_t)ppmapin(pp, PROT_READ | PROT_WRITE,
3057             (caddr_t)-1);

3059     /*
3060      * Copy from src to dest
3061      */
3062     if (copyin_nowatch(src, kaddr + off, len) != 0)
3063         error = EFAULT;
3064     /*
3065      * Unmap destination page from kernel
3066      */
3067     if (kpm_enable)
3068         hat_kpm_mapout(pp, (struct kpme *)NULL, kaddr);
3069     else
3070         ppmapout(kaddr);
3071     /*
3072      * Unlock destination page
3073      */
3074     as_pageunlock(as, pplist, rdest, PAGESIZE, S_WRITE);
3075     return (error);
3076 }

3078 /*
3079  * General kernel upcall using doors
3080  * Returns 0 on success, errno for failures.

```

```

3081 * Caller must have a hold on the door based vnode, and on any
3082 * references passed in desc_ptr. The references are released
3083 * in the event of an error, and passed without duplication
3084 * otherwise. Note that param->rbuf must be 64-bit aligned in
3085 * a 64-bit kernel, since it may be used to store door descriptors
3086 * if they are returned by the server. The caller is responsible
3087 * for holding a reference to the cred passed in.
3088 */
3089 int
3090 door_upcall(vnode_t *vp, door_arg_t *param, struct cred *cred,
3091            size_t max_data, uint_t max_descs)
3092 {
3093     /* Locals */
3094     door_upcall_t *dup;
3095     door_node_t *dp;
3096     kthread_t *server_thread;
3097     int error = 0;
3098     lwp_t *lwp;
3099     door_client_t *ct; /* curthread door_data */
3100     door_server_t *st; /* server thread door_data */
3101     int gotresults = 0;
3102     int cancel_pending;

3104     if (vp->v_type != VDOOR) {
3105         if (param->desc_num)
3106             door_fd_rele(param->desc_ptr, param->desc_num, 1);
3107         return (EINVAL);
3108     }

3110     lwp = ttolwp(curthread);
3111     ct = door_my_client(1);
3112     dp = VTOD(vp); /* Convert to a door_node_t */

3114     dup = kmem_zalloc(sizeof (*dup), KM_SLEEP);
3115     dup->du_cred = (cred != NULL) ? cred : curthread->t_cred;
3116     dup->du_max_data = max_data;
3117     dup->du_max_descs = max_descs;

3119     /*
3120     * This should be done in shuttle_resume(), just before going to
3121     * sleep, but we want to avoid overhead while holding door_knob.
3122     * prstop() is just a no-op if we don't really go to sleep.
3123     * We test not-kernel-address-space for the sake of clustering code.
3124     */
3125     if (lwp && lwp->lwp_nostop == 0 && curproc->p_as != &kas)
3126         prstop(PR_REQUESTED, 0);

3128     mutex_enter(&door_knob);
3129     if (DOOR_INVALID(dp)) {
3130         mutex_exit(&door_knob);
3131         if (param->desc_num)
3132             door_fd_rele(param->desc_ptr, param->desc_num, 1);
3133         error = EBADF;
3134         goto out;
3135     }

3137     if (dp->door_target == &p0) {
3138         /* Can't do an upcall to a kernel server */
3139         mutex_exit(&door_knob);
3140         if (param->desc_num)
3141             door_fd_rele(param->desc_ptr, param->desc_num, 1);
3142         error = EINVAL;
3143         goto out;
3144     }

3146     error = door_check_limits(dp, param, 1);

```

```

3147     if (error != 0) {
3148         mutex_exit(&door_knob);
3149         if (param->desc_num)
3150             door_fd_rele(param->desc_ptr, param->desc_num, 1);
3151         goto out;
3152     }

3154     /*
3155     * Get a server thread from the target domain
3156     */
3157     if ((server_thread = door_get_server(dp)) == NULL) {
3158         if (DOOR_INVALID(dp))
3159             error = EBADF;
3160         else
3161             error = EAGAIN;
3162         mutex_exit(&door_knob);
3163         if (param->desc_num)
3164             door_fd_rele(param->desc_ptr, param->desc_num, 1);
3165         goto out;
3166     }

3168     st = DOOR_SERVER(server_thread->t_door);
3169     ct->d_buf = param->data_ptr;
3170     ct->d_bufsize = param->data_size;
3171     ct->d_args = *param; /* structure assignment */

3173     if (ct->d_args.desc_num) {
3174         /*
3175         * Move data from client to server
3176         */
3177         DOOR_T_HOLD(st);
3178         mutex_exit(&door_knob);
3179         error = door_translate_out();
3180         mutex_enter(&door_knob);
3181         DOOR_T_RELEASE(st);
3182         if (error) {
3183             /*
3184             * We're not going to resume this thread after all
3185             */
3186             door_release_server(dp, server_thread);
3187             shuttle_sleep(server_thread);
3188             mutex_exit(&door_knob);
3189             goto out;
3190         }
3191     }

3193     ct->d_upcall = dup;
3194     if (param->rsize == 0)
3195         ct->d_noreresults = 1;
3196     else
3197         ct->d_noreresults = 0;

3199     dp->door_active++;

3201     ct->d_error = DOOR_WAIT;
3202     st->d_caller = curthread;
3203     st->d_active = dp;

3205     shuttle_resume(server_thread, &door_knob);

3207     mutex_enter(&door_knob);
3208     shuttle_return:
3209     if ((error = ct->d_error) < 0) { /* DOOR_WAIT or DOOR_EXIT */
3210         /*
3211         * Premature wakeup. Find out why (stop, forkall, sig, exit ...)
3212         */

```

```

3213     mutex_exit(&door_knob);          /* May block in ISSIG */
3214     cancel_pending = 0;
3215     if (lwp && (ISSIG(curthread, FORREAL) || lwp->lwp_sysabort ||
3216         MUSTRETURN(curproc, curthread) ||
3217         (cancel_pending = schedctl_cancel_pending()) != 0)) {
3218         /* Signal, forkall, ... */
3219         if (cancel_pending)
3220             schedctl_cancel_eintr();
3221         lwp->lwp_sysabort = 0;
3222         mutex_enter(&door_knob);
3223         error = EINTR;
3224         /*
3225          * If the server has finished processing our call,
3226          * or exited (calling door_slam()), then d_error
3227          * will have changed. If the server hasn't finished
3228          * yet, d_error will still be DOOR_WAIT, and we
3229          * let it know we are not interested in any
3230          * results by sending a SIGCANCEL, unless the door
3231          * is marked with DOOR_NO_CANCEL.
3232          */
3233         if (ct->d_error == DOOR_WAIT &&
3234             st->d_caller == curthread) {
3235             proc_t *p = ttoproc(server_thread);
3236
3237             st->d_active = NULL;
3238             st->d_caller = NULL;
3239             if (!(dp->door_flags & DOOR_NO_CANCEL)) {
3240                 DOOR_T_HOLD(st);
3241                 mutex_exit(&door_knob);
3242
3243                 mutex_enter(&p->p_lock);
3244                 sigtoproc(p, server_thread, SIGCANCEL);
3245                 mutex_exit(&p->p_lock);
3246
3247                 mutex_enter(&door_knob);
3248                 DOOR_T_RELEASE(st);
3249             }
3250         }
3251     } else {
3252         /*
3253          * Return from stop(), server exit...
3254          *
3255          * Note that the server could have done a
3256          * door_return while the client was in stop state
3257          * (ISSIG), in which case the error condition
3258          * is updated by the server.
3259          */
3260         mutex_enter(&door_knob);
3261         if (ct->d_error == DOOR_WAIT) {
3262             /* Still waiting for a reply */
3263             shuttle_swth(&door_knob);
3264             mutex_enter(&door_knob);
3265             if (lwp)
3266                 lwp->lwp_asleep = 0;
3267             goto shuttle_return;
3268         } else if (ct->d_error == DOOR_EXIT) {
3269             /* Server exit */
3270             error = EINTR;
3271         } else {
3272             /* Server did a door_return during ISSIG */
3273             error = ct->d_error;
3274         }
3275     }
3276     /*
3277     * Can't exit if the server is currently copying
3278     * results for me

```

```

3279     /*
3280     while (DOOR_T_HELD(ct))
3281         cv_wait(&ct->d_cv, &door_knob);
3282
3283     /*
3284     * Find out if results were successfully copied.
3285     */
3286     if (ct->d_error == 0)
3287         gotresults = 1;
3288     }
3289     if (lwp) {
3290         lwp->lwp_asleep = 0;          /* /proc */
3291         lwp->lwp_sysabort = 0;       /* /proc */
3292     }
3293     if (--dp->door_active == 0 && (dp->door_flags & DOOR_DELAY))
3294         door_deliver_unref(dp);
3295     mutex_exit(&door_knob);
3296
3297     /*
3298     * Translate returned doors (if any)
3299     */
3300     if (ct->d_noresults)
3301         goto out;
3302
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     }
3314
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;
3320
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;
3324
3325         while (n--) {
3326             struct file *fp;
3327
3328             fp = *fpp;
3329             if (VOP_REALVP(fp->f_vnode, &vp, NULL))
3330                 vp = fp->f_vnode;
3331
3332             didpp->d_attributes = DOOR_HANDLE |
3333                 (VTOD(vp)->door_flags & DOOR_ATTR_MASK);
3334             didpp->d_data.d_handle = FTODH(fp);
3335
3336             fpp++; didpp++;
3337         }
3338     }
3339
3340     /* on return data is in rbuf */
3341     *param = ct->d_args;          /* structure assignment */
3342
3343     out;

```

```

3345     kmem_free(dup, sizeof (*dup));
3347     if (ct->d_fpp) {
3348         kmem_free(ct->d_fpp, ct->d_fpp_size);
3349         ct->d_fpp = NULL;
3350         ct->d_fpp_size = 0;
3351     }
3353     ct->d_upcall = NULL;
3354     ct->d_noresults = 0;
3355     ct->d_buf = NULL;
3356     ct->d_bufsize = 0;
3357     return (error);
3358 }
3360 /*
3361  * Add a door to the per-process list of active doors for which the
3362  * process is a server.
3363  */
3364 static void
3365 door_list_insert(door_node_t *dp)
3366 {
3367     proc_t *p = dp->door_target;
3369     ASSERT(MUTEX_HELD(&door_knob));
3370     dp->door_list = p->p_door_list;
3371     p->p_door_list = dp;
3372 }
3374 /*
3375  * Remove a door from the per-process list of active doors.
3376  */
3377 void
3378 door_list_delete(door_node_t *dp)
3379 {
3380     door_node_t **pp;
3382     ASSERT(MUTEX_HELD(&door_knob));
3383     /*
3384      * Find the door in the list.  If the door belongs to another process,
3385      * it's OK to use p_door_list since that process can't exit until all
3386      * doors have been taken off the list (see door_exit).
3387      */
3388     pp = &(dp->door_target->p_door_list);
3389     while (*pp != dp)
3390         pp = &((*pp)->door_list);
3392     /* found it, take it off the list */
3393     *pp = dp->door_list;
3394 }
3397 /*
3398  * External kernel interfaces for doors.  These functions are available
3399  * outside the doorfs module for use in creating and using doors from
3400  * within the kernel.
3401  */
3403 /*
3404  * door_ki_upcall invokes a user-level door server from the kernel, with
3405  * the credentials associated with curthread.
3406  */
3407 int
3408 door_ki_upcall(door_handle_t dh, door_arg_t *param)
3409 {
3410     return (door_ki_upcall_limited(dh, param, NULL, SIZE_MAX, UINT_MAX));

```

```

3411 }
3413 /*
3414  * door_ki_upcall_limited invokes a user-level door server from the
3415  * kernel with the given credentials and reply limits.  If the "cred"
3416  * argument is NULL, uses the credentials associated with current
3417  * thread.  max_data limits the maximum length of the returned data (the
3418  * client will get E2BIG if they go over), and max_desc limits the
3419  * number of returned descriptors (the client will get EMFILE if they
3420  * go over).
3421  */
3422 int
3423 door_ki_upcall_limited(door_handle_t dh, door_arg_t *param, struct cred *cred,
3424     size_t max_data, uint_t max_desc)
3425 {
3426     file_t *fp = DHTOF(dh);
3427     vnode_t *realvp;
3429     if (VOP_REALVP(fp->f_vnode, &realvp, NULL))
3430         realvp = fp->f_vnode;
3431     return (door_upcall(realvp, param, cred, max_data, max_desc));
3432 }
3434 /*
3435  * Function call to create a "kernel" door server.  A kernel door
3436  * server provides a way for a user-level process to invoke a function
3437  * in the kernel through a door_call.  From the caller's point of
3438  * view, a kernel door server looks the same as a user-level one
3439  * (except the server pid is 0).  Unlike normal door calls, the
3440  * kernel door function is invoked via a normal function call in the
3441  * same thread and context as the caller.
3442  */
3443 int
3444 door_ki_create(void (*pc_cookie)(), void *data_cookie, uint_t attributes,
3445     door_handle_t *dhp)
3446 {
3447     int err;
3448     file_t *fp;
3450     /* no DOOR_PRIVATE */
3451     if ((attributes & ~DOOR_KI_CREATE_MASK) ||
3452         (attributes & (DOOR_UNREF | DOOR_UNREF_MULTTI)) ==
3453         (DOOR_UNREF | DOOR_UNREF_MULTTI))
3454         return (EINVAL);
3456     err = door_create_common(pc_cookie, data_cookie, attributes,
3457         1, NULL, &fp);
3458     if (err == 0 && (attributes & (DOOR_UNREF | DOOR_UNREF_MULTTI)) &&
3459         p0.p_unref_thread == 0) {
3460         /* need to create unref thread for process 0 */
3461         (void) thread_create(NULL, 0, door_unref_kernel, NULL, 0, &p0,
3462             TS_RUN, minclsyspri);
3463     }
3464     if (err == 0) {
3465         *dhp = FTODH(fp);
3466     }
3467     return (err);
3468 }
3470 void
3471 door_ki_hold(door_handle_t dh)
3472 {
3473     file_t *fp = DHTOF(dh);
3475     mutex_enter(&fp->f_tlock);
3476     fp->f_count++;

```

```

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

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

3485     (void) closef(fp);
3486 }

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

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

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

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

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

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

```

```

3543 }

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

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

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

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

```

```

*****
16755 Wed Jul 13 01:32:36 2016
new/usr/src/uts/common/fs/sockfs/sockcommon.c
XXX 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     avl_create(&so->so_pid_tree, pid_node_comparator, 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_tree_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     avl_destroy(&so->so_pid_tree);
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_tree_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_flowctrlrd = 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_tree_lock);
669     while ((pn = avl_first(&so->so_pid_tree)) != NULL) {
670         avl_remove(&so->so_pid_tree, pn);
671         kmem_free(pn, sizeof (*pn));
672     }
673     mutex_exit(&so->so_pid_tree_lock);
675 #endif /* ! codereview */
676     ASSERT(list_is_empty(&so->so_acceptq_list));
677     ASSERT(list_is_empty(&so->so_acceptq_defer));
678     ASSERT(!list_link_active(&so->so_acceptq_node));
680     ASSERT(so->so_rcv_queued == 0);
681     ASSERT(so->so_rcv_q_head == NULL);
682     ASSERT(so->so_rcv_q_last_head == NULL);
683     ASSERT(so->so_rcv_head == NULL);
684     ASSERT(so->so_rcv_last_head == NULL);
685 }
687 void
688 sonode_insert_pid(struct sonode *so, pid_t pid)
689 {
690     pid_node_t *pn, lookup_pn;
691     avl_index_t idx_pn;
693     lookup_pn.pn_pid = pid;
694     mutex_enter(&so->so_pid_tree_lock);
695     pn = avl_find(&so->so_pid_tree, &lookup_pn, &idx_pn);

```

```
697     if (pn != NULL) {
698         pn->pn_count++;
699     } else {
700         pn = kmem_zalloc(sizeof (*pn), KM_SLEEP);
701         pn->pn_pid = pid;
702         pn->pn_count = 1;
703         avl_insert(&so->so_pid_tree, pn, idx_pn);
704     }
705     mutex_exit(&so->so_pid_tree_lock);
706 }

708 void
709 sonode_remove_pid(struct sonode *so, pid_t pid)
710 {
711     pid_node_t *pn, lookup_pn;
712
713     lookup_pn.pn_pid = pid;
714     mutex_enter(&so->so_pid_tree_lock);
715     pn = avl_find(&so->so_pid_tree, &lookup_pn, NULL);
716
717     if (pn != NULL) {
718         if (pn->pn_count > 1) {
719             pn->pn_count--;
720         } else {
721             avl_remove(&so->so_pid_tree, pn);
722             kmem_free(pn, sizeof (*pn));
723         }
724     }
725     mutex_exit(&so->so_pid_tree_lock);
726 #endif /* ! codereview */
727 }
```

```

*****
9811 Wed Jul 13 01:32:37 2016
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_sendmblock_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, 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_sendmblock_impl(struct sonode *, struct nmsgHdr *, int,
139     struct cred *, mblk_t **, struct sof_instance *, boolean_t);
140 extern int so_sendmblock(struct sonode *, struct nmsgHdr *, int,
141     struct cred *, mblk_t **);
142 extern int so_rcvmsg(struct sonode *, struct nmsgHdr *, struct uio *,
143     struct cred *);
144 extern int so_shutdown(struct sonode *, int, struct cred *);
145 extern int so_close(struct sonode *, int, struct cred *);

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

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

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

```

```

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

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

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

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

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

193 extern mblk_t *socopyinuiouio(uio_t *, ssize_t, size_t, ssize_t, size_t, int *);
194 extern mblk_t *socopyoutuiouio(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 sorecvuob(struct sonode *, struct nmsgHdr *, struct uio *,
199     int, boolean_t);

201 extern void so_timer_callback(void *);

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

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

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

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

```

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

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

249 extern sonodeops_t so_sonodeops;
250 extern sock_upcalls_t so_upcalls;

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

```

*****
49290 Wed Jul 13 01:32:37 2016
new/usr/src/uts/common/fs/sockfs/sockcommon_sops.c
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

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

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

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

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

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

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

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

57 #include <sys/socket_proto.h>

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

```

```

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

64 extern int xnet_skip_checks;
65 extern int xnet_check_print;

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

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

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

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

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

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

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

116 /*
117  * Generic Socket Ops
118 */

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

127 int

```

```

128 so_bind(struct sonode *so, struct sockaddr *name, socklen_t namelen,
129         int flags, struct cred *cr)
130 {
131     int error;
132
133     SO_BLOCK_FALLBACK(so, SOP_BIND(so, name, namelen, flags, cr));
134
135     ASSERT(flags == _SOBIND_XPG4_2 || flags == _SOBIND_SOCKETBSD);
136
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     }
146
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;
153
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         }
162
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;
187
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     }
198
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 */
229
230     break;
231 }
232 default:
233     /* Just pass the request to the protocol */
234     goto dobind;
235 }
236
237 /*
238  * First we check if either NCA or KSSL has been enabled for
239  * the requested address, and if so, we fall back to TPI.
240  * If neither of those two services are enabled, then we just
241  * pass the request to the protocol.
242  *
243  * Note that KSSL can only be enabled on a socket if NCA is NOT
244  * enabled for that socket, hence the else-statement below.
245  */
246 if (nl7c_enabled && ((so->so_family == AF_INET ||
247                    so->so_family == AF_INET6) &&
248                    nl7c_lookup_addr(name, namelen) != NULL)) {
249     /*
250      * NL7C is not supported in non-global zones,
251      * we enforce this restriction here.
252      */
253     if (so->so_zoneid == GLOBAL_ZONEID) {
254         /* NCA should be used, so fall back to TPI */
255         error = so_tpi_fallback(so, cr);
256         SO_UNBLOCK_FALLBACK(so);
257         if (error)
258             return (error);
259     } else

```

```

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

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

273     return (error);
274 }

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

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

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

289     SO_UNBLOCK_FALLBACK(so);

291     return (error);
292 }

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

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

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

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

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

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

```

```

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

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

340     *nsop = NULL;

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

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

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

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

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

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

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

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

```

```

392     SO_UNBLOCK_FALLBACK(so);
393     return (EOPNOTSUPP);
394 }
395
396 if ((so->so_mode & SM_ATOMIC) &&
397     uiop->uio_resid > so->so_proto_props.sopp_maxpsz &&
398     so->so_proto_props.sopp_maxpsz != -1) {
399     SO_UNBLOCK_FALLBACK(so);
400     return (EMSGSIZE);
401 }
402
403 /*
404  * For atomic sends we will only do one iteration.
405  */
406 do {
407     if (so->so_state & SS_CANTSENDMORE) {
408         error = EPIPE;
409         break;
410     }
411
412     if (so->so_error != 0) {
413         mutex_enter(&so->so_lock);
414         error = sogeterr(so, B_TRUE);
415         mutex_exit(&so->so_lock);
416         if (error != 0)
417             break;
418     }
419
420     /*
421      * Send down OOB messages even if the send path is being
422      * flow controlled (assuming the protocol supports OOB data).
423      */
424     if (flags & MSG_OOB) {
425         if ((so->so_mode & SM_EXDATA) == 0) {
426             error = EOPNOTSUPP;
427             break;
428         }
429     } else if (SO_SND_FLOWCTRLD(so)) {
430         /*
431          * Need to wait until the protocol is ready to receive
432          * more data for transmission.
433          */
434         if ((error = so_snd_wait_qnotfull(so, dontblock)) != 0)
435             break;
436     }
437
438     /*
439      * Time to send data to the protocol. We either copy the
440      * data into mblks or pass the uio directly to the protocol.
441      * We decide what to do based on the available down calls.
442      */
443     if (so->so_downcalls->sd_send_uio != NULL) {
444         error = (*so->so_downcalls->sd_send_uio)
445             (so->so_proto_handle, uiop, msg, cr);
446         if (error != 0)
447             break;
448     } else {
449         /* save the resid in case of failure */
450         orig_resid = uiop->uio_resid;
451
452         if ((mp = socopyniuio(uiop,
453             so->so_proto_props.sopp_maxpsz,
454             so->so_proto_props.sopp_wroff,
455             so->so_proto_props.sopp_maxblk,
456             so->so_proto_props.sopp_tail, &error)) == NULL) {
457             break;

```

```

458     }
459     ASSERT(uiop->uio_resid >= 0);
460
461     if (so->so_filter_active > 0 &&
462         ((mp = SOF_FILTER_DATA_OUT(so, mp, msg, cr,
463             &error)) == NULL)) {
464         if (error != 0)
465             break;
466         continue;
467     }
468     error = (*so->so_downcalls->sd_send)
469         (so->so_proto_handle, mp, msg, cr);
470     if (error != 0) {
471         /*
472          * The send failed. We do not have to free the
473          * mblks, because that is the protocol's
474          * responsibility. However, uio_resid must
475          * remain accurate, so adjust that here.
476          */
477         uiop->uio_resid = orig_resid;
478         break;
479     }
480 } while (uiop->uio_resid > 0);
481
482 SO_UNBLOCK_FALLBACK(so);
483
484 return (error);
485 }
486
487 int
488 so_sendmblk_impl(struct sonode *so, struct nmsg_hdr *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;
496
497     if (so->so_downcalls->sd_send == NULL)
498         return (EOPNOTSUPP);
499
500     error = 0;
501     dontblock = (msg->msg_flags & MSG_DONTWAIT) ||
502         (fflag & (FNONBLOCK|FNDELAY));
503     size = msgdsize(mp);
504
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     }
511
512     while (mp != NULL) {
513         mblk_t *nmp, *last_mblk;
514         size_t mlen;
515
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

```

```

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

```

```

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

```

```

656 so_getsockname(struct sonode *so, struct sockaddr *addr,
657 socklen_t *addrlen, struct cred *cr)
658 {
659     int error;
661     SO_BLOCK_FALLBACK(so, SOP_GETSOCKNAME(so, addr, addrlen, cr));
663     if (so->so_filter_active == 0 ||
664         (error = sof_filter_getsockname(so, addr, addrlen, cr)) < 0)
665         error = (*so->so_downcalls->sd_getsockname)
666             (so->so_proto_handle, addr, addrlen, cr);
668     SO_UNBLOCK_FALLBACK(so);
669     return (error);
670 }
672 int
673 so_getpeername(struct sonode *so, struct sockaddr *addr,
674 socklen_t *addrlen, boolean_t accept, struct cred *cr)
675 {
676     int error;
678     SO_BLOCK_FALLBACK(so, SOP_GETPEERNAME(so, addr, addrlen, accept, cr));
680     if (accept) {
681         error = (*so->so_downcalls->sd_getpeername)
682             (so->so_proto_handle, addr, addrlen, cr);
683     } else if (!(so->so_state & SS_ISCONNECTED)) {
684         error = ENOTCONN;
685     } else if ((so->so_state & SS_CANTSENDMORE) && !xnet_skip_checks) {
686         /* Added this check for X/Open */
687         error = EINVAL;
688         if (xnet_check_print) {
689             printf("sockfs: X/Open getpeername check => EINVAL\n");
690         }
691     } else if (so->so_filter_active == 0 ||
692         (error = sof_filter_getpeername(so, addr, addrlen, cr)) < 0) {
693         error = (*so->so_downcalls->sd_getpeername)
694             (so->so_proto_handle, addr, addrlen, cr);
695     }
697     SO_UNBLOCK_FALLBACK(so);
698     return (error);
699 }
701 int
702 so_getsockopt(struct sonode *so, int level, int option_name,
703 void *optval, socklen_t *optlenp, int flags, struct cred *cr)
704 {
705     int error = 0;
707     if (level == SOL_FILTER)
708         return (sof_getsockopt(so, option_name, optval, optlenp, cr));
710     SO_BLOCK_FALLBACK(so,
711         SOP_GETSOCKOPT(so, level, option_name, optval, optlenp, flags, cr));
713     if ((so->so_filter_active == 0 ||
714         (error = sof_filter_getsockopt(so, level, option_name, optval,
715         optlenp, cr)) < 0) &&
716         (error = socket_getopt_common(so, level, option_name, optval,
717         optlenp, flags)) < 0) {
718         error = (*so->so_downcalls->sd_getsockopt)
719             (so->so_proto_handle, level, option_name, optval, optlenp,
720             cr);
721         if (error == ENOPROTOOPT) {

```

```

722         if (level == SOL_SOCKET) {
723             /*
724              * If a protocol does not support a particular
725              * socket option, set can fail (not allowed)
726              * but get can not fail. This is the previous
727              * sockfs behavior.
728              */
729             switch (option_name) {
730                 case SO_LINGER:
731                     if (*optlenp < (t_uscalar_t)
732                         sizeof (struct linger)) {
733                         error = EINVAL;
734                         break;
735                     }
736                     error = 0;
737                     bzero(optval, sizeof (struct linger));
738                     *optlenp = sizeof (struct linger);
739                     break;
740                 case SO_RCVTIMEO:
741                 case SO_SNDTIMEO:
742                     if (*optlenp < (t_uscalar_t)
743                         sizeof (struct timeval)) {
744                         error = EINVAL;
745                         break;
746                     }
747                     error = 0;
748                     bzero(optval, sizeof (struct timeval));
749                     *optlenp = sizeof (struct timeval);
750                     break;
751                 case SO_SND_BUFINFO:
752                     if (*optlenp < (t_uscalar_t)
753                         sizeof (struct so_snd_bufinfo)) {
754                         error = EINVAL;
755                         break;
756                     }
757                     error = 0;
758                     bzero(optval,
759                         sizeof (struct so_snd_bufinfo));
760                     *optlenp =
761                         sizeof (struct so_snd_bufinfo);
762                     break;
763                 case SO_DEBUG:
764                 case SO_REUSEADDR:
765                 case SO_KEEPAVIVE:
766                 case SO_DONTROUTE:
767                 case SO_BROADCAST:
768                 case SO_USELOOPBACK:
769                 case SO_OOBINLINE:
770                 case SO_DGRAM_ERRIND:
771                 case SO_SNDBUF:
772                 case SO_RCVBUF:
773                     error = 0;
774                     *((int32_t *)optval) = 0;
775                     *optlenp = sizeof (int32_t);
776                     break;
777                 default:
778                     break;
779             }
780         }
781     }
782 }
784     SO_UNBLOCK_FALLBACK(so);
785     return (error);
786 }

```

```

788 int
789 so_setsockopt(struct sonode *so, int level, int option_name,
790              const void *optval, socklen_t optlen, struct cred *cr)
791 {
792     int error = 0;
793     struct timeval tl;
794     const void *opt = optval;
795
796     if (level == SOL_FILTER)
797         return (sof_setsockopt(so, option_name, optval, optlen, cr));
798
799     SO_BLOCK_FALLBACK(so,
800                      SOP_SETSOCKOPT(so, level, option_name, optval, optlen, cr));
801
802     /* X/Open requires this check */
803     if (so->so_state & SS_CANTSENDMORE && !xnet_skip_checks) {
804         SO_UNBLOCK_FALLBACK(so);
805         if (xnet_check_print)
806             printf("sockfs: X/Open setsockopt check => EINVAL\n");
807         return (EINVAL);
808     }
809
810     if (so->so_filter_active > 0 &&
811         (error = sof_filter_setsockopt(so, level, option_name,
812                                       (void *)optval, &optlen, cr)) >= 0)
813         goto done;
814
815     if (level == SOL_SOCKET) {
816         switch (option_name) {
817             case SO_RCVTIMEO:
818             case SO_SNDTIMEO: {
819                 /*
820                  * We pass down these two options to protocol in order
821                  * to support some third part protocols which need to
822                  * know them. For those protocols which don't care
823                  * these two options, simply return 0.
824                  */
825                 clock_t t_usec;
826
827                 if (get_udatamodel() == DATAMODEL_NONE ||
828                     get_udatamodel() == DATAMODEL_NATIVE) {
829                     if (optlen != sizeof (struct timeval)) {
830                         error = EINVAL;
831                         goto done;
832                     }
833                     bcopy((struct timeval *)optval, &tl,
834                          sizeof (struct timeval));
835                 } else {
836                     if (optlen != sizeof (struct timeval32)) {
837                         error = EINVAL;
838                         goto done;
839                     }
840                     TIMEVAL32_TO_TIMEVAL(&tl,
841                                         (struct timeval32 *)optval);
842                 }
843                 opt = &tl;
844                 optlen = sizeof (tl);
845                 t_usec = tl.tv_sec * 1000 * 1000 + tl.tv_usec;
846                 mutex_enter(&so->so_lock);
847                 if (option_name == SO_RCVTIMEO)
848                     so->so_rcvtimeo = drv_usec2tohz(t_usec);
849                 else
850                     so->so_sndtimeo = drv_usec2tohz(t_usec);
851                 mutex_exit(&so->so_lock);
852                 break;
853             }

```

```

854         case SO_RCVBUF:
855             /*
856              * XXX XPG 4.2 applications retrieve SO_RCVBUF from
857              * sockfs since the transport might adjust the value
858              * and not return exactly what was set by the
859              * application.
860              */
861             so->so_xpg_rcvbuf = *(int32_t *)optval;
862             break;
863         }
864     }
865     error = (*so->so_downcalls->sd_setsockopt)
866           (so->so_proto_handle, level, option_name, opt, optlen, cr);
867 done:
868     SO_UNBLOCK_FALLBACK(so);
869     return (error);
870 }
871
872 int
873 so_ioctl(struct sonode *so, int cmd, intptr_t arg, int mode,
874          struct cred *cr, int32_t *rvalp)
875 {
876     int error = 0;
877
878     SO_BLOCK_FALLBACK(so, SOP_IOCTL(so, cmd, arg, mode, cr, rvalp));
879
880     /*
881      * If there is a pending error, return error
882      * This can happen if a non blocking operation caused an error.
883      */
884     if (so->so_error != 0) {
885         mutex_enter(&so->so_lock);
886         error = sogeterr(so, B_TRUE);
887         mutex_exit(&so->so_lock);
888         if (error != 0)
889             goto done;
890     }
891
892     /*
893      * calling stioctl can result in the socket falling back to TPI,
894      * if that is supported.
895      */
896     if ((so->so_filter_active == 0 ||
897         (error = sof_filter_ioctl(so, cmd, arg, mode,
898                                   rvalp, cr)) < 0) &&
899         (error = socket_ioctl_common(so, cmd, arg, mode, cr, rvalp)) < 0 &&
900         (error = socket_stioctl_common(so, cmd, arg, mode, cr, rvalp)) < 0) {
901         error = (*so->so_downcalls->sd_ioctl)(so->so_proto_handle,
902                                             cmd, arg, mode, rvalp, cr);
903     }
904
905 done:
906     SO_UNBLOCK_FALLBACK(so);
907
908     return (error);
909 }
910
911 int
912 so_poll(struct sonode *so, short events, int anyyet, short *reventsp,
913         struct pollhead **phpp)
914 {
915     int state = so->so_state, mask;
916     *reventsp = 0;
917
918     /*
919      * In sockets the errors are represented as input/output events

```

```

920  */
921  if (so->so_error != 0 &&
922      ((POLLIN|POLLRDNORM|POLLOUT) & events) != 0) {
923      *reventsp = (POLLIN|POLLRDNORM|POLLOUT) & events;
924      return (0);
925  }

927  /*
928  * If the socket is in a state where it can send data
929  * turn on POLLWRBAND and POLLOUT events.
930  */
931  if ((so->so_mode & SM_CONNREQUIRED) == 0 || (state & SS_ISCONNECTED)) {
932      /*
933      * out of band data is allowed even if the connection
934      * is flow controlled
935      */
936      *reventsp |= POLLWRBAND & events;
937      if (ISO_SND_FLOWCTRLD(so)) {
938          /*
939          * As long as there is buffer to send data
940          * turn on POLLOUT events
941          */
942          *reventsp |= POLLOUT & events;
943      }
944  }

946  /*
947  * Turn on POLLIN whenever there is data on the receive queue,
948  * or the socket is in a state where no more data will be received.
949  * Also, if the socket is accepting connections, flip the bit if
950  * there is something on the queue.
951  *
952  * We do an initial check for events without holding locks. However,
953  * if there are no event available, then we redo the check for POLLIN
954  * events under the lock.
955  */

957  /* Pending connections */
958  if (!list_is_empty(&so->so_acceptq_list))
959      *reventsp |= (POLLIN|POLLRDNORM) & events;

961  /*
962  * If we're looking for POLLRDHUP, indicate it if we have sent the
963  * last rx signal for the socket.
964  */
965  if ((events & POLLRDHUP) && (state & SS_SENTLASTREADSIG))
966      *reventsp |= POLLRDHUP;

968  /* Data */
969  /* so_downcalls is null for sctp */
970  if (so->so_downcalls != NULL && so->so_downcalls->sd_poll != NULL) {
971      *reventsp |= (*so->so_downcalls->sd_poll)
972      (so->so_proto_handle, events & SO_PROTO_POLLEV, anyyet,
973      CRED()) & events;
974      ASSERT((*reventsp & ~events) == 0);
975      /* do not recheck events */
976      events &= ~SO_PROTO_POLLEV;
977  } else {
978      if (SO_HAVE_DATA(so))
979          *reventsp |= (POLLIN|POLLRDNORM) & events;

981      /* Urgent data */
982      if ((state & SS_OOBPEND) != 0) {
983          *reventsp |= (POLLRDBAND | POLLPRI) & events;
984      }

```

```

986  /*
987  * If the socket has become disconnected, we set POLLHUP.
988  * Note that if we are in this state, we will have set POLLIN
989  * (SO_HAVE_DATA() is true on a disconnected socket), but not
990  * POLLOUT (SS_ISCONNECTED is false). This is in keeping with
991  * the semantics of POLLHUP, which is defined to be mutually
992  * exclusive with respect to POLLOUT but not POLLIN. We are
993  * therefore setting POLLHUP primarily for the benefit of
994  * those not polling on POLLIN, as they have no other way of
995  * knowing that the socket has been disconnected.
996  */
997  mask = SS_SENTLASTREADSIG | SS_SENTLASTWRITESIG;

999  if ((state & (mask | SS_ISCONNECTED)) == mask)
1000      *reventsp |= POLLHUP;
1001  }

1003  if ((*reventsp && !anyyet) || (events & POLLET)) {
1004      /* Check for read events again, but this time under lock */
1005      if (events & (POLLIN|POLLRDNORM)) {
1006          mutex_enter(&so->so_lock);
1007          if (SO_HAVE_DATA(so) ||
1008              !list_is_empty(&so->so_acceptq_list)) {
1009              if (events & POLLET) {
1010                  so->so_pollev |= SO_POLLEV_IN;
1011                  *phpp = &so->so_poll_list;
1012              }

1014              mutex_exit(&so->so_lock);
1015              *reventsp |= (POLLIN|POLLRDNORM) & events;

1017              return (0);
1018          } else {
1019              so->so_pollev |= SO_POLLEV_IN;
1020              mutex_exit(&so->so_lock);
1021          }
1022      }
1023      *phpp = &so->so_poll_list;
1024  }
1025  return (0);
1026  }

1028  /*
1029  * Generic Upcalls
1030  */
1031  void
1032  so_connected(sock_upper_handle_t sock_handle, sock_connid_t id,
1033              cred_t *peer_cred, pid_t peer_cpuid)
1034  {
1035      struct sonode *so = (struct sonode *)sock_handle;

1037      mutex_enter(&so->so_lock);
1038      ASSERT(so->so_proto_handle != NULL);

1040      if (peer_cred != NULL) {
1041          if (so->so_peercred != NULL)
1042              crfree(so->so_peercred);
1043          crhold(peer_cred);
1044          so->so_peercred = peer_cred;
1045          so->so_cpuid = peer_cpuid;
1046      }

1048      so->so_proto_connid = id;
1049      soisconnected(so);
1050      /*
1051      * Wake ones who're waiting for conn to become established.

```

```

1052     */
1053     so_notify_connected(so);
1054 }

1056 int
1057 so_disconnected(sock_upper_handle_t sock_handle, sock_connid_t id, int error)
1058 {
1059     struct sonode *so = (struct sonode *)sock_handle;
1060     boolean_t connect_failed;

1062     mutex_enter(&so->so_lock);

1064     /*
1065     * If we aren't currently connected, then this isn't a disconnect but
1066     * rather a failure to connect.
1067     */
1068     connect_failed = !(so->so_state & SS_ISCONNECTED);

1070     so->so_proto_connid = id;
1071     soisdisconnected(so, error);
1072     so_notify_disconnected(so, connect_failed, error);

1074     return (0);
1075 }

1077 void
1078 so_opctl(sock_upper_handle_t sock_handle, sock_opctl_action_t action,
1079          uintptr_t arg)
1080 {
1081     struct sonode *so = (struct sonode *)sock_handle;

1083     switch (action) {
1084     case SOCK_OPCTL_SHUT_SEND:
1085         mutex_enter(&so->so_lock);
1086         socantsendmore(so);
1087         so_notify_disconnecting(so);
1088         break;
1089     case SOCK_OPCTL_SHUT_RECV: {
1090         mutex_enter(&so->so_lock);
1091         socantrcvmore(so);
1092         so_notify_eof(so);
1093         break;
1094     }
1095     case SOCK_OPCTL_ENAB_ACCEPT:
1096         mutex_enter(&so->so_lock);
1097         so->so_state |= SS_ACCEPTCONN;
1098         so->so_backlog = (unsigned int)arg;
1099         /*
1100          * The protocol can stop generating newconn upcalls when
1101          * the backlog is full, so to make sure the listener does
1102          * not end up with a queue full of deferred connections
1103          * we reduce the backlog by one. Thus the listener will
1104          * start closing deferred connections before the backlog
1105          * is full.
1106          */
1107         if (so->so_filter_active > 0)
1108             so->so_backlog = MAX(1, so->so_backlog - 1);
1109         mutex_exit(&so->so_lock);
1110         break;
1111     default:
1112         ASSERT(0);
1113         break;
1114     }
1115 }

1117 void

```

```

1118 so_txq_full(sock_upper_handle_t sock_handle, boolean_t qfull)
1119 {
1120     struct sonode *so = (struct sonode *)sock_handle;

1122     if (qfull) {
1123         so_snd_qfull(so);
1124     } else {
1125         so_snd_qnotfull(so);
1126         mutex_enter(&so->so_lock);
1127         /* so_notify_writable drops so_lock */
1128         so_notify_writable(so);
1129     }
1130 }

1132 sock_upper_handle_t
1133 so_newconn(sock_upper_handle_t parenthandle,
1134            sock_lower_handle_t proto_handle, sock_downcalls_t *sock_downcalls,
1135            struct cred *peer_cred, pid_t peer_cpuid, sock_upcalls_t **sock_upcallsp)
1136 {
1137     struct sonode *so = (struct sonode *)parenthandle;
1138     struct sonode *nso;
1139     int error;

1141     ASSERT(proto_handle != NULL);

1143     if ((so->so_state & SS_ACCEPTCONN) == 0 ||
1144         (so->so_acceptq_len >= so->so_backlog &&
1145          (so->so_filter_active == 0 || !sof_sonode_drop_deferred(so)))) {
1146         return (NULL);
1147     }

1149     nso = socket_newconn(so, proto_handle, sock_downcalls, SOCKET_NOSLEEP,
1150                        &error);
1151     if (nso == NULL)
1152         return (NULL);

1154     if (peer_cred != NULL) {
1155         crhold(peer_cred);
1156         nso->so_peercred = peer_cred;
1157         nso->so_cpuid = peer_cpuid;
1158     }
1159     nso->so_listener = so;

1161     /*
1162     * The new socket (nso), proto_handle and sock_upcallsp are all
1163     * valid at this point. But as soon as nso is placed in the accept
1164     * queue that can no longer be assumed (since an accept() thread may
1165     * pull it off the queue and close the socket).
1166     */
1167     *sock_upcallsp = &so_upcalls;

1169     mutex_enter(&so->so_acceptq_lock);
1170     if (so->so_state & (SS_CLOSING|SS_FALLBACK_PENDING|SS_FALLBACK_COMP)) {
1171         mutex_exit(&so->so_acceptq_lock);
1172         ASSERT(nso->so_count == 1);
1173         nso->so_count--;
1174         nso->so_listener = NULL;
1175         /* drop proto ref */
1176         VN_RELE(SOTOV(nso));
1177         socket_destroy(nso);
1178         return (NULL);
1179     } else {
1180         so->so_acceptq_len++;
1181         if (nso->so_state & SS_FIL_DEFER) {
1182             list_insert_tail(&so->so_acceptq_defer, nso);
1183             mutex_exit(&so->so_acceptq_lock);

```

```

1184     } else {
1185         list_insert_tail(&so->so_acceptq_list, nso);
1186         cv_signal(&so->so_acceptq_cv);
1187         mutex_exit(&so->so_acceptq_lock);
1188         mutex_enter(&so->so_lock);
1189         so_notify_newconn(so);
1190     }
1191
1192     return ((sock_upper_handle_t)nso);
1193 }
1194 }
1195
1196 void
1197 so_set_prop(sock_upper_handle_t sock_handle, struct sock_proto_props *sopp)
1198 {
1199     struct sonode *so;
1200
1201     so = (struct sonode *)sock_handle;
1202
1203     mutex_enter(&so->so_lock);
1204
1205     if (sopp->sopp_flags & SOCKOPT_MAXBLK)
1206         so->so_proto_props.sopp_maxblk = sopp->sopp_maxblk;
1207     if (sopp->sopp_flags & SOCKOPT_WROFF)
1208         so->so_proto_props.sopp_wroff = sopp->sopp_wroff;
1209     if (sopp->sopp_flags & SOCKOPT_TAIL)
1210         so->so_proto_props.sopp_tail = sopp->sopp_tail;
1211     if (sopp->sopp_flags & SOCKOPT_RCVHIWAT)
1212         so->so_proto_props.sopp_rxhiwat = sopp->sopp_rxhiwat;
1213     if (sopp->sopp_flags & SOCKOPT_RCVLOWAT)
1214         so->so_proto_props.sopp_rxlwat = sopp->sopp_rxlwat;
1215     if (sopp->sopp_flags & SOCKOPT_MAXPSZ)
1216         so->so_proto_props.sopp_maxpsz = sopp->sopp_maxpsz;
1217     if (sopp->sopp_flags & SOCKOPT_MINPSZ)
1218         so->so_proto_props.sopp_minpsz = sopp->sopp_minpsz;
1219     if (sopp->sopp_flags & SOCKOPT_ZCOPY) {
1220         if (sopp->sopp_zcopyflag & ZCVMSAFE) {
1221             so->so_proto_props.sopp_zcopyflag |= STZCVMSAFE;
1222             so->so_proto_props.sopp_zcopyflag &= ~STZCVMUNSAFE;
1223         } else if (sopp->sopp_zcopyflag & ZCVMUNSAFE) {
1224             so->so_proto_props.sopp_zcopyflag |= STZCVMUNSAFE;
1225             so->so_proto_props.sopp_zcopyflag &= ~STZCVMSAFE;
1226         }
1227     }
1228     if (sopp->sopp_zcopyflag & COPYCACHED) {
1229         so->so_proto_props.sopp_zcopyflag |= STRCOPYCACHED;
1230     }
1231 }
1232 if (sopp->sopp_flags & SOCKOPT_OOBLINE)
1233     so->so_proto_props.sopp_oobinline = sopp->sopp_oobinline;
1234 if (sopp->sopp_flags & SOCKOPT_RCVTIMER)
1235     so->so_proto_props.sopp_rcvtimer = sopp->sopp_rcvtimer;
1236 if (sopp->sopp_flags & SOCKOPT_RCVTHRESH)
1237     so->so_proto_props.sopp_rcvthresh = sopp->sopp_rcvthresh;
1238 if (sopp->sopp_flags & SOCKOPT_MAXADDRLEN)
1239     so->so_proto_props.sopp_maxaddrlen = sopp->sopp_maxaddrlen;
1240 if (sopp->sopp_flags & SOCKOPT_LOOPBACK)
1241     so->so_proto_props.sopp_loopback = sopp->sopp_loopback;
1242
1243 mutex_exit(&so->so_lock);
1244
1245 if (so->so_filter_active > 0) {
1246     sof_instance_t *inst;
1247     ssize_t maxblk;
1248     ushort_t wroff, tail;
1249     maxblk = so->so_proto_props.sopp_maxblk;

```

```

1250     wroff = so->so_proto_props.sopp_wroff;
1251     tail = so->so_proto_props.sopp_tail;
1252     for (inst = so->so_filter_bottom; inst != NULL;
1253          inst = inst->sofi_prev) {
1254         if (SOF_INTERESTED(inst, mblk_prop)) {
1255             (*inst->sofi_ops->sofop_mblk_prop)(
1256                 (sof_handle_t)inst, inst->sofi_cookie,
1257                 &maxblk, &wroff, &tail);
1258         }
1259     }
1260     mutex_enter(&so->so_lock);
1261     so->so_proto_props.sopp_maxblk = maxblk;
1262     so->so_proto_props.sopp_wroff = wroff;
1263     so->so_proto_props.sopp_tail = tail;
1264     mutex_exit(&so->so_lock);
1265 }
1266 #ifdef DEBUG
1267     sopp->sopp_flags &= ~(SOCKOPT_MAXBLK | SOCKOPT_WROFF | SOCKOPT_TAIL |
1268         SOCKOPT_RCVHIWAT | SOCKOPT_RCVLOWAT | SOCKOPT_MAXPSZ |
1269         SOCKOPT_ZCOPY | SOCKOPT_OOBLINE | SOCKOPT_RCVTIMER |
1270         SOCKOPT_RCVTHRESH | SOCKOPT_MAXADDRLEN | SOCKOPT_MINPSZ |
1271         SOCKOPT_LOOPBACK);
1272     ASSERT(sopp->sopp_flags == 0);
1273 #endif
1274 }
1275
1276 /* ARGSUSED */
1277 ssize_t
1278 so_queue_msg_impl(struct sonode *so, mblk_t *mp,
1279     size_t msg_size, int flags, int *errorp, boolean_t *force_pushp,
1280     sof_instance_t *filter)
1281 {
1282     boolean_t force_push = B_TRUE;
1283     int space_left;
1284     sodirect_t *sodp = so->so_direct;
1285
1286     ASSERT(errorp != NULL);
1287     *errorp = 0;
1288     if (mp == NULL) {
1289         if (so->so_downcalls->sd_recv_uio != NULL) {
1290             mutex_enter(&so->so_lock);
1291             /* the notify functions will drop the lock */
1292             if (flags & MSG_OOB)
1293                 so_notify_oobdata(so, IS_SO_OOB_INLINE(so));
1294             else
1295                 so_notify_data(so, msg_size);
1296             return (0);
1297         }
1298         ASSERT(msg_size == 0);
1299         mutex_enter(&so->so_lock);
1300         goto space_check;
1301     }
1302
1303     ASSERT(mp->b_next == NULL);
1304     ASSERT(DB_TYPE(mp) == M_DATA || DB_TYPE(mp) == M_PROTO);
1305     ASSERT(msg_size == msgdsize(mp));
1306
1307     if (DB_TYPE(mp) == M_PROTO && !__TPI_PRIM_ISALIGNED(mp->b_rptr)) {
1308         /* The read pointer is not aligned correctly for TPI */
1309         zcmn_err(getzoneid(), CE_WARN,
1310             "sockfs: Unaligned TPI message received. rptr = %p\n",
1311             (void *)mp->b_rptr);
1312         freemsg(mp);
1313         mutex_enter(&so->so_lock);
1314         if (sodp != NULL)
1315             SOD_UIOAFINI(sodp);

```

```

1316         goto space_check;
1317     }

1319     if (so->so_filter_active > 0) {
1320         for (; filter != NULL; filter = filter->sofi_prev) {
1321             if (!SOF_INTERESTED(filter, data_in))
1322                 continue;
1323             mp = (*filter->sofi_ops->sofop_data_in)(
1324                 (sof_handle_t)filter, filter->sofi_cookie, mp,
1325                 flags, &msg_size);
1326             ASSERT(msgdsz(mp) == msg_size);
1327             DTRACE_PROBE2(filter_data, (sof_instance_t), filter,
1328                 (mblk_t *), mp);
1329             /* Data was consumed/dropped, just do space check */
1330             if (msg_size == 0) {
1331                 mutex_enter(&so->so_lock);
1332                 goto space_check;
1333             }
1334         }
1335     }

1337     if (flags & MSG_OOB) {
1338         so_queue_oob(so, mp, msg_size);
1339         mutex_enter(&so->so_lock);
1340         goto space_check;
1341     }

1343     if (force_pushp != NULL)
1344         force_push = *force_pushp;

1346     mutex_enter(&so->so_lock);
1347     if (so->so_state & (SS_FALLBACK_DRAIN | SS_FALLBACK_COMP)) {
1348         if (sodp != NULL)
1349             SOD_DISABLE(sodp);
1350         mutex_exit(&so->so_lock);
1351         *errorp = EOPNOTSUPP;
1352         return (-1);
1353     }
1354     if (so->so_state & (SS_CANTRCVMORE | SS_CLOSING)) {
1355         freemsg(mp);
1356         if (sodp != NULL)
1357             SOD_DISABLE(sodp);
1358         mutex_exit(&so->so_lock);
1359         return (0);
1360     }

1362     /* process the mblk via I/OAT if capable */
1363     if (sodp != NULL && sodp->sod_enabled) {
1364         if (DBL_TYPE(mp) == M_DATA) {
1365             sod_uioa_mblk_init(sodp, mp, msg_size);
1366         } else {
1367             SOD_UIOAFINI(sodp);
1368         }
1369     }

1371     if (mp->b_next == NULL) {
1372         so_enqueue_msg(so, mp, msg_size);
1373     } else {
1374         do {
1375             mblk_t *nmp;

1377             if ((nmp = mp->b_next) != NULL) {
1378                 mp->b_next = NULL;
1379             }
1380             so_enqueue_msg(so, mp, msgdsz(mp));
1381             mp = nmp;

```

```

1382         } while (mp != NULL);
1383     }

1385     space_left = so->so_rcvbuf - so->so_rcv_queued;
1386     if (space_left <= 0) {
1387         so->so_flowctrlld = B_TRUE;
1388         *errorp = ENOSPC;
1389         space_left = -1;
1390     }

1392     if (force_push || so->so_rcv_queued >= so->so_rcv_thresh ||
1393         so->so_rcv_queued >= so->so_rcv_wanted) {
1394         SOCKET_TIMER_CANCEL(so);
1395         /*
1396          * so_notify_data will release the lock
1397          */
1398         so_notify_data(so, so->so_rcv_queued);

1400         if (force_pushp != NULL)
1401             *force_pushp = B_TRUE;
1402         goto done;
1403     } else if (so->so_rcv_timer_tid == 0) {
1404         /* Make sure the rcv push timer is running */
1405         SOCKET_TIMER_START(so);
1406     }

1408 done_unlock:
1409     mutex_exit(&so->so_lock);
1410 done:
1411     return (space_left);

1413 space_check:
1414     space_left = so->so_rcvbuf - so->so_rcv_queued;
1415     if (space_left <= 0) {
1416         so->so_flowctrlld = B_TRUE;
1417         *errorp = ENOSPC;
1418         space_left = -1;
1419     }
1420     goto done_unlock;
1421 }

1423 #pragma inline(so_queue_msg_impl)

1425 ssize_t
1426 so_queue_msg(sock_upper_handle_t sock_handle, mblk_t *mp,
1427     size_t msg_size, int flags, int *errorp, boolean_t *force_pushp)
1428 {
1429     struct sonode *so = (struct sonode *)sock_handle;

1431     return (so_queue_msg_impl(so, mp, msg_size, flags, errorp, force_pushp,
1432         so->so_filter_bottom));
1433 }

1435 /*
1436  * Set the offset of where the oob data is relative to the bytes in
1437  * queued. Also generate SIGURG
1438  */
1439 void
1440 so_signal_oob(sock_upper_handle_t sock_handle, ssize_t offset)
1441 {
1442     struct sonode *so;

1444     ASSERT(offset >= 0);
1445     so = (struct sonode *)sock_handle;
1446     mutex_enter(&so->so_lock);
1447     if (so->so_direct != NULL)

```

```

1448         SOD_UIOAFINI(so->so_direct);
1450     /*
1451     * New urgent data on the way so forget about any old
1452     * urgent data.
1453     */
1454     so->so_state &= ~(SS_HAVEOOBDATA|SS_HADOOBDATA);
1456     /*
1457     * Record that urgent data is pending.
1458     */
1459     so->so_state |= SS_OOBPEND;
1461     if (so->so_oobmsg != NULL) {
1462         dprintso(so, 1, ("sock: discarding old oob\n"));
1463         freemsg(so->so_oobmsg);
1464         so->so_oobmsg = NULL;
1465     }
1467     /*
1468     * set the offset where the urgent byte is
1469     */
1470     so->so_oobmark = so->so_rcv_queued + offset;
1471     if (so->so_oobmark == 0)
1472         so->so_state |= SS_RCVATMARK;
1473     else
1474         so->so_state &= ~SS_RCVATMARK;
1476     so_notify_oobsig(so);
1477 }
1479 /*
1480 * Queue the OOB byte
1481 */
1482 static void
1483 so_queue_oob(struct sonode *so, mblk_t *mp, size_t len)
1484 {
1485     mutex_enter(&so->so_lock);
1486     if (so->so_direct != NULL)
1487         SOD_UIOAFINI(so->so_direct);
1489     ASSERT(mp != NULL);
1490     if (!IS_SO_OOB_INLINE(so)) {
1491         so->so_oobmsg = mp;
1492         so->so_state |= SS_HAVEOOBDATA;
1493     } else {
1494         so_enqueue_msg(so, mp, len);
1495     }
1497     so_notify_oobdata(so, IS_SO_OOB_INLINE(so));
1498 }
1500 int
1501 so_close(struct sonode *so, int flag, struct cred *cr)
1502 {
1503     int error;
1505     /*
1506     * No new data will be enqueued once the CLOSING flag is set.
1507     */
1508     mutex_enter(&so->so_lock);
1509     so->so_state |= SS_CLOSING;
1510     ASSERT(so_verify_oobstate(so));
1511     so_rcv_flush(so);
1512     mutex_exit(&so->so_lock);

```

```

1514     if (so->so_filter_active > 0)
1515         sof_sonode_closing(so);
1517     if (so->so_state & SS_ACCEPTCONN) {
1518         /*
1519         * We grab and release the accept lock to ensure that any
1520         * thread about to insert a socket in so_newconn completes
1521         * before we flush the queue. Any thread calling so_newconn
1522         * after we drop the lock will observe the SS_CLOSING flag,
1523         * which will stop it from inserting the socket in the queue.
1524         */
1525         mutex_enter(&so->so_acceptq_lock);
1526         mutex_exit(&so->so_acceptq_lock);
1528         so_acceptq_flush(so, B_TRUE);
1529     }
1531     error = (*so->so_downcalls->sd_close)(so->so_proto_handle, flag, cr);
1532     switch (error) {
1533     default:
1534         /* Protocol made a synchronous close; remove proto ref */
1535         VN_RELE(SOTOV(so));
1536         break;
1537     case EINPROGRESS:
1538         /*
1539         * Protocol is in the process of closing, it will make a
1540         * 'closed' upcall to remove the reference.
1541         */
1542         error = 0;
1543         break;
1544     }
1546     return (error);
1547 }
1549 /*
1550 * Upcall made by the protocol when it's doing an asynchronous close. It
1551 * will drop the protocol's reference on the socket.
1552 */
1553 void
1554 so_closed(sock_upper_handle_t sock_handle)
1555 {
1556     struct sonode *so = (struct sonode *)sock_handle;
1558     VN_RELE(SOTOV(so));
1559 }
1561 mblk_t *
1562 so_get_sock_pid_mblk(sock_upper_handle_t sock_handle)
1563 {
1564     ulong_t sz, n;
1565     mblk_t *mblk;
1566     pid_node_t *pn;
1567     pid_t *pids;
1568     conn_pid_info_t *cpi;
1569     struct sonode *so = (struct sonode *)sock_handle;
1571     mutex_enter(&so->so_pid_tree_lock);
1573     n = avl_numnodes(&so->so_pid_tree);
1574     sz = sizeof (conn_pid_info_t);
1575     sz += (n > 1) ? ((n - 1) * sizeof (pid_t)) : 0;
1576     if ((mblk = allocb(sz, BPRI_HI)) == NULL) {
1577         mutex_exit(&so->so_pid_tree_lock);
1578         return (NULL);
1579     }

```

```

1580     mblk->b_wptr += sz;
1581     cpi = (conn_pid_info_t *)mblk->b_datap->db_base;

1583     cpi->cpi_contents = CONN_PID_INFO_SOC;
1584     cpi->cpi_pids_cnt = n;
1585     cpi->cpi_tot_size = sz;
1586     cpi->cpi_pids[0] = 0;

1588     if (cpi->cpi_pids_cnt > 0) {
1589         pids = cpi->cpi_pids;
1590         for (pn = avl_first(&so->so_pid_tree); pn != NULL;
1591             pids++, pn = AVL_NEXT(&so->so_pid_tree, pn))
1592             *pids = pn->pn_pid;
1593     }
1594     mutex_exit(&so->so_pid_tree_lock);
1595     return (mblk);
1596 }

1598 #endif /* ! codereview */
1599 void
1600 so_zcopy_notify(sock_upper_handle_t sock_handle)
1601 {
1602     struct sonode *so = (struct sonode *)sock_handle;

1604     mutex_enter(&so->so_lock);
1605     so->so_copyflag |= STZCNOTIFY;
1606     cv_broadcast(&so->so_copy_cv);
1607     mutex_exit(&so->so_lock);
1608 }

1610 void
1611 so_set_error(sock_upper_handle_t sock_handle, int error)
1612 {
1613     struct sonode *so = (struct sonode *)sock_handle;

1615     mutex_enter(&so->so_lock);

1617     soseterror(so, error);

1619     so_notify_error(so);
1620 }

1622 /*
1623  * so_recvmmsg - read data from the socket
1624  *
1625  * There are two ways of obtaining data; either we ask the protocol to
1626  * copy directly into the supplied buffer, or we copy data from the
1627  * sonode's receive queue. The decision which one to use depends on
1628  * whether the protocol has a sd_rcv_uio down call.
1629  */
1630 int
1631 so_recvmmsg(struct sonode *so, struct nmsg_hdr *msg, struct uio *uiop,
1632             struct cred *cr)
1633 {
1634     rval_t      rval;
1635     int         flags = 0;
1636     t_uscalar_t ctrlollen, namelen;
1637     int         error = 0;
1638     int         ret;
1639     mblk_t      *mctlp = NULL;
1640     union T_primitives *tpr;
1641     void        *control;
1642     ssize_t     saved_resid;
1643     struct uio  *suiop;

1644     SO_BLOCK_FALLBACK(so, SOP_RECVMMSG(so, msg, uiop, cr));

```

```

1647     if ((so->so_state & (SS_ISCONNECTED|SS_CANTRCVMORE)) == 0 &&
1648         (so->so_mode & SM_CONNREQUIRED)) {
1649         SO_UNBLOCK_FALLBACK(so);
1650         return (ENOTCONN);
1651     }

1653     if (msg->msg_flags & MSG_PEEK)
1654         msg->msg_flags &= ~MSG_WAITALL;

1656     if (so->so_mode & SM_ATOMIC)
1657         msg->msg_flags |= MSG_TRUNC;

1659     if (msg->msg_flags & MSG_OOB) {
1660         if ((so->so_mode & SM_EXDATA) == 0) {
1661             error = EOPNOTSUPP;
1662         } else if (so->so_downcalls->sd_rcv_uio != NULL) {
1663             error = (*so->so_downcalls->sd_rcv_uio)
1664                 (so->so_proto_handle, uiop, msg, cr);
1665         } else {
1666             error = sorecvoob(so, msg, uiop, msg->msg_flags,
1667                 IS_SO_OOB_INLINE(so));
1668         }
1669         SO_UNBLOCK_FALLBACK(so);
1670         return (error);
1671     }

1673     /*
1674     * If the protocol has the rcv down call, then pass the request
1675     * down.
1676     */
1677     if (so->so_downcalls->sd_rcv_uio != NULL) {
1678         error = (*so->so_downcalls->sd_rcv_uio)
1679             (so->so_proto_handle, uiop, msg, cr);
1680         SO_UNBLOCK_FALLBACK(so);
1681         return (error);
1682     }

1684     /*
1685     * Reading data from the socket buffer
1686     */
1687     flags = msg->msg_flags;
1688     msg->msg_flags = 0;

1690     /*
1691     * Set msg_ctrlollen and msg_namelen to zero here to make it
1692     * simpler in the cases that no control or name is returned.
1693     */
1694     ctrlollen = msg->msg_ctrlollen;
1695     namelen = msg->msg_namelen;
1696     msg->msg_ctrlollen = 0;
1697     msg->msg_namelen = 0;

1699     mutex_enter(&so->so_lock);
1700     /* Set SOREADLOCKED */
1701     error = so_lock_read_intr(so,
1702         uiop->uio_fmode | ((flags & MSG_DONTWAIT) ? FNONBLOCK : 0));
1703     mutex_exit(&so->so_lock);
1704     if (error) {
1705         SO_UNBLOCK_FALLBACK(so);
1706         return (error);
1707     }

1709     suiop = sod_rcv_init(so, flags, &uiop);
1710     retry:
1711     saved_resid = uiop->uio_resid;

```

```

1712 error = so_dequeue_msg(so, &mctlp, uiop, &rval, flags);
1713 if (error != 0) {
1714     goto out;
1715 }
1716 /*
1717  * For datagrams the MOREDATA flag is used to set MSG_TRUNC.
1718  * For non-datagrams MOREDATA is used to set MSG_EOR.
1719  */
1720 ASSERT(!(rval.r_vall & MORECTL));
1721 if ((rval.r_vall & MOREDATA) && (so->so_mode & SM_ATOMIC))
1722     msg->msg_flags |= MSG_TRUNC;
1723 if (mctlp == NULL) {
1724     dprintso(so, 1, ("so_recvmg: got M_DATA\n"));
1725
1726     mutex_enter(&so->so_lock);
1727     /* Set MSG_EOR based on MOREDATA */
1728     if (!(rval.r_vall & MOREDATA)) {
1729         if (so->so_state & SS_SAVEDDEOR) {
1730             msg->msg_flags |= MSG_EOR;
1731             so->so_state &= ~SS_SAVEDDEOR;
1732         }
1733     }
1734     /*
1735      * If some data was received (i.e. not EOF) and the
1736      * read/recv* has not been satisfied wait for some more.
1737      */
1738     if ((flags & MSG_WAITALL) && !(msg->msg_flags & MSG_EOR) &&
1739         uiop->uio_resid != saved_resid && uiop->uio_resid > 0) {
1740         mutex_exit(&so->so_lock);
1741         flags |= MSG_NOMARK;
1742         goto retry;
1743     }
1744
1745     goto out_locked;
1746 }
1747 /* so_queue_msg has already verified length and alignment */
1748 tpr = (union T_primitives *)mctlp->b_rptr;
1749 dprintso(so, 1, ("so_recvmg: type %d\n", tpr->type));
1750 switch (tpr->type) {
1751 case T_DATA_IND: {
1752     /*
1753      * Set msg_flags to MSG_EOR based on
1754      * MORE flag and MOREDATA.
1755      */
1756     mutex_enter(&so->so_lock);
1757     so->so_state &= ~SS_SAVEDDEOR;
1758     if (!(tpr->data_ind.MORE_flag & 1)) {
1759         if (!(rval.r_vall & MOREDATA))
1760             msg->msg_flags |= MSG_EOR;
1761         else
1762             so->so_state |= SS_SAVEDDEOR;
1763     }
1764     freemsg(mctlp);
1765     /*
1766      * If some data was received (i.e. not EOF) and the
1767      * read/recv* has not been satisfied wait for some more.
1768      */
1769     if ((flags & MSG_WAITALL) && !(msg->msg_flags & MSG_EOR) &&
1770         uiop->uio_resid != saved_resid && uiop->uio_resid > 0) {
1771         mutex_exit(&so->so_lock);
1772         flags |= MSG_NOMARK;
1773         goto retry;
1774     }
1775     goto out_locked;
1776 }
1777 case T_UNITDATA_IND: {

```

```

1778 void *addr;
1779 t_uscalar_t addrlen;
1780 void *abuf;
1781 t_uscalar_t optlen;
1782 void *opt;
1783
1784 if (namelen != 0) {
1785     /* Caller wants source address */
1786     addrlen = tpr->unitdata_ind.SRC_length;
1787     addr = sogetoff(mctlp, tpr->unitdata_ind.SRC_offset,
1788         addrlen, 1);
1789     if (addr == NULL) {
1790         freemsg(mctlp);
1791         error = EPROTO;
1792         eprintsoline(so, error);
1793         goto out;
1794     }
1795     ASSERT(so->so_family != AF_UNIX);
1796 }
1797 optlen = tpr->unitdata_ind.OPT_length;
1798 if (optlen != 0) {
1799     t_uscalar_t ncontrollen;
1800
1801     /*
1802      * Extract any source address option.
1803      * Determine how large cmsg buffer is needed.
1804      */
1805     opt = sogetoff(mctlp, tpr->unitdata_ind.OPT_offset,
1806         optlen, __TPI_ALIGN_SIZE);
1807
1808     if (opt == NULL) {
1809         freemsg(mctlp);
1810         error = EPROTO;
1811         eprintsoline(so, error);
1812         goto out;
1813     }
1814     if (so->so_family == AF_UNIX)
1815         so_getopt_srcaddr(opt, optlen, &addr, &addrlen);
1816     ncontrollen = so_cmsglen(mctlp, opt, optlen,
1817         !(flags & MSG_XPG4_2));
1818     if (controllen != 0)
1819         controllen = ncontrollen;
1820     else if (ncontrollen != 0)
1821         msg->msg_flags |= MSG_CTRUNC;
1822 } else {
1823     controllen = 0;
1824 }
1825
1826 if (namelen != 0) {
1827     /*
1828      * Return address to caller.
1829      * Caller handles truncation if length
1830      * exceeds msg_namelen.
1831      * NOTE: AF_UNIX NUL termination is ensured by
1832      * the sender's copyin_name().
1833      */
1834     abuf = kmem_alloc(addrlen, KM_SLEEP);
1835
1836     bcopy(addr, abuf, addrlen);
1837     msg->msg_name = abuf;
1838     msg->msg_namelen = addrlen;
1839 }
1840
1841 if (controllen != 0) {
1842     /*
1843      * Return control msg to caller.

```

```

1844     * Caller handles truncation if length
1845     * exceeds msg_controllen.
1846     */
1847     control = kmem_zalloc(controllen, KM_SLEEP);

1849     error = so_opt2cmsg(mctlp, opt, optlen,
1850     !(flags & MSG_XPG4_2), control, controllen);
1851     if (error) {
1852         freemsg(mctlp);
1853         if (msg->msg_namelen != 0)
1854             kmem_free(msg->msg_name,
1855             msg->msg_namelen);
1856         kmem_free(control, controllen);
1857         eprintsoline(so, error);
1858         goto out;
1859     }
1860     msg->msg_control = control;
1861     msg->msg_controllen = controllen;
1862 }

1864     freemsg(mctlp);
1865     goto out;
1866 }
1867 case T_OPTDATA_IND: {
1868     struct T_optdata_req *tdr;
1869     void *opt;
1870     t_uscalar_t optlen;

1872     tdr = (struct T_optdata_req *)mctlp->b_rptr;
1873     optlen = tdr->OPT_length;
1874     if (optlen != 0) {
1875         t_uscalar_t ncontrollen;
1876         /*
1877          * Determine how large cmsg buffer is needed.
1878          */
1879         opt = sogetoff(mctlp,
1880         tpr->optdata_ind.OPT_offset, optlen,
1881         _TPI_ALIGN_SIZE);

1883         if (opt == NULL) {
1884             freemsg(mctlp);
1885             error = EPROTO;
1886             eprintsoline(so, error);
1887             goto out;
1888         }

1890         ncontrollen = so_cmsglen(mctlp, opt, optlen,
1891         !(flags & MSG_XPG4_2));
1892         if (controllen != 0)
1893             controllen = ncontrollen;
1894         else if (ncontrollen != 0)
1895             msg->msg_flags |= MSG_CTRUNC;
1896     } else {
1897         controllen = 0;
1898     }

1900     if (controllen != 0) {
1901         /*
1902          * Return control msg to caller.
1903          * Caller handles truncation if length
1904          * exceeds msg_controllen.
1905          */
1906         control = kmem_zalloc(controllen, KM_SLEEP);

1908         error = so_opt2cmsg(mctlp, opt, optlen,
1909         !(flags & MSG_XPG4_2), control, controllen);

```

```

1910         if (error) {
1911             freemsg(mctlp);
1912             kmem_free(control, controllen);
1913             eprintsoline(so, error);
1914             goto out;
1915         }
1916         msg->msg_control = control;
1917         msg->msg_controllen = controllen;
1918     }

1920     /*
1921     * Set msg_flags to MSG_EOR based on
1922     * DATA flag and MOREDATA.
1923     */
1924     mutex_enter(&so->so_lock);
1925     so->so_state &= ~SS_SAVEDDEOR;
1926     if (!(tpr->data_ind.MORE_flag & 1)) {
1927         if (!(rval.r_vall & MOREDATA))
1928             msg->msg_flags |= MSG_EOR;
1929         else
1930             so->so_state |= SS_SAVEDDEOR;
1931     }
1932     freemsg(mctlp);
1933     /*
1934     * If some data was received (i.e. not EOF) and the
1935     * read/recv* has not been satisfied wait for some more.
1936     * Not possible to wait if control info was received.
1937     */
1938     if ((flags & MSG_WAITALL) && !(msg->msg_flags & MSG_EOR) &&
1939     controllen == 0 &&
1940     uiop->uio_resid != saved_resid && uiop->uio_resid > 0) {
1941         mutex_exit(&so->so_lock);
1942         flags |= MSG_NOMARK;
1943         goto retry;
1944     }
1945     goto out_locked;
1946 }
1947 default:
1948     cmn_err(CE_CONT, "so_recvmmsg bad type %x \n",
1949     tpr->type);
1950     freemsg(mctlp);
1951     error = EPROTO;
1952     ASSERT(0);
1953 }

1954 out:
1955     mutex_enter(&so->so_lock);
1956 out_locked:
1957     ret = sod_rcv_done(so, suiop, uiop);
1958     if (ret != 0 && error == 0)
1959         error = ret;

1961     so_unlock_read(so); /* Clear SOREADLOCKED */
1962     mutex_exit(&so->so_lock);

1964     SO_UNBLOCK_FALLBACK(so);

1966     return (error);
1967 }

1969 sonodeops_t so_sonodeops = {
1970     so_init, /* sop_init */
1971     so_accept, /* sop_accept */
1972     so_bind, /* sop_bind */
1973     so_listen, /* sop_listen */
1974     so_connect, /* sop_connect */
1975     so_recvmmsg, /* sop_recvmmsg */

```

```
1976     so_sendmsg,          /* sop_sendmsg */
1977     so_sendmbk,          /* sop_sendmbk */
1978     so_getpeername,     /* sop_getpeername */
1979     so_getsockname,     /* sop_getsockname */
1980     so_shutdown,       /* sop_shutdown */
1981     so_getsockopt,     /* sop_getsockopt */
1982     so_setsockopt,     /* sop_setsockopt */
1983     so_ioctl,         /* sop_ioctl */
1984     so_poll,          /* sop_poll */
1985     so_close,         /* sop_close */
1986 };
```

```
1988 sock_upcalls_t so_upcalls = {
1989     so_newconn,
1990     so_connected,
1991     so_disconnected,
1992     so_opctl,
1993     so_queue_msg,
1994     so_set_prop,
1995     so_txq_full,
1996     so_signal_oob,
1997     so_zcopy_notify,
1998     so_set_error,
1999     so_closed,
2000     so_get_sock_pid_mblk
    40     so_closed
2001 };
_____unchanged_portion_omitted_
```

```

*****
12990 Wed Jul 13 01:32:37 2016
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     if (!(curproc->p_flag & SSYS))
127         sonode_insert_pid(so, curproc->p_pidp->pid_id);

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

133     return (0);
134 }

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

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

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

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

153     if (count > 1) {
154         dprint(2, ("socket_vop_close: count %d\n", count));
155         return (0);
156     }

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

```

```

167     }
169     return (error);
170 }

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

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

183     return (socket_recvmmsg(so, &lmsg, uiop, cr));
184 }

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

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

197     if (!(so->so_mode & SM_BYTESTREAM)) {
198         /*
199          * If the socket is not byte stream set MSG_EOR
200          */
201         lmsg.msg_flags = MSG_EOR;
202     }

204     return (socket_sendmmsg(so, &lmsg, uiop, cr));
205 }

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

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

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

224     case F_DASSOC_PID:
225         if (cr != kcred)
226             return (EPERM);
227         if (!(curproc->p_flag & SSYS))
228             sonode_remove_pid(so, (pid_t)arg);
229         return (0);
230     }
231 #endif /* ! codereview */

```

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

```

```

365     vap->va_seq = 0;
367     return (0);
368 }

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

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

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

403     return (0);
404 }

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

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

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

```

```

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

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

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

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

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

468     ASSERT(!vn_has_cached_data(vp));

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

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

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

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

```

```
497 {  
498     struct sonode *so = VTOSO(vp);  
500     ASSERT(vp->v_type == VSOCK);  
502     return (socket_poll(so, events, anyyet, reventsp, phpp));  
503 }
```

new/usr/src/uts/common/fs/sockfs/socksubr.c

1

```
*****
50096 Wed Jul 13 01:32:37 2016
new/usr/src/uts/common/fs/sockfs/socksubr.c
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
22 /*
23  * Copyright (c) 1995, 2010, Oracle and/or its affiliates. All rights reserved.
24 */
26 #include <sys/types.h>
27 #include <sys/t_lock.h>
28 #include <sys/param.h>
29 #include <sys/system.h>
30 #include <sys/buf.h>
31 #include <sys/conf.h>
32 #include <sys/cred.h>
33 #include <sys/kmem.h>
34 #include <sys/sysmacros.h>
35 #include <sys/vfs.h>
36 #include <sys/vfs_opreg.h>
37 #include <sys/vnode.h>
38 #include <sys/debug.h>
39 #include <sys/errno.h>
40 #include <sys/time.h>
41 #include <sys/file.h>
42 #include <sys/open.h>
43 #include <sys/user.h>
44 #include <sys/termios.h>
45 #include <sys/stream.h>
46 #include <sys/strsubr.h>
47 #include <sys/strsun.h>
48 #include <sys/esunddi.h>
49 #include <sys/flock.h>
50 #include <sys/modctl.h>
51 #include <sys/cmn_err.h>
52 #include <sys/mkdev.h>
53 #include <sys/pathname.h>
54 #include <sys/ddi.h>
55 #include <sys/stat.h>
56 #include <sys/fs/snnode.h>
57 #include <sys/fs/dv_node.h>
58 #include <sys/zone.h>
60 #include <sys/socket.h>
61 #include <sys/socketvar.h>
```

new/usr/src/uts/common/fs/sockfs/socksubr.c

2

```
62 #include <netinet/in.h>
63 #include <sys/un.h>
64 #include <sys/ucred.h>
66 #include <sys/tiuser.h>
67 #define _SUN_TPI_VERSION 2
68 #include <sys/tihdr.h>
70 #include <c2/audit.h>
72 #include <fs/sockfs/nl7c.h>
73 #include <fs/sockfs/sockcommon.h>
74 #include <fs/sockfs/sockfilter_impl.h>
75 #include <fs/sockfs/socktpi.h>
76 #include <fs/sockfs/socktpi_impl.h>
77 #include <fs/sockfs/sodirect.h>
79 /*
80  * Macros that operate on struct cmsghdr.
81  * The MSG_VALID macro does not assume that the last option buffer is padded.
82  */
83 #define MSG_CONTENT(msg) (&((msg)[1]))
84 #define MSG_CONTENTLEN(msg) ((msg)->cmsgh_len - sizeof (struct cmsghdr))
85 #define MSG_VALID(msg, start, end) \
86     (ISALIGNED_cmsghdr(msg) && \
87     ((uintptr_t)(msg) >= (uintptr_t)(start)) && \
88     ((uintptr_t)(msg) < (uintptr_t)(end)) && \
89     ((ssize_t)(msg)->cmsgh_len >= sizeof (struct cmsghdr)) && \
90     ((uintptr_t)(msg) + (msg)->cmsgh_len <= (uintptr_t)(end)))
91 #define SO_LOCK_WAKEUP_TIME 3000 /* Wakeup time in milliseconds */
93 dev_t sockdev; /* For fsid in getattr */
94 int sockfs_defer_nl7c_init = 0;
96 struct socklist socklist;
98 struct kmem_cache *socket_cache;
100 /*
101  * sockconf_lock protects the socket configuration (socket types and
102  * socket filters) which is changed via the sockconfig system call.
103  */
104 krwlock_t sockconf_lock;
106 static int sockfs_update(kstat_t *, int);
107 static int sockfs_snapshot(kstat_t *, void *, int);
108 extern smod_info_t *sotpi_smod_create(void);
110 extern void sendfile_init();
112 extern void nl7c_init(void);
114 extern int modrootloaded;
116 #define ADRSTRLEN (2 * sizeof (void *) + 1)
117 /*
118  * kernel structure for passing the sockinfo data back up to the user.
119  * the strings array allows us to convert AF_UNIX addresses into strings
120  * with a common method regardless of which n-bit kernel we're running.
121  */
122 struct k_sockinfo {
123     struct sockinfo ks_si;
124     char ks_straddr[3][ADRSTRLEN];
125 };
116 /*
```

```

117 * Translate from a device pathname (e.g. "/dev/tcp") to a vnode.
118 * Returns with the vnode held.
119 */
120 int
121 sogetvp(char *devpath, vnode_t **vpp, int uioflag)
122 {
123     struct snode *csp;
124     vnode_t *vp, *dvp;
125     major_t maj;
126     int error;
127
128     ASSERT(uioflag == UIO_SYSSPACE || uioflag == UIO_USERSPACE);
129
130     /*
131      * Lookup the underlying filesystem vnode.
132      */
133     error = lookupname(devpath, uioflag, FOLLOW, NULLVPP, &vp);
134     if (error)
135         return (error);
136
137     /* Check that it is the correct vnode */
138     if (vp->v_type != VCHR) {
139         VN_RELE(vp);
140         return (ENOTSOCK);
141     }
142
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     }
163
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     }
170
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;
729
730     dprint(1, ("fdbuf_extract: %d fds, len %d\n",
731             fdbuf->fd_numfd, rightslen));
732
733     numfd = fdbuf->fd_numfd;
734     ASSERT(rightslen == numfd * (int)sizeof (int));
735
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
759         /*
760          * Add the current pid to the list associated with this
761          * descriptor.
762          */
763         if (fp->f_vnode != NULL)
764             (void) VOP_IOCTL(fp->f_vnode, F_ASSOCI_PID,
765                 (intptr_t)curproc->p_pidp->pid_id, FKIOCTL, kcred,
766                 NULL, NULL);
767
768     #endif /* ! codereview */
769     if (AU_AUDITING())
770         audit_fdrecv(fd, fp);
771     dprint(1, ("fdbuf_extract: [%d] = %d, %p refcnt %d\n",
772             i, fd, (void *)fp, fp->f_count));
773 }
774     return (0);
775
776 cleanup:
777     /*
778      * Undo whatever partial work the loop above has done.
779      */
780     {
781         int j;
782
783         rp = (int *)rights;
784         for (j = 0; j < i; j++) {
785             dprint(0,
786                 ("fdbuf_extract: cleanup[%d] = %d\n", j, *rp));
787             (void) closeandsetf(*rp++, NULL);
788         }
789     }

```

```

791     return (EMFILE);
792 }

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

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

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

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

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

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

```

```

857     return ((int)FDBUF_HDRSIZE + (numfd * (int)sizeof (struct file *)));
858 }

860 static t_uscalar_t
861 fdbuf_cmsglen(int fdbuflen)
862 {
863     return (t_uscalar_t)((fdbuflen - FDBUF_HDRSIZE) /
864         (int)sizeof (struct file *) * (int)sizeof (int));
865 }

868 /*
869  * Return non-zero if the mblk and fdbuf are consistent.
870  */
871 static int
872 fdbuf_verify(mblk_t *mp, struct fdbuf *fdbuf, int fdbuflen)
873 {
874     if (fdbuflen >= FDBUF_HDRSIZE &&
875         fdbuflen == fdbuf->fd_size) {
876         frtn_t *frp = mp->b_datap->db_frtnp;
877         /*
878          * Check that the SO_FILEP portion of the
879          * message has not been modified by
880          * the loopback transport. The sending sockfs generates
881          * a message that is esballoc'ed with the free function
882          * being fdbuf_free() and where free_arg contains the
883          * identical information as the SO_FILEP content.
884          *
885          * If any of these constraints are not satisfied we
886          * silently ignore the option.
887          */
888         ASSERT(mp);
889         if (frp != NULL &&
890             frp->free_func == fdbuf_free &&
891             frp->free_arg != NULL &&
892             bcmp(frp->free_arg, fdbuf, fdbuflen) == 0) {
893             dprint(1, ("fdbuf_verify: fdbuf %p len %d\n",
894                 (void *)fdbuf, fdbuflen));
895             return (1);
896         } else {
897             zcmn_err(getzoneid(), CE_WARN,
898                 "sockfs: mismatched fdbuf content (%p)",
899                 (void *)mp);
900             return (0);
901         }
902     } else {
903         zcmn_err(getzoneid(), CE_WARN,
904             "sockfs: mismatched fdbuf len %d, %d\n",
905             fdbuflen, fdbuf->fd_size);
906         return (0);
907     }
908 }

910 /*
911  * When the file descriptors returned by sorecvmsg can not be passed
912  * to the application this routine will cleanup the references on
913  * the files. Start at startoff bytes into the buffer.
914  */
915 static void
916 close_fds(void *fdbuf, int fdbuflen, int startoff)
917 {
918     int *fds = (int *)fdbuf;
919     int numfd = fdbuflen / (int)sizeof (int);
920     int i;

```

```

922     dprint(1, ("close_fds(%p, %d, %d)\n", fdbuf, fdbuflen, startoff));
924     for (i = 0; i < numfd; i++) {
925         if (startoff < 0)
926             startoff = 0;
927         if (startoff < (int)sizeof (int)) {
928             /*
929              * This file descriptor is partially or fully after
930              * the offset
931              */
932             dprint(0,
933                 ("close_fds: cleanup[%d] = %d\n", i, fds[i]));
934             (void) closeandsetf(fds[i], NULL);
935         }
936         startoff -= (int)sizeof (int);
937     }
938 }

940 /*
941  * Close all file descriptors contained in the control part starting at
942  * the startoffset.
943  */
944 void
945 so_closefds(void *control, t_uscalar_t controllen, int oldflg,
946             int startoff)
947 {
948     struct cmsghdr *cmsg;

950     if (control == NULL)
951         return;

953     if (oldflg) {
954         close_fds(control, controllen, startoff);
955         return;
956     }
957     /* Scan control part for file descriptors. */
958     for (cmsg = (struct cmsghdr *)control;
959          MSG_VALID(cmsg, control, (uintptr_t)control + controllen);
960          cmsg = MSG_NEXT(cmsg)) {
961         if (cmsg->cmsg_level == SOL_SOCKET &&
962             cmsg->cmsg_type == SCM_RIGHTS) {
963             close_fds(MSG_CONTENT(cmsg),
964                 (int)MSG_CONTENTLEN(cmsg),
965                 startoff - (int)sizeof (struct cmsghdr));
966         }
967         startoff -= cmsg->cmsg_len;
968     }
969 }

971 /*
972  * Returns a pointer/length for the file descriptors contained
973  * in the control buffer. Returns with *fdlenp == -1 if there are no
974  * file descriptor options present. This is different than there being
975  * a zero-length file descriptor option.
976  * Fail if there are multiple SCM_RIGHT cmsgs.
977  */
978 int
979 so_getfdopt(void *control, t_uscalar_t controllen, int oldflg,
980             void **fdsp, int *fdlenp)
981 {
982     struct cmsghdr *cmsg;
983     void *fds;
984     int fdlen;

986     if (control == NULL) {
987         *fdsp = NULL;

```

```

988         *fdlenp = -1;
989         return (0);
990     }

992     if (oldflg) {
993         *fdsp = control;
994         if (controllen == 0)
995             *fdlenp = -1;
996         else
997             *fdlenp = controllen;
998         dprint(1, ("so_getfdopt: old %d\n", *fdlenp));
999         return (0);
1000     }

1002     fds = NULL;
1003     fdlen = 0;

1005     for (cmsg = (struct cmsghdr *)control;
1006          MSG_VALID(cmsg, control, (uintptr_t)control + controllen);
1007          cmsg = MSG_NEXT(cmsg)) {
1008         if (cmsg->cmsg_level == SOL_SOCKET &&
1009             cmsg->cmsg_type == SCM_RIGHTS) {
1010             if (fds != NULL)
1011                 return (EINVAL);
1012             fds = MSG_CONTENT(cmsg);
1013             fdlen = (int)MSG_CONTENTLEN(cmsg);
1014             dprint(1, ("so_getfdopt: new %lu\n",
1015                 (size_t)MSG_CONTENTLEN(cmsg)));
1016         }
1017     }

1018     if (fds == NULL) {
1019         dprint(1, ("so_getfdopt: NONE\n"));
1020         *fdlenp = -1;
1021     } else
1022         *fdlenp = fdlen;
1023     *fdsp = fds;
1024     return (0);
1025 }

1027 /*
1028  * Return the length of the options including any file descriptor options.
1029  */
1030 t_uscalar_t
1031 so_optlen(void *control, t_uscalar_t controllen, int oldflg)
1032 {
1033     struct cmsghdr *cmsg;
1034     t_uscalar_t optlen = 0;
1035     t_uscalar_t len;

1037     if (control == NULL)
1038         return (0);

1040     if (oldflg)
1041         return ((t_uscalar_t)(sizeof (struct T_opthdr) +
1042             fdbuf_optlen(controllen)));

1044     for (cmsg = (struct cmsghdr *)control;
1045          MSG_VALID(cmsg, control, (uintptr_t)control + controllen);
1046          cmsg = MSG_NEXT(cmsg)) {
1047         if (cmsg->cmsg_level == SOL_SOCKET &&
1048             cmsg->cmsg_type == SCM_RIGHTS) {
1049             len = fdbuf_optlen((int)MSG_CONTENTLEN(cmsg));
1050         } else {
1051             len = (t_uscalar_t)MSG_CONTENTLEN(cmsg);
1052         }
1053         optlen += (t_uscalar_t)(_TPI_ALIGN_TOPT(len) +

```

```

1054         sizeof (struct T_opthdr));
1055     }
1056     dprint(1, ("so_optlen: controllen %d, flg %d -> optlen %d\n",
1057         controllen, oldflg, optlen));
1058     return (optlen);
1059 }

1061 /*
1062  * Copy options from control to the mblk. Skip any file descriptor options.
1063  */
1064 void
1065 so_cmsg2opt(void *control, t_uscalar_t controllen, int oldflg, mblk_t *mp)
1066 {
1067     struct T_opthdr toh;
1068     struct cmsghdr *cmsg;

1070     if (control == NULL)
1071         return;

1073     if (oldflg) {
1074         /* No real options - caller has handled file descriptors */
1075         return;
1076     }
1077     for (cmsg = (struct cmsghdr *)control;
1078         MSG_VALID(cmsg, control, (uintptr_t)control + controllen);
1079         cmsg = CMSG_NEXT(cmsg)) {
1080         /*
1081          * Note: The caller handles file descriptors prior
1082          * to calling this function.
1083          */
1084         t_uscalar_t len;

1086         if (cmsg->cmsg_level == SOL_SOCKET &&
1087             cmsg->cmsg_type == SCM_RIGHTS)
1088             continue;

1090         len = (t_uscalar_t)CMSG_CONTENTLEN(cmsg);
1091         toh.level = cmsg->cmsg_level;
1092         toh.name = cmsg->cmsg_type;
1093         toh.len = len + (t_uscalar_t)sizeof (struct T_opthdr);
1094         toh.status = 0;

1096         soappendmsg(mp, &toh, sizeof (toh));
1097         soappendmsg(mp, CMSG_CONTENT(cmsg), len);
1098         mp->b_wptr += _TPI_ALIGN_TOPT(len) - len;
1099         ASSERT(mp->b_wptr <= mp->b_datap->db_lim);
1100     }
1101 }

1103 /*
1104  * Return the length of the control message derived from the options.
1105  * Exclude SO_SRCADDR and SO_UNIX_CLOSE options. Include SO_FILEP.
1106  * When oldflg is set only include SO_FILEP.
1107  * so_opt2cmsg and so_cmsglen are inter-related since so_cmsglen
1108  * allocates the space that so_opt2cmsg fills. If one changes, the other should
1109  * also be checked for any possible impacts.
1110  */
1111 t_uscalar_t
1112 so_cmsglen(mblk_t *mp, void *opt, t_uscalar_t optlen, int oldflg)
1113 {
1114     t_uscalar_t cmsglen = 0;
1115     struct T_opthdr *tohp;
1116     t_uscalar_t len;
1117     t_uscalar_t last_roundup = 0;

1119     ASSERT(!_TPI_TOPT_ISALIGNED(opt));

```

```

1121     for (tohp = (struct T_opthdr *)opt;
1122         tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1123         tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1124         dprint(1, ("so_cmsglen: level 0x%x, name %d, len %d\n",
1125             tohp->level, tohp->name, tohp->len));
1126         if (tohp->level == SOL_SOCKET &&
1127             (tohp->name == SO_SRCADDR ||
1128              tohp->name == SO_UNIX_CLOSE)) {
1129             continue;
1130         }
1131         if (tohp->level == SOL_SOCKET && tohp->name == SO_FILEP) {
1132             struct fdbuf *fdbuf;
1133             int fdbuflen;

1135             fdbuf = (struct fdbuf *)_TPI_TOPT_DATA(tohp);
1136             fdbuflen = (int)_TPI_TOPT_DATALEN(tohp);

1138             if (!fdbuf_verify(mp, fdbuf, fdbuflen))
1139                 continue;
1140             if (oldflg) {
1141                 cmsglen += fdbuf_cmsglen(fdbuflen);
1142                 continue;
1143             }
1144             len = fdbuf_cmsglen(fdbuflen);
1145         } else if (tohp->level == SOL_SOCKET &&
1146             tohp->name == SCM_TIMESTAMP) {
1147             if (oldflg)
1148                 continue;

1150             if (get_udatamodel() == DATAMODEL_NATIVE) {
1151                 len = sizeof (struct timeval);
1152             } else {
1153                 len = sizeof (struct timeval32);
1154             }
1155         } else {
1156             if (oldflg)
1157                 continue;
1158             len = (t_uscalar_t)_TPI_TOPT_DATALEN(tohp);
1159         }
1160         /*
1161          * Exclude roundup for last option to not set
1162          * MSG_TRUNC when the cmsg fits but the padding doesn't fit.
1163          */
1164         last_roundup = (t_uscalar_t)
1165             (ROUNDUP_cmsglen(len + (int)sizeof (struct cmsghdr)) -
1166              (len + (int)sizeof (struct cmsghdr)));
1167         cmsglen += (t_uscalar_t)(len + (int)sizeof (struct cmsghdr)) +
1168             last_roundup;
1169     }
1170     cmsglen -= last_roundup;
1171     dprint(1, ("so_cmsglen: optlen %d, flg %d -> cmsglen %d\n",
1172         optlen, oldflg, cmsglen));
1173     return (cmsglen);
1174 }

1176 /*
1177  * Copy options from options to the control. Convert SO_FILEP to
1178  * file descriptors.
1179  * Returns errno or zero.
1180  * so_opt2cmsg and so_cmsglen are inter-related since so_cmsglen
1181  * allocates the space that so_opt2cmsg fills. If one changes, the other should
1182  * also be checked for any possible impacts.
1183  */
1184 int
1185 so_opt2cmsg(mblk_t *mp, void *opt, t_uscalar_t optlen, int oldflg,

```

```

1186 void *control, t_uscalar_t controllen)
1187 {
1188     struct T_opthdr *tohp;
1189     struct cmsghdr *cmsg;
1190     struct fdbuf *fdbuf;
1191     int fdbuflen;
1192     int error;
1193 #if defined(DEBUG) || defined(__lint)
1194     struct cmsghdr *cend = (struct cmsghdr *)
1195         (((uint8_t *)control) + ROUNDUP_cmsglen(controllen));
1196 #endif
1197     cmsg = (struct cmsghdr *)control;
1199     ASSERT(__TPI_TOPT_ISALIGNED(opt));
1201     for (tohp = (struct T_opthdr *)opt;
1202          tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1203          tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1204         dprint(1, ("so_opt2cmsg: level 0x%x, name %d, len %d\n",
1205                 tohp->level, tohp->name, tohp->len));
1207         if (tohp->level == SOL_SOCKET &&
1208             (tohp->name == SO_SRCADDR ||
1209              tohp->name == SO_UNIX_CLOSE)) {
1210             continue;
1211         }
1212         ASSERT((uintptr_t)cmsg <= (uintptr_t)control + controllen);
1213         if (tohp->level == SOL_SOCKET && tohp->name == SO_FILEP) {
1214             fdbuf = (struct fdbuf *)_TPI_TOPT_DATA(tohp);
1215             fdbuflen = (int)_TPI_TOPT_DATALEN(tohp);
1217             if (!fdbuf_verify(mp, fdbuf, fdbuflen))
1218                 return (EPROTO);
1219             if (oldflg) {
1220                 error = fdbuf_extract(fdbuf, control,
1221                                     (int)controllen);
1222                 if (error != 0)
1223                     return (error);
1224                 continue;
1225             } else {
1226                 int fdlen;
1228                 fdlen = (int)fdbuf_cmsglen(
1229                     (int)_TPI_TOPT_DATALEN(tohp));
1231                 cmsg->cmsg_level = tohp->level;
1232                 cmsg->cmsg_type = SCM_RIGHTS;
1233                 cmsg->cmsg_len = (socklen_t)(fdlen +
1234                                     sizeof (struct cmsghdr));
1236                 error = fdbuf_extract(fdbuf,
1237                                     MSG_CONTENT(cmsg), fdlen);
1238                 if (error != 0)
1239                     return (error);
1240             }
1241         } else if (tohp->level == SOL_SOCKET &&
1242                  tohp->name == SCM_TIMESTAMP) {
1243             timestruc_t *timestamp;
1245             if (oldflg)
1246                 continue;
1248             cmsg->cmsg_level = tohp->level;
1249             cmsg->cmsg_type = tohp->name;
1251             timestamp =

```

```

1252             (timestruc_t *)P2ROUNDUP((intptr_t)&tohp[1],
1253                                     sizeof (intptr_t));
1255             if (get_umatamodel() == DATAMODEL_NATIVE) {
1256                 struct timeval tv;
1258                 cmsg->cmsg_len = sizeof (struct timeval) +
1259                     sizeof (struct cmsghdr);
1260                 tv.tv_sec = timestamp->tv_sec;
1261                 tv.tv_usec = timestamp->tv_nsec /
1262                     (NANOSEC / MICROSEC);
1263                 /*
1264                  * on LP64 systems, the struct timeval in
1265                  * the destination will not be 8-byte aligned,
1266                  * so use bcopy to avoid alignment trouble
1267                  */
1268                 bcopy(&tv, MSG_CONTENT(cmsg), sizeof (tv));
1269             } else {
1270                 struct timeval32 *time32;
1272                 cmsg->cmsg_len = sizeof (struct timeval32) +
1273                     sizeof (struct cmsghdr);
1274                 time32 = (struct timeval32 *)MSG_CONTENT(cmsg);
1275                 time32->tv_sec = (time32_t)timestamp->tv_sec;
1276                 time32->tv_usec =
1277                     (int32_t)(timestamp->tv_nsec /
1278                             (NANOSEC / MICROSEC));
1281             } else {
1282                 if (oldflg)
1283                     continue;
1285                 cmsg->cmsg_level = tohp->level;
1286                 cmsg->cmsg_type = tohp->name;
1287                 cmsg->cmsg_len = (socklen_t)(_TPI_TOPT_DATALEN(tohp) +
1288                                     sizeof (struct cmsghdr));
1290                 /* copy content to control data part */
1291                 bcopy(&tohp[1], MSG_CONTENT(cmsg),
1292                     MSG_CONTENTLEN(cmsg));
1293             }
1294             /* move to next MSG structure! */
1295             cmsg = MSG_NEXT(cmsg);
1296         }
1297         dprint(1, ("so_opt2cmsg: buf %p len %d; cend %p; final cmsg %p\n",
1298                 control, controllen, (void *)cend, (void *)cmsg));
1299         ASSERT(cmsg <= cend);
1300         return (0);
1301     }
1303 /*
1304  * Extract the SO_SRCADDR option value if present.
1305  */
1306 void
1307 so_getopt_srcaddr(void *opt, t_uscalar_t optlen, void **srctp,
1308                  t_uscalar_t *srclenp)
1309 {
1310     struct T_opthdr *tohp;
1312     ASSERT(__TPI_TOPT_ISALIGNED(opt));
1314     ASSERT(srctp != NULL && srclenp != NULL);
1315     *srctp = NULL;
1316     *srclenp = 0;

```

```

1318     for (tohp = (struct T_opthdr *)opt;
1319         tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1320         tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1321         dprint(1, ("so_getopt_srcaddr: level 0x%x, name %d, len %d\n",
1322             tohp->level, tohp->name, tohp->len));
1323         if (tohp->level == SOL_SOCKET &&
1324             tohp->name == SO_SRCADDR) {
1325             *srctp = _TPI_TOPT_DATA(tohp);
1326             *srclenp = (t_uscalar_t)_TPI_TOPT_DATALEN(tohp);
1327         }
1328     }
1329 }

1331 /*
1332  * Verify if the SO_UNIX_CLOSE option is present.
1333  */
1334 int
1335 so_getopt_unix_close(void *opt, t_uscalar_t optlen)
1336 {
1337     struct T_opthdr      *tohp;
1338
1339     ASSERT(__TPI_TOPT_ISALIGNED(opt));
1340
1341     for (tohp = (struct T_opthdr *)opt;
1342         tohp && _TPI_TOPT_VALID(tohp, opt, (uintptr_t)opt + optlen);
1343         tohp = _TPI_TOPT_NEXTHDR(opt, optlen, tohp)) {
1344         dprint(1,
1345             ("so_getopt_unix_close: level 0x%x, name %d, len %d\n",
1346             tohp->level, tohp->name, tohp->len));
1347         if (tohp->level == SOL_SOCKET &&
1348             tohp->name == SO_UNIX_CLOSE)
1349             return (1);
1350     }
1351     return (0);
1352 }

1354 /*
1355  * Allocate an M_PROTO message.
1356  *
1357  * If allocation fails the behavior depends on sleepflg:
1358  *   _ALLOC_NOSLEEP fail immediately
1359  *   _ALLOC_INTR   sleep for memory until a signal is caught
1360  *   _ALLOC_SLEEP  sleep forever. Don't return NULL.
1361  */
1362 mblk_t *
1363 soallocproto(size_t size, int sleepflg, cred_t *cr)
1364 {
1365     mblk_t *mp;
1366
1367     /* Round up size for reuse */
1368     size = MAX(size, 64);
1369     if (cr != NULL)
1370         mp = allocb_cred(size, cr, curproc->p_pid);
1371     else
1372         mp = allocb(size, BPRI_MED);
1373
1374     if (mp == NULL) {
1375         int error;          /* Dummy - error not returned to caller */
1376
1377         switch (sleepflg) {
1378         case _ALLOC_SLEEP:
1379             if (cr != NULL) {
1380                 mp = allocb_cred_wait(size, STR_NOSIG, &error,
1381                     cr, curproc->p_pid);
1382             } else {
1383                 mp = allocb_wait(size, BPRI_MED, STR_NOSIG,

```

```

1384         &error);
1385     }
1386     ASSERT(mp);
1387     break;
1388 case _ALLOC_INTR:
1389     if (cr != NULL) {
1390         mp = allocb_cred_wait(size, 0, &error, cr,
1391             curproc->p_pid);
1392     } else {
1393         mp = allocb_wait(size, BPRI_MED, 0, &error);
1394     }
1395     if (mp == NULL) {
1396         /* Caught signal while sleeping for memory */
1397         eprintline(ENOBUFS);
1398         return (NULL);
1399     }
1400     break;
1401 case _ALLOC_NOSLEEP:
1402 default:
1403     eprintline(ENOBUFS);
1404     return (NULL);
1405 }
1406 }
1407 DB_TYPE(mp) = M_PROTO;
1408 return (mp);
1409 }

1411 /*
1412  * Allocate an M_PROTO message with a single component.
1413  * len is the length of buf. size is the amount to allocate.
1414  *
1415  * buf can be NULL with a non-zero len.
1416  * This results in a bzero'ed chunk being placed the message.
1417  */
1418 mblk_t *
1419 soallocproto1(const void *buf, ssize_t len, ssize_t size, int sleepflg,
1420     cred_t *cr)
1421 {
1422     mblk_t *mp;
1423
1424     if (size == 0)
1425         size = len;
1426
1427     ASSERT(size >= len);
1428     /* Round up size for reuse */
1429     size = MAX(size, 64);
1430     mp = soallocproto(size, sleepflg, cr);
1431     if (mp == NULL)
1432         return (NULL);
1433     mp->b_datap->db_type = M_PROTO;
1434     if (len != 0) {
1435         if (buf != NULL)
1436             bcopy(buf, mp->b_wptr, len);
1437         else
1438             bzero(mp->b_wptr, len);
1439         mp->b_wptr += len;
1440     }
1441     return (mp);
1442 }

1444 /*
1445  * Append buf/len to mp.
1446  * The caller has to ensure that there is enough room in the mblk.
1447  *
1448  * buf can be NULL with a non-zero len.
1449  * This results in a bzero'ed chunk being placed the message.

```

```

1450 */
1451 void
1452 soappendmsg(mblk_t *mp, const void *buf, ssize_t len)
1453 {
1454     ASSERT(mp);
1455
1456     if (len != 0) {
1457         /* Assert for room left */
1458         ASSERT(mp->b_datap->db_lim - mp->b_wptr >= len);
1459         if (buf != NULL)
1460             bcopy(buf, mp->b_wptr, len);
1461         else
1462             bzero(mp->b_wptr, len);
1463     }
1464     mp->b_wptr += len;
1465 }
1466
1467 /*
1468 * Create a message using two kernel buffers.
1469 * If size is set that will determine the allocation size (e.g. for future
1470 * soappendmsg calls). If size is zero it is derived from the buffer
1471 * lengths.
1472 */
1473 mblk_t *
1474 soallocproto2(const void *buf1, ssize_t len1, const void *buf2, ssize_t len2,
1475              ssize_t size, int sleepflg, cred_t *cr)
1476 {
1477     mblk_t *mp;
1478
1479     if (size == 0)
1480         size = len1 + len2;
1481     ASSERT(size >= len1 + len2);
1482
1483     mp = soallocproto1(buf1, len1, size, sleepflg, cr);
1484     if (mp)
1485         soappendmsg(mp, buf2, len2);
1486     return (mp);
1487 }
1488
1489 /*
1490 * Create a message using three kernel buffers.
1491 * If size is set that will determine the allocation size (for future
1492 * soappendmsg calls). If size is zero it is derived from the buffer
1493 * lengths.
1494 */
1495 mblk_t *
1496 soallocproto3(const void *buf1, ssize_t len1, const void *buf2, ssize_t len2,
1497              const void *buf3, ssize_t len3, ssize_t size, int sleepflg, cred_t *cr)
1498 {
1499     mblk_t *mp;
1500
1501     if (size == 0)
1502         size = len1 + len2 + len3;
1503     ASSERT(size >= len1 + len2 + len3);
1504
1505     mp = soallocproto1(buf1, len1, size, sleepflg, cr);
1506     if (mp != NULL) {
1507         soappendmsg(mp, buf2, len2);
1508         soappendmsg(mp, buf3, len3);
1509     }
1510     return (mp);
1511 }
1512
1513 #ifdef DEBUG
1514 char *
1515 pr_state(uint_t state, uint_t mode)

```

```

1516 {
1517     static char buf[1024];
1518
1519     buf[0] = 0;
1520     if (state & SS_ISCONNECTED)
1521         (void) strcat(buf, "ISCONNECTED ");
1522     if (state & SS_ISCONNECTING)
1523         (void) strcat(buf, "ISCONNECTING ");
1524     if (state & SS_ISDISCONNECTING)
1525         (void) strcat(buf, "ISDISCONNECTING ");
1526     if (state & SS_CANTSENDMORE)
1527         (void) strcat(buf, "CANTSENDMORE ");
1528
1529     if (state & SS_CANTRCVMORE)
1530         (void) strcat(buf, "CANTRCVMORE ");
1531     if (state & SS_ISBOUND)
1532         (void) strcat(buf, "ISBOUND ");
1533     if (state & SS_NDELAY)
1534         (void) strcat(buf, "NDELAY ");
1535     if (state & SS_NONBLOCK)
1536         (void) strcat(buf, "NONBLOCK ");
1537
1538     if (state & SS_ASYNC)
1539         (void) strcat(buf, "ASYNC ");
1540     if (state & SS_ACCEPTCONN)
1541         (void) strcat(buf, "ACCEPTCONN ");
1542     if (state & SS_SAVEDEOR)
1543         (void) strcat(buf, "SAVEDEOR ");
1544
1545     if (state & SS_RCVATMARK)
1546         (void) strcat(buf, "RCVATMARK ");
1547     if (state & SS_OOBPEND)
1548         (void) strcat(buf, "OOBPEND ");
1549     if (state & SS_HAVEOOBDATA)
1550         (void) strcat(buf, "HAVEOOBDATA ");
1551     if (state & SS_HADOOBDATA)
1552         (void) strcat(buf, "HADOOBDATA ");
1553
1554     if (mode & SM_PRIV)
1555         (void) strcat(buf, "PRIV ");
1556     if (mode & SM_ATOMIC)
1557         (void) strcat(buf, "ATOMIC ");
1558     if (mode & SM_ADDR)
1559         (void) strcat(buf, "ADDR ");
1560     if (mode & SM_CONNREQUIRED)
1561         (void) strcat(buf, "CONNREQUIRED ");
1562
1563     if (mode & SM_FDPASSING)
1564         (void) strcat(buf, "FDPASSING ");
1565     if (mode & SM_EXDATA)
1566         (void) strcat(buf, "EXDATA ");
1567     if (mode & SM_OPTDATA)
1568         (void) strcat(buf, "OPTDATA ");
1569     if (mode & SM_BYTESTREAM)
1570         (void) strcat(buf, "BYTESTREAM ");
1571     return (buf);
1572 }
1573
1574 char *
1575 pr_addr(int family, struct sockaddr *addr, t_uscalar_t addrlen)
1576 {
1577     static char buf[1024];
1578
1579     if (addr == NULL || addrlen == 0) {
1580         (void) sprintf(buf, "(len %d) %p", addrlen, (void *)addr);
1581         return (buf);

```

```

1582     }
1583     switch (family) {
1584     case AF_INET: {
1585         struct sockaddr_in sin;
1586
1587         bcopy(addr, &sin, sizeof (sin));
1588
1589         (void) sprintf(buf, "(len %d) %x/%d",
1590             addrlen, ntohl(sin.sin_addr.s_addr), ntohs(sin.sin_port));
1591         break;
1592     }
1593     case AF_INET6: {
1594         struct sockaddr_in6 sin6;
1595         uint16_t *piece = (uint16_t *)&sin6.sin6_addr;
1596
1597         bcopy((char *)addr, (char *)&sin6, sizeof (sin6));
1598         (void) sprintf(buf, "(len %d) %x:%x:%x:%x:%x:%x:%x:%x/%d",
1599             addrlen,
1600             ntohs(piece[0]), ntohs(piece[1]),
1601             ntohs(piece[2]), ntohs(piece[3]),
1602             ntohs(piece[4]), ntohs(piece[5]),
1603             ntohs(piece[6]), ntohs(piece[7]),
1604             ntohs(sin6.sin6_port));
1605         break;
1606     }
1607     case AF_UNIX: {
1608         struct sockaddr_un *soun = (struct sockaddr_un *)addr;
1609
1610         (void) sprintf(buf, "(len %d) %s", addrlen,
1611             (soun == NULL) ? "(none)" : soun->sun_path);
1612         break;
1613     }
1614     default:
1615         (void) sprintf(buf, "(unknown af %d)", family);
1616         break;
1617     }
1618     return (buf);
1619 }
1620
1621 /* The logical equivalence operator (a if-and-only-if b) */
1622 #define EQUIVALENT(a, b)    (((a) && (b)) || (!(a) && !(b)))
1623
1624 /*
1625  * Verify limitations and invariants on oob state.
1626  * Return 1 if OK, otherwise 0 so that it can be used as
1627  * ASSERT(verify_oobstate(so));
1628  */
1629 int
1630 so_verify_oobstate(struct sonode *so)
1631 {
1632     boolean_t havemark;
1633
1634     ASSERT(MUTEX_HELD(&so->so_lock));
1635
1636     /*
1637      * The possible state combinations are:
1638      * 0
1639      * SS_OOBPEND
1640      * SS_OOBPEND|SS_HAVEOBDATA
1641      * SS_OOBPEND|SS_HADOOBDATA
1642      * SS_HADOOBDATA
1643      */
1644     switch (so->so_state & (SS_OOBPEND|SS_HAVEOBDATA|SS_HADOOBDATA)) {
1645     case 0:
1646     case SS_OOBPEND:
1647     case SS_OOBPEND|SS_HAVEOBDATA:

```

```

1648     case SS_OOBPEND|SS_HADOOBDATA:
1649     case SS_HADOOBDATA:
1650         break;
1651     default:
1652         printf("Bad oob state 1 (%p): state %s\n",
1653             (void *)so, pr_state(so->so_state, so->so_mode));
1654         return (0);
1655     }
1656
1657     /* SS_RCVATMARK should only be set when SS_OOBPEND is set */
1658     if ((so->so_state & (SS_RCVATMARK|SS_OOBPEND)) == SS_RCVATMARK) {
1659         printf("Bad oob state 2 (%p): state %s\n",
1660             (void *)so, pr_state(so->so_state, so->so_mode));
1661         return (0);
1662     }
1663
1664     /*
1665      * (havemark != 0 or SS_RCVATMARK) iff SS_OOBPEND
1666      * For TPI, the presence of a "mark" is indicated by sti_oobsigcnt.
1667      */
1668     havemark = (SOCK_IS_NONSTR(so) ? so->so_oobmark > 0 :
1669         SOTOTPI(so)->sti_oobsigcnt > 0);
1670
1671     if (!EQUIVALENT(havemark || (so->so_state & SS_RCVATMARK),
1672         so->so_state & SS_OOBPEND)) {
1673         printf("Bad oob state 3 (%p): state %s\n",
1674             (void *)so, pr_state(so->so_state, so->so_mode));
1675         return (0);
1676     }
1677
1678     /*
1679      * Unless SO_OOBINLINE we have so_oobmsg != NULL iff SS_HAVEOBDATA
1680      */
1681     if (!(so->so_options & SO_OOBINLINE) &&
1682         !EQUIVALENT(so->so_oobmsg != NULL, so->so_state & SS_HAVEOBDATA)) {
1683         printf("Bad oob state 4 (%p): state %s\n",
1684             (void *)so, pr_state(so->so_state, so->so_mode));
1685         return (0);
1686     }
1687
1688     if (!SOCK_IS_NONSTR(so) &&
1689         SOTOTPI(so)->sti_oobsigcnt < SOTOTPI(so)->sti_oobcnt) {
1690         printf("Bad oob state 5 (%p): counts %d/%d state %s\n",
1691             (void *)so, SOTOTPI(so)->sti_oobsigcnt,
1692             SOTOTPI(so)->sti_oobcnt,
1693             pr_state(so->so_state, so->so_mode));
1694         return (0);
1695     }
1696
1697     return (1);
1698 }
1699 #undef EQUIVALENT
1700 #endif /* DEBUG */
1701
1702 /* initialize sockfs zone specific kstat related items */
1703 void *
1704 sock_kstat_init(zoneid_t zoneid)
1705 {
1706     kstat_t *ksp;
1707
1708     ksp = kstat_create_zone("sockfs", 0, "sock_unix_list", "misc",
1709         KSTAT_TYPE_RAW, 0, KSTAT_FLAG_VAR_SIZE|KSTAT_FLAG_VIRTUAL, zoneid);
1710
1711     if (ksp != NULL) {
1712         ksp->ks_update = sockfs_update;
1713         ksp->ks_snapshot = sockfs_snapshot;

```

```

1714         ksp->ks_lock = &socklist.sl_lock;
1715         ksp->ks_private = (void *) (uintptr_t) zoneid;
1716         kstat_install(ksp);
1717     }
1719     return (ksp);
1720 }

1722 /* tear down sockfs zone specific kstat related items */
1723 /*ARGSUSED*/
1724 void
1725 sock_kstat_fini(zoneid_t zoneid, void *arg)
1726 {
1727     kstat_t *ksp = (kstat_t *) arg;

1729     if (ksp != NULL) {
1730         ASSERT(zoneid == (zoneid_t) (uintptr_t) ksp->ks_private);
1731         kstat_delete(ksp);
1732     }
1733 }

1735 /*
1736  * Zones:
1737  * Note that nactive is going to be different for each zone.
1738  * This means we require kstat to call sockfs_update and then sockfs_snapshot
1739  * for the same zone, or sockfs_snapshot will be taken into the wrong size
1740  * buffer. This is safe, but if the buffer is too small, user will not be
1741  * given details of all sockets. However, as this kstat has a ks_lock, kstat
1742  * driver will keep it locked between the update and the snapshot, so no
1743  * other process (zone) can currently get inbetween resulting in a wrong size
1744  * buffer allocation.
1745  */
1746 static int
1747 sockfs_update(kstat_t *ksp, int rw)
1748 {
1749     uint_t n, nactive = 0;          /* # of active AF_UNIX sockets */
1750     uint_t tsze = 0;
1751     uint_t nactive = 0;          /* # of active AF_UNIX sockets */
1752     struct sonode *so;          /* current sonode on socklist */
1753     zoneid_t myzoneid = (zoneid_t) (uintptr_t) ksp->ks_private;

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_tree_lock);
1767             n = avl_numnodes(&so->so_pid_tree);
1768             mutex_exit(&so->so_pid_tree_lock);

1770             tsze += sizeof (struct sockinfo);
1771             tsze += (n > 1) ? ((n - 1) * sizeof (pid_t)) : 0;
1772 #endif /* ! codereview */
1773         }
1774     }
1775     ksp->ks_ndata = nactive;
1776     ksp->ks_data_size = tsze;
1777     ksp->ks_data_size = nactive * sizeof (struct k_sockinfo);

```

```

1778         return (0);
1779     }

1781 static int
1782 sockfs_snapshot(kstat_t *ksp, void *buf, int rw)
1783 {
1784     int ns;          /* # of sonodes we've copied */
1785     struct sonode *so; /* current sonode on socklist */
1786     struct sockinfo *psi; /* where we put sockinfo data */
1787     struct k_sockinfo *pksi; /* where we put sockinfo data */
1788     t_uscalar_t sn_len; /* soa_len */
1789     zoneid_t myzoneid = (zoneid_t) (uintptr_t) ksp->ks_private;
1790     sotpi_info_t *sti;

1791     uint_t size;
1792     mblk_t *mblk;
1793     conn_pid_info_t *cpi;

1795 #endif /* ! codereview */
1796     ASSERT((zoneid_t) (uintptr_t) ksp->ks_private == getzoneid());

1798     ksp->ks_snaptime = gethrtime();

1800     if (rw == KSTAT_WRITE) {      /* bounce all writes */
1801         return (EACCES);
1802     }

1804     /*
1805      * for each sonode on the socklist, we massage the important
1806      * info into buf, in k_sockinfo format.
1807      */
1808     psi = (struct sockinfo *) buf;
1809     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         }

1816         mblk = so_get_sock_pid_mblk((sock_upper_handle_t) so);
1817         if (mblk == NULL) {
1818             continue;
1819         }
1820         cpi = (conn_pid_info_t *) mblk->b_datap->db_base;
1821         size = sizeof (struct sockinfo);
1822         size += (cpi->cpi_pids_cnt > 1) ?
1823             ((cpi->cpi_pids_cnt - 1) * sizeof (pid_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. In the
1830          * time window between the update and the snapshot, the size of
1831          * sockinfo may change, as new pids are added/removed to/from
1832          * the list. We have to take that into consideration and only
1833          * include the sockinfo if we have enough space. That means the
1834          * number of entries we return by snapshot might not equal the
1835          * the number of entries calculated by update.
1836          * snapshot, we're done - as this is only a snapshot.
1837          */
1838         if (((caddr_t) (psi) + size) >
1839             ((caddr_t) buf + ksp->ks_data_size)) {
1840             if ((caddr_t) (pksi) >= (caddr_t) buf + ksp->ks_data_size) {
1841                 break;
1842             }

```

```

1840     }
1842     sti = SOTOTPI(so);
1843     /* copy important info into buf: */
1844     psi->si_size = size;
1845     psi->si_family = so->so_family;
1846     psi->si_type = so->so_type;
1847     psi->si_flag = so->so_flag;
1848     psi->si_state = so->so_state;
1849     psi->si_serv_type = sti->sti_serv_type;
1850     psi->si_ux_laddr_sou_magic =
812     pksi->ks_si.si_size = sizeof (struct k_sockinfo);
813     pksi->ks_si.si_family = so->so_family;
814     pksi->ks_si.si_type = so->so_type;
815     pksi->ks_si.si_flag = so->so_flag;
816     pksi->ks_si.si_state = so->so_state;
817     pksi->ks_si.si_serv_type = sti->sti_serv_type;
818     pksi->ks_si.si_ux_laddr_sou_magic =
1851     sti->sti_ux_laddr_soua_magic;
1852     psi->si_ux_faddr_sou_magic =
820     pksi->ks_si.si_ux_faddr_sou_magic =
1853     sti->sti_ux_faddr_soua_magic;
1854     psi->si_laddr_soa_len = sti->sti_laddr.soa_len;
1855     psi->si_faddr_soa_len = sti->sti_faddr.soa_len;
1856     psi->si_szoneid = so->so_szoneid;
1857     psi->si_faddr_noxlate = sti->sti_faddr_noxlate;

822     pksi->ks_si.si_laddr_soa_len = sti->sti_laddr.soa_len;
823     pksi->ks_si.si_faddr_soa_len = sti->sti_faddr.soa_len;
824     pksi->ks_si.si_szoneid = so->so_szoneid;
825     pksi->ks_si.si_faddr_noxlate = sti->sti_faddr_noxlate;

1860     mutex_enter(&so->so_lock);

1862     if (sti->sti_laddr_sa != NULL) {
1863         ASSERT(sti->sti_laddr_sa->sa_data != NULL);
1864         sn_len = sti->sti_laddr_len;
1865         ASSERT(sn_len <= sizeof (short) +
1866             sizeof (psi->si_laddr_sun_path));
833         sizeof (pksi->ks_si.si_laddr_sun_path));

1868     psi->si_laddr_family =
835     pksi->ks_si.si_laddr_family =
1869     sti->sti_laddr_sa->sa_family;
1870     if (sn_len != 0) {
1871         /* AF_UNIX socket names are NULL terminated */
1872         (void) strncpy(psi->si_laddr_sun_path,
839         (void) strncpy(pksi->ks_si.si_laddr_sun_path,
1873             sti->sti_laddr_sa->sa_data,
1874             sizeof (psi->si_laddr_sun_path));
1875         sn_len = strlen(psi->si_laddr_sun_path);
841         sizeof (pksi->ks_si.si_laddr_sun_path));
842         sn_len = strlen(pksi->ks_si.si_laddr_sun_path);
1876     }
1877     psi->si_laddr_sun_path[sn_len] = 0;
844     pksi->ks_si.si_laddr_sun_path[sn_len] = 0;
1878 }

1880     if (sti->sti_faddr_sa != NULL) {
1881         ASSERT(sti->sti_faddr_sa->sa_data != NULL);
1882         sn_len = sti->sti_faddr_len;
1883         ASSERT(sn_len <= sizeof (short) +
1884             sizeof (psi->si_faddr_sun_path));
851         sizeof (pksi->ks_si.si_faddr_sun_path));

1886     psi->si_faddr_family =

```

```

853     pksi->ks_si.si_faddr_family =
1887     sti->sti_faddr_sa->sa_family;
1888     if (sn_len != 0) {
1889         (void) strncpy(psi->si_faddr_sun_path,
856         (void) strncpy(pksi->ks_si.si_faddr_sun_path,
1890             sti->sti_faddr_sa->sa_data,
1891             sizeof (psi->si_faddr_sun_path));
1892         sn_len = strlen(psi->si_faddr_sun_path);
858         sizeof (pksi->ks_si.si_faddr_sun_path));
859         sn_len = strlen(pksi->ks_si.si_faddr_sun_path);
1893     }
1894     psi->si_faddr_sun_path[sn_len] = 0;
861     pksi->ks_si.si_faddr_sun_path[sn_len] = 0;
1895 }

1897     mutex_exit(&so->so_lock);

1899     (void) sprintf(psi->si_son_straddr, "%p", (void *)so);
1900     (void) sprintf(psi->si_lvn_straddr, "%p",
866     (void) sprintf(pksi->ks_straddr[0], "%p", (void *)so);
867     (void) sprintf(pksi->ks_straddr[1], "%p",
1901         (void *)sti->sti_ux_laddr.soua_vp);
1902     (void) sprintf(psi->si_fvn_straddr, "%p",
869     (void) sprintf(pksi->ks_straddr[2], "%p",
1903         (void *)sti->sti_ux_faddr.soua_vp);

1905     psi->si_pids[0] = 0;
1906     if ((psi->si_pn_cnt = cpi->cpi_pids_cnt) > 0) {
1907         (void) memcpy(psi->si_pids, cpi->cpi_pids,
1908             psi->si_pn_cnt * sizeof (pid_t));
1909     }

1911     freemsg(mblk);

1913     psi = (struct sockinfo *)((caddr_t)psi + psi->si_size);
1914 #endif /* ! codereview */
1915     ns++;
872     pksi++;
1916 }

1918     ksp->ks_ndata = ns;
1919     return (0);
1920 }

```

_____unchanged_portion_omitted_____

```
*****
80972 Wed Jul 13 01:32:38 2016
new/usr/src/uts/common/inet/ip/ipclassifier.c
XXX adding PID information to netstat output
*****
_____unchanged_portion_omitted_
2724 #endif

2726 mblk_t *
2727 conn_get_pid_mblk(conn_t *connp)
2728 {
2729     mblk_t *mblk;
2730     conn_pid_info_t *cpi;

2732     /*
2733      * If the connection is closing, it is not safe to make an upcall or
2734      * access the stream associated with the connection.
2735      */
2736     if (!(connp->conn_state_flags & CONN_CLOSING)) {
2737         if (connp->conn_upper_handle != NULL) {
2738             return (*connp->conn_upcalls->su_get_sock_pid_mblk)
2739                 (connp->conn_upper_handle);
2740         } else if (!IPCL_IS_NONSTR(connp) && connp->conn_rq != NULL &&
2741             connp->conn_rq->q_stream != NULL) {
2742             return (sh_get_pid_mblk(connp->conn_rq->q_stream));
2743         }
2744     }

2746     /* return an empty mblk */
2747     if ((mblk = alloca(sizeof (conn_pid_info_t), BPRI_HI)) == NULL)
2748         return (NULL);
2749     mblk->b_wptr += sizeof (conn_pid_info_t);
2750     cpi = (conn_pid_info_t *)mblk->b_datap->db_base;
2751     cpi->cpi_contents = CONN_PID_INFO_NON;
2752     cpi->cpi_pids_cnt = 0;
2753     cpi->cpi_tot_size = sizeof (conn_pid_info_t);
2754     cpi->cpi_pids[0] = 0;
2755     return (mblk);
2756 }
2757 #endif /* ! codereview */
```

```

*****
26483 Wed Jul 13 01:32:38 2016
new/usr/src/uts/common/inet/ipclassifier.h
XXXX adding PID information to netstat output
*****
_____unchanged_portion_omitted_____

508 /*
509  * For use with subsystems within ip which use ALL_ZONES as a wildcard
510  */
511 #define IPCL_ZONEID(connp) \
512     ((connp)->conn_allzones ? ALL_ZONES : (connp)->conn_zoneid)

514 /*
515  * For matching between a conn_t and a zoneid.
516  */
517 #define IPCL_ZONE_MATCH(connp, zoneid) \
518     (((connp)->conn_allzones) || \
519      ((zoneid) == ALL_ZONES) || \
520      (connp)->conn_zoneid == (zoneid))

522 /*
523  * On a labeled system, we must treat bindings to ports
524  * on shared IP addresses by sockets with MAC exemption
525  * privilege as being in all zones, as there's
526  * otherwise no way to identify the right receiver.
527  */

529 #define IPCL_CONNS_MAC(conn1, conn2) \
530     (((conn1)->conn_mac_mode != CONN_MAC_DEFAULT) || \
531      ((conn2)->conn_mac_mode != CONN_MAC_DEFAULT))

533 #define IPCL_BIND_ZONE_MATCH(conn1, conn2) \
534     (IPCL_CONNS_MAC(conn1, conn2) || \
535      IPCL_ZONE_MATCH(conn1, conn2->conn_zoneid) || \
536      IPCL_ZONE_MATCH(conn2, conn1->conn_zoneid))

539 #define _IPCL_V4_MATCH(v6addr, v4addr) \
540     (V4_PART_OF_V6((v6addr)) == (v4addr) && IN6_IS_ADDR_V4MAPPED(&(v6addr)))

542 #define _IPCL_V4_MATCH_ANY(addr) \
543     (IN6_IS_ADDR_V4MAPPED_ANY(&(addr)) || IN6_IS_ADDR_UNSPECIFIED(&(addr)))

546 /*
547  * IPCL_PROTO_MATCH() and IPCL_PROTO_MATCH_V6() only matches conns with
548  * the specified ira_zoneid or conn_allzones by calling conn_wantpacket.
549  */
550 #define IPCL_PROTO_MATCH(connp, ira, ipha) \
551     (((connp)->conn_laddr_v4 == INADDR_ANY) || \
552      ((connp)->conn_laddr_v4 == ((ipha)->ipha_dst)) && \
553      (((connp)->conn_faddr_v4 == INADDR_ANY) || \
554       ((connp)->conn_faddr_v4 == ((ipha)->ipha_src)))) && \
555     conn_wantpacket((connp), (ira), (ipha))

557 #define IPCL_PROTO_MATCH_V6(connp, ira, ip6h) \
558     ((IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
559      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &((ip6h)->ip6_dst)) && \
560      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) || \
561      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &((ip6h)->ip6_src)))))) && \
562      (conn_wantpacket_v6((connp), (ira), (ip6h))))

564 #define IPCL_CONN_HASH(src, ports, ipst) \
565     ((unsigned)(ntohl((src)) ^ ((ports) >> 24) ^ ((ports) >> 16) ^ \
566      ((ports) >> 8) ^ (ports)) % (ipst)->ips_ipcl_conn_fanout_size)

```

```

568 #define IPCL_CONN_HASH_V6(src, ports, ipst) \
569     IPCL_CONN_HASH(V4_PART_OF_V6((src)), (ports), (ipst))

571 #define IPCL_CONN_MATCH(connp, proto, src, dst, ports) \
572     ((connp)->conn_proto == (proto) && \
573      (connp)->conn_ports == (ports) && \
574      _IPCL_V4_MATCH((connp)->conn_faddr_v6, (src)) && \
575      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (dst)) && \
576      !(connp)->conn_ipv6_v6only)

578 #define IPCL_CONN_MATCH_V6(connp, proto, src, dst, ports) \
579     ((connp)->conn_proto == (proto) && \
580      (connp)->conn_ports == (ports) && \
581      IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(src)) && \
582      IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(dst)))

584 #define IPCL_PORT_HASH(port, size) \
585     (((port) >> 8) ^ (port) & ((size) - 1))

587 #define IPCL_BIND_HASH(lport, ipst) \
588     ((unsigned)(((lport) >> 8) ^ (lport)) % \
589      (ipst)->ips_ipcl_bind_fanout_size)

591 #define IPCL_BIND_MATCH(connp, proto, laddr, lport) \
592     ((connp)->conn_proto == (proto) && \
593      (connp)->conn_lport == (lport) && \
594      (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
595      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))) && \
596      !(connp)->conn_ipv6_v6only)

598 #define IPCL_BIND_MATCH_V6(connp, proto, laddr, lport) \
599     ((connp)->conn_proto == (proto) && \
600      (connp)->conn_lport == (lport) && \
601      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) || \
602      IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6)))

604 /*
605  * We compare conn_laddr since it captures both connected and a bind to
606  * a multicast or broadcast address.
607  * The caller needs to match the zoneid and also call conn_wantpacket
608  * for multicast, broadcast, or when conn_incoming_ifindex is set.
609  */
610 #define IPCL_UDP_MATCH(connp, lport, laddr, fport, faddr) \
611     (((connp)->conn_lport == (lport)) && \
612      (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
613      _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))) && \
614      (_IPCL_V4_MATCH_ANY((connp)->conn_faddr_v6) || \
615      _IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)) && \
616      (connp)->conn_fport == (fport)))) && \
617     !(connp)->conn_ipv6_v6only)

619 /*
620  * We compare conn_laddr since it captures both connected and a bind to
621  * a multicast or broadcast address.
622  * The caller needs to match the zoneid and also call conn_wantpacket_v6
623  * for multicast or when conn_incoming_ifindex is set.
624  */
625 #define IPCL_UDP_MATCH_V6(connp, lport, laddr, fport, faddr) \
626     (((connp)->conn_lport == (lport)) && \
627      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
628      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr)) && \
629      (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_faddr_v6) || \
630      (IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, &(faddr)) && \
631      (connp)->conn_fport == (fport))))))

```

```

633 #define IPCL_IPTUN_HASH(laddr, faddr) \
634 ((ntohl(laddr) ^ (ntohl(faddr) << 24) | (ntohl(faddr) >> 8))) % \
635 ipcl_iptun_fanout_size)

637 #define IPCL_IPTUN_HASH_V6(laddr, faddr) \
638 IPCL_IPTUN_HASH((laddr)->s6_addr32[0] ^ (laddr)->s6_addr32[1] ^ \
639 (faddr)->s6_addr32[2] ^ (faddr)->s6_addr32[3], \
640 (faddr)->s6_addr32[0] ^ (faddr)->s6_addr32[1] ^ \
641 (laddr)->s6_addr32[2] ^ (laddr)->s6_addr32[3])

643 #define IPCL_IPTUN_MATCH(connp, laddr, faddr) \
644 (_IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr)) && \
645 _IPCL_V4_MATCH((connp)->conn_faddr_v6, (faddr)))

647 #define IPCL_IPTUN_MATCH_V6(connp, laddr, faddr) \
648 (IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, (laddr)) && \
649 IN6_ARE_ADDR_EQUAL(&(connp)->conn_faddr_v6, (faddr)))

651 #define IPCL_UDP_HASH(lport, ipst) \
652 IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_udp_fanout_size)

654 #define CONN_G_HASH_SIZE 1024

656 /* Raw socket hash function. */
657 #define IPCL_RAW_HASH(lport, ipst) \
658 IPCL_PORT_HASH(lport, (ipst)->ips_ipcl_raw_fanout_size)

660 /*
661 * This is similar to IPCL_BIND_MATCH except that the local port check
662 * is changed to a wildcard port check.
663 * We compare conn_laddr since it captures both connected and a bind to
664 * a multicast or broadcast address.
665 */
666 #define IPCL_RAW_MATCH(connp, proto, laddr) \
667 ((connp)->conn_proto == (proto) && \
668 (connp)->conn_lport == 0 && \
669 (_IPCL_V4_MATCH_ANY((connp)->conn_laddr_v6) || \
670 _IPCL_V4_MATCH((connp)->conn_laddr_v6, (laddr))))

672 #define IPCL_RAW_MATCH_V6(connp, proto, laddr) \
673 ((connp)->conn_proto == (proto) && \
674 (connp)->conn_lport == 0 && \
675 (IN6_IS_ADDR_UNSPECIFIED(&(connp)->conn_laddr_v6) || \
676 IN6_ARE_ADDR_EQUAL(&(connp)->conn_laddr_v6, &(laddr))))

678 /* Function prototypes */
679 extern void ipcl_g_init(void);
680 extern void ipcl_init(ip_stack_t *);
681 extern void ipcl_g_destroy(void);
682 extern void ipcl_destroy(ip_stack_t *);
683 extern conn_t *ipcl_conn_create(uint32_t, int, netstack_t *);
684 extern void ipcl_conn_destroy(conn_t *);

686 void ipcl_hash_insert_wildcard(connf_t *, conn_t *);
687 void ipcl_hash_remove(conn_t *);
688 void ipcl_hash_remove_locked(conn_t *connp, connf_t *connfp);

690 extern int ipcl_bind_insert(conn_t *);
691 extern int ipcl_bind_insert_v4(conn_t *);
692 extern int ipcl_bind_insert_v6(conn_t *);
693 extern int ipcl_conn_insert(conn_t *);
694 extern int ipcl_conn_insert_v4(conn_t *);
695 extern int ipcl_conn_insert_v6(conn_t *);
696 extern conn_t *ipcl_get_next_conn(connf_t *, conn_t *, uint32_t);

698 conn_t *ipcl_classify_v4(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,

```

```

699 ip_stack_t *);
700 conn_t *ipcl_classify_v6(mblk_t *, uint8_t, uint_t, ip_recv_attr_t *,
701 ip_stack_t *);
702 conn_t *ipcl_classify(mblk_t *, ip_recv_attr_t *, ip_stack_t *);
703 conn_t *ipcl_classify_raw(mblk_t *, uint8_t, uint32_t, ipha_t *,
704 ip6_t *, ip_recv_attr_t *, ip_stack_t *);
705 conn_t *ipcl_iptun_classify_v4(ipaddr_t *, ipaddr_t *, ip_stack_t *);
706 conn_t *ipcl_iptun_classify_v6(in6_addr_t *, in6_addr_t *, ip_stack_t *);
707 void ipcl_globalhash_insert(conn_t *);
708 void ipcl_globalhash_remove(conn_t *);
709 void ipcl_walk(pfv_t, void *, ip_stack_t *);
710 conn_t *ipcl_tcp_lookup_reversed_ipv4(ipha_t *, tcpha_t *, int, ip_stack_t *);
711 conn_t *ipcl_tcp_lookup_reversed_ipv6(ip6_t *, tcpha_t *, int, uint_t,
712 ip_stack_t *);
713 conn_t *ipcl_lookup_listener_v4(uint16_t, ipaddr_t, zoneid_t, ip_stack_t *);
714 conn_t *ipcl_lookup_listener_v6(uint16_t, in6_addr_t *, uint_t, zoneid_t,
715 ip_stack_t *);
716 int conn_trace_ref(conn_t *);
717 int conn_untrace_ref(conn_t *);
718 void ipcl_conn_cleanup(conn_t *);
719 extern uint_t conn_recvancillary_size(conn_t *, crb_t, ip_recv_attr_t *,
720 mblk_t *, ip_pkt_t *);
721 extern void conn_recvancillary_add(conn_t *, crb_t, ip_recv_attr_t *,
722 ip_pkt_t *, uchar_t *, uint_t);
723 conn_t *ipcl_conn_tcp_lookup_reversed_ipv4(conn_t *, ipha_t *, tcpha_t *,
724 ip_stack_t *);
725 conn_t *ipcl_conn_tcp_lookup_reversed_ipv6(conn_t *, ip6_t *, tcpha_t *,
726 ip_stack_t *);

728 extern int ip_create_helper_stream(conn_t *, ldi_ident_t);
729 extern void ip_free_helper_stream(conn_t *);
730 extern int ip_helper_stream_setup(queue_t *, dev_t *, int, int,
731 cred_t *, boolean_t);
732 extern mblk_t *conn_get_pid_mblk(conn_t *);
733 #endif /* ! codereview */

735 #ifdef __cplusplus
736 }
737 #endif

739 #endif /* _INET_IPCLASSIFIER_H */

```

```

*****
60158 Wed Jul 13 01:32:38 2016
new/usr/src/uts/common/inet/mib2.h
XXXX adding PID information to netstat output
*****
_____unchanged_portion_omitted_____

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

164 #define MIB2_UNKNOWN_INTERFACE 0
165 #define MIB2_UNKNOWN_PROCESS 0

167 /*
168 * IP group
169 */
170 #define MIB2_IP_ADDR 20 /* ipAddrEntry */
171 #define MIB2_IP_ROUTE 21 /* ipRouteEntry */
172 #define MIB2_IP_MEDIA 22 /* ipNetToMediaEntry */
173 #define MIB2_IP6_ROUTE 23 /* ipv6RouteEntry */
174 #define MIB2_IP6_MEDIA 24 /* ipv6NetToMediaEntry */
175 #define MIB2_IP6_ADDR 25 /* ipv6AddrEntry */
176 #define MIB2_IP_TRAFFIC_STATS 31 /* ipIfStatsEntry (IPv4) */
177 #define EXPER_IP_GROUP_MEMBERSHIP 100
178 #define EXPER_IP6_GROUP_MEMBERSHIP 101
179 #define EXPER_IP_GROUP_SOURCES 102
180 #define EXPER_IP6_GROUP_SOURCES 103
181 #define EXPER_IP_RTATTR 104
182 #define EXPER_IP_DCE 105

184 /*
185 * There can be one of each of these tables per transport (MIB2_* above).
186 */
187 #define EXPER_XPORT_MLP 105 /* transportMLPEntry */
188 #define EXPER_XPORT_PROC_INFO 106 /* conn_pid_node entry */
189 #endif /* ! codereview */

191 /* Old names retained for compatibility */
192 #define MIB2_IP_20 MIB2_IP_ADDR
193 #define MIB2_IP_21 MIB2_IP_ROUTE
194 #define MIB2_IP_22 MIB2_IP_MEDIA

196 typedef struct mib2_ip {
197     /* forwarder? 1 gateway, 2 NOT gateway {ip 1} RW */
198     int ipForwarding;
199     /* default Time-to-Live for iph {ip 2} RW */
200     int ipDefaultTTL;
201     /* # of input datagrams {ip 3} */
202     Counter ipInReceives;
203     /* # of dg discards for iph error {ip 4} */
204     Counter ipInHdrErrors;
205     /* # of dg discards for bad addr {ip 5} */
206     Counter ipInAddrErrors;
207     /* # of dg being forwarded {ip 6} */
208     Counter ipForwDatagrams;
209     /* # of dg discards for unk protocol {ip 7} */
210     Counter ipInUnknownProtos;
211     /* # of dg discards of good dg's {ip 8} */
212     Counter ipInDiscards;

```

```

213     /* # of dg sent upstream {ip 9} */
214     Counter ipInDelivers;
215     /* # of outdgs recvd from upstream {ip 10} */
216     Counter ipOutRequests;
217     /* # of good outdgs discarded {ip 11} */
218     Counter ipOutDiscards;
219     /* # of outdg discards: no route found {ip 12} */
220     Counter ipOutNoRoutes;
221     /* sec's recvd frags held for reass. {ip 13} */
222     int ipReasmTimeout;
223     /* # of ip frags needing reassembly {ip 14} */
224     Counter ipReasmReqds;
225     /* # of dg's reassembled {ip 15} */
226     Counter ipReasmOKs;
227     /* # of reassembly failures (not dg cnt){ip 16} */
228     Counter ipReasmFails;
229     /* # of dg's fragged {ip 17} */
230     Counter ipFragOKs;
231     /* # of dg discards for no frag set {ip 18} */
232     Counter ipFragFails;
233     /* # of dg frags from fragmentation {ip 19} */
234     Counter ipFragCreates;
235     /* {ip 20} */
236     int ipAddrEntrySize;
237     /* {ip 21} */
238     int ipRouteEntrySize;
239     /* {ip 22} */
240     int ipNetToMediaEntrySize;
241     /* # of valid route entries discarded {ip 23} */
242     Counter ipRoutingDiscards;
243 /*
244 * following defined in MIB-II as part of TCP & UDP groups:
245 */
246     /* total # of segments recvd with error {tcp 14} */
247     Counter tcpInErrs;
248     /* # of recvd dg's not deliverable (no appl.) {udp 2} */
249     Counter udpNoPorts;
250 /*
251 * In addition to MIB-II
252 */
253     /* # of bad IP header checksums */
254     Counter ipInCksumErrs;
255     /* # of complete duplicates in reassembly */
256     Counter ipReasmDuplicates;
257     /* # of partial duplicates in reassembly */
258     Counter ipReasmPartDups;
259     /* # of packets not forwarded due to administrative reasons */
260     Counter ipForwProhibits;
261     /* # of UDP packets with bad UDP checksums */
262     Counter udpInCksumErrs;
263     /* # of UDP packets dropped due to queue overflow */
264     Counter udpInOverflows;
265     /*
266     * # of RAW IP packets (all IP protocols except UDP, TCP
267     * and ICMP) dropped due to queue overflow
268     */
269     Counter rawipInOverflows;

271 /*
272 * Following are private IPSEC MIB.
273 */
274 /* # of incoming packets that succeeded policy checks */
275 Counter ipsecInSucceeded;
276 /* # of incoming packets that failed policy checks */
277 Counter ipsecInFailed;
278 /* Compatible extensions added here */

```

```

279     int     ipMemberEntrySize; /* Size of ip_member_t */
280     int     ipGroupSourceEntrySize; /* Size of ip_grpsrc_t */

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

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

291 /*
292 *     ipv6IfStatsEntry OBJECT-TYPE
293 *     SYNTAX      Ipv6IfStatsEntry
294 *     MAX-ACCESS not-accessible
295 *     STATUS      current
296 *     DESCRIPTION
297 *         "An interface statistics entry containing objects
298 *         at a particular IPv6 interface."
299 *     AUGMENTS { ipv6IfEntry }
300 *     ::= { ipv6IfStatsTable 1 }
301 *
302 * Per-interface IPv6 statistics table
303 */

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

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

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

321     /* # input datagrams (incl errors) { ipv6IfStatsEntry 1 } */
322     Counter ipv6InReceives;
323     /* # errors in IPv6 headers and options { ipv6IfStatsEntry 2 } */
324     Counter ipv6InHdrErrors;
325     /* # exceeds outgoing link MTU { ipv6IfStatsEntry 3 } */
326     Counter ipv6InTooBigErrors;
327     /* # discarded due to no route to dest { ipv6IfStatsEntry 4 } */
328     Counter ipv6InNoRoutes;
329     /* # invalid or unsupported addresses { ipv6IfStatsEntry 5 } */
330     Counter ipv6InAddrErrors;
331     /* # unknown next header { ipv6IfStatsEntry 6 } */
332     Counter ipv6InUnknownProtos;
333     /* # too short packets { ipv6IfStatsEntry 7 } */
334     Counter ipv6InTruncatedPkts;
335     /* # discarded e.g. due to no buffers { ipv6IfStatsEntry 8 } */
336     Counter ipv6InDiscards;
337     /* # delivered to upper layer protocols { ipv6IfStatsEntry 9 } */
338     Counter ipv6InDelivers;
339     /* # forwarded out interface { ipv6IfStatsEntry 10 } */
340     Counter ipv6OutForwDatagrams;
341     /* # originated out interface { ipv6IfStatsEntry 11 } */
342     Counter ipv6OutRequests;
343     /* # discarded e.g. due to no buffers { ipv6IfStatsEntry 12 } */
344     Counter ipv6OutDiscards;

```

```

345     /* # successfully fragmented packets { ipv6IfStatsEntry 13 } */
346     Counter ipv6OutFragOKs;
347     /* # fragmentation failed { ipv6IfStatsEntry 14 } */
348     Counter ipv6OutFragFails;
349     /* # fragments created { ipv6IfStatsEntry 15 } */
350     Counter ipv6OutFragCreates;
351     /* # fragments to reassemble { ipv6IfStatsEntry 16 } */
352     Counter ipv6ReasmReqds;
353     /* # packets after reassembly { ipv6IfStatsEntry 17 } */
354     Counter ipv6ReasmOKs;
355     /* # reassembly failed { ipv6IfStatsEntry 18 } */
356     Counter ipv6ReasmFails;
357     /* # received multicast packets { ipv6IfStatsEntry 19 } */
358     Counter ipv6InMcastPkts;
359     /* # transmitted multicast packets { ipv6IfStatsEntry 20 } */
360     Counter ipv6OutMcastPkts;
361 /*
362 * In addition to defined MIBs
363 */
364     /* # discarded due to no route to dest */
365     Counter ipv6OutNoRoutes;
366     /* # of complete duplicates in reassembly */
367     Counter ipv6ReasmDuplicates;
368     /* # of partial duplicates in reassembly */
369     Counter ipv6ReasmPartDups;
370     /* # of packets not forwarded due to administrative reasons */
371     Counter ipv6ForwProhibits;
372     /* # of UDP packets with bad UDP checksums */
373     Counter udpInCksmErrs;
374     /* # of UDP packets dropped due to queue overflow */
375     Counter udpInOverflows;
376     /*
377     * # of RAW IPv6 packets (all IPv6 protocols except UDP, TCP
378     * and ICMPv6) dropped due to queue overflow
379     */
380     Counter rawipInOverflows;

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

390 /*
391 * Per interface IP statistics, both v4 and v6.
392 *
393 * Some applications expect to get mib2_ipv6IfStatsEntry_t structs back when
394 * making a request. To ensure backwards compatibility, the first
395 * sizeof(mib2_ipv6IfStatsEntry_t) bytes of the structure is identical to
396 * mib2_ipv6IfStatsEntry_t. This should work as long as the application is
397 * written correctly (i.e., using ipv6IfStatsEntrySize to get the size of
398 * the struct)
399 *
400 * RFC4293 introduces several new counters, as well as defining 64-bit
401 * versions of existing counters. For a new counters, if they have both 32-
402 * and 64-bit versions, then we only added the latter. However, for already
403 * existing counters, we have added the 64-bit versions without removing the
404 * old (32-bit) ones. The 64- and 32-bit counters will only be synchronized
405 * when the structure contains IPv6 statistics, which is done to ensure
406 * backwards compatibility.
407 */

409 /* The following are defined in RFC 4001 and are used for ipIfStatsIPVersion */
410 #define MIB2_INETADDRESSSTYPE_unknown 0

```

```

411 #define MIB2_INETADDRESSTYPE_ipv4      1
412 #define MIB2_INETADDRESSTYPE_ipv6      2

414 /*
415  * On amd64, the alignment requirements for long long's is different for
416  * 32 and 64 bits. If we have a struct containing long long's that is being
417  * passed between a 64-bit kernel to a 32-bit application, then it is very
418  * likely that the size of the struct will differ due to padding. Therefore, we
419  * pack the data to ensure that the struct size is the same for 32- and
420  * 64-bits.
421  */
422 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
423 #pragma pack(4)
424 #endif

426 typedef struct mib2_ipIfStatsEntry {

428     /* Local ifindex to identify the interface */
429     DeviceIndex    ipIfStatsIfIndex;

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

437     int             ipIfStatsEntrySize;
438     int             ipIfStatsAddrEntrySize;
439     int             ipIfStatsRouteEntrySize;
440     int             ipIfStatsNetToMediaEntrySize;
441     int             ipIfStatsMemberEntrySize;
442     int             ipIfStatsGroupSourceEntrySize;

444     /* # input datagrams (incl errors)      { ipIfStatsEntry 3 } */
445     Counter ipIfStatsInReceives;
446     /* # errors in IP headers and options  { ipIfStatsEntry 7 } */
447     Counter ipIfStatsInHdrErrors;
448     /* # exceeds outgoing link MTU(v6 only) { ipv6IfStatsEntry 3 } */
449     Counter ipIfStatsInTooBigErrors;
450     /* # discarded due to no route to dest { ipIfStatsEntry 8 } */
451     Counter ipIfStatsInNoRoutes;
452     /* # invalid or unsupported addresses { ipIfStatsEntry 9 } */
453     Counter ipIfStatsInAddrErrors;
454     /* # unknown next header              { ipIfStatsEntry 10 } */
455     Counter ipIfStatsInUnknownProtos;
456     /* # too short packets                 { ipIfStatsEntry 11 } */
457     Counter ipIfStatsInTruncatedPkts;
458     /* # discarded e.g. due to no buffers { ipIfStatsEntry 17 } */
459     Counter ipIfStatsInDiscards;
460     /* # delivered to upper layer protocols { ipIfStatsEntry 18 } */
461     Counter ipIfStatsInDelivers;
462     /* # forwarded out interface          { ipIfStatsEntry 23 } */
463     Counter ipIfStatsOutForwDatagrams;
464     /* # originated out interface        { ipIfStatsEntry 20 } */
465     Counter ipIfStatsOutRequests;
466     /* # discarded e.g. due to no buffers { ipIfStatsEntry 25 } */
467     Counter ipIfStatsOutDiscards;
468     /* # successfully fragmented packets { ipIfStatsEntry 27 } */
469     Counter ipIfStatsOutFragOKs;
470     /* # fragmentation failed             { ipIfStatsEntry 28 } */
471     Counter ipIfStatsOutFragFails;
472     /* # fragments created                { ipIfStatsEntry 29 } */
473     Counter ipIfStatsOutFragCreates;
474     /* # fragments to reassemble         { ipIfStatsEntry 14 } */
475     Counter ipIfStatsReasmReqds;
476     /* # packets after reassembly        { ipIfStatsEntry 15 } */

```

```

477     Counter ipIfStatsReasmOKs;
478     /* # reassembly failed                { ipIfStatsEntry 16 } */
479     Counter ipIfStatsReasmFails;
480     /* # received multicast packets      { ipIfStatsEntry 34 } */
481     Counter ipIfStatsInMcastPkts;
482     /* # transmitted multicast packets   { ipIfStatsEntry 38 } */
483     Counter ipIfStatsOutMcastPkts;

485     /*
486     * In addition to defined MIBs
487     */

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

510     /*
511     * # of IP packets received with the wrong version (i.e., not equal
512     * to ipIfStatsIPVersion) and that were dropped.
513     */
514     Counter ipIfStatsInWrongIPVersion;
515     /*
516     * This counter is no longer used
517     */
518     Counter ipIfStatsOutWrongIPVersion;
519     /*
520     * This counter is no longer used
521     */
522     Counter ipIfStatsOutSwitchIPVersion;

524     /*
525     * Fields defined in RFC 4293
526     */

528     /* ip version                          { ipIfStatsEntry 1 } */
529     int             ipIfStatsIPVersion;
530     /* # input datagrams (incl errors)      { ipIfStatsEntry 4 } */
531     Counter64      ipIfStatsHCInReceives;
532     /* # input octets (incl errors)         { ipIfStatsEntry 6 } */
533     Counter64      ipIfStatsHCInOctets;
534     /*
535     * # input datagrams for which a forwarding attempt was made
536     */
537     Counter64      ipIfStatsHCInForwDatagrams;
538     /* # delivered to upper layer protocols { ipIfStatsEntry 19 } */
539     Counter64      ipIfStatsHCInDelivers;
540     /* # originated out interface          { ipIfStatsEntry 21 } */
541     Counter64      ipIfStatsHCOutRequests;

```

```

543 /* # forwarded out interface { ipIfStatsEntry 23 } */
544 Counter64 ipIfStatsHCOutForwDatagrams;
545 /* # dg's requiring fragmentation { ipIfStatsEntry 26 } */
546 Counter ipIfStatsOutFragReqds;
547 /* # output datagrams { ipIfStatsEntry 31 } */
548 Counter64 ipIfStatsHCOutTransmits;
549 /* # output octets { ipIfStatsEntry 33 } */
550 Counter64 ipIfStatsHCOutOctets;
551 /* # received multicast datagrams { ipIfStatsEntry 35 } */
552 Counter64 ipIfStatsHCInMcastPkts;
553 /* # received multicast octets { ipIfStatsEntry 37 } */
554 Counter64 ipIfStatsHCInMcastOctets;
555 /* # transmitted multicast datagrams { ipIfStatsEntry 39 } */
556 Counter64 ipIfStatsHCOutMcastPkts;
557 /* # transmitted multicast octets { ipIfStatsEntry 41 } */
558 Counter64 ipIfStatsHCOutMcastOctets;
559 /* # received broadcast datagrams { ipIfStatsEntry 43 } */
560 Counter64 ipIfStatsHCInBcastPkts;
561 /* # transmitted broadcast datagrams { ipIfStatsEntry 45 } */
562 Counter64 ipIfStatsHCOutBcastPkts;

564 /*
565 * Fields defined in mib2_ip_t
566 */

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

586 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
587 #pragma pack()
588 #endif

590 /*
591 * The IP address table contains this entity's IP addressing information.
592 *
593 * ipAddrTable OBJECT-TYPE
594 * SYNTAX SEQUENCE OF IpAddrEntry
595 * ACCESS not-accessible
596 * STATUS mandatory
597 * DESCRIPTION
598 * "The table of addressing information relevant to
599 * this entity's IP addresses."
600 * ::= { ip 20 }
601 */

603 typedef struct mib2_ipAddrEntry {
604 /* IP address of this entry {ipAddrEntry 1} */
605 IpAddress ipAdEntAddr;
606 /* Unique interface index {ipAddrEntry 2} */
607 DeviceName ipAdEntIfIndex;
608 /* Subnet mask for this IP addr {ipAddrEntry 3} */

```

```

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

635 /*
636 * ipv6AddrTable OBJECT-TYPE
637 * SYNTAX SEQUENCE OF Ipv6AddrEntry
638 * MAX-ACCESS not-accessible
639 * STATUS current
640 * DESCRIPTION
641 * "The table of addressing information relevant to
642 * this node's interface addresses."
643 * ::= { ipv6MIBObjects 8 }
644 */

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

650 /* IPv6 address of this entry { ipv6AddrEntry 1 } */
651 Ip6Address ipv6AddrAddress;
652 /* Prefix length { ipv6AddrEntry 2 } */
653 uint_t ipv6AddrPfxLength;
654 /* Type: stateless(1), stateful(2), unknown(3) { ipv6AddrEntry 3 } */
655 uint_t ipv6AddrType;
656 /* Anycast: true(1), false(2) { ipv6AddrEntry 4 } */
657 uint_t ipv6AddrAnycastFlag;
658 /*
659 * Address status: preferred(1), deprecated(2), invalid(3),
660 * inaccessible(4), unknown(5) { ipv6AddrEntry 5 }
661 */
662 uint_t ipv6AddrStatus;
663 struct ipv6AddrInfo_s {
664 Gauge ae_mtu;
665 /* BSD if metric */
666 int ae_metric;
667 /* point-point dest addr */
668 Ip6Address ae_pp_dst_addr;
669 int ae_flags; /* IFF_* flags in if.h */
670 Counter ae_ibcnt; /* Inbound packets */
671 Counter ae_obcnt; /* Outbound packets */
672 Counter ae_focnt; /* Forwarded packets */
673 Ip6Address ae_subnet; /* Subnet prefix */
674 int ae_subnet_len; /* Subnet prefix length */

```

```

675     IpAddress    ae_src_addr; /* Source address */
676 }
677     uint32_t     ipv6AddrReasmMaxSize; /* InterfaceReasmMaxSize */
678 IpAddress      ipv6AddrIdentifier; /* InterfaceIdentifier */
679     uint32_t     ipv6AddrIdentifierLen;
680     uint32_t     ipv6AddrReachableTime; /* InterfaceReachableTime */
681     uint32_t     ipv6AddrRetransmitTime; /* InterfaceRetransmitTime */
682 } mib2_ipv6AddrEntry_t;
683 #define MIB_FIRST_NEW_ELM_mib2_ipv6AddrEntry_t ipv6AddrReasmMaxSize

685 /*
686 * The IP routing table contains an entry for each route presently known to
687 * this entity. (for IPv4 routes)
688 *
689 * ipRouteTable OBJECT-TYPE
690 *     SYNTAX SEQUENCE OF IpRouteEntry
691 *     ACCESS not-accessible
692 *     STATUS mandatory
693 *     DESCRIPTION
694 *         "This entity's IP Routing table."
695 *     ::= { ip 21 }
696 */

698 typedef struct mib2_ipRouteEntry {
699     /* dest ip addr for this route {ipRouteEntry 1 } RW */
700     IpAddress    ipRouteDest;
701     /* unique interface index for this hop {ipRouteEntry 2 } RW */
702     DeviceName  ipRouteIfIndex;
703     /* primary route metric {ipRouteEntry 3 } RW */
704     int         ipRouteMetric1;
705     /* alternate route metric {ipRouteEntry 4 } RW */
706     int         ipRouteMetric2;
707     /* alternate route metric {ipRouteEntry 5 } RW */
708     int         ipRouteMetric3;
709     /* alternate route metric {ipRouteEntry 6 } RW */
710     int         ipRouteMetric4;
711     /* ip addr of next hop on this route {ipRouteEntry 7 } RW */
712     IpAddress    ipRouteNextHop;
713     /* other(1), inval(2), dir(3), indir(4) {ipRouteEntry 8 } RW */
714     int         ipRouteType;
715     /* mechanism by which route was learned {ipRouteEntry 9 } */
716     int         ipRouteProto;
717     /* sec's since last update of route {ipRouteEntry 10} RW */
718     int         ipRouteAge;
719     /*
720     IpAddress    ipRouteMask;
721     /* alternate route metric {ipRouteEntry 12} RW */
722     int         ipRouteMetric5;
723     /* additional info from ire's {ipRouteEntry 13 } */
724     struct ipRouteInfo_s {
725         Gauge    re_max_frag;
726         Gauge    re_rtt;
727         Counter  re_ref;
728         int      re_frag_flag;
729         IpAddress re_src_addr;
730         int      re_ire_type;
731         Counter  re_obpkt;
732         Counter  re_ibpkt;
733         int      re_flags;
734     /*
735     * The following two elements (re_in_ill and re_in_src_addr)
736     * are no longer used but are left here for the benefit of
737     * old Apps that won't be able to handle the change in the
738     * size of this struct. These elements will always be
739     * set to zeroes.
740     */

```

```

741     DeviceName  re_in_ill; /* Input interface */
742     IpAddress    re_in_src_addr; /* Input source address */
743 }
744 } mib2_ipRouteEntry_t;

746 /*
747 * The IPv6 routing table contains an entry for each route presently known to
748 * this entity.
749 *
750 * ipv6RouteTable OBJECT-TYPE
751 *     SYNTAX SEQUENCE OF IpRouteEntry
752 *     ACCESS not-accessible
753 *     STATUS current
754 *     DESCRIPTION
755 *         "IPv6 Routing table. This table contains
756 *         an entry for each valid IPv6 unicast route
757 *         that can be used for packet forwarding
758 *         determination."
759 *     ::= { ipv6MIBObjects 11 }
760 */

762 typedef struct mib2_ipv6RouteEntry {
763     /* dest ip addr for this route { ipv6RouteEntry 1 } */
764     IpAddress    ipv6RouteDest;
765     /* prefix length { ipv6RouteEntry 2 } */
766     int         ipv6RoutePfxLength;
767     /* unique route index { ipv6RouteEntry 3 } */
768     unsigned    ipv6RouteIndex;
769     /* unique interface index for this hop { ipv6RouteEntry 4 } */
770     DeviceName  ipv6RouteIfIndex;
771     /* IPv6 addr of next hop on this route { ipv6RouteEntry 5 } */
772     IpAddress    ipv6RouteNextHop;
773     /* other(1), discard(2), local(3), remote(4) */
774     /* { ipv6RouteEntry 6 } */
775     int         ipv6RouteType;
776     /* mechanism by which route was learned { ipv6RouteEntry 7 } */
777     /*
778     * other(1), local(2), netmgmt(3), ndisc(4), rip(5), ospf(6),
779     * bgp(7), idrp(8), igrp(9)
780     */
781     int         ipv6RouteProtocol;
782     /* policy hook or traffic class { ipv6RouteEntry 8 } */
783     unsigned    ipv6RoutePolicy;
784     /* sec's since last update of route { ipv6RouteEntry 9 } */
785     int         ipv6RouteAge;
786     /* Routing domain ID of the next hop { ipv6RouteEntry 10 } */
787     unsigned    ipv6RouteNextHopRDI;
788     /* route metric { ipv6RouteEntry 11 } */
789     unsigned    ipv6RouteMetric;
790     /* preference (impl specific) { ipv6RouteEntry 12 } */
791     unsigned    ipv6RouteWeight;
792     /* additional info from ire's { } */
793     struct ipv6RouteInfo_s {
794         Gauge    re_max_frag;
795         Gauge    re_rtt;
796         Counter  re_ref;
797         int      re_frag_flag;
798         IpAddress re_src_addr;
799         int      re_ire_type;
800         Counter  re_obpkt;
801         Counter  re_ibpkt;
802         int      re_flags;
803     } ipv6RouteInfo;
804 } mib2_ipv6RouteEntry_t;

806 /*

```

```

807 * The IPv4 and IPv6 routing table entries on a trusted system also have
808 * security attributes in the form of label ranges. This experimental
809 * interface provides information about these labels.
810 *
811 * Each entry in this table contains a label range and an index that refers
812 * back to the entry in the routing table to which it applies. There may be 0,
813 * 1, or many label ranges for each routing table entry.
814 *
815 * (opthdr.level is set to MIB2_IP for IPv4 entries and MIB2_IP6 for IPv6.
816 * opthdr.name is set to EXPER_IP_GWATTR.)
817 *
818 *     ipRouteAttributeTable OBJECT-TYPE
819 *         SYNTAX SEQUENCE OF IpAttributeEntry
820 *         ACCESS not-accessible
821 *         STATUS current
822 *         DESCRIPTION
823 *             "IPv4 routing attributes table. This table contains
824 *             an entry for each valid trusted label attached to a
825 *             route in the system."
826 *         ::= { ip 102 }
827 *
828 *     ipv6RouteAttributeTable OBJECT-TYPE
829 *         SYNTAX SEQUENCE OF IpAttributeEntry
830 *         ACCESS not-accessible
831 *         STATUS current
832 *         DESCRIPTION
833 *             "IPv6 routing attributes table. This table contains
834 *             an entry for each valid trusted label attached to a
835 *             route in the system."
836 *         ::= { ip6 102 }
837 */
838
839 typedef struct mib2_ipAttributeEntry {
840     uint_t     iae_routeidx;
841     int        iae_doi;
842     brange_t   iae_slrange;
843 } mib2_ipAttributeEntry_t;
844
845 /*
846 * The IP address translation table contain the IPAddress to
847 * 'physical' address equivalences. Some interfaces do not
848 * use translation tables for determining address
849 * equivalences (e.g., DDN-X.25 has an algorithmic method);
850 * if all interfaces are of this type, then the Address
851 * Translation table is empty, i.e., has zero entries.
852 *
853 *     ipNetToMediaTable OBJECT-TYPE
854 *         SYNTAX SEQUENCE OF IpNetToMediaEntry
855 *         ACCESS not-accessible
856 *         STATUS mandatory
857 *         DESCRIPTION
858 *             "The IP Address Translation table used for mapping
859 *             from IP addresses to physical addresses."
860 *         ::= { ip 22 }
861 */
862
863 typedef struct mib2_ipNetToMediaEntry {
864     /* Unique interface index          { ipNetToMediaEntry 1 } RW */
865     DeviceName  ipNetToMediaIfIndex;
866     /* Media dependent physical addr    { ipNetToMediaEntry 2 } RW */
867     PhysAddress ipNetToMediaPhysAddress;
868     /* ip addr for this physical addr   { ipNetToMediaEntry 3 } RW */
869     IPAddress   ipNetToMediaNetAddress;
870     /* other(1), inval(2), dyn(3), stat(4) { ipNetToMediaEntry 4 } RW */
871     int         ipNetToMediaType;
872     struct ipNetToMediaInfo_s {

```

```

873         PhysAddress  ntm_mask;          /* subnet mask for entry */
874         int          ntm_flags;        /* ACE_F_* flags in arp.h */
875     } mib2_ipNetToMediaInfo;
876 } mib2_ipNetToMediaEntry_t;
877
878 /*
879 *     ipv6NetToMediaTable OBJECT-TYPE
880 *         SYNTAX SEQUENCE OF Ipv6NetToMediaEntry
881 *         MAX-ACCESS not-accessible
882 *         STATUS current
883 *         DESCRIPTION
884 *             "The IPv6 Address Translation table used for
885 *             mapping from IPv6 addresses to physical addresses.
886 *
887 *             The IPv6 address translation table contain the
888 *             Ipv6Address to 'physical' address equivalencies.
889 *             Some interfaces do not use translation tables
890 *             for determining address equivalencies; if all
891 *             interfaces are of this type, then the Address
892 *             Translation table is empty, i.e., has zero
893 *             entries."
894 *         ::= { ipv6MIBObjects 12 }
895 */
896
897 typedef struct mib2_ipv6NetToMediaEntry {
898     /* Unique interface index          { Part of INDEX } */
899     DeviceIndex  ipv6NetToMediaIfIndex;
900
901     /* ip addr for this physical addr   { ipv6NetToMediaEntry 1 } */
902     IpAddress    ipv6NetToMediaNetAddress;
903     /* Media dependent physical addr    { ipv6NetToMediaEntry 2 } */
904     PhysAddress  ipv6NetToMediaPhysAddress;
905     /*
906     * Type of mapping
907     * other(1), dynamic(2), static(3), local(4)
908     *                                     { ipv6NetToMediaEntry 3 }
909     */
910     int          ipv6NetToMediaType;
911     /*
912     * NUD state
913     * reachable(1), stale(2), delay(3), probe(4), invalid(5), unknown(6)
914     * Note: The kernel returns ND_* states.
915     *                                     { ipv6NetToMediaEntry 4 }
916     */
917     int          ipv6NetToMediaState;
918     /* sysUpTime last time entry was updated { ipv6NetToMediaEntry 5 } */
919     int          ipv6NetToMediaLastUpdated;
920 } mib2_ipv6NetToMediaEntry_t;
921
922 /*
923 * List of group members per interface
924 */
925
926 typedef struct ip_member {
927     /* Interface index */
928     DeviceName  ipGroupMemberIfIndex;
929     /* IP Multicast address */
930     IPAddress   ipGroupMemberAddress;
931     /* Number of member sockets */
932     Counter     ipGroupMemberRefCnt;
933     /* Filter mode: 1 => include, 2 => exclude */
934     int         ipGroupMemberFilterMode;
935 } ip_member_t;
936
937
938 /*

```

```

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

952 /*
953 * This is used to mark transport layer entities (e.g., TCP connections) that
954 * are capable of receiving packets from a range of labels. 'level' is set to
955 * the protocol of interest (e.g., MIB2_TCP), and 'name' is set to
956 * EXPER_XPORT_MLP. The tme_connidix refers back to the entry in MIB2_TCP_CONN,
957 * MIB2_TCP6_CONN, or MIB2_SCTP_CONN.
958 *
959 * It is also used to report connections that receive packets at a single label
960 * that's other than the zone's label. This is the case when a TCP connection
961 * is accepted from a particular peer using an MLP listener.
962 */
963 typedef struct mib2_transportMLPEntry {
964     uint_t         tme_connidix;
965     uint_t         tme_flags;
966     int            tme_doi;
967     bslabel_t     tme_label;
968 } mib2_transportMLPEntry_t;

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

989 /*
990 * List of IPv6 source addresses being filtered per interface
991 */
992 typedef struct ipv6_grpsrc {
993     /* Interface index */
994     DeviceIndex    ipv6GroupSourceIfIndex;
995     /* IP Multicast address */
996     Ip6Address     ipv6GroupSourceGroup;
997     /* IP Source address */
998     Ip6Address     ipv6GroupSourceAddress;
999 } ipv6_grpsrc_t;

1002 /*
1003 * List of destination cache entries
1004 */

```

```

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

1017 /*
1018 * ICMP Group
1019 */
1020 typedef struct mib2_icmp {
1021     /* total # of rcv'd ICMP msgs          { icmp 1 } */
1022     Counter icmpInMsgs;
1023     /* rcv'd ICMP msgs with errors        { icmp 2 } */
1024     Counter icmpInErrors;
1025     /* rcv'd "dest unreachable" msg's     { icmp 3 } */
1026     Counter icmpInDestUnreachs;
1027     /* rcv'd "time exceeded" msg's        { icmp 4 } */
1028     Counter icmpInTimeExcds;
1029     /* rcv'd "parameter problem" msg's    { icmp 5 } */
1030     Counter icmpInParmProbs;
1031     /* rcv'd "source quench" msg's        { icmp 6 } */
1032     Counter icmpInSrcQuenchs;
1033     /* rcv'd "ICMP redirect" msg's        { icmp 7 } */
1034     Counter icmpInRedirects;
1035     /* rcv'd "echo request" msg's         { icmp 8 } */
1036     Counter icmpInEchos;
1037     /* rcv'd "echo reply" msg's           { icmp 9 } */
1038     Counter icmpInEchoReps;
1039     /* rcv'd "timestamp" msg's           { icmp 10 } */
1040     Counter icmpInTimestamps;
1041     /* rcv'd "timestamp reply" msg's      { icmp 11 } */
1042     Counter icmpInTimestampReps;
1043     /* rcv'd "address mask request" msg's { icmp 12 } */
1044     Counter icmpInAddrMasks;
1045     /* rcv'd "address mask reply" msg's   { icmp 13 } */
1046     Counter icmpInAddrMaskReps;
1047     /* total # of sent ICMP msg's         { icmp 14 } */
1048     Counter icmpOutMsgs;
1049     /* # of msg's not sent for internal icmp errors { icmp 15 } */
1050     Counter icmpOutErrors;
1051     /* # of "dest unreachable" msg's sent { icmp 16 } */
1052     Counter icmpOutDestUnreachs;
1053     /* # of "time exceeded" msg's sent    { icmp 17 } */
1054     Counter icmpOutTimeExcds;
1055     /* # of "parameter problem" msg's sent { icmp 18 } */
1056     Counter icmpOutParmProbs;
1057     /* # of "source quench" msg's sent    { icmp 19 } */
1058     Counter icmpOutSrcQuenchs;
1059     /* # of "ICMP redirect" msg's sent    { icmp 20 } */
1060     Counter icmpOutRedirects;
1061     /* # of "Echo request" msg's sent     { icmp 21 } */
1062     Counter icmpOutEchos;
1063     /* # of "Echo reply" msg's sent       { icmp 22 } */
1064     Counter icmpOutEchoReps;
1065     /* # of "timestamp request" msg's sent { icmp 23 } */
1066     Counter icmpOutTimestamps;
1067     /* # of "timestamp reply" msg's sent  { icmp 24 } */
1068     Counter icmpOutTimestampReps;
1069     /* # of "address mask request" msg's sent { icmp 25 } */
1070     Counter icmpOutAddrMasks;

```

```

1071 /* # of "address mask reply" msg's sent { icmp 26 } */
1072 Counter icmpOutAddrMaskReps;
1073 /*
1074 * In addition to MIB-II
1075 */
1076 /* # of received packets with checksum errors */
1077 Counter icmpInCksumErrs;
1078 /* # of received packets with unknow codes */
1079 Counter icmpInUnknowns;
1080 /* # of received unreachable with "fragmentation needed" */
1081 Counter icmpInFragNeeded;
1082 /* # of sent unreachables with "fragmentation needed" */
1083 Counter icmpOutFragNeeded;
1084 /*
1085 * # of msg's not sent since original packet was broadcast/multicast
1086 * or an ICMP error packet
1087 */
1088 Counter icmpOutDrops;
1089 /* # of ICMP packets dropped due to queue overflow */
1090 Counter icmpInOverflows;
1091 /* recv'd "ICMP redirect" msg's that are bad thus ignored */
1092 Counter icmpInBadRedirects;
1093 } mib2_icmp_t;

1096 /*
1097 * ipv6IfIcmpEntry OBJECT-TYPE
1098 * SYNTAX Ipv6IfIcmpEntry
1099 * MAX-ACCESS not-accessible
1100 * STATUS current
1101 * DESCRIPTION
1102 * "An ICMPv6 statistics entry containing
1103 * objects at a particular IPv6 interface.
1104 *
1105 * Note that a receiving interface is
1106 * the interface to which a given ICMPv6 message
1107 * is addressed which may not be necessarily
1108 * the input interface for the message.
1109 *
1110 * Similarly, the sending interface is
1111 * the interface that sources a given
1112 * ICMP message which is usually but not
1113 * necessarily the output interface for the message."
1114 * AUGMENTS { ipv6IfEntry }
1115 * ::= { ipv6IfIcmpTable 1 }
1116 *
1117 * Per-interface ICMPv6 statistics table
1118 */

1120 typedef struct mib2_ipv6IfIcmpEntry {
1121 /* Local ifindex to identify the interface */
1122 DeviceIndex ipv6IfIcmpIfIndex;

1124 int ipv6IfIcmpEntrySize; /* Size of ipv6IfIcmpEntry */

1126 /* The total # ICMP msgs rcvd includes ipv6IfIcmpInErrors */
1127 Counter32 ipv6IfIcmpInMsgs;
1128 /* # ICMP with ICMP-specific errors (bad checksum, length, etc) */
1129 Counter32 ipv6IfIcmpInErrors;
1130 /* # ICMP Destination Unreachable */
1131 Counter32 ipv6IfIcmpInDestUnreachs;
1132 /* # ICMP destination unreachable/communication admin prohibited */
1133 Counter32 ipv6IfIcmpInAdminProhibs;
1134 Counter32 ipv6IfIcmpInTimeExcds;
1135 Counter32 ipv6IfIcmpInParmProblems;
1136 Counter32 ipv6IfIcmpInPktTooBig;

```

```

1137 Counter32 ipv6IfIcmpInEchoes;
1138 Counter32 ipv6IfIcmpInEchoReplies;
1139 Counter32 ipv6IfIcmpInRouterSolicits;
1140 Counter32 ipv6IfIcmpInRouterAdvertisements;
1141 Counter32 ipv6IfIcmpInNeighborSolicits;
1142 Counter32 ipv6IfIcmpInNeighborAdvertisements;
1143 Counter32 ipv6IfIcmpInRedirects;
1144 Counter32 ipv6IfIcmpInGroupMembQueries;
1145 Counter32 ipv6IfIcmpInGroupMembResponses;
1146 Counter32 ipv6IfIcmpInGroupMembReductions;
1147 /* Total # ICMP messages attempted to send (includes OutErrors) */
1148 Counter32 ipv6IfIcmpOutMsgs;
1149 /* # ICMP messages not sent due to ICMP problems (e.g. no buffers) */
1150 Counter32 ipv6IfIcmpOutErrors;
1151 Counter32 ipv6IfIcmpOutDestUnreachs;
1152 Counter32 ipv6IfIcmpOutAdminProhibs;
1153 Counter32 ipv6IfIcmpOutTimeExcds;
1154 Counter32 ipv6IfIcmpOutParmProblems;
1155 Counter32 ipv6IfIcmpOutPktTooBig;
1156 Counter32 ipv6IfIcmpOutEchoes;
1157 Counter32 ipv6IfIcmpOutEchoReplies;
1158 Counter32 ipv6IfIcmpOutRouterSolicits;
1159 Counter32 ipv6IfIcmpOutRouterAdvertisements;
1160 Counter32 ipv6IfIcmpOutNeighborSolicits;
1161 Counter32 ipv6IfIcmpOutNeighborAdvertisements;
1162 Counter32 ipv6IfIcmpOutRedirects;
1163 Counter32 ipv6IfIcmpOutGroupMembQueries;
1164 Counter32 ipv6IfIcmpOutGroupMembResponses;
1165 Counter32 ipv6IfIcmpOutGroupMembReductions;
1166 /* Additions beyond the MIB */
1167 Counter32 ipv6IfIcmpInOverflows;
1168 /* recv'd "ICMPv6 redirect" msg's that are bad thus ignored */
1169 Counter32 ipv6IfIcmpBadHoplimit;
1170 Counter32 ipv6IfIcmpInBadNeighborAdvertisements;
1171 Counter32 ipv6IfIcmpInBadNeighborSolicitations;
1172 Counter32 ipv6IfIcmpInBadRedirects;
1173 Counter32 ipv6IfIcmpInGroupMembTotal;
1174 Counter32 ipv6IfIcmpInGroupMembBadQueries;
1175 Counter32 ipv6IfIcmpInGroupMembBadReports;
1176 Counter32 ipv6IfIcmpInGroupMembOurReports;
1177 } mib2_ipv6IfIcmpEntry_t;

1179 /*
1180 * the TCP group
1181 *
1182 * Note that instances of object types that represent
1183 * information about a particular TCP connection are
1184 * transient; they persist only as long as the connection
1185 * is in question.
1186 */
1187 #define MIB2_TCP_CONN 13 /* tcpConnEntry */
1188 #define MIB2_TCP6_CONN 14 /* tcp6ConnEntry */

1190 /* Old name retained for compatibility */
1191 #define MIB2_TCP_13 MIB2_TCP_CONN

1193 /* Pack data in mib2_tcp to make struct size the same for 32- and 64-bits */
1194 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1195 #pragma pack(4)
1196 #endif
1197 typedef struct mib2_tcp {
1198 /* algorithm used for transmit timeout value { tcp 1 } */
1199 int tcpRtoAlgorithm;
1200 /* minimum retransmit timeout (ms) { tcp 2 } */
1201 int tcpRtoMin;
1202 /* maximum retransmit timeout (ms) { tcp 3 } */

```

```

1203     int      tcpRtoMax;
1204         /* maximum # of connections supported      { tcp 4 } */
1205     int      tcpMaxConn;
1206         /* # of direct transitions CLOSED -> SYN-SENT { tcp 5 } */
1207     Counter  tcpActiveOpens;
1208         /* # of direct transitions LISTEN -> SYN-RCVD { tcp 6 } */
1209     Counter  tcpPassiveOpens;
1210         /* # of direct SIN-SENT/RCVD -> CLOSED/LISTEN { tcp 7 } */
1211     Counter  tcpAttemptFails;
1212         /* # of direct ESTABLISHED/CLOSE-WAIT -> CLOSED { tcp 8 } */
1213     Counter  tcpEstabResets;
1214         /* # of connections ESTABLISHED or CLOSE-WAIT { tcp 9 } */
1215     Gauge   tcpCurrEstab;
1216         /* total # of segments rcv'd                { tcp 10 } */
1217     Counter  tcpInSegs;
1218         /* total # of segments sent                  { tcp 11 } */
1219     Counter  tcpOutSegs;
1220         /* total # of segments retransmitted         { tcp 12 } */
1221     Counter  tcpRetransSegs;
1222         /* { tcp 13 } */
1223     int      tcpConnTableSize; /* Size of tcpConnEntry_t */
1224         /* in ip { tcp 14 } */
1225     Counter  tcpOutRsts; /* # of segments sent with RST flag { tcp 15 } */
1226
1227 /* In addition to MIB-II */
1228 /* Sender */
1229     /* total # of data segments sent */
1230     Counter  tcpOutDataSegs;
1231     /* total # of bytes in data segments sent */
1232     Counter  tcpOutDataBytes;
1233     /* total # of bytes in segments retransmitted */
1234     Counter  tcpRetransBytes;
1235     /* total # of acks sent */
1236     Counter  tcpOutAck;
1237     /* total # of delayed acks sent */
1238     Counter  tcpOutAckDelayed;
1239     /* total # of segments sent with the urg flag on */
1240     Counter  tcpOutUrg;
1241     /* total # of window updates sent */
1242     Counter  tcpOutWinUpdate;
1243     /* total # of zero window probes sent */
1244     Counter  tcpOutWinProbe;
1245     /* total # of control segments sent (syn, fin, rst) */
1246     Counter  tcpOutControl;
1247     /* total # of segments sent due to "fast retransmit" */
1248     Counter  tcpOutFastRetrans;
1249 /* Receiver */
1250     /* total # of ack segments received */
1251     Counter  tcpInAckSegs;
1252     /* total # of bytes acked */
1253     Counter  tcpInAckBytes;
1254     /* total # of duplicate acks */
1255     Counter  tcpInDupAck;
1256     /* total # of acks acking unsent data */
1257     Counter  tcpInAckUnsent;
1258     /* total # of data segments received in order */
1259     Counter  tcpInDataInorderSegs;
1260     /* total # of data bytes received in order */
1261     Counter  tcpInDataInorderBytes;
1262     /* total # of data segments received out of order */
1263     Counter  tcpInDataUnorderSegs;
1264     /* total # of data bytes received out of order */
1265     Counter  tcpInDataUnorderBytes;
1266     /* total # of complete duplicate data segments received */
1267     Counter  tcpInDataDupSegs;
1268     /* total # of bytes in the complete duplicate data segments received */

```

```

1269     Counter  tcpInDataDupBytes;
1270         /* total # of partial duplicate data segments received */
1271     Counter  tcpInDataPartDupSegs;
1272         /* total # of bytes in the partial duplicate data segments received */
1273     Counter  tcpInDataPartDupBytes;
1274         /* total # of data segments received past the window */
1275     Counter  tcpInDataPastWinSegs;
1276         /* total # of data bytes received part the window */
1277     Counter  tcpInDataPastWinBytes;
1278         /* total # of zero window probes received */
1279     Counter  tcpInWinProbe;
1280         /* total # of window updates received */
1281     Counter  tcpInWinUpdate;
1282         /* total # of data segments received after the connection has closed */
1283     Counter  tcpInClosed;
1284 /* Others */
1285     /* total # of failed attempts to update the rtt estimate */
1286     Counter  tcpRttNoUpdate;
1287     /* total # of successful attempts to update the rtt estimate */
1288     Counter  tcpRttUpdate;
1289     /* total # of retransmit timeouts */
1290     Counter  tcpTimRetrans;
1291     /* total # of retransmit timeouts dropping the connection */
1292     Counter  tcpTimRetransDrop;
1293     /* total # of keepalive timeouts */
1294     Counter  tcpTimKeepalive;
1295     /* total # of keepalive timeouts sending a probe */
1296     Counter  tcpTimKeepaliveProbe;
1297     /* total # of keepalive timeouts dropping the connection */
1298     Counter  tcpTimKeepaliveDrop;
1299     /* total # of connections refused due to backlog full on listen */
1300     Counter  tcpListenDrop;
1301     /* total # of connections refused due to half-open queue (q0) full */
1302     Counter  tcpListenDropQ0;
1303     /* total # of connections dropped from a full half-open queue (q0) */
1304     Counter  tcpHalfOpenDrop;
1305     /* total # of retransmitted segments by SACK retransmission */
1306     Counter  tcpOutSackRetransSegs;
1307
1308     int      tcp6ConnTableSize; /* Size of tcp6ConnEntry_t */
1309
1310     /*
1311     * fields from RFC 4022
1312     */
1313
1314     /* total # of segments rcv'd                { tcp 17 } */
1315     Counter64 tcpHCInSegs;
1316     /* total # of segments sent                { tcp 18 } */
1317     Counter64 tcpHCOutSegs;
1318 } mib2_tcp_t;
1319 #define MIB_FIRST_NEW_ELM_mib2_tcp_t    tcpHCInSegs
1320
1321 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1322 #pragma pack()
1323 #endif
1324
1325 /*
1326 * The TCP/IPv4 connection table {tcp 13} contains information about this
1327 * entity's existing TCP connections over IPv4.
1328 */
1329 /* For tcpConnState and tcp6ConnState */
1330 #define MIB2_TCP_closed      1
1331 #define MIB2_TCP_listen     2
1332 #define MIB2_TCP_synSent    3
1333 #define MIB2_TCP_synReceived 4
1334 #define MIB2_TCP_established 5

```

```

1335 #define MIB2_TCP_finWait1      6
1336 #define MIB2_TCP_finWait2      7
1337 #define MIB2_TCP_closeWait     8
1338 #define MIB2_TCP_lastAck       9
1339 #define MIB2_TCP_closing       10
1340 #define MIB2_TCP_timeWait      11
1341 #define MIB2_TCP_deleteTCB     12          /* only writeable value */

1343 /* Pack data to make struct size the same for 32- and 64-bits */
1344 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1345 #pragma pack(4)
1346 #endif
1347 typedef struct mib2_tcpConnEntry {
1348     /* state of tcp connection          { tcpConnEntry 1 } RW */
1349     int          tcpConnState;
1350     /* local ip addr for this connection { tcpConnEntry 2 } */
1351     IpAddress    tcpConnLocalAddress;
1352     /* local port for this connection   { tcpConnEntry 3 } */
1353     int          tcpConnLocalPort;     /* In host byte order */
1354     /* remote ip addr for this connection { tcpConnEntry 4 } */
1355     IpAddress    tcpConnRemAddress;
1356     /* remote port for this connection   { tcpConnEntry 5 } */
1357     int          tcpConnRemPort;       /* In host byte order */
1358     struct tcpConnEntryInfo_s {
1359         /* seq # of next segment to send */
1360         Gauge    ce_snxt;
1361         /* seq # of of last segment unacknowledged */
1362         Gauge    ce_suna;
1363         /* current send window size */
1364         Gauge    ce_swnd;
1365         /* seq # of next expected segment */
1366         Gauge    ce_rnxt;
1367         /* seq # of last ack'd segment */
1368         Gauge    ce_rack;
1369         /* current receive window size */
1370         Gauge    ce_rwnd;
1371         /* current rto (retransmit timeout) */
1372         Gauge    ce_rto;
1373         /* current max segment size */
1374         Gauge    ce_mss;
1375         /* actual internal state */
1376         int      ce_state;
1377     }
1378     tcpConnEntryInfo;

1379     /* pid of the processes that created this connection */
1380     uint32_t    tcpConnCreationProcess;
1381     /* system uptime when the connection was created */
1382     uint64_t    tcpConnCreationTime;
1383 } mib2_tcpConnEntry_t;
1384 #define MIB_FIRST_NEW_ELM_mib2_tcpConnEntry_t  tcpConnCreationProcess

1386 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1387 #pragma pack(4)
1388 #endif

1391 /*
1392  * The TCP/IPV6 connection table {tcp 14} contains information about this
1393  * entity's existing TCP connections over IPV6.
1394  */

1396 /* Pack data to make struct size the same for 32- and 64-bits */
1397 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1398 #pragma pack(4)
1399 #endif
1400 typedef struct mib2_tcp6ConnEntry {

```

```

1401     /* local ip addr for this connection { ipv6TcpConnEntry 1 } */
1402     Ip6Address  tcp6ConnLocalAddress;
1403     /* local port for this connection   { ipv6TcpConnEntry 2 } */
1404     int          tcp6ConnLocalPort;
1405     /* remote ip addr for this connection { ipv6TcpConnEntry 3 } */
1406     Ip6Address  tcp6ConnRemAddress;
1407     /* remote port for this connection   { ipv6TcpConnEntry 4 } */
1408     int          tcp6ConnRemPort;
1409     /* interface index or zero          { ipv6TcpConnEntry 5 } */
1410     DeviceIndex tcp6ConnIfIndex;
1411     /* state of tcp6 connection        { ipv6TcpConnEntry 6 } RW */
1412     int          tcp6ConnState;
1413     struct tcp6ConnEntryInfo_s {
1414         /* seq # of next segment to send */
1415         Gauge    ce_snxt;
1416         /* seq # of of last segment unacknowledged */
1417         Gauge    ce_suna;
1418         /* current send window size */
1419         Gauge    ce_swnd;
1420         /* seq # of next expected segment */
1421         Gauge    ce_rnxt;
1422         /* seq # of last ack'd segment */
1423         Gauge    ce_rack;
1424         /* current receive window size */
1425         Gauge    ce_rwnd;
1426         /* current rto (retransmit timeout) */
1427         Gauge    ce_rto;
1428         /* current max segment size */
1429         Gauge    ce_mss;
1430         /* actual internal state */
1431         int      ce_state;
1432     }
1433     tcp6ConnEntryInfo;

1434     /* pid of the processes that created this connection */
1435     uint32_t    tcp6ConnCreationProcess;
1436     /* system uptime when the connection was created */
1437     uint64_t    tcp6ConnCreationTime;
1438 } mib2_tcp6ConnEntry_t;
1439 #define MIB_FIRST_NEW_ELM_mib2_tcp6ConnEntry_t  tcp6ConnCreationProcess

1441 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1442 #pragma pack(4)
1443 #endif

1445 /*
1446  * the UDP group
1447  */
1448 #define MIB2_UDP_ENTRY 5          /* udpEntry */
1449 #define MIB2_UDP6_ENTRY 6        /* udp6Entry */

1451 /* Old name retained for compatibility */
1452 #define MIB2_UDP_5          MIB2_UDP_ENTRY

1454 /* Pack data to make struct size the same for 32- and 64-bits */
1455 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1456 #pragma pack(4)
1457 #endif
1458 typedef struct mib2_udp {
1459     /* total # of UDP datagrams sent upstream { udp 1 } */
1460     Counter udpInDatagrams;
1461     /* in ip { udp 2 } */
1462     Counter udpInErrors;
1463     /* # of rcv'd dg's not deliverable (other) { udp 3 } */
1464     Counter udpInDatagrams;
1465     /* total # of dg's sent { udp 4 } */
1466     Counter udpOutDatagrams;
1467     /* { udp 5 } */

```

```

1467     int     udpEntrySize;      /* Size of udpEntry_t */
1468     int     udp6EntrySize;     /* Size of udp6Entry_t */
1469     Counter udpOutErrors;

1471     /*
1472     * fields from RFC 4113
1473     */

1475     /* total # of UDP datagrams sent upstream      { udp 8 } */
1476     Counter64  udpHCInDatagrams;
1477     /* total # of dg's sent                        { udp 9 } */
1478     Counter64  udpHCOutDatagrams;
1479 } mib2_udp_t;
1480 #define MIB_FIRST_NEW_ELM_mib2_udp_t    udpHCInDatagrams

1482 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1483 #pragma pack()
1484 #endif

1486 /*
1487 * The UDP listener table contains information about this entity's UDP
1488 * end-points on which a local application is currently accepting datagrams.
1489 */

1491 /* For both IPv4 and IPv6 ue_state: */
1492 #define MIB2_UDP_unbound      1
1493 #define MIB2_UDP_idle        2
1494 #define MIB2_UDP_connected   3
1495 #define MIB2_UDP_unknown    4

1497 /* Pack data to make struct size the same for 32- and 64-bits */
1498 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1499 #pragma pack(4)
1500 #endif
1501 typedef struct mib2_udpEntry {
1502     /* local ip addr of listener          { udpEntry 1 } */
1503     IpAddress  udpLocalAddress;
1504     /* local port of listener             { udpEntry 2 } */
1505     int        udpLocalPort; /* In host byte order */
1506     struct udpEntryInfo_s {
1507         int        ue_state;
1508         IpAddress  ue_RemoteAddress;
1509         int        ue_RemotePort; /* In host byte order */
1510     }          udpEntryInfo;

1512     /*
1513     * RFC 4113
1514     */

1516     /* Unique id for this 4-tuple          { udpEndpointEntry 7 } */
1517     uint32_t   udpInstance;
1518     /* pid of the processes that created this endpoint */
1519     uint32_t   udpCreationProcess;
1520     /* system uptime when the endpoint was created */
1521     uint64_t   udpCreationTime;
1522 } mib2_udpEntry_t;
1523 #define MIB_FIRST_NEW_ELM_mib2_udpEntry_t    udpInstance

1525 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1526 #pragma pack()
1527 #endif

1529 /*
1530 * The UDP (for IPv6) listener table contains information about this
1531 * entity's UDP end-points on which a local application is
1532 * currently accepting datagrams.

```

```

1533     */

1535     /* Pack data to make struct size the same for 32- and 64-bits */
1536     #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1537     #pragma pack(4)
1538     #endif
1539     typedef struct mib2_udp6Entry {
1540         /* local ip addr of listener          { ipv6UdpEntry 1 } */
1541         Ip6Address  udp6LocalAddress;
1542         /* local port of listener             { ipv6UdpEntry 2 } */
1543         int        udp6LocalPort; /* In host byte order */
1544         /* interface index or zero           { ipv6UdpEntry 3 } */
1545         DeviceIndex  udp6IfIndex;
1546         struct udp6EntryInfo_s {
1547             int        ue_state;
1548             Ip6Address  ue_RemoteAddress;
1549             int        ue_RemotePort; /* In host byte order */
1550         }          udp6EntryInfo;

1552         /*
1553         * RFC 4113
1554         */

1556         /* Unique id for this 4-tuple          { udpEndpointEntry 7 } */
1557         uint32_t   udp6Instance;
1558         /* pid of the processes that created this endpoint */
1559         uint32_t   udp6CreationProcess;
1560         /* system uptime when the endpoint was created */
1561         uint64_t   udp6CreationTime;
1562     } mib2_udp6Entry_t;
1563 #define MIB_FIRST_NEW_ELM_mib2_udp6Entry_t    udp6Instance

1565 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1566 #pragma pack()
1567 #endif

1569     /*
1570     * the RAWIP group
1571     */
1572     typedef struct mib2_rawip {
1573         /* total # of RAWIP datagrams sent upstream */
1574         Counter  rawipInDatagrams;
1575         /* # of RAWIP packets with bad IPV6_CHECKSUM checksums */
1576         Counter  rawipInCksumErrrs;
1577         /* # of recv'd dg's not deliverable (other) */
1578         Counter  rawipInErrors;
1579         /* total # of dg's sent */
1580         Counter  rawipOutDatagrams;
1581         /* total # of dg's not sent (e.g. no memory) */
1582         Counter  rawipOutErrors;
1583     } mib2_rawip_t;

1585     /* DVMRP group */
1586     #define EXPER_DVMRP_VIF      1
1587     #define EXPER_DVMRP_MRT     2

1590     /*
1591     * The SCTP group
1592     */
1593     #define MIB2_SCTP_CONN      15
1594     #define MIB2_SCTP_CONN_LOCAL 16
1595     #define MIB2_SCTP_CONN_REMOTE 17

1597     #define MIB2_SCTP_closed    1
1598     #define MIB2_SCTP_cookieWait 2

```

```

1599 #define MIB2_SCTP_cookieEchoed      3
1600 #define MIB2_SCTP_established        4
1601 #define MIB2_SCTP_shutdownPending    5
1602 #define MIB2_SCTP_shutdownSent       6
1603 #define MIB2_SCTP_shutdownReceived   7
1604 #define MIB2_SCTP_shutdownAckSent    8
1605 #define MIB2_SCTP_deleteTCB          9
1606 #define MIB2_SCTP_listen              10      /* Not in the MIB */

1608 #define MIB2_SCTP_ACTIVE               1
1609 #define MIB2_SCTP_INACTIVE             2

1611 #define MIB2_SCTP_ADDR_V4              1
1612 #define MIB2_SCTP_ADDR_V6             2

1614 #define MIB2_SCTP_RTOALGO_OTHER       1
1615 #define MIB2_SCTP_RTOALGO_VANJ       2

1617 typedef struct mib2_sctpConnEntry {
1618     /* connection identifier          { sctpAssocEntry 1 } */
1619     uint32_t      sctpAssocId;
1620     /* remote hostname (not used)     { sctpAssocEntry 2 } */
1621     Octet_t      sctpAssocRemHostName;
1622     /* local port number               { sctpAssocEntry 3 } */
1623     uint32_t      sctpAssocLocalPort;
1624     /* remote port number              { sctpAssocEntry 4 } */
1625     uint32_t      sctpAssocRemPort;
1626     /* type of primary remote addr    { sctpAssocEntry 5 } */
1627     int           sctpAssocRemPrimAddrType;
1628     /* primary remote address         { sctpAssocEntry 6 } */
1629     Ip6Address    sctpAssocRemPrimAddr;
1630     /* local address                  */
1631     Ip6Address    sctpAssocLocPrimAddr;
1632     /* current heartbeat interval     { sctpAssocEntry 7 } */
1633     uint32_t      sctpAssocHeartBeatInterval;
1634     /* state of this association       { sctpAssocEntry 8 } */
1635     int           sctpAssocState;
1636     /* # of inbound streams            { sctpAssocEntry 9 } */
1637     uint32_t      sctpAssocInStreams;
1638     /* # of outbound streams           { sctpAssocEntry 10 } */
1639     uint32_t      sctpAssocOutStreams;
1640     /* max # of data retrans           { sctpAssocEntry 11 } */
1641     uint32_t      sctpAssocMaxRetr;
1642     /* sysId for assoc owner          { sctpAssocEntry 12 } */
1643     uint32_t      sctpAssocPrimProcess;
1644     /* # of rxmit timeouts during handshake */
1645     Counter32    sctpAssocT1expired; /* { sctpAssocEntry 13 } */
1646     /* # of rxmit timeouts during shutdown */
1647     Counter32    sctpAssocT2expired; /* { sctpAssocEntry 14 } */
1648     /* # of rxmit timeouts during data transfer */
1649     Counter32    sctpAssocRtxChunks; /* { sctpAssocEntry 15 } */
1650     /* assoc start-up time            { sctpAssocEntry 16 } */
1651     uint32_t      sctpAssocStartTime;
1652     struct sctpConnEntryInfo_s {
1653         /* amount of data in send Q */
1654         Gauge     ce_sendq;
1655         /* amount of data in recv Q */
1656         Gauge     ce_recvq;
1657         /* current send window size */
1658         Gauge     ce_swnd;
1659         /* current receive window size */
1660         Gauge     ce_rwnd;
1661         /* current max segment size */
1662         Gauge     ce_mss;
1663     } sctpConnEntryInfo;
1664 } mib2_sctpConnEntry_t;

```

```

1666 typedef struct mib2_sctpConnLocalAddrEntry {
1667     /* connection identifier */
1668     uint32_t      sctpAssocId;
1669     /* type of local addr       { sctpAssocLocalEntry 1 } */
1670     int           sctpAssocLocalAddrType;
1671     /* local address           { sctpAssocLocalEntry 2 } */
1672     Ip6Address    sctpAssocLocalAddr;
1673 } mib2_sctpConnLocalEntry_t;

1675 typedef struct mib2_sctpConnRemoteAddrEntry {
1676     /* connection identifier */
1677     uint32_t      sctpAssocId;
1678     /* remote addr type       { sctpAssocRemEntry 1 } */
1679     int           sctpAssocRemAddrType;
1680     /* remote address         { sctpAssocRemEntry 2 } */
1681     Ip6Address    sctpAssocRemAddr;
1682     /* is the address active   { sctpAssocRemEntry 3 } */
1683     int           sctpAssocRemAddrActive;
1684     /* whether heartbeat is active { sctpAssocRemEntry 4 } */
1685     int           sctpAssocRemAddrHBActive;
1686     /* current RTO            { sctpAssocRemEntry 5 } */
1687     uint32_t      sctpAssocRemAddrRTO;
1688     /* max # of rexmits before becoming inactive */
1689     uint32_t      sctpAssocRemAddrMaxPathRtx; /* { sctpAssocRemEntry 6 } */
1690     /* # of rexmits to this dest { sctpAssocRemEntry 7 } */
1691     uint32_t      sctpAssocRemAddrRtx;
1692 } mib2_sctpConnRemoteEntry_t;

1696 /* Pack data in mib2_sctp to make struct size the same for 32- and 64-bits */
1697 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1698 #pragma pack(4)
1699 #endif

1701 typedef struct mib2_sctp {
1702     /* algorithm used to determine rto { sctpParams 1 } */
1703     int           sctpRtoAlgorithm;
1704     /* min RTO in msecs                { sctpParams 2 } */
1705     uint32_t      sctpRtoMin;
1706     /* max RTO in msecs                { sctpParams 3 } */
1707     uint32_t      sctpRtoMax;
1708     /* initial RTO in msecs            { sctpParams 4 } */
1709     uint32_t      sctpRtoInitial;
1710     /* max # of assocs                 { sctpParams 5 } */
1711     int32_t       sctpMaxAssocs;
1712     /* cookie lifetime in msecs        { sctpParams 6 } */
1713     uint32_t      sctpValCookieLife;
1714     /* max # of retrans in startup     { sctpParams 7 } */
1715     uint32_t      sctpMaxInitRetr;
1716     /* # of conns ESTABLISHED, SHUTDOWN-RECEIVED or SHUTDOWN-PENDING */
1717     Counter32    sctpCurrEstab; /* { sctpStats 1 } */
1718     /* # of active opens                { sctpStats 2 } */
1719     Counter32    sctpActiveEstab;
1720     /* # of passive opens               { sctpStats 3 } */
1721     Counter32    sctpPassiveEstab;
1722     /* # of aborted conns              { sctpStats 4 } */
1723     Counter32    sctpAborted;
1724     /* # of graceful shutdowns        { sctpStats 5 } */
1725     Counter32    sctpShutdowns;
1726     /* # of OOB packets                { sctpStats 6 } */
1727     Counter32    sctpOutOfBlue;
1728     /* # of packets discarded due to cksum { sctpStats 7 } */
1729     Counter32    sctpChecksumError;
1730     /* # of control chunks sent        { sctpStats 8 } */

```

```

1731 Counter64 sctpOutCtrlChunks;
1732 /* # of ordered data chunks sent { sctpStats 9 } */
1733 Counter64 sctpOutOrderChunks;
1734 /* # of unordered data chunks sent { sctpStats 10 } */
1735 Counter64 sctpOutUnorderChunks;
1736 /* # of retransmitted data chunks */
1737 Counter64 sctpRetransChunks;
1738 /* # of SACK chunks sent */
1739 Counter sctpOutAck;
1740 /* # of delayed ACK timeouts */
1741 Counter sctpOutAckDelayed;
1742 /* # of SACK chunks sent to update window */
1743 Counter sctpOutWinUpdate;
1744 /* # of fast retransmits */
1745 Counter sctpOutFastRetrans;
1746 /* # of window probes sent */
1747 Counter sctpOutWinProbe;
1748 /* # of control chunks received { sctpStats 11 } */
1749 Counter64 sctpInCtrlChunks;
1750 /* # of ordered data chunks rcvd { sctpStats 12 } */
1751 Counter64 sctpInOrderChunks;
1752 /* # of unord data chunks rcvd { sctpStats 13 } */
1753 Counter64 sctpInUnorderChunks;
1754 /* # of received SACK chunks */
1755 Counter sctpInAck;
1756 /* # of received SACK chunks with duplicate TSN */
1757 Counter sctpInDupAck;
1758 /* # of SACK chunks acking unsent data */
1759 Counter sctpInAckUnsent;
1760 /* # of Fragmented User Messages { sctpStats 14 } */
1761 Counter64 sctpFragUsrMsgs;
1762 /* # of Reassembled User Messages { sctpStats 15 } */
1763 Counter64 sctpReasmUsrMsgs;
1764 /* # of Sent SCTP Packets { sctpStats 16 } */
1765 Counter64 sctpOutSCTPPkts;
1766 /* # of Received SCTP Packets { sctpStats 17 } */
1767 Counter64 sctpInSCTPPkts;
1768 /* # of invalid cookies received */
1769 Counter sctpInInvalidCookie;
1770 /* total # of retransmit timeouts */
1771 Counter sctpTimRetrans;
1772 /* total # of retransmit timeouts dropping the connection */
1773 Counter sctpTimRetransDrop;
1774 /* total # of heartbeat probes */
1775 Counter sctpTimHeartBeatProbe;
1776 /* total # of heartbeat timeouts dropping the connection */
1777 Counter sctpTimHeartBeatDrop;
1778 /* total # of conns refused due to backlog full on listen */
1779 Counter sctpListenDrop;
1780 /* total # of pkts received after the association has closed */
1781 Counter sctpInClosed;
1782 int sctpEntrySize;
1783 int sctpLocalEntrySize;
1784 int sctpRemoteEntrySize;
1785 } mib2_sctp_t;

1787 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
1788 #pragma pack()
1789 #endif

1792 #ifdef __cplusplus
1793 }
1794 #endif

1796 #endif /* _INET_MIB2_H */

```

```

*****
32229 Wed Jul 13 01:32:38 2016
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 association 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 ophdr *optp;
545     sctp_t      *sctp, *sctp_prev = NULL;
546     sctp_faddr_t *fp;
547     mib2_sctpConnEntry_t sce;
548     mib2_sctpConnLocalEntry_t scl;
549     mib2_sctpConnRemoteEntry_t scre;
550     mib2_transportMLPEnt_t mlp;
551     int          i;
552     int          l;
553     int          scanned = 0;
554     zoneid_t    zoneid = Q_TO_CONN(q)->conn_zoneid;
555     conn_t      *connp;
556     boolean_t   needattr;
557     int         idx;
558     mib2_sctp_t sctp_mib;

560     /*
561     * Make copies of the original message.
562     * mpctl will hold SCTP counters,
563     * mp_conn_ctl will hold list of connections.
564     */
565     mp_ret = copymsg(mpctl);
566     mp_conn_ctl = copymsg(mpctl);
567     mp_pidnode_ctl = copymsg(mpctl);
568 #endif /* ! codereview */
569     mp_local_ctl = copymsg(mpctl);
570     mp_rem_ctl = copymsg(mpctl);
571     mp_attr_ctl = copymsg(mpctl);

573     mpdata = mpctl->b_cont;

575     if (mp_conn_ctl == NULL || mp_pidnode_ctl == NULL ||
576         mp_local_ctl == NULL || mp_rem_ctl == NULL || mp_attr_ctl == NULL ||

```

```

577     mpdata == NULL) {
581     if (mp_conn_ctl == NULL || mp_local_ctl == NULL ||
582         mp_rem_ctl == NULL || mp_attr_ctl == NULL || mpdata == NULL) {
583         freemsg(mp_attr_ctl);
584         freemsg(mp_rem_ctl);
585         freemsg(mp_local_ctl);
586         freemsg(mp_pidnode_ctl);
587     #endif /* ! codereview */
588     freemsg(mp_conn_ctl);
589     freemsg(mp_ret);
590     freemsg(mpctl);
591     return (NULL);
592 }
593 mp_conn_data = mp_conn_ctl->b_cont;
594 mp_pidnode_data = mp_pidnode_ctl->b_cont;
595 #endif /* ! codereview */
596 mp_local_data = mp_local_ctl->b_cont;
597 mp_rem_data = mp_rem_ctl->b_cont;
598 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;

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

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

```

```

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

670     sce.sctpAssocId = ntohl(sctp->sctp_lvtag);
671     sce.sctpAssocLocalPort = ntohs(sctp->sctp_connp->conn_lport);
672     sce.sctpAssocRemPort = ntohs(sctp->sctp_connp->conn_fport);

674     RUN_SCTP(sctp);
675     if (sctp->sctp_primary != NULL) {
676         fp = sctp->sctp_primary;

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

699     /*
700     * Table for local addresses
701     */
702     scanned = 0;
703     for (i = 0; i < SCTP_IPIF_HASH; i++) {
704         sctp_saddr_ipif_t *obj;

706         if (sctp->sctp_saddrs[i].ipif_count == 0)

```

```

707         continue;
708         obj = list_head(&sctp->sctp_saddrs[i].sctp_ipif_list);
709         for (l = 0; l < sctp->sctp_saddrs[i].ipif_count; l++) {
710             sctp_ipif_t *sctp_ipif;
711             in6_addr_t addr;

713             sctp_ipif = obj->saddr_ipif;
714             addr = sctp_ipif->sctp_ipif_saddr;
715             scanned++;
716             scl.e.sctpAssocId = ntohl(sctp->sctp_lvtag);
717             if (IN6_IS_ADDR_V4MAPPED(&addr)) {
718                 scl.e.sctpAssocLocalAddrType =
719                     MIB2_SCTP_ADDR_V4;
720             } else {
721                 scl.e.sctpAssocLocalAddrType =
722                     MIB2_SCTP_ADDR_V6;
723             }
724             scl.e.sctpAssocLocalAddr = addr;
725             (void) snmp_append_data2(mp_local_data,
726                 &mp_local_tail, (char *)&scl.e,
727                 sizeof (scl.e));
728             if (scanned >= sctp->sctp_nsaddrs)
729                 goto done;
730             obj = list_next(&sctp->
731                 sctp_saddrs[i].sctp_ipif_list, obj);
732         }
733     }
734 done:
735     /*
736     * Table for remote addresses
737     */
738     for (fp = sctp->sctp_faddrs; fp; fp = fp->sf_next) {
739         scre.sctpAssocId = ntohl(sctp->sctp_lvtag);
740         if (IN6_IS_ADDR_V4MAPPED(&fp->sf_faddr)) {
741             scre.sctpAssocRemAddrType = MIB2_SCTP_ADDR_V4;
742         } else {
743             scre.sctpAssocRemAddrType = MIB2_SCTP_ADDR_V6;
744         }
745         scre.sctpAssocRemAddr = fp->sf_faddr;
746         if (fp->sf_state == SCTP_FADDRS_ALIVE) {
747             scre.sctpAssocRemAddrActive =
748                 scre.sctpAssocRemAddrHBActive =
749                     MIB2_SCTP_ACTIVE;
750         } else {
751             scre.sctpAssocRemAddrActive =
752                 scre.sctpAssocRemAddrHBActive =
753                     MIB2_SCTP_INACTIVE;
754         }
755         scre.sctpAssocRemAddrRTO = TICK_TO_MSEC(fp->sf_rto);
756         scre.sctpAssocRemAddrMaxPathRtx = fp->sf_max_retr;
757         scre.sctpAssocRemAddrRtx = fp->sf_T3expire;
758         (void) snmp_append_data2(mp_rem_data, &mp_rem_tail,
759             (char *)&scre, sizeof (scre));
760     }
761     connp = sctp->sctp_connp;
762     needattr = B_FALSE;
763     bzero(&mlp, sizeof (mlp));
764     if (connp->conn_mlp_type != mlptSingle) {
765         if (connp->conn_mlp_type == mlptShared ||
766             connp->conn_mlp_type == mlptBoth)
767             mlp.tme_flags |= MIB2_TMEF_SHARED;
768         if (connp->conn_mlp_type == mlptPrivate ||
769             connp->conn_mlp_type == mlptBoth)
770             mlp.tme_flags |= MIB2_TMEF_PRIVATE;
771         needattr = B_TRUE;
772     }

```

```

773     if (connp->conn_anon_mlp) {
774         mlp.tme_flags |= MIB2_TMEF_ANONMLP;
775         needattr = B_TRUE;
776     }
777     switch (connp->conn_mac_mode) {
778     case CONN_MAC_DEFAULT:
779         break;
780     case CONN_MAC_AWARE:
781         mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
782         needattr = B_TRUE;
783         break;
784     case CONN_MAC_IMPLICIT:
785         mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
786         needattr = B_TRUE;
787         break;
788     }
789     if (sctp->sctp_connp->conn_ixa->ixa_tsl != NULL) {
790         ts_label_t *tsl;
791
792         tsl = sctp->sctp_connp->conn_ixa->ixa_tsl;
793         mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
794         mlp.tme_doi = label2doi(tsl);
795         mlp.tme_label = *label2bslabel(tsl);
796         needattr = B_TRUE;
797     }
798     WAKE_SCTP(sctp);
799     sce.sctpAssocState = sctp_snmp_state(sctp);
800     sce.sctpAssocInStreams = sctp->sctp_num_istr;
801     sce.sctpAssocOutStreams = sctp->sctp_num_ostr;
802     sce.sctpAssocMaxRetr = sctp->sctp_pa_max_rxt;
803     /* A 0 here indicates that no primary process is known */
804     sce.sctpAssocPrimProcess = 0;
805     sce.sctpAssocTlexpired = sctp->sctp_Tlexpire;
806     sce.sctpAssocT2expired = sctp->sctp_T2expire;
807     sce.sctpAssocRtxChunks = sctp->sctp_T3expire;
808     sce.sctpAssocStartTime = sctp->sctp_assoc_start_time;
809     sce.sctpConnEntryInfo.ce_sendq = sctp->sctp_unacked +
810         sctp->sctp_unsent;
811     sce.sctpConnEntryInfo.ce_recvq = sctp->sctp_rxqueued;
812     sce.sctpConnEntryInfo.ce_swnd = sctp->sctp_frwnd;
813     sce.sctpConnEntryInfo.ce_rwnd = sctp->sctp_rwnd;
814     sce.sctpConnEntryInfo.ce_mss = sctp->sctp_mss;
815     (void) snmp_append_data2(mp_conn_data, &mp_conn_tail,
816         (char *)&sce, sizeof (sce));
817
818     (void) snmp_append_data2(mp_pidnode_data, &mp_pidnode_tail,
819         (char *)&sce, sizeof (sce));
820
821     (void) snmp_append_mblk2(mp_pidnode_data, &mp_pidnode_tail,
822         conn_get_pid_mblk(connp));
823
824 #endif /* ! codereview */
825     mlp.tme_connid = idx++;
826     if (needattr)
827         (void) snmp_append_data2(mp_attr_ctl->b_cont,
828             &mp_attr_tail, (char *)&mlp, sizeof (mlp));
829 next_sctp:
830     sctp_prev = sctp;
831     mutex_enter(&sctps->sctps_g_lock);
832     sctp = list_next(&sctps->sctps_g_list, sctp);
833 }
834 mutex_exit(&sctps->sctps_g_lock);
835 if (sctp_prev != NULL)
836     SCTP_REFRELE(sctp_prev);
837
838 sctp_sum_mib(sctps, &sctp_mib);

```

```

840     optp = (struct ophdr *)&mpctl->b_rprtr[sizeof (struct T_optmgmt_ack)];
841     optp->level = MIB2_SCTP;
842     optp->name = 0;
843     (void) snmp_append_data(mpdata, (char *)&sctp_mib, sizeof (sctp_mib));
844     optp->len = msgsize(mpdata);
845     qreply(q, mpctl);
846
847     /* table of connections... */
848     optp = (struct ophdr *)&mp_conn_ctl->b_rprtr[
849         sizeof (struct T_optmgmt_ack)];
850     optp->level = MIB2_SCTP;
851     optp->name = MIB2_SCTP_CONN;
852     optp->len = msgsize(mp_conn_data);
853     qreply(q, mp_conn_ctl);
854
855     /* table of EXPR_XPORT_PROC_INFO */
856     optp = (struct ophdr *)&mp_pidnode_ctl->b_rprtr[
857         sizeof (struct T_optmgmt_ack)];
858     optp->level = MIB2_SCTP;
859     optp->name = EXPR_XPORT_PROC_INFO;
860     optp->len = msgsize(mp_pidnode_data);
861     qreply(q, mp_pidnode_ctl);
862 #endif /* ! codereview */
863
864     /* assoc local address table */
865     optp = (struct ophdr *)&mp_local_ctl->b_rprtr[
866         sizeof (struct T_optmgmt_ack)];
867     optp->level = MIB2_SCTP;
868     optp->name = MIB2_SCTP_CONN_LOCAL;
869     optp->len = msgsize(mp_local_data);
870     qreply(q, mp_local_ctl);
871
872     /* assoc remote address table */
873     optp = (struct ophdr *)&mp_rem_ctl->b_rprtr[
874         sizeof (struct T_optmgmt_ack)];
875     optp->level = MIB2_SCTP;
876     optp->name = MIB2_SCTP_CONN_REMOTE;
877     optp->len = msgsize(mp_rem_data);
878     qreply(q, mp_rem_ctl);
879
880     /* table of MLP attributes */
881     optp = (struct ophdr *)&mp_attr_ctl->b_rprtr[
882         sizeof (struct T_optmgmt_ack)];
883     optp->level = MIB2_SCTP;
884     optp->name = EXPR_XPORT_MLP;
885     optp->len = msgsize(mp_attr_data);
886     if (optp->len == 0)
887         freemsg(mp_attr_ctl);
888     else
889         qreply(q, mp_attr_ctl);
890
891     return (mp_ret);
892 }
893
894 /* Translate SCTP state to MIB2 SCTP state. */
895 static int
896 sctp_snmp_state(sctp_t *sctp)
897 {
898     if (sctp == NULL)
899         return (0);
900
901     switch (sctp->sctp_state) {
902     case SCTPS_IDLE:
903     case SCTPS_BOUND:
904         return (MIB2_SCTP_closed);

```

```

905     case SCTPS_LISTEN:
906         return (MIB2_SCTP_listen);
907     case SCTPS_COOKIE_WAIT:
908         return (MIB2_SCTP_cookieWait);
909     case SCTPS_COOKIE_ECHORD:
910         return (MIB2_SCTP_cookieEchoed);
911     case SCTPS_ESTABLISHED:
912         return (MIB2_SCTP_established);
913     case SCTPS_SHUTDOWN_PENDING:
914         return (MIB2_SCTP_shutdownPending);
915     case SCTPS_SHUTDOWN_SENT:
916         return (MIB2_SCTP_shutdownSent);
917     case SCTPS_SHUTDOWN_RECEIVED:
918         return (MIB2_SCTP_shutdownReceived);
919     case SCTPS_SHUTDOWN_ACK_SENT:
920         return (MIB2_SCTP_shutdownAckSent);
921     default:
922         return (0);
923     }
924 }

926 /*
927  * To sum up all MIB2 stats for a sctp_stack_t from all per CPU stats. The
928  * caller should initialize the target mib2_sctp_t properly as this function
929  * just adds up all the per CPU stats.
930  */
931 static void
932 sctp_sum_mib(sctp_stack_t *sctps, mib2_sctp_t *sctp_mib)
933 {
934     int i;
935     int cnt;

937     /* Static componets of mib2_sctp.t. */
938     SET_MIB(sctp_mib->sctpRtoAlgorithm, MIB2_SCTP_RTOALGO_VANJ);
939     SET_MIB(sctp_mib->sctpRtoMin, sctps->sctps_rto_ming);
940     SET_MIB(sctp_mib->sctpRtoMax, sctps->sctps_rto_maxg);
941     SET_MIB(sctp_mib->sctpRtoInitial, sctps->sctps_rto_initialg);
942     SET_MIB(sctp_mib->sctpMaxAssocs, -1);
943     SET_MIB(sctp_mib->sctpValCookieLife, sctps->sctps_cookie_life);
944     SET_MIB(sctp_mib->sctpMaxInitRetr, sctps->sctps_max_init_retr);

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

953     /*
954      * sctps_sc_cnt may change in the middle of the loop. It is better
955      * to get its value first.
956      */
957     cnt = sctps->sctps_sc_cnt;
958     for (i = 0; i < cnt; i++)
959         sctp_add_mib(&sctps->sctps_sc[i]->sctp_sc_mib, sctp_mib);
960 }

962 static void
963 sctp_add_mib(mib2_sctp_t *from, mib2_sctp_t *to)
964 {
965     to->sctpActiveEstab += from->sctpActiveEstab;
966     to->sctpPassiveEstab += from->sctpPassiveEstab;
967     to->sctpAborted += from->sctpAborted;
968     to->sctpShutdowns += from->sctpShutdowns;
969     to->sctpOutOfBlue += from->sctpOutOfBlue;
970     to->sctpChecksumError += from->sctpChecksumError;

```

```

971     to->sctpOutCtrlChunks += from->sctpOutCtrlChunks;
972     to->sctpOutOrderChunks += from->sctpOutOrderChunks;
973     to->sctpOutUnorderChunks += from->sctpOutUnorderChunks;
974     to->sctpRetransChunks += from->sctpRetransChunks;
975     to->sctpOutAck += from->sctpOutAck;
976     to->sctpOutAckDelayed += from->sctpOutAckDelayed;
977     to->sctpOutWinUpdate += from->sctpOutWinUpdate;
978     to->sctpOutFastRetrans += from->sctpOutFastRetrans;
979     to->sctpOutWinProbe += from->sctpOutWinProbe;
980     to->sctpInCtrlChunks += from->sctpInCtrlChunks;
981     to->sctpInOrderChunks += from->sctpInOrderChunks;
982     to->sctpInUnorderChunks += from->sctpInUnorderChunks;
983     to->sctpInAck += from->sctpInAck;
984     to->sctpInDupAck += from->sctpInDupAck;
985     to->sctpInAckUnsent += from->sctpInAckUnsent;
986     to->sctpFragUsrMsgs += from->sctpFragUsrMsgs;
987     to->sctpReasmUsrMsgs += from->sctpReasmUsrMsgs;
988     to->sctpOutSCTPPkts += from->sctpOutSCTPPkts;
989     to->sctpInSCTPPkts += from->sctpInSCTPPkts;
990     to->sctpInInvalidCookie += from->sctpInInvalidCookie;
991     to->sctpTimRetrans += from->sctpTimRetrans;
992     to->sctpTimRetransDrop += from->sctpTimRetransDrop;
993     to->sctpTimHeartBeatProbe += from->sctpTimHeartBeatProbe;
994     to->sctpTimHeartBeatDrop += from->sctpTimHeartBeatDrop;
995     to->sctpListenDrop += from->sctpListenDrop;
996     to->sctpInClosed += from->sctpInClosed;
997 }

```

```

*****
9932 Wed Jul 13 01:32:39 2016
new/usr/src/uts/common/inet/snmpcom.c
XXXX adding PID information to netstat output
*****
_____unchanged_portion_omitted_____

110 int
111 snmp_append_mblk(mblk_t *mpdata, mblk_t *mblk)
112 {
113     if (!mpdata || !mblk)
114         return (0);
115     while (mpdata->b_cont)
116         mpdata = mpdata->b_cont;
117     mpdata->b_cont = mblk;
118     return (1);
119 }

121 #endif /* ! codereview */
122 /*
123  * Need a form which avoids O(n^2) behavior locating the end of the
124  * chain every time. This is it.
125  */
126 int
127 snmp_append_data2(mblk_t *mpdata, mblk_t **last_mpp, char *blob, int len)
128 {
129
130     if (!mpdata)
131         return (0);
132     if (*last_mpp == NULL) {
133         while (mpdata->b_cont)
134             mpdata = mpdata->b_cont;
135         *last_mpp = mpdata;
136     }
137     if ((*last_mpp)->b_wptr + len >= (*last_mpp)->b_datap->db_lim) {
138         (*last_mpp)->b_cont = allocb(DATA_MBLK_SIZE, BPRI_HI);
139         *last_mpp = (*last_mpp)->b_cont;
140         if (!*last_mpp)
141             return (0);
142     }
143     bcopy(blob, (char *)(*last_mpp)->b_wptr, len);
144     (*last_mpp)->b_wptr += len;
145     return (1);
146 }

148 int
149 snmp_append_mblk2(mblk_t *mpdata, mblk_t **last_mpp, mblk_t *mblk)
150 {
151     if (!mpdata || !mblk)
152         return (0);
153     if (*last_mpp == NULL) {
154         while (mpdata->b_cont)
155             mpdata = mpdata->b_cont;
156         *last_mpp = mpdata;
157     }
158     (*last_mpp)->b_cont = mblk;
159     *last_mpp = (*last_mpp)->b_cont;
160 #endif /* ! codereview */
161     return (1);
162 }

164 /*
165  * SNMP requests are issued using putmsg() on a stream containing all
166  * relevant modules. The ctl part contains a O_T_OPTMGMT_REQ message,
167  * and the data part is NULL
168  * to process this msg. If snmpcom_req() returns FALSE, then the module

```

```

169  * will try optcom_req to see if its some sort of SOCKET or IP option.
170  * snmpcom_req returns TRUE whenever the first option is recognized as
171  * an SNMP request, even if a bad one.
172  *
173  * "get" is done by a single O_T_OPTMGMT_REQ with MGMT_flags set to T_CURRENT.
174  * All modules respond with one or msg's about what they know. Responses
175  * are in T_OPTMGMT_ACK format. The ophdr level/name fields identify what
176  * is begin returned, the len field how big it is (in bytes). The info
177  * itself is in the data portion of the msg. Fixed length info returned
178  * in one msg; each table in a separate msg.
179  *
180  * setfn() returns 1 if things ok, 0 if set request invalid or otherwise
181  * messed up.
182  *
183  * If the passed q is at the bottom of the module chain (q_next == NULL,
184  * a ctl msg with req->name, level, len all zero is sent upstream. This
185  * is and EOD flag to the caller.
186  *
187  * IMPORTANT:
188  * - The msg type is M_PROTO, not M_PCPROTO!!! This is by design,
189  * since multiple messages will be sent to stream head and we want
190  * them queued for reading, not discarded.
191  * - All requests which match a table entry are sent to all get/set functions
192  * of each module. The functions must simply ignore requests not meant
193  * for them: getfn() returns 0, setfn() returns 1.
194  */
195 boolean_t
196 snmpcom_req(queue_t *q, mblk_t *mp, pfi_t setfn, pfi_t getfn, cred_t *credp)
197 {
198     mblk_t                *mpctl;
199     struct ophdr          *req;
200     struct ophdr          *next_req;
201     struct ophdr          *req_end;
202     struct ophdr          *req_start;
203     sor_t                 *sreq;
204     struct T_optmgmt_req  *tor = (struct T_optmgmt_req *)mp->b_rptr;
205     struct T_optmgmt_ack  *toa;
206     boolean_t             legacy_req;

208     if (mp->b_cont) { /* don't deal with multiple mblk's */
209         freemsg(mp->b_cont);
210         mp->b_cont = (mblk_t *)0;
211         optcom_err_ack(q, mp, TSYSERR, EBADMSG);
212         return (B_TRUE);
213     }
214     if ((mp->b_wptr - mp->b_rptr) < sizeof (struct T_optmgmt_req) ||
215         !(req_start = (struct ophdr *)mi_offset_param(mp,
216             tor->OPT_offset, tor->OPT_length)))
217         goto bad_req;
218     if (! __TPI_OPT_ISALIGNED(req_start))
219         goto bad_req;

221     /*
222      * if first option not in the MIB2 or EXPER range, return false so
223      * optcom_req can scope things out. Otherwise it's passed to each
224      * calling module to process or ignore as it sees fit.
225      */
226     if ((!(req_start->level >= MIB2_RANGE_START &&
227         req_start->level <= MIB2_RANGE_END)) &&
228         (!(req_start->level >= EXPER_RANGE_START &&
229         req_start->level <= EXPER_RANGE_END)))
230         return (B_FALSE);

232     switch (tor->MGMT_flags) {
234     case T_NEGOTIATE:

```

```

235     if (secpolicy_ip_config(credp, B_FALSE) != 0) {
236         optcom_err_ack(q, mp, TACCES, 0);
237         return (B_TRUE);
238     }
239     req_end = (struct ophdr *)((uchar_t *)req_start +
240         tor->OPT_length);
241     for (req = req_start; req < req_end; req = next_req) {
242         next_req =
243             (struct ophdr *)((uchar_t *)&req[1] +
244                 TPI_ALIGN_OPT(req->len));
245         if (next_req > req_end)
246             goto bad_req2;
247         for (sreq = req_arr; sreq < A_END(req_arr); sreq++) {
248             if (req->level == sreq->sor_group &&
249                 req->name == sreq->sor_code)
250                 break;
251         }
252         if (sreq >= A_END(req_arr))
253             goto bad_req3;
254         if (!(*setfn)(q, req->level, req->name,
255             (uchar_t *)&req[1], req->len))
256             goto bad_req4;
257     }
258     if (q->q_next != NULL)
259         putnext(q, mp);
260     else
261         freemsg(mp);
262     return (B_TRUE);
263
264 case OLD_T_CURRENT:
265 case T_CURRENT:
266     mpctl = allocb(TOAHDR_SIZE, BPRI_MED);
267     if (!mpctl) {
268         optcom_err_ack(q, mp, TSYSERR, ENOMEM);
269         return (B_TRUE);
270     }
271     mpctl->b_cont = allocb(DATA_MBLK_SIZE, BPRI_MED);
272     if (!mpctl->b_cont) {
273         freemsg(mpctl);
274         optcom_err_ack(q, mp, TSYSERR, ENOMEM);
275         return (B_TRUE);
276     }
277     mpctl->b_datap->db_type = M_PROTO;
278     mpctl->b_wptr += TOAHDR_SIZE;
279     toa = (struct T_optmgmt_ack *)mpctl->b_rptr;
280     toa->PRIM_type = T_OPTMGMT_ACK;
281     toa->OPT_offset = sizeof (struct T_optmgmt_ack);
282     toa->OPT_length = sizeof (struct ophdr);
283     toa->MGMT_flags = T_SUCCESS;
284     /*
285     * If the current process is running inside a solaris10-
286     * branded zone and len is 0 then it's a request for
287     * legacy data.
288     */
289     if (PROC_IS_BRANDED(curproc) &&
290         (strcmp(curproc->p_brand->b_name, "solaris10") == 0) &&
291         (req_start->len == 0))
292         legacy_req = B_TRUE;
293     else
294         legacy_req = B_FALSE;
295     if (!(*getfn)(q, mpctl, req_start->level, legacy_req))
296         freemsg(mpctl);
297     /*
298     * all data for this module has now been sent upstream. If
299     * this is bottom module of stream, send up an EOD ctl msg,
300     * otherwise pass onto the next guy for processing.

```

```

301     */
302     if (q->q_next != NULL) {
303         putnext(q, mp);
304         return (B_TRUE);
305     }
306     if (mp->b_cont) {
307         freemsg(mp->b_cont);
308         mp->b_cont = NULL;
309     }
310     mpctl = realloc(mp, TOAHDR_SIZE, 1);
311     if (!mpctl) {
312         optcom_err_ack(q, mp, TSYSERR, ENOMEM);
313         return (B_TRUE);
314     }
315     mpctl->b_datap->db_type = M_PROTO;
316     mpctl->b_wptr = mpctl->b_rptr + TOAHDR_SIZE;
317     toa = (struct T_optmgmt_ack *)mpctl->b_rptr;
318     toa->PRIM_type = T_OPTMGMT_ACK;
319     toa->OPT_offset = sizeof (struct T_optmgmt_ack);
320     toa->OPT_length = sizeof (struct ophdr);
321     toa->MGMT_flags = T_SUCCESS;
322     req = (struct ophdr *)&toa[1];
323     req->level = 0;
324     req->name = 0;
325     req->len = 0;
326     qreply(q, mpctl);
327     return (B_TRUE);
328
329     default:
330         optcom_err_ack(q, mp, TBADFLAG, 0);
331         return (B_TRUE);
332     }
333
334 bad_req1:;
335     printf("snmpcom bad_req1\n");
336     goto bad_req;
337 bad_req2:;
338     printf("snmpcom bad_req2\n");
339     goto bad_req;
340 bad_req3:;
341     printf("snmpcom bad_req3\n");
342     goto bad_req;
343 bad_req4:;
344     printf("snmpcom bad_req4\n");
345     /* FALLTHRU */
346 bad_req:;
347     optcom_err_ack(q, mp, TBADOPT, 0);
348     return (B_TRUE);
349
350 }

```

new/usr/src/uts/common/inet/snmpcom.h

1

1768 Wed Jul 13 01:32:39 2016

new/usr/src/uts/common/inet/snmpcom.h

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) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
23 */
24 /* Copyright (c) 1990 Mentat Inc. */

26 #ifndef _INET_SNMPCOM_H
27 #define _INET_SNMPCOM_H

29 #ifdef __cplusplus
30 extern "C" {
31 #endif

33 #if defined(_KERNEL) && defined(__STDC__)

35 /* snmpcom_req function prototypes */
36 typedef int (*snmp_setf_t)(queue_t *, int, int, uchar_t *, int);
37 typedef int (*snmp_getf_t)(queue_t *, mblk_t *, int, boolean_t);

39 extern int      snmp_append_data(mblk_t *mpdata, char *blob, int len);
40 extern int      snmp_append_mblk(mblk_t *mpdata, mblk_t *mblk);
41 #endif /* ! codereview */
42 extern int      snmp_append_data2(mblk_t *mpdata, mblk_t **last_mpp,
43                                  char *blob, int len);
44 extern int      snmp_append_mblk2(mblk_t *mpdata, mblk_t **last_mpp,
45                                  mblk_t *mblk);
46 #endif /* ! codereview */

48 extern boolean_t      snmpcom_req(queue_t *q, mblk_t *mp,
49                                  snmp_setf_t setfn, snmp_getf_t getfn, cred_t *cr);

51 #endif /* defined(_KERNEL) && defined(__STDC__) */

53 #ifdef __cplusplus
54 }
55 #endif

57 #endif /* _INET_SNMPCOM_H */
```

```

*****
54263 Wed Jul 13 01:32:39 2016
new/usr/src/uts/common/inet/sockmods/socksctp.c
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2015 Joyent, Inc. All rights reserved.
25  */

27 #include <sys/types.h>
28 #include <sys/t_lock.h>
29 #include <sys/param.h>
30 #include <sys/system.h>
31 #include <sys/buf.h>
32 #include <sys/vfs.h>
33 #include <sys/vnode.h>
34 #include <sys/fcntl.h>
35 #endif /* ! codereview */
36 #include <sys/debug.h>
37 #include <sys/errno.h>
38 #include <sys/stropts.h>
39 #include <sys/cmn_err.h>
40 #include <sys/sysmacros.h>
41 #include <sys/filio.h>
42 #include <sys/policy.h>

44 #include <sys/project.h>
45 #include <sys/tihdr.h>
46 #include <sys/strsubr.h>
47 #include <sys/esunddi.h>
48 #include <sys/ddi.h>

50 #include <sys/sockio.h>
51 #include <sys/socket.h>
52 #include <sys/socketvar.h>
53 #include <sys/strsun.h>

55 #include <netinet/sctp.h>
56 #include <inet/sctp_if.h>
57 #include <fs/sockfs/sockcommon.h>
58 #include "socksctp.h"

60 /*
61  * SCTP sockfs sonode operations, 1-1 socket

```

```

62 */
63 static int soscetp_init(struct sonode *, struct sonode *, struct cred *, int);
64 static int soscetp_accept(struct sonode *, int, struct cred *, struct sonode **);
65 static int soscetp_bind(struct sonode *, struct sockaddr *, socklen_t, int,
66     struct cred *);
67 static int soscetp_listen(struct sonode *, int, struct cred *);
68 static int soscetp_connect(struct sonode *, struct sockaddr *, socklen_t,
69     int, int, struct cred *);
70 static int soscetp_recvmmsg(struct sonode *, struct nmsg_hdr *, struct uio *,
71     struct cred *);
72 static int soscetp_sendmmsg(struct sonode *, struct nmsg_hdr *, struct uio *,
73     struct cred *);
74 static int soscetp_getpeername(struct sonode *, struct sockaddr *, socklen_t *,
75     boolean_t, struct cred *);
76 static int soscetp_getsockname(struct sonode *, struct sockaddr *, socklen_t *,
77     struct cred *);
78 static int soscetp_shutdown(struct sonode *, int, struct cred *);
79 static int soscetp_getsockopt(struct sonode *, int, int, void *, socklen_t *,
80     struct cred *);
81 static int soscetp_setsockopt(struct sonode *, int, int, const void *,
82     socklen_t, struct cred *);
83 static int soscetp_ioctl(struct sonode *, int, intptr_t, int, struct cred *,
84     int32_t *);
85 static int soscetp_close(struct sonode *, int, struct cred *);
86 void soscetp_fini(struct sonode *, struct cred *);

88 /*
89  * SCTP sockfs sonode operations, 1-N socket
90  */
91 static int soscetp_seq_connect(struct sonode *, struct sockaddr *,
92     socklen_t, int, int, struct cred *);
93 static int soscetp_seq_sendmmsg(struct sonode *, struct nmsg_hdr *, struct uio *,
94     struct cred *);

96 /*
97  * Socket association upcalls, 1-N socket connection
98  */
99 sock_upper_handle_t sctp_assoc_newconn(sock_upper_handle_t,
100     sock_lower_handle_t, sock_downcalls_t *, struct cred *, pid_t,
101     sock_upcalls_t **);
102 static void sctp_assoc_connected(sock_upper_handle_t, sock_connid_t,
103     struct cred *, pid_t);
104 static int sctp_assoc_disconnected(sock_upper_handle_t, sock_connid_t, int);
105 static void sctp_assoc_disconnecting(sock_upper_handle_t, sock_opctl_action_t,
106     uintptr_t arg);
107 static ssize_t sctp_assoc_recv(sock_upper_handle_t, mblk_t *, size_t, int,
108     int *, boolean_t *);
109 static void sctp_assoc_xmitted(sock_upper_handle_t, boolean_t);
110 static void sctp_assoc_properties(sock_upper_handle_t,
111     struct sock_proto_props *);
112 static mblk_t *sctp_get_sock_pid_mblk(sock_upper_handle_t);
113 #endif /* ! codereview */

115 sonodeops_t soscetp_sonodeops = {
116     soscetp_init,           /* sop_init */
117     soscetp_accept,       /* sop_accept */
118     soscetp_bind,         /* sop_bind */
119     soscetp_listen,       /* sop_listen */
120     soscetp_connect,      /* sop_connect */
121     soscetp_recvmmsg,     /* sop_recvmmsg */
122     soscetp_sendmmsg,     /* sop_sendmmsg */
123     so_sendmblock_notsupp, /* sop_sendmblock */
124     soscetp_getpeername,  /* sop_getpeername */
125     soscetp_getsockname,  /* sop_getsockname */
126     soscetp_shutdown,     /* sop_shutdown */
127     soscetp_getsockopt,   /* sop_getsockopt */

```

```

128     sosctp_setsockopt,          /* sop_setsockopt */
129     sosctp_ioctl,             /* sop_ioctl   */
130     so_poll,                  /* sop_poll    */
131     sosctp_close,             /* sop_close   */
132 };

134 sonodeops_t sosctp_seq_sonodeops = {
135     sosctp_init,               /* sop_init    */
136     so_accept_notsupp,        /* sop_accept  */
137     sosctp_bind,              /* sop_bind    */
138     sosctp_listen,            /* sop_listen  */
139     sosctp_seq_connect,       /* sop_connect */
140     sosctp_rcvmsg,            /* sop_rcvmsg  */
141     sosctp_seq_sendmsg,       /* sop_sendmsg */
142     so_sendmblock_notsupp,    /* sop_sendmblock */
143     so_getpeername_notsupp,    /* sop_getpeername */
144     sosctp_getsockname,       /* sop_getsockname */
145     so_shutdown_notsupp,     /* sop_shutdown */
146     sosctp_getsockopt,        /* sop_getsockopt */
147     sosctp_setsockopt,        /* sop_setsockopt */
148     sosctp_ioctl,             /* sop_ioctl   */
149     so_poll,                  /* sop_poll    */
150     sosctp_close,             /* sop_close   */
151 };

153 /* All the upcalls expect the upper handle to be sonode. */
154 sock_upcalls_t sosctp_sock_upcalls = {
155     so_newconn,
156     so_connected,
157     so_disconnected,
158     so_opctl,
159     so_queue_msg,
160     so_set_prop,
161     so_txq_full,
162     NULL, /* su_signal_oob */
163 };

165 /* All the upcalls expect the upper handle to be sctp_sonode/sctp_soassoc. */
166 sock_upcalls_t sosctp_assoc_upcalls = {
167     sctp_assoc_newconn,
168     sctp_assoc_connected,
169     sctp_assoc_disconnected,
170     sctp_assoc_disconnecting,
171     sctp_assoc_rcv,
172     sctp_assoc_properties,
173     sctp_assoc_xmitted,
174     NULL, /* su_rcv_space */
175     NULL, /* su_signal_oob */
176     NULL, /* su_set_error */
177     NULL, /* su_closed */
178     sctp_get_sock_pid_mblock
179 #endif /* !codereview */
180 };

182 /* ARGSUSED */
183 static int
184 sosctp_init(struct sonode *so, struct sonode *pso, struct cred *cr, int flags)
185 {
186     struct sctp_sonode *ss;
187     struct sctp_sonode *pss;
188     sctp_sockbuf_limits_t sbl;
189     int err;

191     ss = SOTOSSO(so);

193     if (pso != NULL) {

```

```

194     /*
195     * Passive open, just inherit settings from parent. We should
196     * not end up here for SOCK_SEQPACKET type sockets, since no
197     * new sonode is created in that case.
198     */
199     ASSERT(so->so_type == SOCK_STREAM);
200     pss = SOTOSSO(pso);

202     mutex_enter(&pso->so_lock);
203     so->so_state |= (SS_ISBOUND | SS_ISCONNECTED |
204         (pso->so_state & SS_ASYNC));
205     sosctp_so_inherit(pss, ss);
206     so->so_proto_props = pso->so_proto_props;
207     so->so_mode = pso->so_mode;
208     mutex_exit(&pso->so_lock);

210     return (0);
211 }

213 if ((err = secpolicy_basic_net_access(cr)) != 0)
214     return (err);

216 if (so->so_type == SOCK_STREAM) {
217     so->so_proto_handle = (sock_lower_handle_t)sctp_create(so,
218         NULL, so->so_family, so->so_type, SCTP_CAN_BLOCK,
219         &sosctp_sock_upcalls, &sbl, cr);
220     so->so_mode = SM_CONNREQUIRED;
221 } else {
222     ASSERT(so->so_type == SOCK_SEQPACKET);
223     so->so_proto_handle = (sock_lower_handle_t)sctp_create(ss,
224         NULL, so->so_family, so->so_type, SCTP_CAN_BLOCK,
225         &sosctp_assoc_upcalls, &sbl, cr);
226 }

228 if (so->so_proto_handle == NULL)
229     return (ENOMEM);

231 so->so_rcvbuf = sbl.sbl_rxbuf;
232 so->so_rcvlowat = sbl.sbl_rxlowat;
233 so->so_sndbuf = sbl.sbl_txbuf;
234 so->so_sndlowat = sbl.sbl_txlowat;

236     return (0);
237 }

239 /*
240 * Accept incoming connection.
241 */
242 /* ARGSUSED */
243 static int
244 sosctp_accept(struct sonode *so, int fflag, struct cred *cr,
245     struct sonode **nsop)
246 {
247     int error = 0;

249     if ((so->so_state & SS_ACCEPTCONN) == 0)
250         return (EINVAL);

252     error = so_acceptq_dequeue(so, (fflag & (FNONBLOCK|FNDELAY)), nsop);

254     return (error);
255 }

257 /*
258 * Bind local endpoint.
259 */

```

```

260 /*ARGSUSED*/
261 static int
262 sosctp_bind(struct sonode *so, struct sockaddr *name, socklen_t namelen,
263             int flags, struct cred *cr)
264 {
265     int error;
266
267     if (!(flags & _SOBIND_LOCK_HELD)) {
268         mutex_enter(&so->so_lock);
269         so_lock_single(so); /* Set SOLOCKED */
270     } else {
271         ASSERT(MUTEX_HELD(&so->so_lock));
272     }
273
274     /*
275      * X/Open requires this check
276      */
277     if (so->so_state & SS_CANTSENDMORE) {
278         error = EINVAL;
279         goto done;
280     }
281
282     /*
283      * Protocol module does address family checks.
284      */
285     mutex_exit(&so->so_lock);
286
287     error = sctp_bind((struct sctp_s *)so->so_proto_handle, name, namelen);
288
289     mutex_enter(&so->so_lock);
290     if (error == 0) {
291         so->so_state |= SS_ISBOUND;
292     } else {
293         eprintsoline(so, error);
294     }
295 done:
296     if (!(flags & _SOBIND_LOCK_HELD)) {
297         so_unlock_single(so, SOLOCKED);
298         mutex_exit(&so->so_lock);
299     } else {
300         /* If the caller held the lock don't release it here */
301         ASSERT(MUTEX_HELD(&so->so_lock));
302         ASSERT(so->so_flag & SOLOCKED);
303     }
304
305     return (error);
306 }
307
308 /*
309  * Turn socket into a listen socket.
310  */
311 /* ARGSUSED */
312 static int
313 sosctp_listen(struct sonode *so, int backlog, struct cred *cr)
314 {
315     int error = 0;
316
317     mutex_enter(&so->so_lock);
318     so_lock_single(so);
319
320     /*
321      * If this socket is trying to do connect, or if it has
322      * been connected, disallow.
323      */
324     if (so->so_state & (SS_ISCONNECTING | SS_ISCONNECTED |

```

```

325     SS_ISDISCONNECTING | SS_CANTRCVMORE | SS_CANTSENDMORE)) {
326         error = EINVAL;
327         eprintsoline(so, error);
328         goto done;
329     }
330
331     if (backlog < 0) {
332         backlog = 0;
333     }
334
335     /*
336      * If listen() is only called to change backlog, we don't
337      * need to notify protocol module.
338      */
339     if (so->so_state & SS_ACCEPTCONN) {
340         so->so_backlog = backlog;
341         goto done;
342     }
343
344     mutex_exit(&so->so_lock);
345     error = sctp_listen((struct sctp_s *)so->so_proto_handle);
346     mutex_enter(&so->so_lock);
347     if (error == 0) {
348         so->so_state |= (SS_ACCEPTCONN|SS_ISBOUND);
349         so->so_backlog = backlog;
350     } else {
351         eprintsoline(so, error);
352     }
353 done:
354     so_unlock_single(so, SOLOCKED);
355     mutex_exit(&so->so_lock);
356
357     return (error);
358 }
359
360 /*
361  * Active open.
362  */
363 /* ARGSUSED */
364 static int
365 sosctp_connect(struct sonode *so, struct sockaddr *name,
366               socklen_t namelen, int fflag, int flags, struct cred *cr)
367 {
368     int error = 0;
369     pid_t pid = curproc->p_pid;
370
371     ASSERT(so->so_type == SOCK_STREAM);
372
373     mutex_enter(&so->so_lock);
374     so_lock_single(so);
375
376     /*
377      * Can't connect() after listen(), or if the socket is already
378      * connected.
379      */
380     if (so->so_state & (SS_ACCEPTCONN|SS_ISCONNECTED|SS_ISCONNECTING)) {
381         if (so->so_state & SS_ISCONNECTED) {
382             error = EISCONN;
383         } else if (so->so_state & SS_ISCONNECTING) {
384             error = EALREADY;
385         } else {
386             error = EOPNOTSUPP;
387         }
388         eprintsoline(so, error);
389         goto done;
390     }
391

```

```

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  }
401
402  /*
403  * Connection is closing, or closed, don't allow reconnect.
404  * TCP allows this to proceed, but the socket remains unwritable.
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  }
413
414  if (name == NULL || namelen == 0) {
415      error = EINVAL;
416      eprintsoline(so, error);
417      goto done;
418  }
419
420  soisconnecting(so);
421  mutex_exit(&so->so_lock);
422
423  error = sctp_connect((struct sctp_s *)so->so_proto_handle,
424      name, namelen, cr, pid);
425
426  mutex_enter(&so->so_lock);
427  if (error == 0) {
428      /*
429      * Allow other threads to access the socket
430      */
431      error = sowaitconnected(so, fflag, 0);
432  }
433  done:
434  so_unlock_single(so, SOLOCKED);
435  mutex_exit(&so->so_lock);
436  return (error);
437  }
438
439  /*
440  * Active open for 1-N sockets, create a new association and
441  * call connect on that.
442  * If there parent hasn't been bound yet (this is the first association),
443  * make it so.
444  */
445  static int
446  sosctp_seq_connect(struct sonode *so, struct sockaddr *name,
447      socklen_t namelen, int fflag, int flags, struct cred *cr)
448  {
449      struct sctp_soassoc *ssa;
450      struct sctp_sonode *ss;
451      int error;
452
453      ASSERT(so->so_type == SOCK_SEQPACKET);
454
455      mutex_enter(&so->so_lock);
456      so_lock_single(so);

```

```

458      if (name == NULL || namelen == 0) {
459          error = EINVAL;
460          eprintsoline(so, error);
461          goto done;
462      }
463
464      ss = SOTOSSO(so);
465
466      error = sosctp_assoc_createconn(ss, name, namelen, NULL, 0, fflag,
467          cr, &ssa);
468      if (error != 0) {
469          if ((error == EHOSTUNREACH) && (flags & _SOCONNECT_XPG4_2)) {
470              error = ENETUNREACH;
471          }
472      }
473      if (ssa != NULL) {
474          SSA_REFRELE(ss, ssa);
475      }
476
477  done:
478      so_unlock_single(so, SOLOCKED);
479      mutex_exit(&so->so_lock);
480      return (error);
481  }
482
483  /*
484  * Receive data.
485  */
486  /* ARGSUSED */
487  static int
488  sosctp_recvmsg(struct sonode *so, struct nmsgHdr *msg, struct uio *uiop,
489      struct cred *cr)
490  {
491      struct sctp_sonode *ss = SOTOSSO(so);
492      struct sctp_soassoc *ssa = NULL;
493      int flags, error = 0;
494      struct T_unitdata_ind *tind;
495      ssize_t orig_resid = uiop->uio_resid;
496      int len, count, readcnt = 0;
497      socklen_t controllen, namelen;
498      void *opt;
499      mblk_t *mp;
500      rval_t rval;
501
502      controllen = msg->msg_controllen;
503      namelen = msg->msg_namelen;
504      flags = msg->msg_flags;
505      msg->msg_flags = 0;
506      msg->msg_controllen = 0;
507      msg->msg_namelen = 0;
508
509      if (so->so_type == SOCK_STREAM) {
510          if (!(so->so_state & (SS_ISCONNECTED|SS_ISCONNECTING|
511              SS_CANTRCVMORE))) {
512              return (ENOTCONN);
513          }
514      } else {
515          /* NOTE: Will come here from vop_read() as well */
516          /* For 1-N socket, recv() cannot be used. */
517          if (namelen == 0)
518              return (EOPNOTSUPP);
519          /*
520          * If there are no associations, and no new connections are
521          * coming in, there's not going to be new messages coming
522          * in either.
523          */

```

```

524         if (so->so_rcv_q_head == NULL && so->so_rcv_head == NULL &&
525             ss->ss_assoccnt == 0 && !(so->so_state & SS_ACCEPTCONN)) {
526             return (ENOTCONN);
527         }
528     }
529
530     /*
531     * out-of-band data not supported.
532     */
533     if (flags & MSG_OOB) {
534         return (EOPNOTSUPP);
535     }
536
537     /*
538     * flag possibilities:
539     *
540     * MSG_PEEK      Don't consume data
541     * MSG_WAITALL  Wait for full quantity of data (ignored if MSG_PEEK)
542     * MSG_DONTWAIT Non-blocking (same as FNDELAY | FNONBLOCK)
543     *
544     * MSG_WAITALL can return less than the full buffer if either
545     *
546     * 1. we would block and we are non-blocking
547     * 2. a full message cannot be delivered
548     *
549     * Given that we always get a full message from proto below,
550     * MSG_WAITALL is not meaningful.
551     */
552
553     mutex_enter(&so->so_lock);
554
555     /*
556     * Allow just one reader at a time.
557     */
558     error = so_lock_read_intr(so,
559         uiop->uio_fmode | ((flags & MSG_DONTWAIT) ? FNONBLOCK : 0));
560     if (error) {
561         mutex_exit(&so->so_lock);
562         return (error);
563     }
564     mutex_exit(&so->so_lock);
565
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         }
572
573         tind = (struct T_unitdata_ind *)mp->b_rptr;
574
575         len = tind->SRC_length;
576
577         if (namelen > 0 && len > 0) {
578             opt = sogetoff(mp, tind->SRC_offset, len, 1);
579
580             ASSERT(opt != NULL);
581
582             msg->msg_name = kmem_alloc(len, KM_SLEEP);
583             msg->msg_namelen = len;
584
585             bcopy(opt, msg->msg_name, len);
586         }
587
588         len = tind->OPT_length;
589         if (controllen == 0) {

```

```

590             if (len > 0) {
591                 msg->msg_flags |= MSG_CTRUNC;
592             }
593         } else if (len > 0) {
594             opt = sogetoff(mp, tind->OPT_offset, len,
595                 _TPI_ALIGN_SIZE);
596
597             ASSERT(opt != NULL);
598             socksctp_pack_cmsg(opt, msg, len);
599         }
600
601         if (mp->b_flag & SCTP_NOTIFICATION) {
602             msg->msg_flags |= MSG_NOTIFICATION;
603         }
604
605         if (!(mp->b_flag & SCTP_PARTIAL_DATA) &&
606             !(rval.r_val1 & MOREDATA)) {
607             msg->msg_flags |= MSG_EOR;
608         }
609         freemsg(mp);
610     }
611     done:
612     if (!(flags & MSG_PEEK))
613         readcnt = orig_resid - uiop->uio_resid;
614
615     /*
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;
624
625         ASSERT(so->so_rcv_q_head != NULL ||
626             so->so_rcv_head != NULL ||
627             so->so_rcv_queued == 0);
628
629         so_unlock_read(so);
630
631         /*
632         * so_dequeue_msg() sets r_val2 to true if flow control was
633         * cleared and we need to update SCTP. so_flowctrlrd 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_rcvvd((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;
653
654         so_unlock_read(so);
655
656         if (readcnt > 0 && ssa->ssa_flowctrlrd &&

```

```

656         ssa->ssa_rcv_queued < so->so_rcvlowat) {
657             /*
658              * Need to clear ssa_flowctrlrd, different from l-1
659              * style.
660              */
661             ssa->ssa_flowctrlrd = B_FALSE;
662             mutex_exit(&so->so_lock);
663             sctp_rcvrd(ssa->ssa_conn, count);
664             mutex_enter(&so->so_lock);
665         }
666
667     /*
668      * MOREDATA flag is set if all data could not be copied
669      */
670     if (!(flags & MSG_PEEK) && !(rval.r_vall & MOREDATA)) {
671         SSA_REFRELE(ss, ssa);
672     }
673     mutex_exit(&so->so_lock);
674 }
675
676 return (error);
677 }
678
679 int
680 sosctp_uiomove(mblk_t *hdr_mp, ssize_t count, ssize_t blk_size, int wroff,
681               struct uio *uiop, int flags)
682 {
683     ssize_t size;
684     int error;
685     mblk_t *mp;
686     dblk_t *dp;
687
688     if (blk_size == INFP SZ)
689         blk_size = count;
690
691     /*
692      * Loop until we have all data copied into mblk's.
693      */
694     while (count > 0) {
695         size = MIN(count, blk_size);
696
697         /*
698          * As a message can be splitted up and sent in different
699          * packets, each mblk will have the extra space before
700          * data to accommodate what SCTP wants to put in there.
701          */
702         while ((mp = allocb(size + wroff, BPRI_MED)) == NULL) {
703             if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
704                 (flags & MSG_DONTWAIT)) {
705                 return (EAGAIN);
706             }
707             if ((error = strwaitbuf(size + wroff, BPRI_MED))) {
708                 return (error);
709             }
710         }
711
712         dp = mp->b_datap;
713         dp->db_cpuid = curproc->p_pid;
714         ASSERT(wroff <= dp->db_lim - mp->b_wptra);
715         mp->b_rptr += wroff;
716         error = uiomove(mp->b_rptr, size, UIO_WRITE, uiop);
717         if (error != 0) {
718             freeb(mp);
719             return (error);
720         }
721         mp->b_wptra = mp->b_rptr + size;

```

```

722         count -= size;
723         hdr_mp->b_cont = mp;
724         hdr_mp = mp;
725     }
726     return (0);
727 }
728
729 /*
730  * Send message.
731  */
732 static int
733 sosctp_sendmsg(struct sonode *so, struct nmsg_hdr *msg, struct uio *uiop,
734               struct cred *cr)
735 {
736     mblk_t *mctl;
737     struct cmsghdr *cmsg;
738     struct sctp_sndrcvinfo *sinfo;
739     int optlen, flags, fflag;
740     ssize_t count, msglen;
741     int error;
742
743     ASSERT(so->so_type == SOCK_STREAM);
744
745     flags = msg->msg_flags;
746     if (flags & MSG_OOB) {
747         /*
748          * No out-of-band data support.
749          */
750         return (EOPNOTSUPP);
751     }
752
753     if (msg->msg_controllen != 0) {
754         optlen = msg->msg_controllen;
755         cmsg = sosctp_find_cmsg(msg->msg_control, optlen, SCTP_SNDRCV);
756         if (cmsg != NULL) {
757             if (cmsg->cmsg_len <
758                 (sizeof (*sinfo) + sizeof (*cmsg))) {
759                 eprintsoline(so, EINVAL);
760                 return (EINVAL);
761             }
762             sinfo = (struct sctp_sndrcvinfo *) (cmsg + 1);
763
764             /* Both flags should not be set together. */
765             if ((sinfo->sinfo_flags & MSG_EOF) &&
766                 (sinfo->sinfo_flags & MSG_ABORT)) {
767                 eprintsoline(so, EINVAL);
768                 return (EINVAL);
769             }
770
771             /* Initiate a graceful shutdown. */
772             if (sinfo->sinfo_flags & MSG_EOF) {
773                 /* Can't include data in MSG_EOF message. */
774                 if (uiop->uio_resid != 0) {
775                     eprintsoline(so, EINVAL);
776                     return (EINVAL);
777                 }
778             }
779
780             /*
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;

```

```

788         mutex_exit(&so->so_lock);
790
791         pollwakeuper(&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);
796
797         mutex_enter(&so->so_lock);
798         so_unlock_single(so, SOLOCKED);
799         mutex_exit(&so->so_lock);
800         return (error);
801     }
802 } else {
803     optlen = 0;
804 }
805
806 mutex_enter(&so->so_lock);
807 for (;;) {
808     if (so->so_state & SS_CANTSENDMORE) {
809         mutex_exit(&so->so_lock);
810         return (EPIPE);
811     }
812
813     if (so->so_error != 0) {
814         error = sogeterr(so, B_TRUE);
815         mutex_exit(&so->so_lock);
816         return (error);
817     }
818
819     if (!so->so_snd_qfull)
820         break;
821
822     if (so->so_state & SS_CLOSING) {
823         mutex_exit(&so->so_lock);
824         return (EINTR);
825     }
826     /*
827     * Xmit window full in a blocking socket.
828     */
829     if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
830         (flags & MSG_DONTWAIT)) {
831         mutex_exit(&so->so_lock);
832         return (EAGAIN);
833     } else {
834         /*
835         * Wait for space to become available and try again.
836         */
837         error = cv_wait_sig(&so->so_snd_cv, &so->so_lock);
838         if (!error) { /* signal */
839             mutex_exit(&so->so_lock);
840             return (EINTR);
841         }
842     }
843 }
844 msglen = count = uiop->uio_resid;
845
846 /* Don't allow sending a message larger than the send buffer size. */
847 /* XXX Transport module need to enforce this */
848 if (msglen > so->so_sndbuf) {
849     mutex_exit(&so->so_lock);
850     return (EMSGSIZE);
851 }
852
853 /*

```

```

854     * Allow piggybacking data on handshake messages (SS_ISCONNECTING).
855     */
856     if (!(so->so_state & (SS_ISCONNECTING | SS_ISCONNECTED))) {
857         /*
858         * We need to check here for listener so that the
859         * same error will be returned as with a TCP socket.
860         * In this case, sosctp_connect() returns EOPNOTSUPP
861         * while a TCP socket returns ENOTCONN instead. Catch it
862         * here to have the same behavior as a TCP socket.
863         */
864         * We also need to make sure that the peer address is
865         * provided before we attempt to do the connect.
866         */
867         if ((so->so_state & SS_ACCEPTCONN) ||
868             msg->msg_name == NULL) {
869             mutex_exit(&so->so_lock);
870             error = ENOTCONN;
871             goto error_nofree;
872         }
873         mutex_exit(&so->so_lock);
874         fflag = uiop->uio_fmode;
875         if (flags & MSG_DONTWAIT) {
876             fflag |= FNDELAY;
877         }
878         error = sosctp_connect(so, msg->msg_name, msg->msg_namelen,
879                               fflag, (so->so_version == SOV_XPG4_2) * _SOCONNECT_XPG4_2,
880                               cr);
881         if (error) {
882             /*
883             * Check for non-fatal errors, socket connected
884             * while the lock had been lifted.
885             */
886             if (error != EISCONN && error != EALREADY) {
887                 goto error_nofree;
888             }
889             error = 0;
890         }
891     } else {
892         mutex_exit(&so->so_lock);
893     }
894
895     mctl = sctp_alloc_hdr(msg->msg_name, msg->msg_namelen,
896                          msg->msg_control, optlen, SCTP_CAN_BLOCK);
897     if (mctl == NULL) {
898         error = EINTR;
899         goto error_nofree;
900     }
901
902     /* Copy in the message. */
903     if ((error = sosctp_uicomove(mctl, count, so->so_proto_props.sopp_maxblk,
904                                so->so_proto_props.sopp_wroff, uiop, flags)) != 0) {
905         goto error_ret;
906     }
907     error = sctp_sendmsg((struct sctp_s *)so->so_proto_handle, mctl, 0);
908     if (error == 0)
909         return (0);
910
911 error_ret:
912     freemsg(mctl);
913 error_nofree:
914     mutex_enter(&so->so_lock);
915     if ((error == EPIPE) && (so->so_state & SS_CANTSENDMORE)) {
916         /*
917         * We received shutdown between the time lock was
918         * lifted and call to sctp_sendmsg().
919         */

```

```

920         mutex_exit(&so->so_lock);
921         return (EPIPE);
922     }
923     mutex_exit(&so->so_lock);
924     return (error);
925 }

927 /*
928  * Send message on 1-N socket. Connects automatically if there is
929  * no association.
930  */
931 static int
932 sosctp_seq_sendmsg(struct sonode *so, struct nmsg_hdr *msg, struct uio *uiop,
933                  struct cred *cr)
934 {
935     struct sctp_sonode *ss;
936     struct sctp_soassoc *ssa;
937     struct cmsghdr *cmsgh;
938     struct sctp_sndrcvinfo *sinfo;
939     int aid = 0;
940     mblk_t *mctl;
941     int namelen, optlen, flags;
942     ssize_t count, msglen;
943     int error;
944     uint16_t s_flags = 0;

946     ASSERT(so->so_type == SOCK_SEQPACKET);

948     /*
949      * There shouldn't be problems with alignment, as the memory for
950      * msg_control was allocated with kmem_alloc.
951      */
952     cmsg = sosctp_find_cmsg(msg->msg_control, msg->msg_controllen,
953                          Sctp_Sndrcv);
954     if (cmsg != NULL) {
955         if (cmsg->cmsg_len < (sizeof (*sinfo) + sizeof (*cmsg))) {
956             eprintsoline(so, EINVAL);
957             return (EINVAL);
958         }
959         sinfo = (struct sctp_sndrcvinfo *) (cmsg + 1);
960         s_flags = sinfo->sinfo_flags;
961         aid = sinfo->sinfo_assoc_id;
962     }

964     ss = SOTOSSO(so);
965     namelen = msg->msg_namelen;

967     if (msg->msg_controllen > 0) {
968         optlen = msg->msg_controllen;
969     } else {
970         optlen = 0;
971     }

973     mutex_enter(&so->so_lock);

975     /*
976      * If there is no association id, connect to address specified
977      * in msg_name. Otherwise look up the association using the id.
978      */
979     if (aid == 0) {
980         /*
981          * Connect and shutdown cannot be done together, so check for
982          * MSG_EOF.
983          */
984         if (msg->msg_name == NULL || namelen == 0 ||
985             (s_flags & MSG_EOF)) {

```

```

986         error = EINVAL;
987         eprintsoline(so, error);
988         goto done;
989     }
990     flags = uiop->uio_fmode;
991     if (msg->msg_flags & MSG_DONTWAIT) {
992         flags |= FNDELAY;
993     }
994     so_lock_single(so);
995     error = sosctp_assoc_createconn(ss, msg->msg_name, namelen,
996                                   msg->msg_control, optlen, flags, cr, &ssa);
997     if (error) {
998         if ((so->so_version == SOV_XPG4_2) &&
999             (error == EHOSTUNREACH)) {
1000             error = ENETUNREACH;
1001         }
1002         if (ssa == NULL) {
1003             /*
1004              * Fatal error during connect(). Bail out.
1005              * If ssa exists, it means that the handshake
1006              * is in progress.
1007              */
1008             eprintsoline(so, error);
1009             so_unlock_single(so, SOLOCKED);
1010             goto done;
1011         }
1012         /*
1013          * All the errors are non-fatal ones, don't return
1014          * e.g. EINPROGRESS from sendmsg().
1015          */
1016         error = 0;
1017     }
1018     so_unlock_single(so, SOLOCKED);
1019 } else {
1020     if ((error = sosctp_assoc(ss, aid, &ssa)) != 0) {
1021         eprintsoline(so, error);
1022         goto done;
1023     }
1024 }

1026     /*
1027      * Now we have an association.
1028      */
1029     flags = msg->msg_flags;

1031     /*
1032      * MSG_EOF initiates graceful shutdown.
1033      */
1034     if (s_flags & MSG_EOF) {
1035         if (uiop->uio_resid) {
1036             /*
1037              * Can't include data in MSG_EOF message.
1038              */
1039             error = EINVAL;
1040         } else {
1041             mutex_exit(&so->so_lock);
1042             ssa->ssa_state |= SS_ISDISCONNECTING;
1043             sctp_rcvvd(ssa->ssa_conn, so->so_rcvbuf);
1044             error = sctp_disconnect(ssa->ssa_conn);
1045             mutex_enter(&so->so_lock);
1046         }
1047         goto refrele;
1048     }

1050     for (;;) {
1051         if (ssa->ssa_state & SS_CANTSENDMORE) {

```

```

1052         SSA_REFRELE(ss, ssa);
1053         mutex_exit(&so->so_lock);
1054         return (EPIPE);
1055     }
1056     if (ssa->ssa_error != 0) {
1057         error = ssa->ssa_error;
1058         ssa->ssa_error = 0;
1059         goto refrele;
1060     }
1062     if (!ssa->ssa_snd_qfull)
1063         break;
1065     if (so->so_state & SS_CLOSING) {
1066         error = EINTR;
1067         goto refrele;
1068     }
1069     if ((uiop->uio_fmode & (FNDELAY|FNONBLOCK)) ||
1070         (flags & MSG_DONTWAIT)) {
1071         error = EAGAIN;
1072         goto refrele;
1073     } else {
1074         /*
1075          * Wait for space to become available and try again.
1076          */
1077         error = cv_wait_sig(&so->so_snd_cv, &so->so_lock);
1078         if (!error) { /* signal */
1079             error = EINTR;
1080             goto refrele;
1081         }
1082     }
1083 }
1085 msglen = count = uiop->uio_resid;
1087 /* Don't allow sending a message larger than the send buffer size. */
1088 if (msglen > so->so_sndbuf) {
1089     error = EMSGSIZE;
1090     goto refrele;
1091 }
1093 /*
1094  * Update TX buffer usage here so that we can lift the socket lock.
1095  */
1096 mutex_exit(&so->so_lock);
1098 mctl = sctp_alloc_hdr(msg->msg_name, namelen, msg->msg_control,
1099     optlen, SCTP_CAN_BLOCK);
1100 if (mctl == NULL) {
1101     error = EINTR;
1102     goto lock_rele;
1103 }
1105 /* Copy in the message. */
1106 if ((error = sosctp_uicomove(mctl, count, ssa->ssa_wrsz,
1107     ssa->ssa_wroff, uiop, flags)) != 0) {
1108     goto lock_rele;
1109 }
1110 error = sctp_sendmsg((struct sctp_s *)ssa->ssa_conn, mctl, 0);
1111 lock_rele:
1112 mutex_enter(&so->so_lock);
1113 if (error != 0) {
1114     freemsg(mctl);
1115     if ((error == EPIPE) && (ssa->ssa_state & SS_CANTSENDMORE)) {
1116         /*
1117          * We received shutdown between the time lock was

```

```

1118         * lifted and call to sctp_sendmsg().
1119         */
1120         SSA_REFRELE(ss, ssa);
1121         mutex_exit(&so->so_lock);
1122         return (EPIPE);
1123     }
1124 }
1126 refrele:
1127     SSA_REFRELE(ss, ssa);
1128 done:
1129     mutex_exit(&so->so_lock);
1130     return (error);
1131 }
1133 /*
1134  * Get address of remote node.
1135  */
1136 /* ARGSUSED */
1137 static int
1138 sosctp_getpeername(struct sonode *so, struct sockaddr *addr, socklen_t *addrlen,
1139     boolean_t accept, struct cred *cr)
1140 {
1141     return (sctp_getpeername((struct sctp_s *)so->so_proto_handle, addr,
1142         addrlen));
1143 }
1145 /*
1146  * Get local address.
1147  */
1148 /* ARGSUSED */
1149 static int
1150 sosctp_getsockname(struct sonode *so, struct sockaddr *addr, socklen_t *addrlen,
1151     struct cred *cr)
1152 {
1153     return (sctp_getsockname((struct sctp_s *)so->so_proto_handle, addr,
1154         addrlen));
1155 }
1157 /*
1158  * Called from shutdown().
1159  */
1160 /* ARGSUSED */
1161 static int
1162 sosctp_shutdown(struct sonode *so, int how, struct cred *cr)
1163 {
1164     uint_t state_change;
1165     int wakesig = 0;
1166     int error = 0;
1168     mutex_enter(&so->so_lock);
1169     /*
1170      * Record the current state and then perform any state changes.
1171      * Then use the difference between the old and new states to
1172      * determine which needs to be done.
1173      */
1174     state_change = so->so_state;
1176     switch (how) {
1177     case SHUT_RD:
1178         socantrcvmore(so);
1179         break;
1180     case SHUT_WR:
1181         socantsendmore(so);
1182         break;
1183     case SHUT_RDWR:

```

```

1184         socantsendmore(so);
1185         socantrcvmore(so);
1186         break;
1187     default:
1188         mutex_exit(&so->so_lock);
1189         return (EINVAL);
1190     }
1191
1192     state_change = so->so_state & ~state_change;
1193
1194     if (state_change & SS_CANTRCVMORE) {
1195         if (so->so_rcv_q_head == NULL) {
1196             cv_signal(&so->so_rcv_cv);
1197         }
1198         wakesig = POLLIN|POLLRDNORM;
1199
1200         socket_sendsig(so, SOCKETSIG_READ);
1201     }
1202     if (state_change & SS_CANTSENDMORE) {
1203         cv_broadcast(&so->so_snd_cv);
1204         wakesig |= POLLOUT;
1205
1206         so->so_state |= SS_ISDISCONNECTING;
1207     }
1208     mutex_exit(&so->so_lock);
1209
1210     pollwakeuper(so->so_poll_list, wakesig);
1211
1212     if (state_change & SS_CANTSENDMORE) {
1213         sctp_rcvrd((struct sctp_s *)so->so_proto_handle, so->so_rcvbuf);
1214         error = sctp_disconnect((struct sctp_s *)so->so_proto_handle);
1215     }
1216
1217     /*
1218     * HACK: sctp_disconnect() may return EWOULDBLOCK. But this error is
1219     * not documented in standard socket API. Catch it here.
1220     */
1221     if (error == EWOULDBLOCK)
1222         error = 0;
1223     return (error);
1224 }
1225
1226 /*
1227 * Get socket options.
1228 */
1229 /* ARGSUSED5 */
1230 static int
1231 sosctp_getsockopt(struct sonode *so, int level, int option_name,
1232                 void *optval, socklen_t *optlenp, int flags, struct cred *cr)
1233 {
1234     socklen_t maxlen = *optlenp;
1235     socklen_t len;
1236     socklen_t optlen;
1237     uint8_t buffer[4];
1238     void *optbuf = &buffer;
1239     int error = 0;
1240
1241     if (level == SOL_SOCKET) {
1242         switch (option_name) {
1243             /* Not supported options */
1244             case SO_SNDTIMEO:
1245             case SO_RCVTIMEO:
1246             case SO_EXCLBIND:
1247                 eprintsoline(so, ENOPROTOOPT);
1248                 return (ENOPROTOOPT);
1249             default:

```

```

1250         error = socket_getopt_common(so, level, option_name,
1251                                     optval, optlenp, flags);
1252         if (error >= 0)
1253             return (error);
1254         /* Pass the request to the protocol */
1255         break;
1256     }
1257 }
1258
1259 if (level == IPPROTO_SCTP) {
1260     /*
1261     * Should go through ioctl().
1262     */
1263     return (EINVAL);
1264 }
1265
1266 if (maxlen > sizeof (buffer)) {
1267     optbuf = kmem_alloc(maxlen, KM_SLEEP);
1268 }
1269 optlen = maxlen;
1270
1271 /*
1272 * If the resulting optlen is greater than the provided maxlen, then
1273 * we silently truncate.
1274 */
1275 error = sctp_get_opt((struct sctp_s *)so->so_proto_handle, level,
1276                    option_name, optbuf, &optlen);
1277
1278 if (error != 0) {
1279     eprintsoline(so, error);
1280     goto free;
1281 }
1282 len = optlen;
1283
1284 copyout:
1285     len = MIN(len, maxlen);
1286     bcopy(optbuf, optval, len);
1287     *optlenp = optlen;
1288 free:
1289     if (optbuf != &buffer) {
1290         kmem_free(optbuf, maxlen);
1291     }
1292
1293     return (error);
1294 }
1295
1296 /*
1297 * Set socket options
1298 */
1299 /*
1300 * ARGSUSED */
1301 static int
1302 sosctp_setsockopt(struct sonode *so, int level, int option_name,
1303                 const void *optval, t_uscalar_t optlen, struct cred *cr)
1304 {
1305     struct sctp_sonode *ss = SOTOSO(so);
1306     struct sctp_scassoc *ssa = NULL;
1307     sctp_assoc_t id;
1308     int error, rc;
1309     void *conn = NULL;
1310
1311     mutex_enter(&so->so_lock);
1312
1313     /*
1314     * For some SCTP level options, one can select the association this
1315     * applies to.

```

```

1316     */
1317     if (so->so_type == SOCK_STREAM) {
1318         conn = so->so_proto_handle;
1319     } else {
1320         /*
1321          * SOCK_SEQPACKET only
1322          */
1323         id = 0;
1324         if (level == IPPROTO_SCTP) {
1325             switch (option_name) {
1326                 case SCTP_RTOINFO:
1327                 case SCTP_ASSOCINFO:
1328                 case SCTP_SET_PEER_PRIMARY_ADDR:
1329                 case SCTP_PRIMARY_ADDR:
1330                 case SCTP_PEER_ADDR_PARAMS:
1331                     /*
1332                      * Association ID is the first element
1333                      * params struct
1334                      */
1335                     if (optlen < sizeof (sctp_assoc_t) {
1336                         error = EINVAL;
1337                         eprintsoline(so, error);
1338                         goto done;
1339                     }
1340                     id = *(sctp_assoc_t *)optval;
1341                     break;
1342                 case SCTP_DEFAULT_SEND_PARAM:
1343                     if (optlen != sizeof (struct sctp_sndrcvinfo) {
1344                         error = EINVAL;
1345                         eprintsoline(so, error);
1346                         goto done;
1347                     }
1348                     id = ((struct sctp_sndrcvinfo *)
1349                         optval)->sinfo_assoc_id;
1350                     break;
1351                 case SCTP_INITMSG:
1352                     /*
1353                      * Only applies to future associations
1354                      */
1355                     conn = so->so_proto_handle;
1356                     break;
1357                 default:
1358                     break;
1359             }
1360         } else if (level == SOL_SOCKET) {
1361             if (option_name == SO_LINGER) {
1362                 error = EOPNOTSUPP;
1363                 eprintsoline(so, error);
1364                 goto done;
1365             }
1366             /*
1367              * These 2 options are applied to all associations.
1368              * The other socket level options are only applied
1369              * to the socket (not associations).
1370              */
1371             if ((option_name != SO_RCVBUF) &&
1372                 (option_name != SO_SNDBUF)) {
1373                 conn = so->so_proto_handle;
1374             }
1375         } else {
1376             conn = NULL;
1377         }
1378     }
1379     /*
1380     * If association ID was specified, do op on that assoc.
1381     * Otherwise set the default setting of a socket.

```

```

1382     */
1383     if (id != 0) {
1384         if ((error = sosctp_assoc(ss, id, &ssa)) != 0) {
1385             eprintsoline(so, error);
1386             goto done;
1387         }
1388         conn = ssa->ssa_conn;
1389     }
1390 }
1391 dprint(2, ("sosctp_setsockopt %p (%d) - conn %p %d %d id:%d\n",
1392     (void *)ss, so->so_type, (void *)conn, level, option_name, id));
1393
1394 ASSERT(ssa == NULL || (ssa != NULL && conn != NULL));
1395 if (conn != NULL) {
1396     mutex_exit(&so->so_lock);
1397     error = sctp_set_opt((struct sctp_s *)conn, level, option_name,
1398         optval, optlen);
1399     mutex_enter(&so->so_lock);
1400     if (ssa != NULL)
1401         SSA_REFRELE(ss, ssa);
1402 } else {
1403     /*
1404     * 1-N socket, and we have to apply the operation to ALL
1405     * associations. Like with anything of this sort, the
1406     * problem is what to do if the operation fails.
1407     * Just try to apply the setting to everyone, but store
1408     * error number if someone returns such. And since we are
1409     * looping through all possible aids, some of them can be
1410     * invalid. We just ignore this kind (sosctp_assoc()) of
1411     * errors.
1412     */
1413     sctp_assoc_t aid;
1414
1415     mutex_exit(&so->so_lock);
1416     error = sctp_set_opt((struct sctp_s *)so->so_proto_handle,
1417         level, option_name, optval, optlen);
1418     mutex_enter(&so->so_lock);
1419     for (aid = 1; aid < ss->ss_maxassoc; aid++) {
1420         if (sosctp_assoc(ss, aid, &ssa) != 0)
1421             continue;
1422         mutex_exit(&so->so_lock);
1423         rc = sctp_set_opt((struct sctp_s *)ssa->ssa_conn, level,
1424             option_name, optval, optlen);
1425         mutex_enter(&so->so_lock);
1426         SSA_REFRELE(ss, ssa);
1427         if (error == 0) {
1428             error = rc;
1429         }
1430     }
1431 }
1432 done:
1433     mutex_exit(&so->so_lock);
1434     return (error);
1435 }
1436
1437 /*ARGSUSED*/
1438 static int
1439 sosctp_ioctl(struct sonode *so, int cmd, intptr_t arg, int mode,
1440     struct cred *cr, int32_t *rvalp)
1441 {
1442     struct sctp_sonode *ss;
1443     int32_t value;
1444     int error;
1445     int intval;
1446     pid_t pid;
1447     struct sctp_soassoc *ssa;

```

```

1448 void *conn;
1449 void *buf;
1450 STRUCT_DECL(sctptopt, opt);
1451 uint32_t optlen;
1452 int buflen;

1454 ss = SOTOSO(so);

1456 /* handle socket specific ioctls */
1457 switch (cmd) {
1458 case FIONBIO:
1459     if (so_copyin((void *)arg, &value, sizeof (int32_t),
1460                 (mode & (int)FKIOCTL)) {
1461         return (EFAULT);
1462     }
1463     mutex_enter(&so->so_lock);
1464     if (value) {
1465         so->so_state |= SS_NDELAY;
1466     } else {
1467         so->so_state &= ~SS_NDELAY;
1468     }
1469     mutex_exit(&so->so_lock);
1470     return (0);

1472 case FIOASYNC:
1473     if (so_copyin((void *)arg, &value, sizeof (int32_t),
1474                 (mode & (int)FKIOCTL)) {
1475         return (EFAULT);
1476     }
1477     mutex_enter(&so->so_lock);

1479     if (value) {
1480         /* Turn on SIGIO */
1481         so->so_state |= SS_ASYNC;
1482     } else {
1483         /* Turn off SIGIO */
1484         so->so_state &= ~SS_ASYNC;
1485     }
1486     mutex_exit(&so->so_lock);
1487     return (0);

1489 case SIOCSPGRP:
1490 case FIOSETOWN:
1491     if (so_copyin((void *)arg, &pid, sizeof (pid_t),
1492                 (mode & (int)FKIOCTL)) {
1493         return (EFAULT);
1494     }
1495     mutex_enter(&so->so_lock);

1497     error = (pid != so->so_pgrp) ? socket_chgpgrp(so, pid) : 0;
1498     mutex_exit(&so->so_lock);
1499     return (error);

1501 case SIOCGPGRP:
1502 case FIOGETOWN:
1503     if (so_copyout(&so->so_pgrp, (void *)arg,
1504                 sizeof (pid_t), (mode & (int)FKIOCTL))
1505         return (EFAULT);
1506     return (0);

1508 case FIONREAD:
1509     /* XXX: Cannot be used unless standard buffer is used */
1510     /*
1511     * Return number of bytes of data in all data messages
1512     * in queue in "arg".
1513     * For stream socket, amount of available data.

```

```

1514     * For sock_dgram, # of available bytes + addresses.
1515     */
1516     intval = (so->so_state & SS_ACCEPTCONN) ? 0 :
1517         MIN(so->so_rcv_queued, INT_MAX);
1518     if (so_copyout(&intval, (void *)arg, sizeof (intval),
1519                 (mode & (int)FKIOCTL))
1520         return (EFAULT);
1521     return (0);
1522 case SIOCATMARK:
1523     /*
1524     * No support for urgent data.
1525     */
1526     intval = 0;

1528     if (so_copyout(&intval, (void *)arg, sizeof (int),
1529                 (mode & (int)FKIOCTL))
1530         return (EFAULT);
1531     return (0);
1532 case _I_GETPEERCRED: {
1533     int error = 0;

1535     if ((mode & FKIOCTL) == 0)
1536         return (EINVAL);

1538     mutex_enter(&so->so_lock);
1539     if ((so->so_mode & SM_CONNREQUIRED) == 0) {
1540         error = ENOTSUP;
1541     } else if ((so->so_state & SS_ISCONNECTED) == 0) {
1542         error = ENOTCONN;
1543     } else if (so->so_peercred != NULL) {
1544         k_peercred_t *kp = (k_peercred_t *)arg;
1545         kp->pc_cr = so->so_peercred;
1546         kp->pc_cpuid = so->so_cpuid;
1547         crhold(so->so_peercred);
1548     } else {
1549         error = EINVAL;
1550     }
1551     mutex_exit(&so->so_lock);
1552     return (error);
1553 }
1554 case SIOCSTPGOPT:
1555     STRUCT_INIT(opt, mode);

1557     if (so_copyin((void *)arg, STRUCT_BUF(opt), STRUCT_SIZE(opt),
1558                 (mode & (int)FKIOCTL)) {
1559         return (EFAULT);
1560     }
1561     if ((optlen = STRUCT_FGET(opt, sopt_len)) > SO_MAXARGSIZE)
1562         return (EINVAL);

1564     /*
1565     * Find the correct sctp_t based on whether it is 1-N socket
1566     * or not.
1567     */
1568     intval = STRUCT_FGET(opt, sopt_aid);
1569     mutex_enter(&so->so_lock);
1570     if ((so->so_type == SOCK_SEQPACKET) && intval) {
1571         if ((error = sosctp_assoc(ss, intval, &ssa)) != 0) {
1572             mutex_exit(&so->so_lock);
1573             return (error);
1574         }
1575         conn = ssa->ssa_conn;
1576         ASSERT(conn != NULL);
1577     } else {
1578         conn = so->so_proto_handle;
1579         ssa = NULL;

```

```

1580     }
1581     mutex_exit(&so->so_lock);

1583     /* Copyin the option buffer and then call sctp_get_opt(). */
1584     buflen = optlen;
1585     /* Let's allocate a buffer enough to hold an int */
1586     if (buflen < sizeof (uint32_t))
1587         buflen = sizeof (uint32_t);
1588     buf = kmem_alloc(buflen, KM_SLEEP);
1589     if (so_copyin(STRUCT_FGETP(opt, sopt_val), buf, optlen,
1590         (mode & (int)FKIOCTL)) {
1591         if (ssa != NULL) {
1592             mutex_enter(&so->so_lock);
1593             SSA_REFRELE(ss, ssa);
1594             mutex_exit(&so->so_lock);
1595         }
1596         kmem_free(buf, buflen);
1597         return (EFAULT);
1598     }
1599     /* The option level has to be IPPROTO_SCTP */
1600     error = sctp_get_opt((struct sctp_s *)conn, IPPROTO_SCTP,
1601         STRUCT_FGET(opt, sopt_name), buf, &optlen);
1602     if (ssa != NULL) {
1603         mutex_enter(&so->so_lock);
1604         SSA_REFRELE(ss, ssa);
1605         mutex_exit(&so->so_lock);
1606     }
1607     optlen = MIN(buflen, optlen);
1608     /* No error, copyout the result with the correct buf len. */
1609     if (error == 0) {
1610         STRUCT_FSET(opt, sopt_len, optlen);
1611         if (so_copyout(STRUCT_BUF(opt), (void *)arg,
1612             STRUCT_SIZE(opt), (mode & (int)FKIOCTL)) {
1613             error = EFAULT;
1614         } else if (so_copyout(buf, STRUCT_FGETP(opt, sopt_val),
1615             optlen, (mode & (int)FKIOCTL)) {
1616             error = EFAULT;
1617         }
1618     }
1619     kmem_free(buf, buflen);
1620     return (error);

1622 case SIOCCTPSOFT:
1623     STRUCT_INIT(opt, mode);

1625     if (so_copyin((void *)arg, STRUCT_BUF(opt), STRUCT_SIZE(opt),
1626         (mode & (int)FKIOCTL)) {
1627         return (EFAULT);
1628     }
1629     if ((optlen = STRUCT_FGET(opt, sopt_len)) > SO_MAXARGSIZE)
1630         return (EINVAL);

1632     /*
1633     * Find the correct sctp_t based on whether it is 1-N socket
1634     * or not.
1635     */
1636     intval = STRUCT_FGET(opt, sopt_aid);
1637     mutex_enter(&so->so_lock);
1638     if (intval != 0) {
1639         if ((error = sosctp_assoc(ss, intval, &ssa)) != 0) {
1640             mutex_exit(&so->so_lock);
1641             return (error);
1642         }
1643         conn = ssa->ssa_conn;
1644         ASSERT(conn != NULL);
1645     } else {

```

```

1646         conn = so->so_proto_handle;
1647         ssa = NULL;
1648     }
1649     mutex_exit(&so->so_lock);

1651     /* Copyin the option buffer and then call sctp_set_opt(). */
1652     buf = kmem_alloc(optlen, KM_SLEEP);
1653     if (so_copyin(STRUCT_FGETP(opt, sopt_val), buf, optlen,
1654         (mode & (int)FKIOCTL)) {
1655         if (ssa != NULL) {
1656             mutex_enter(&so->so_lock);
1657             SSA_REFRELE(ss, ssa);
1658             mutex_exit(&so->so_lock);
1659         }
1660         kmem_free(buf, intval);
1661         return (EFAULT);
1662     }
1663     /* The option level has to be IPPROTO_SCTP */
1664     error = sctp_set_opt((struct sctp_s *)conn, IPPROTO_SCTP,
1665         STRUCT_FGET(opt, sopt_name), buf, optlen);
1666     if (ssa) {
1667         mutex_enter(&so->so_lock);
1668         SSA_REFRELE(ss, ssa);
1669         mutex_exit(&so->so_lock);
1670     }
1671     kmem_free(buf, optlen);
1672     return (error);

1674 case SIOCCTPPEELOFF: {
1675     struct sonode *nso;
1676     struct sctp_uc_swap us;
1677     int nfd;
1678     struct file *nfp;
1679     struct vnode *nvp = NULL;
1680     struct sockparams *sp;

1682     dprint(2, ("sctppeeloff %p\n", (void *)ss));

1684     if (so->so_type != SOCK_SEQPACKET) {
1685         return (EOPNOTSUPP);
1686     }
1687     if (so_copyin((void *)arg, &intval, sizeof (intval),
1688         (mode & (int)FKIOCTL)) {
1689         return (EFAULT);
1690     }
1691     if (intval == 0) {
1692         return (EINVAL);
1693     }

1695     /*
1696     * Find sockparams. This is different from parent's entry,
1697     * as the socket type is different.
1698     */
1699     error = solookup(so->so_family, SOCK_STREAM, so->so_protocol,
1700         &sp);
1701     if (error != 0)
1702         return (error);

1704     /*
1705     * Allocate the user fd.
1706     */
1707     if ((nfd = ufalloc(0)) == -1) {
1708         eprintsoline(so, EMFILE);
1709         SOCKPARAMS_DEC_REF(sp);
1710         return (EMFILE);
1711     }

```

```

1713     /*
1714     * Copy the fd out.
1715     */
1716     if (so_copyout(&nfd, (void *)arg, sizeof (nfd),
1717         (mode & (int)FKIOCTL)) {
1718         error = EFAULT;
1719         goto err;
1720     }
1721     mutex_enter(&so->so_lock);

1723     /*
1724     * Don't use sosctp_assoc() in order to peel off disconnected
1725     * associations.
1726     */
1727     ssa = ((uint32_t)intval >= ss->ss_maxassoc) ? NULL :
1728         ss->ss_assocs[intval].ssi_assoc;
1729     if (ssa == NULL) {
1730         mutex_exit(&so->so_lock);
1731         error = EINVAL;
1732         goto err;
1733     }
1734     SSA_REFHOLD(ssa);

1736     nso = socksctp_create(sp, so->so_family, SOCK_STREAM,
1737         so->so_protocol, so->so_version, SOCKET_NOSLEEP,
1738         &error, cr);
1739     if (nso == NULL) {
1740         SSA_REFRELE(ss, ssa);
1741         mutex_exit(&so->so_lock);
1742         goto err;
1743     }
1744     nvp = SOTOV(nso);
1745     so_lock_single(so);
1746     mutex_exit(&so->so_lock);

1748     /* cannot fail, only inheriting properties */
1749     (void) sosctp_init(nso, so, CRED(), 0);

1751     /*
1752     * We have a single ref on the new socket. This is normally
1753     * handled by socket_{create,newconn}, but since they are not
1754     * used we have to do it here.
1755     */
1756     nso->so_count = 1;

1758     us.sus_handle = nso;
1759     us.sus_upcalls = &sosctp_sock_upcalls;

1761     /*
1762     * Upcalls to new socket are blocked for the duration of
1763     * downcall.
1764     */
1765     mutex_enter(&nso->so_lock);

1767     error = sctp_set_opt((struct sctp_s *)ssa->ssa_conn,
1768         IPPROTO_SCTP, SCTP_UC_SWAP, &us, sizeof (us));
1769     if (error) {
1770         goto peelerr;
1771     }
1772     error = falloc(nvp, FWRITE|FREAD, &nfp, NULL);
1773     if (error) {
1774         goto peelerr;
1775     }
1777     /*

```

```

1778         * fill in the entries that falloc reserved
1779         */
1780         nfp->f_vnode = nvp;
1781         mutex_exit(&nfp->f_tlock);
1782         setf(nfd, nfp);

1784         /* Add pid to the list associated with that socket. */
1785         if (nfp->f_vnode != NULL) {
1786             (void) VOP_IOCTL(nfp->f_vnode, F_ASSOCI_PID,
1787                 (intptr_t)curproc->p_pidp->pid_id, FKIOCTL, kcred,
1788                 NULL, NULL);
1789         }

1791 #endif /* ! codereview */
1792     mutex_enter(&so->so_lock);

1794     sosctp_assoc_move(ss, SOTOSSO(nso), ssa);

1796     mutex_exit(&nso->so_lock);

1798     ssa->ssa_conn = NULL;
1799     sosctp_assoc_free(ss, ssa);

1801     so_unlock_single(so, SOLOCKED);
1802     mutex_exit(&so->so_lock);

1804     return (0);

1806 err:
1807     SOCKPARAMS_DEC_REF(sp);
1808     setf(nfd, NULL);
1809     eprintsoline(so, error);
1810     return (error);

1812 peelerr:
1813     mutex_exit(&nso->so_lock);
1814     mutex_enter(&so->so_lock);
1815     ASSERT(nso->so_count == 1);
1816     nso->so_count = 0;
1817     so_unlock_single(so, SOLOCKED);
1818     SSA_REFRELE(ss, ssa);
1819     mutex_exit(&so->so_lock);

1821     setf(nfd, NULL);
1822     ASSERT(nvp->v_count == 1);
1823     socket_destroy(nso);
1824     eprintsoline(so, error);
1825     return (error);
1826     }
1827     default:
1828         return (EINVAL);
1829     }
1830 }

1832 /*ARGSUSED*/
1833 static int
1834 sosctp_close(struct sonode *so, int flag, struct cred *cr)
1835 {
1836     struct sctp_sonode *ss;
1837     struct sctp_sa_id *ssi;
1838     struct sctp_soassoc *ssa;
1839     int32_t i;

1841     ss = SOTOSSO(so);
1843     /*

```

```

1844     * Initiate connection shutdown. Tell SCTP if there is any data
1845     * left unread.
1846     */
1847     sctp_recvd((struct sctp_s *)so->so_proto_handle,
1848              so->so_rcvbuf - so->so_rcv_queued);
1849     (void) sctp_disconnect((struct sctp_s *)so->so_proto_handle);

1851     /*
1852     * New associations can't come in, but old ones might get
1853     * closed in upcall. Protect against that by taking a reference
1854     * on the association.
1855     */
1856     mutex_enter(&so->so_lock);
1857     ssi = ss->ss_assoc;
1858     for (i = 0; i < ss->ss_maxassoc; i++, ssi++) {
1859         if ((ssa = ssi->ssi_assoc) != NULL) {
1860             SSA_REFHOLD(ssa);
1861             sosctp_assoc_isdisconnected(ssa, 0);
1862             mutex_exit(&so->so_lock);

1864             sctp_recvd(ssa->ssa_conn, so->so_rcvbuf -
1865                       ssa->ssa_rcv_queued);
1866             (void) sctp_disconnect(ssa->ssa_conn);

1868             mutex_enter(&so->so_lock);
1869             SSA_REFRELE(ss, ssa);
1870         }
1871     }
1872     mutex_exit(&so->so_lock);

1874     return (0);
1875 }

1877 /*
1878 * Closes incoming connections which were never accepted, frees
1879 * resources.
1880 */
1881 /* ARGSUSED */
1882 void
1883 sosctp_fini(struct sonode *so, struct cred *cr)
1884 {
1885     struct sctp_sonode *ss;
1886     struct sctp_sa_id *ssi;
1887     struct sctp_soassoc *ssa;
1888     int32_t i;

1890     ss = SOTOSSO(so);

1892     ASSERT(so->so_ops == &sosctp_sonodeops ||
1893           so->so_ops == &sosctp_seq_sonodeops);

1895     /* We are the sole owner of so now */
1896     mutex_enter(&so->so_lock);

1898     /* Free all pending connections */
1899     so_acceptq_flush(so, B_TRUE);

1901     ssi = ss->ss_assoc;
1902     for (i = 0; i < ss->ss_maxassoc; i++, ssi++) {
1903         if ((ssa = ssi->ssi_assoc) != NULL) {
1904             SSA_REFHOLD(ssa);
1905             mutex_exit(&so->so_lock);

1907             sctp_close((struct sctp_s *)ssa->ssa_conn);

1909             mutex_enter(&so->so_lock);

```

```

1910             ssa->ssa_conn = NULL;
1911             sosctp_assoc_free(ss, ssa);
1912         }
1913     }
1914     if (ss->ss_assoc != NULL) {
1915         ASSERT(ss->ss_assoccnt == 0);
1916         kmem_free(ss->ss_assoc,
1917                  ss->ss_maxassoc * sizeof (struct sctp_sa_id));
1918     }
1919     mutex_exit(&so->so_lock);

1921     if (so->so_proto_handle)
1922         sctp_close((struct sctp_s *)so->so_proto_handle);
1923     so->so_proto_handle = NULL;

1925     /*
1926     * Note until sctp_close() is called, SCTP can still send up
1927     * messages, such as event notifications. So we should flush
1928     * the receive buffer after calling sctp_close().
1929     */
1930     mutex_enter(&so->so_lock);
1931     so_rcv_flush(so);
1932     mutex_exit(&so->so_lock);

1934     sonode_fini(so);
1935 }

1937 /*
1938 * Upcalls from SCTP
1939 */

1941 /*
1942 * This is the upcall function for 1-N (SOCK_SEQPACKET) socket when a new
1943 * association is created. Note that the first argument (handle) is of type
1944 * sctp_sonode *, which is the one changed to a listener for new
1945 * associations. All the other upcalls for 1-N socket take sctp_soassoc *
1946 * as handle. The only exception is the su_properties upcall, which
1947 * can take both types as handle.
1948 */
1949 /* ARGSUSED */
1950 sock_upper_handle_t
1951 sctp_assoc_newconn(sock_upper_handle_t parenthandle,
1952                   sock_lower_handle_t connind, sock_downcalls_t *dc,
1953                   struct cred *peer_cred, pid_t peer_cpuid, sock_upcalls_t **ucp)
1954 {
1955     struct sctp_sonode *lss = (struct sctp_sonode *)parenthandle;
1956     struct sonode *lso = &lss->ss_so;
1957     struct sctp_soassoc *ssa;
1958     sctp_assoc_t id;

1960     ASSERT(lss->ss_type == SOSCTP_SOCKET);
1961     ASSERT(lso->so_state & SS_ACCEPTCONN);
1962     ASSERT(lso->so_proto_handle != NULL); /* closed conn */
1963     ASSERT(lso->so_type == SOCK_SEQPACKET);

1965     mutex_enter(&lso->so_lock);

1967     if ((id = sosctp_aid_get(lss)) == -1) {
1968         /*
1969         * Array not large enough; increase size.
1970         */
1971         if (sosctp_aid_grow(lss, lss->ss_maxassoc, KM_NOSLEEP) < 0) {
1972             mutex_exit(&lso->so_lock);
1973             return (NULL);
1974         }
1975         id = sosctp_aid_get(lss);

```

```

1976         ASSERT(id != -1);
1977     }
1979     /*
1980     * Create soassoc for this connection
1981     */
1982     ssa = sosctp_assoc_create(lss, KM_NOSLEEP);
1983     if (ssa == NULL) {
1984         mutex_exit(&lso->so_lock);
1985         return (NULL);
1986     }
1987     sosctp_aid_reserve(lss, id, 1);
1988     lss->ss_assoc[id].ssi_assoc = ssa;
1989     ++lss->ss_assoccnt;
1990     ssa->ssa_id = id;
1991     ssa->ssa_conn = (struct sctp_s *)connind;
1992     ssa->ssa_state = (SS_ISBOUND | SS_ISCONNECTED);
1993     ssa->ssa_wroff = lss->ss_wroff;
1994     ssa->ssa_wrsz = lss->ss_wrsz;
1996     mutex_exit(&lso->so_lock);
1998     *ucp = &sosctp_assoc_upcalls;
2000     return ((sock_upper_handle_t)ssa);
2001 }
2003 /* ARGSUSED */
2004 static void
2005 sctp_assoc_connected(sock_upper_handle_t handle, sock_connid_t id,
2006     struct cred *peer_cred, pid_t peer_cpuid)
2007 {
2008     struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2009     struct sonode *so = &ssa->ssa_sonode->ss_so;
2011     ASSERT(so->so_type == SOCK_SEQPACKET);
2012     ASSERT(ssa->ssa_conn);
2014     mutex_enter(&so->so_lock);
2015     sosctp_assoc_isdisconnected(ssa);
2016     mutex_exit(&so->so_lock);
2017 }
2019 /* ARGSUSED */
2020 static int
2021 sctp_assoc_disconnected(sock_upper_handle_t handle, sock_connid_t id, int error)
2022 {
2023     struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2024     struct sonode *so = &ssa->ssa_sonode->ss_so;
2025     int ret;
2027     ASSERT(so->so_type == SOCK_SEQPACKET);
2028     ASSERT(ssa->ssa_conn != NULL);
2030     mutex_enter(&so->so_lock);
2031     sosctp_assoc_isdisconnected(ssa, error);
2032     if (ssa->ssa_refcnt == 1) {
2033         ret = 1;
2034         ssa->ssa_conn = NULL;
2035     } else {
2036         ret = 0;
2037     }
2038     SSA_REFRELE(SOTOSO(so), ssa);
2040     cv_broadcast(&so->so_snd_cv);

```

```

2042     mutex_exit(&so->so_lock);
2044     return (ret);
2045 }
2047 /* ARGSUSED */
2048 static void
2049 sctp_assoc_disconnecting(sock_upper_handle_t handle, sock_opctl_action_t action,
2050     uintptr_t arg)
2051 {
2052     struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2053     struct sonode *so = &ssa->ssa_sonode->ss_so;
2055     ASSERT(so->so_type == SOCK_SEQPACKET);
2056     ASSERT(ssa->ssa_conn != NULL);
2057     ASSERT(action == SOCK_OPCTL_SHUT_SEND);
2059     mutex_enter(&so->so_lock);
2060     sosctp_assoc_isdisconnecting(ssa);
2061     mutex_exit(&so->so_lock);
2062 }
2064 /* ARGSUSED */
2065 static ssize_t
2066 sctp_assoc_recv(sock_upper_handle_t handle, mblk_t *mp, size_t len, int flags,
2067     int *errorp, boolean_t *forcepush)
2068 {
2069     struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2070     struct sctp_sonode *ss = ssa->ssa_sonode;
2071     struct sonode *so = &ss->ss_so;
2072     struct T_unitdata_ind *tind;
2073     mblk_t *mp2;
2074     union sctp_notification *sn;
2075     struct sctp_sndrcvinfo *sinfo;
2076     ssize_t space_available;
2078     ASSERT(ssa->ssa_type == SOSCTP_ASSOC);
2079     ASSERT(so->so_type == SOCK_SEQPACKET);
2080     ASSERT(ssa->ssa_conn != NULL); /* closed conn */
2081     ASSERT(mp != NULL);
2083     ASSERT(errorp != NULL);
2084     *errorp = 0;
2086     /*
2087     * Should be getting T_unitdata_req's only.
2088     * Must have address as part of packet.
2089     */
2090     tind = (struct T_unitdata_ind *)mp->b_rptr;
2091     ASSERT((DB_TYPE(mp) == M_PROTO) &&
2092         (tind->PRIM_type == T_UNITDATA_IND));
2093     ASSERT(tind->SRC_length);
2095     mutex_enter(&so->so_lock);
2097     /*
2098     * For notify messages, need to fill in association id.
2099     * For data messages, sndrcvinfo could be in ancillary data.
2100     */
2101     if (mp->b_flag & SCTP_NOTIFICATION) {
2102         mp2 = mp->b_cont;
2103         sn = (union sctp_notification *)mp2->b_rptr;
2104         switch (sn->sn_header.sn_type) {
2105             case Sctp_Assoc_Change:
2106                 sn->sn_assoc_change.sac_assoc_id = ssa->ssa_id;
2107                 break;

```

```

2108     case SCTP_PEER_ADDR_CHANGE:
2109         sn->sn_paddr_change.spc_assoc_id = ssa->ssa_id;
2110         break;
2111     case SCTP_REMOTE_ERROR:
2112         sn->sn_remote_error.sre_assoc_id = ssa->ssa_id;
2113         break;
2114     case SCTP_SEND_FAILED:
2115         sn->sn_send_failed.ssf_assoc_id = ssa->ssa_id;
2116         break;
2117     case SCTP_SHUTDOWN_EVENT:
2118         sn->sn_shutdown_event.sse_assoc_id = ssa->ssa_id;
2119         break;
2120     case SCTP_ADAPTATION_INDICATION:
2121         sn->sn_adaptation_event.sai_assoc_id = ssa->ssa_id;
2122         break;
2123     case SCTP_PARTIAL_DELIVERY_EVENT:
2124         sn->sn_pdapi_event.pdapi_assoc_id = ssa->ssa_id;
2125         break;
2126     default:
2127         ASSERT(0);
2128         break;
2129     }
2130 } else {
2131     if (tind->OPT_length > 0) {
2132         struct cmsghdr *cmsg;
2133         char *cend;
2134
2135         cmsg = (struct cmsghdr *)
2136             ((uchar_t *)mp->b_rptr + tind->OPT_offset);
2137         cend = (char *)cmsg + tind->OPT_length;
2138         for (;;) {
2139             if ((char *) (cmsg + 1) > cend ||
2140                 ((char *)cmsg + cmsg->cmsg_len) > cend) {
2141                 break;
2142             }
2143             if ((cmsg->cmsg_level == IPPROTO_SCTP) &&
2144                 (cmsg->cmsg_type == SCTP_SNDRCV)) {
2145                 struct sctp_sndrcvinfo *
2146                     sinfo = (struct sctp_sndrcvinfo *)
2147                     (cmsg + 1);
2148                 sinfo->sinfo_assoc_id = ssa->ssa_id;
2149                 break;
2150             }
2151             if (cmsg->cmsg_len > 0) {
2152                 cmsg = (struct cmsghdr *)
2153                     ((uchar_t *)cmsg + cmsg->cmsg_len);
2154             } else {
2155                 break;
2156             }
2157         }
2158     }
2159
2160     /*
2161     * SCTP has reserved space in the header for storing a pointer.
2162     * Put the pointer to association there, and queue the data.
2163     */
2164     SSA_REFHOLD(ssa);
2165     ASSERT((mp->b_rptr - DB_BASE(mp)) >= sizeof (ssa));
2166     *(struct sctp_soassoc **)DB_BASE(mp) = ssa;
2167
2168     ssa->ssa_rcv_queued += len;
2169     space_available = so->so_rcvbuf - ssa->ssa_rcv_queued;
2170     if (space_available <= 0)
2171         ssa->ssa_flowctrlld = B_TRUE;
2172
2173     so_enqueue_msg(so, mp, len);

```

```

2175     /* so_notify_data drops so_lock */
2176     so_notify_data(so, len);
2177
2178     return (space_available);
2179 }
2180
2181 static void
2182 sctp_assoc_xmitted(sock_upper_handle_t handle, boolean_t qfull)
2183 {
2184     struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2185     struct sctp_sonode *ss = ssa->ssa_sonode;
2186
2187     ASSERT(ssa->ssa_type == SOSCTP_ASSOC);
2188     ASSERT(ss->ss_so.so_type == SOCK_SEQPACKET);
2189     ASSERT(ssa->ssa_conn != NULL);
2190
2191     mutex_enter(&ss->ss_so.so_lock);
2192
2193     ssa->ssa_snd_qfull = qfull;
2194
2195     /*
2196     * Wake blocked writers.
2197     */
2198     cv_broadcast(&ss->ss_so.so_snd_cv);
2199
2200     mutex_exit(&ss->ss_so.so_lock);
2201 }
2202
2203 static void
2204 sctp_assoc_properties(sock_upper_handle_t handle,
2205                      struct sock_proto_props *soppp)
2206 {
2207     struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2208     struct sonode *so;
2209
2210     if (ssa->ssa_type == SOSCTP_ASSOC) {
2211         so = &ssa->ssa_sonode->ss_so;
2212
2213         mutex_enter(&so->so_lock);
2214
2215         /* Per assoc_id properties. */
2216         if (soppp->sopp_flags & SOCKOPT_WROFF)
2217             ssa->ssa_wroff = soppp->sopp_wroff;
2218         if (soppp->sopp_flags & SOCKOPT_MAXBLK)
2219             ssa->ssa_wrsz = soppp->sopp_maxblk;
2220     } else {
2221         so = &((struct sctp_sonode *)handle)->ss_so;
2222         mutex_enter(&so->so_lock);
2223
2224         if (soppp->sopp_flags & SOCKOPT_WROFF)
2225             so->so_proto_props.sopp_wroff = soppp->sopp_wroff;
2226         if (soppp->sopp_flags & SOCKOPT_MAXBLK)
2227             so->so_proto_props.sopp_maxblk = soppp->sopp_maxblk;
2228         if (soppp->sopp_flags & SOCKOPT_RCVHIWAT) {
2229             ssize_t lowat;
2230
2231             so->so_rcvbuf = soppp->sopp_rxhiwat;
2232             /*
2233             * The low water mark should be adjusted properly
2234             * if the high water mark is changed. It should
2235             * not be bigger than 1/4 of high water mark.
2236             */
2237             lowat = soppp->sopp_rxhiwat >> 2;
2238             if (so->so_rcvlowat > lowat) {
2239                 /* Sanity check... */

```

```
2240         if (lowat == 0)
2241             so->so_rcvlowat = sopp->sopp_rxhiwat;
2242         else
2243             so->so_rcvlowat = lowat;
2244     }
2245 }
2246 }
2247 mutex_exit(&so->so_lock);
2248 }

2250 static mblk_t *
2251 sctp_get_sock_pid_mblk(sock_upper_handle_t handle)
2252 {
2253     struct sctp_soassoc *ssa = (struct sctp_soassoc *)handle;
2254     struct sonode *so;

2256     if (ssa->ssa_type == SOSCTP_ASSOC)
2257         so = &ssa->ssa_sonode->ss_so;
2258     else
2259         so = &((struct sctp_sonode *)handle)->ss_so;

2261     return (so_get_sock_pid_mblk((sock_upper_handle_t)so));
2262 #endif /* ! codereview */
2263 }
```

```

*****
35181 Wed Jul 13 01:32:39 2016
new/usr/src/uts/common/inet/tcp/tcp_stats.c
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2011, Joyent Inc. All rights reserved.
25  */

27 #include <sys/types.h>
28 #include <sys/tihdr.h>
29 #include <sys/policy.h>
30 #include <sys/tsol/tnet.h>
31 #include <sys/kstat.h>

33 #include <sys/strsun.h>
34 #include <sys/stropts.h>
35 #include <sys/strsubr.h>
36 #include <sys/socket.h>
37 #include <sys/socketvar.h>
38 #include <sys/uio.h>

40 #endif /* ! codereview */
41 #include <inet/common.h>
42 #include <inet/ip.h>
43 #include <inet/tcp.h>
44 #include <inet/tcp_impl.h>
45 #include <inet/tcp_stats.h>
46 #include <inet/kstatcom.h>
47 #include <inet/snmpcom.h>

49 static int tcp_kstat_update(kstat_t *, int);
50 static int tcp_kstat2_update(kstat_t *, int);
51 static void tcp_sum_mib(tcp_stack_t *, mib2_tcp_t *);

53 static void tcp_add_mib(mib2_tcp_t *, mib2_tcp_t *);
54 static void tcp_add_stats(tcp_stat_counter_t *, tcp_stat_t *);
55 static void tcp_clr_stats(tcp_stat_t *);

57 tcp_g_stat_t tcp_g_statistics;
58 kstat_t *tcp_g_kstat;

60 /* Translate TCP state to MIB2 TCP state. */
61 static int

```

```

62 tcp_snmp_state(tcp_t *tcp)
63 {
64     if (tcp == NULL)
65         return (0);

67     switch (tcp->tcp_state) {
68     case TCPS_CLOSED:
69     case TCPS_IDLE: /* RFC1213 doesn't have analogue for IDLE & BOUND */
70     case TCPS_BOUND:
71         return (MIB2_TCP_closed);
72     case TCPS_LISTEN:
73         return (MIB2_TCP_listen);
74     case TCPS_SYN_SENT:
75         return (MIB2_TCP_synSent);
76     case TCPS_SYN_RCVD:
77         return (MIB2_TCP_synReceived);
78     case TCPS_ESTABLISHED:
79         return (MIB2_TCP_established);
80     case TCPS_CLOSE_WAIT:
81         return (MIB2_TCP_closeWait);
82     case TCPS_FIN_WAIT_1:
83         return (MIB2_TCP_finWait1);
84     case TCPS_CLOSING:
85         return (MIB2_TCP_closing);
86     case TCPS_LAST_ACK:
87         return (MIB2_TCP_lastAck);
88     case TCPS_FIN_WAIT_2:
89         return (MIB2_TCP_finWait2);
90     case TCPS_TIME_WAIT:
91         return (MIB2_TCP_timeWait);
92     default:
93         return (0);
94     }
95 }

97 /*
98  * Return SNMP stuff in buffer in mpdata.
99  */
100 mblk_t *
101 tcp_snmp_get(queue_t *q, mblk_t *mpctl, boolean_t legacy_req)
102 {
103     mblk_t *mpdata;
104     mblk_t *mp_conn_ctl = NULL;
105     mblk_t *mp_conn_tail;
106     mblk_t *mp_attr_ctl = NULL;
107     mblk_t *mp_attr_tail;
108     mblk_t *mp_pidnode_ctl = NULL;
109     mblk_t *mp_pidnode_tail;
110 #endif /* ! codereview */
111     mblk_t *mp6_conn_ctl = NULL;
112     mblk_t *mp6_conn_tail;
113     mblk_t *mp6_attr_ctl = NULL;
114     mblk_t *mp6_attr_tail;
115     mblk_t *mp6_pidnode_ctl = NULL;
116     mblk_t *mp6_pidnode_tail;
117 #endif /* ! codereview */
118     struct ophdr *optp;
119     mib2_tcpConnEntry_t tce;
120     mib2_tcp6ConnEntry_t tce6;
121     mib2_transportMLPEntry_t mlp;
122     connf_t *connfp;
123     int i;
124     boolean_t ispriv;
125     zoneid_t zoneid;
126     int v4_conn_idx;
127     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;
134
135     /*
136     * make a copy of the original message
137     */
138     mp2ctl = copymsg(mpctl);
139
140     if (mpctl == NULL ||
141         (mpdata = mpctl->b_cont) == NULL ||
142         (mp_conn_ctl = copymsg(mpctl)) == NULL ||
143         (mp_attr_ctl = copymsg(mpctl)) == NULL ||
144         (mp_pidnode_ctl = copymsg(mpctl)) == NULL ||
145 #endif /* ! codereview */
146         (mp6_conn_ctl = copymsg(mpctl)) == NULL ||
147         (mp6_attr_ctl = copymsg(mpctl)) == NULL ||
148         (mp6_pidnode_ctl = copymsg(mpctl)) == NULL) {
149         (mp6_attr_ctl = copymsg(mpctl)) == NULL) {
150         freemsg(mp_conn_ctl);
151         freemsg(mp_attr_ctl);
152         freemsg(mp_pidnode_ctl);
153 #endif /* ! codereview */
154         freemsg(mp6_conn_ctl);
155         freemsg(mp6_attr_ctl);
156         freemsg(mp6_pidnode_ctl);
157 #endif /* ! codereview */
158         freemsg(mpctl);
159         freemsg(mp2ctl);
160         return (NULL);
161     }
162
163     ipst = connp->conn_netstack->netstack_ip;
164     tcps = connp->conn_netstack->netstack_tcp;
165
166     if (legacy_req) {
167         tcp_mib_size = LEGACY_MIB_SIZE(&tcp_mib, mib2_tcp_t);
168         tce_size = LEGACY_MIB_SIZE(&tce, mib2_tcpConnEntry_t);
169         tce6_size = LEGACY_MIB_SIZE(&tce6, mib2_tcp6ConnEntry_t);
170     } else {
171         tcp_mib_size = sizeof (mib2_tcp_t);
172         tce_size = sizeof (mib2_tcpConnEntry_t);
173         tce6_size = sizeof (mib2_tcp6ConnEntry_t);
174     }
175
176     bzero(&tcp_mib, sizeof (tcp_mib));
177
178     /* build table of connections -- need count in fixed part */
179     SET_MIB(tcp_mib.tcpRtoAlgorithm, 4); /* vanj */
180     SET_MIB(tcp_mib.tcpRtoMin, tcps->tcps_rexmit_interval_min);
181     SET_MIB(tcp_mib.tcpRtoMax, tcps->tcps_rexmit_interval_max);
182     SET_MIB(tcp_mib.tcpMaxConn, -1);
183     SET_MIB(tcp_mib.tcpCurrEstab, 0);
184
185     ispriv =
186     secpolicy_ip_config((Q_TO_CONN(q))->conn_cred, B_TRUE) == 0;
187     zoneid = Q_TO_CONN(q)->conn_zoneid;
188
189     v4_conn_idx = v6_conn_idx = 0;
190     mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;
191     mp_pidnode_tail = mp6_pidnode_tail = NULL;
192 #endif /* ! codereview */

```

```

193     for (i = 0; i < CONN_G_HASH_SIZE; i++) {
194         ipst = tcps->tcps_netstack->netstack_ip;
195
196         connfp = &ipst->ips_ipcl_globalhash_fanout[i];
197
198         connp = NULL;
199
200         while ((connp =
201             ipcl_get_next_conn(connfp, connp, IPCL_TCPCONN)) != NULL) {
202             tcp_t *tcp;
203             boolean_t needattr;
204
205             if (connp->conn_zoneid != zoneid)
206                 continue; /* not in this zone */
207
208             tcp = connp->conn_tcp;
209             TCPS_UPDATE_MIB(tcps, tcpHCInSegs, tcp->tcp_ibsegs);
210             tcp->tcp_ibsegs = 0;
211             TCPS_UPDATE_MIB(tcps, tcpHCOutSegs, tcp->tcp_obsegs);
212             tcp->tcp_obsegs = 0;
213
214             tce6.tcp6ConnState = tce.tcpConnState =
215             tcp_snmp_state(tcp);
216             if (tce.tcpConnState == MIB2_TCP_established ||
217                 tce.tcpConnState == MIB2_TCP_closeWait)
218                 BUMP_MIB(&tcp_mib, tcpCurrEstab);
219
220             needattr = B_FALSE;
221             bzero(&mlp, sizeof (mlp));
222             if (connp->conn_mlp_type != mlptSingle) {
223                 if (connp->conn_mlp_type == mlptShared ||
224                     connp->conn_mlp_type == mlptBoth)
225                     mlp.tme_flags |= MIB2_TMEF_SHARED;
226                 if (connp->conn_mlp_type == mlptPrivate ||
227                     connp->conn_mlp_type == mlptBoth)
228                     mlp.tme_flags |= MIB2_TMEF_PRIVATE;
229                 needattr = B_TRUE;
230             }
231             if (connp->conn_anon_mlp) {
232                 mlp.tme_flags |= MIB2_TMEF_ANONMLP;
233                 needattr = B_TRUE;
234             }
235             switch (connp->conn_mac_mode) {
236             case CONN_MAC_DEFAULT:
237                 break;
238             case CONN_MAC_AWARE:
239                 mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
240                 needattr = B_TRUE;
241                 break;
242             case CONN_MAC_IMPLICIT:
243                 mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
244                 needattr = B_TRUE;
245                 break;
246             }
247             if (connp->conn_ixa->ixa_ts1 != NULL) {
248                 ts_label_t *ts1;
249
250                 ts1 = connp->conn_ixa->ixa_ts1;
251                 mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
252                 mlp.tme_doi = label2doi(ts1);
253                 mlp.tme_label = *label2bslabel(ts1);
254                 needattr = B_TRUE;
255             }
256
257             /* Create a message to report on IPv6 entries */
258             if (connp->conn_ipversion == IPV6_VERSION) {

```

```

259     tce6.tcp6ConnLocalAddress = connp->conn_laddr_v6;
260     tce6.tcp6ConnRemAddress = connp->conn_faddr_v6;
261     tce6.tcp6ConnLocalPort = ntohs(connp->conn_lport);
262     tce6.tcp6ConnRemPort = ntohs(connp->conn_fport);
263     if (connp->conn_ixa->ixa_flags & IXAF_SCOPEID_SET) {
264         tce6.tcp6ConnIfIndex =
265             connp->conn_ixa->ixa_scopeid;
266     } else {
267         tce6.tcp6ConnIfIndex = connp->conn_bound_if;
268     }
269     /* Don't want just anybody seeing these... */
270     if (ispriv) {
271         tce6.tcp6ConnEntryInfo.ce_snxt =
272             tcp->tcp_snxt;
273         tce6.tcp6ConnEntryInfo.ce_suna =
274             tcp->tcp_suna;
275         tce6.tcp6ConnEntryInfo.ce_rnxt =
276             tcp->tcp_rnxt;
277         tce6.tcp6ConnEntryInfo.ce_rack =
278             tcp->tcp_rack;
279     } else {
280         /*
281          * Netstat, unfortunately, uses this to
282          * get send/receive queue sizes. How to fix?
283          * Why not compute the difference only?
284          */
285         tce6.tcp6ConnEntryInfo.ce_snxt =
286             tcp->tcp_snxt - tcp->tcp_suna;
287         tce6.tcp6ConnEntryInfo.ce_suna = 0;
288         tce6.tcp6ConnEntryInfo.ce_rnxt =
289             tcp->tcp_rnxt - tcp->tcp_rack;
290         tce6.tcp6ConnEntryInfo.ce_rack = 0;
291     }
292
293     tce6.tcp6ConnEntryInfo.ce_swnd = tcp->tcp_swnd;
294     tce6.tcp6ConnEntryInfo.ce_rwnd = tcp->tcp_rwnd;
295     tce6.tcp6ConnEntryInfo.ce_rto = tcp->tcp_rto;
296     tce6.tcp6ConnEntryInfo.ce_mss = tcp->tcp_mss;
297     tce6.tcp6ConnEntryInfo.ce_state = tcp->tcp_state;
298
299     tce6.tcp6ConnCreationProcess =
300         (connp->conn_cpuid < 0) ? MIB2_UNKNOWN_PROCESS :
301         connp->conn_cpuid;
302     tce6.tcp6ConnCreationTime = connp->conn_open_time;
303
304     (void) snmp_append_data2(mp6_conn_ctl->b_cont,
305         &mp6_conn_tail, (char *)&tce6, tce6_size);
306
307     (void) snmp_append_data2(mp6_pidnode_ctl->b_cont,
308         &mp6_pidnode_tail, (char *)&tce6, tce6_size);
309
310     (void) snmp_append_mblk2(mp6_pidnode_ctl->b_cont,
311         &mp6_pidnode_tail, conn_get_pid_mblk(connp));
312
313 #endif /* ! codereview */
314     mlp.tme_connidx = v6_conn_idx++;
315     if (needattr)
316         (void) snmp_append_data2(mp6_attr_ctl->b_cont,
317             &mp6_attr_tail, (char *)&mlp, sizeof(mlp));
318     }
319     /*
320      * Create an IPv4 table entry for IPv4 entries and also
321      * for IPv6 entries which are bound to in6addr_any
322      * but don't have IPV6_V6ONLY set.
323      * (i.e. anything an IPv4 peer could connect to)
324      */

```

```

325     if (connp->conn_ipversion == IPV4_VERSION ||
326         (tcp->tcp_state <= TCPS_LISTEN &&
327         !connp->conn_ipv6_v6only &&
328         IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6))) {
329         if (connp->conn_ipversion == IPV6_VERSION) {
330             tce6.tcp6ConnRemAddress = INADDR_ANY;
331             tce6.tcp6ConnLocalAddress = INADDR_ANY;
332         } else {
333             tce6.tcp6ConnRemAddress =
334                 connp->conn_faddr_v4;
335             tce6.tcp6ConnLocalAddress =
336                 connp->conn_laddr_v4;
337         }
338         tce6.tcp6ConnLocalPort = ntohs(connp->conn_lport);
339         tce6.tcp6ConnRemPort = ntohs(connp->conn_fport);
340         /* Don't want just anybody seeing these... */
341         if (ispriv) {
342             tce6.tcp6ConnEntryInfo.ce_snxt =
343                 tcp->tcp_snxt;
344             tce6.tcp6ConnEntryInfo.ce_suna =
345                 tcp->tcp_suna;
346             tce6.tcp6ConnEntryInfo.ce_rnxt =
347                 tcp->tcp_rnxt;
348             tce6.tcp6ConnEntryInfo.ce_rack =
349                 tcp->tcp_rack;
350         } else {
351             /*
352              * Netstat, unfortunately, uses this to
353              * get send/receive queue sizes. How
354              * to fix?
355              * Why not compute the difference only?
356              */
357             tce6.tcp6ConnEntryInfo.ce_snxt =
358                 tcp->tcp_snxt - tcp->tcp_suna;
359             tce6.tcp6ConnEntryInfo.ce_suna = 0;
360             tce6.tcp6ConnEntryInfo.ce_rnxt =
361                 tcp->tcp_rnxt - tcp->tcp_rack;
362             tce6.tcp6ConnEntryInfo.ce_rack = 0;
363         }
364
365         tce6.tcp6ConnEntryInfo.ce_swnd = tcp->tcp_swnd;
366         tce6.tcp6ConnEntryInfo.ce_rwnd = tcp->tcp_rwnd;
367         tce6.tcp6ConnEntryInfo.ce_rto = tcp->tcp_rto;
368         tce6.tcp6ConnEntryInfo.ce_mss = tcp->tcp_mss;
369         tce6.tcp6ConnEntryInfo.ce_state =
370             tcp->tcp_state;
371
372         tce6.tcp6ConnCreationProcess =
373             (connp->conn_cpuid < 0) ?
374             MIB2_UNKNOWN_PROCESS :
375             connp->conn_cpuid;
376         tce6.tcp6ConnCreationTime = connp->conn_open_time;
377
378         (void) snmp_append_data2(mp_conn_ctl->b_cont,
379             &mp_conn_tail, (char *)&tce6, tce6_size);
380
381         (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
382             &mp_pidnode_tail, (char *)&tce6, tce6_size);
383
384         (void) snmp_append_mblk2(mp_pidnode_ctl->b_cont,
385             &mp_pidnode_tail, conn_get_pid_mblk(connp));
386
387 #endif /* ! codereview */
388     mlp.tme_connidx = v4_conn_idx++;
389     if (needattr)
390         (void) snmp_append_data2(

```

```

391         mp_attr_ctl->b_cont,
392         &mp_attr_tail, (char *)&mlp,
393         sizeof (mlp));
394     }
395 }
396 }
397
398 tcp_sum_mib(tcps, &tcp_mib);
399
400 /* Fixed length structure for IPv4 and IPv6 counters */
401 SET_MIB(tcp_mib.tcpConnTableSize, tce_size);
402 SET_MIB(tcp_mib.tcp6ConnTableSize, tce6_size);
403
404 /*
405  * Synchronize 32- and 64-bit counters. Note that tcpInSegs and
406  * tcpOutSegs are not updated anywhere in TCP. The new 64 bits
407  * counters are used. Hence the old counters' values in tcp_sc_mib
408  * are always 0.
409  */
410 SYNC32_MIB(&tcp_mib, tcpInSegs, tcpHCInSegs);
411 SYNC32_MIB(&tcp_mib, tcpOutSegs, tcpHCOutSegs);
412
413 optp = (struct ophdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
414 optp->level = MIB2_TCP;
415 optp->name = 0;
416 (void) snmp_append_data(mpdata, (char *)&tcp_mib, tcp_mib_size);
417 optp->len = msgdsize(mpdata);
418 qreply(q, mpctl);
419
420 /* table of connections... */
421 optp = (struct ophdr *)&mp_conn_ctl->b_rptr[
422     sizeof (struct T_optmgmt_ack)];
423 optp->level = MIB2_TCP;
424 optp->name = MIB2_TCP_CONN;
425 optp->len = msgdsize(mp_conn_ctl->b_cont);
426 qreply(q, mp_conn_ctl);
427
428 /* table of MLP attributes... */
429 optp = (struct ophdr *)&mp_attr_ctl->b_rptr[
430     sizeof (struct T_optmgmt_ack)];
431 optp->level = MIB2_TCP;
432 optp->name = EXPER_XPORT_MLP;
433 optp->len = msgdsize(mp_attr_ctl->b_cont);
434 if (optp->len == 0)
435     freemsg(mp_attr_ctl);
436 else
437     qreply(q, mp_attr_ctl);
438
439 /* table of IPv6 connections... */
440 optp = (struct ophdr *)&mp6_conn_ctl->b_rptr[
441     sizeof (struct T_optmgmt_ack)];
442 optp->level = MIB2_TCP6;
443 optp->name = MIB2_TCP6_CONN;
444 optp->len = msgdsize(mp6_conn_ctl->b_cont);
445 qreply(q, mp6_conn_ctl);
446
447 /* table of IPv6 MLP attributes... */
448 optp = (struct ophdr *)&mp6_attr_ctl->b_rptr[
449     sizeof (struct T_optmgmt_ack)];
450 optp->level = MIB2_TCP6;
451 optp->name = EXPER_XPORT_MLP;
452 optp->len = msgdsize(mp6_attr_ctl->b_cont);
453 if (optp->len == 0)
454     freemsg(mp6_attr_ctl);
455 else
456     qreply(q, mp6_attr_ctl);

```

```

458     /* table of EXPER_XPORT_PROC_INFO ipv4 */
459     optp = (struct ophdr *)&mp_pidnode_ctl->b_rptr[
460         sizeof (struct T_optmgmt_ack)];
461     optp->level = MIB2_TCP;
462     optp->name = EXPER_XPORT_PROC_INFO;
463     optp->len = msgdsize(mp_pidnode_ctl->b_cont);
464     if (optp->len == 0)
465         freemsg(mp_pidnode_ctl);
466     else
467         qreply(q, mp_pidnode_ctl);
468
469     /* table of EXPER_XPORT_PROC_INFO ipv6 */
470     optp = (struct ophdr *)&mp6_pidnode_ctl->b_rptr[
471         sizeof (struct T_optmgmt_ack)];
472     optp->level = MIB2_TCP6;
473     optp->name = EXPER_XPORT_PROC_INFO;
474     optp->len = msgdsize(mp6_pidnode_ctl->b_cont);
475     if (optp->len == 0)
476         freemsg(mp6_pidnode_ctl);
477     else
478         qreply(q, mp6_pidnode_ctl);
479
480 #endif /* ! codereview */
481     return (mp2ctl);
482 }
483
484 /* Return 0 if invalid set request, 1 otherwise, including non-tcp requests */
485 /* ARGSUSED */
486 int
487 tcp_snmp_set(queue_t *q, int level, int name, uchar_t *ptr, int len)
488 {
489     mib2_tcpConnEntry_t *tce = (mib2_tcpConnEntry_t *)ptr;
490
491     switch (level) {
492     case MIB2_TCP:
493         switch (name) {
494         case 13:
495             if (tce->tcpConnState != MIB2_TCP_deleteTCB)
496                 return (0);
497             /* TODO: delete entry defined by tce */
498             return (1);
499         default:
500             return (0);
501         }
502     default:
503         return (1);
504     }
505 }
506
507 /*
508  * TCP Kstats implementation
509  */
510 void *
511 tcp_kstat_init(netstackid_t stackid)
512 {
513     kstat_t *ksp;
514
515     tcp_named_kstat_t template = {
516         { "rtoAlgorithm", KSTAT_DATA_INT32, 0 },
517         { "rtoMin", KSTAT_DATA_INT32, 0 },
518         { "rtoMax", KSTAT_DATA_INT32, 0 },
519         { "maxConn", KSTAT_DATA_INT32, 0 },
520         { "activeOpens", KSTAT_DATA_UINT32, 0 },
521         { "passiveOpens", KSTAT_DATA_UINT32, 0 },
522         { "attemptFails", KSTAT_DATA_UINT32, 0 },

```

```

523     "estabResets",      KSTAT_DATA_UINT32, 0 },
524     "currEstab",       KSTAT_DATA_UINT32, 0 },
525     "inSegs",          KSTAT_DATA_UINT64, 0 },
526     "outSegs",         KSTAT_DATA_UINT64, 0 },
527     "retransSegs",     KSTAT_DATA_UINT32, 0 },
528     "connTableSize",   KSTAT_DATA_INT32,  0 },
529     "outRsts",         KSTAT_DATA_UINT32, 0 },
530     "outDataSegs",     KSTAT_DATA_UINT32, 0 },
531     "outDataBytes",    KSTAT_DATA_UINT32, 0 },
532     "retransBytes",    KSTAT_DATA_UINT32, 0 },
533     "outAck",          KSTAT_DATA_UINT32, 0 },
534     "outAckDelayed",   KSTAT_DATA_UINT32, 0 },
535     "outUrg",          KSTAT_DATA_UINT32, 0 },
536     "outWinUpdate",    KSTAT_DATA_UINT32, 0 },
537     "outWinProbe",     KSTAT_DATA_UINT32, 0 },
538     "outControl",      KSTAT_DATA_UINT32, 0 },
539     "outFastRetrans",  KSTAT_DATA_UINT32, 0 },
540     "inAckSegs",       KSTAT_DATA_UINT32, 0 },
541     "inAckBytes",      KSTAT_DATA_UINT32, 0 },
542     "inDupAck",        KSTAT_DATA_UINT32, 0 },
543     "inAckUnsent",     KSTAT_DATA_UINT32, 0 },
544     "inDataInorderSegs", KSTAT_DATA_UINT32, 0 },
545     "inDataInorderBytes", KSTAT_DATA_UINT32, 0 },
546     "inDataUnorderSegs", KSTAT_DATA_UINT32, 0 },
547     "inDataUnorderBytes", KSTAT_DATA_UINT32, 0 },
548     "inDataDupSegs",   KSTAT_DATA_UINT32, 0 },
549     "inDataDupBytes",  KSTAT_DATA_UINT32, 0 },
550     "inDataPartDupSegs", KSTAT_DATA_UINT32, 0 },
551     "inDataPartDupBytes", KSTAT_DATA_UINT32, 0 },
552     "inDataPastWinSegs", KSTAT_DATA_UINT32, 0 },
553     "inDataPastWinBytes", KSTAT_DATA_UINT32, 0 },
554     "inWinProbe",      KSTAT_DATA_UINT32, 0 },
555     "inWinUpdate",     KSTAT_DATA_UINT32, 0 },
556     "inClosed",        KSTAT_DATA_UINT32, 0 },
557     "rttUpdate",       KSTAT_DATA_UINT32, 0 },
558     "rttNoUpdate",     KSTAT_DATA_UINT32, 0 },
559     "timRetrans",      KSTAT_DATA_UINT32, 0 },
560     "timRetransDrop",  KSTAT_DATA_UINT32, 0 },
561     "timKeepalive",    KSTAT_DATA_UINT32, 0 },
562     "timKeepaliveProbe", KSTAT_DATA_UINT32, 0 },
563     "timKeepaliveDrop", KSTAT_DATA_UINT32, 0 },
564     "listenDrop",      KSTAT_DATA_UINT32, 0 },
565     "listenDropQ0",    KSTAT_DATA_UINT32, 0 },
566     "halfOpenDrop",    KSTAT_DATA_UINT32, 0 },
567     "outSackRetransSegs", KSTAT_DATA_UINT32, 0 },
568     "connTableSize6",  KSTAT_DATA_INT32, 0 }
569 };
571 ksp = kstat_create_netstack(TCP_MOD_NAME, stackid, TCP_MOD_NAME, "mib2",
572     KSTAT_TYPE_NAMED, NUM_OF_FIELDS(tcp_named_kstat_t), 0, stackid);
574 if (ksp == NULL)
575     return (NULL);
577 template.rtoAlgorithm.value.ui32 = 4;
578 template.maxConn.value.i32 = -1;
580 bcopy(&template, ksp->ks_data, sizeof (template));
581 ksp->ks_update = tcp_kstat_update;
582 ksp->ks_private = (void *) (uintptr_t) stackid;
584 /*
585  * If this is an exclusive netstack for a local zone, the global zone
586  * should still be able to read the kstat.
587  */
588 if (stackid != GLOBAL_NETSTACKID)

```

```

589     kstat_zone_add(ksp, GLOBAL_ZONEID);
591     kstat_install(ksp);
592     return (ksp);
593 }
595 void
596 tcp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
597 {
598     if (ksp != NULL) {
599         ASSERT(stackid == (netstackid_t) (uintptr_t) ksp->ks_private);
600         kstat_delete_netstack(ksp, stackid);
601     }
602 }
604 static int
605 tcp_kstat_update(kstat_t *kp, int rw)
606 {
607     tcp_named_kstat_t *tcpkp;
608     tcp_t *tcp;
609     connf_t *connfp;
610     conn_t *connp;
611     int i;
612     netstackid_t stackid = (netstackid_t) (uintptr_t) kp->ks_private;
613     netstack_t *ns;
614     tcp_stack_t *tcps;
615     ip_stack_t *ipst;
616     mib2_tcp_t tcp_mib;
618     if (rw == KSTAT_WRITE)
619         return (EACCES);
621     ns = netstack_find_by_stackid(stackid);
622     if (ns == NULL)
623         return (-1);
624     tcps = ns->netstack_tcp;
625     if (tcps == NULL) {
626         netstack_rele(ns);
627         return (-1);
628     }
630     tcpkp = (tcp_named_kstat_t *) kp->ks_data;
632     tcpkp->currEstab.value.ui32 = 0;
633     tcpkp->rtoMin.value.ui32 = tcps->tcps_rexmit_interval_min;
634     tcpkp->rtoMax.value.ui32 = tcps->tcps_rexmit_interval_max;
636     ipst = ns->netstack_ip;
638     for (i = 0; i < CONN_G_HASH_SIZE; i++) {
639         connfp = &ipst->ips_ipcl_globalhash_fanout[i];
640         connp = NULL;
641         while ((connp =
642             ipcl_get_next_conn(connfp, connp, IPCL_TCPCONN)) != NULL) {
643             tcp = connp->conn_tcp;
644             switch (tcp_snmp_state(tcp)) {
645                 case MIB2_TCP_established:
646                     case MIB2_TCP_closeWait:
647                     tcpkp->currEstab.value.ui32++;
648                     break;
649             }
650         }
651     }
652     bzero(&tcp_mib, sizeof (tcp_mib));
653     tcp_sum_mib(tcps, &tcp_mib);

```

```

655 /* Fixed length structure for IPv4 and IPv6 counters */
656 SET_MIB(tcp_mib.tcpConnTableSize, sizeof(mib2_tcpConnEntry_t));
657 SET_MIB(tcp_mib.tcp6ConnTableSize, sizeof(mib2_tcp6ConnEntry_t));

659 tcpkp->activeOpens.value.ui32 = tcp_mib.tcpActiveOpens;
660 tcpkp->passiveOpens.value.ui32 = tcp_mib.tcpPassiveOpens;
661 tcpkp->attemptFails.value.ui32 = tcp_mib.tcpAttemptFails;
662 tcpkp->estabResets.value.ui32 = tcp_mib.tcpEstabResets;
663 tcpkp->inSegs.value.ui64 = tcp_mib.tcpHCInSegs;
664 tcpkp->outSegs.value.ui64 = tcp_mib.tcpHCOutSegs;
665 tcpkp->retransSegs.value.ui32 = tcp_mib.tcpRetransSegs;
666 tcpkp->connTableSize.value.i32 = tcp_mib.tcpConnTableSize;
667 tcpkp->outRsts.value.ui32 = tcp_mib.tcpOutRsts;
668 tcpkp->outDataSegs.value.ui32 = tcp_mib.tcpOutDataSegs;
669 tcpkp->outDataBytes.value.ui32 = tcp_mib.tcpOutDataBytes;
670 tcpkp->retransBytes.value.ui32 = tcp_mib.tcpRetransBytes;
671 tcpkp->outAck.value.ui32 = tcp_mib.tcpOutAck;
672 tcpkp->outAckDelayed.value.ui32 = tcp_mib.tcpOutAckDelayed;
673 tcpkp->outUrg.value.ui32 = tcp_mib.tcpOutUrg;
674 tcpkp->outWinUpdate.value.ui32 = tcp_mib.tcpOutWinUpdate;
675 tcpkp->outWinProbe.value.ui32 = tcp_mib.tcpOutWinProbe;
676 tcpkp->outControl.value.ui32 = tcp_mib.tcpOutControl;
677 tcpkp->outFastRetrans.value.ui32 = tcp_mib.tcpOutFastRetrans;
678 tcpkp->inAckSegs.value.ui32 = tcp_mib.tcpInAckSegs;
679 tcpkp->inAckBytes.value.ui32 = tcp_mib.tcpInAckBytes;
680 tcpkp->inDupAck.value.ui32 = tcp_mib.tcpInDupAck;
681 tcpkp->inAckUnsent.value.ui32 = tcp_mib.tcpInAckUnsent;
682 tcpkp->inDataInorderSegs.value.ui32 = tcp_mib.tcpInDataInorderSegs;
683 tcpkp->inDataInorderBytes.value.ui32 = tcp_mib.tcpInDataInorderBytes;
684 tcpkp->inDataUnorderSegs.value.ui32 = tcp_mib.tcpInDataUnorderSegs;
685 tcpkp->inDataUnorderBytes.value.ui32 = tcp_mib.tcpInDataUnorderBytes;
686 tcpkp->inDataDupSegs.value.ui32 = tcp_mib.tcpInDataDupSegs;
687 tcpkp->inDataDupBytes.value.ui32 = tcp_mib.tcpInDataDupBytes;
688 tcpkp->inDataPartDupSegs.value.ui32 = tcp_mib.tcpInDataPartDupSegs;
689 tcpkp->inDataPartDupBytes.value.ui32 = tcp_mib.tcpInDataPartDupBytes;
690 tcpkp->inDataPastWinSegs.value.ui32 = tcp_mib.tcpInDataPastWinSegs;
691 tcpkp->inDataPastWinBytes.value.ui32 = tcp_mib.tcpInDataPastWinBytes;
692 tcpkp->inWinProbe.value.ui32 = tcp_mib.tcpInWinProbe;
693 tcpkp->inWinUpdate.value.ui32 = tcp_mib.tcpInWinUpdate;
694 tcpkp->inClosed.value.ui32 = tcp_mib.tcpInClosed;
695 tcpkp->rttNoUpdate.value.ui32 = tcp_mib.tcpRttNoUpdate;
696 tcpkp->rttUpdate.value.ui32 = tcp_mib.tcpRttUpdate;
697 tcpkp->timRetrans.value.ui32 = tcp_mib.tcpTimRetrans;
698 tcpkp->timRetransDrop.value.ui32 = tcp_mib.tcpTimRetransDrop;
699 tcpkp->timKeepalive.value.ui32 = tcp_mib.tcpTimKeepalive;
700 tcpkp->timKeepaliveProbe.value.ui32 = tcp_mib.tcpTimKeepaliveProbe;
701 tcpkp->timKeepaliveDrop.value.ui32 = tcp_mib.tcpTimKeepaliveDrop;
702 tcpkp->listenDrop.value.ui32 = tcp_mib.tcpListenDrop;
703 tcpkp->listenDropQ0.value.ui32 = tcp_mib.tcpListenDropQ0;
704 tcpkp->halfOpenDrop.value.ui32 = tcp_mib.tcpHalfOpenDrop;
705 tcpkp->outSackRetransSegs.value.ui32 = tcp_mib.tcpOutSackRetransSegs;
706 tcpkp->connTableSize6.value.i32 = tcp_mib.tcp6ConnTableSize;

708 netstack_rele(ns);
709 return(0);
710 }

712 /*
713 * kstats related to queues i.e. not per IP instance
714 */
715 void *
716 tcp_g_kstat_init(tcp_g_stat_t *tcp_g_statp)
717 {
718     kstat_t *ksp;

720     tcp_g_stat_t template = {

```

```

721     { "tcp_timermp_allocated",      KSTAT_DATA_UINT64 },
722     { "tcp_timermp_allocfail",     KSTAT_DATA_UINT64 },
723     { "tcp_timermp_allocdblfail",  KSTAT_DATA_UINT64 },
724     { "tcp_freelist_cleanup",      KSTAT_DATA_UINT64 },
725     };

727     ksp = kstat_create(TCP_MOD_NAME, 0, "tcpstat_g", "net",
728                     KSTAT_TYPE_NAMED, sizeof(template) / sizeof(kstat_named_t),
729                     KSTAT_FLAG_VIRTUAL);

731     if (ksp == NULL)
732         return(NULL);

734     bcopy(&template, tcp_g_statp, sizeof(template));
735     ksp->ks_data = (void *)tcp_g_statp;

737     kstat_install(ksp);
738     return(ksp);
739 }

741 void
742 tcp_g_kstat_fini(kstat_t *ksp)
743 {
744     if (ksp != NULL) {
745         kstat_delete(ksp);
746     }
747 }

749 void *
750 tcp_kstat2_init(netstackid_t stackid)
751 {
752     kstat_t *ksp;

754     tcp_stat_t template = {
755         { "tcp_time_wait_syn_success", KSTAT_DATA_UINT64, 0 },
756         { "tcp_clean_death_nondetached", KSTAT_DATA_UINT64, 0 },
757         { "tcp_eager_blowoff_q", KSTAT_DATA_UINT64, 0 },
758         { "tcp_eager_blowoff_q0", KSTAT_DATA_UINT64, 0 },
759         { "tcp_no_listener", KSTAT_DATA_UINT64, 0 },
760         { "tcp_listendrop", KSTAT_DATA_UINT64, 0 },
761         { "tcp_listendropq0", KSTAT_DATA_UINT64, 0 },
762         { "tcp_wsrvcalled", KSTAT_DATA_UINT64, 0 },
763         { "tcp_flwctl_on", KSTAT_DATA_UINT64, 0 },
764         { "tcp_timer_fire_early", KSTAT_DATA_UINT64, 0 },
765         { "tcp_timer_fire_miss", KSTAT_DATA_UINT64, 0 },
766         { "tcp_zcopy_on", KSTAT_DATA_UINT64, 0 },
767         { "tcp_zcopy_off", KSTAT_DATA_UINT64, 0 },
768         { "tcp_zcopy_backoff", KSTAT_DATA_UINT64, 0 },
769         { "tcp_fusion_flowctl", KSTAT_DATA_UINT64, 0 },
770         { "tcp_fusion_backenabld", KSTAT_DATA_UINT64, 0 },
771         { "tcp_fusion_urg", KSTAT_DATA_UINT64, 0 },
772         { "tcp_fusion_putnext", KSTAT_DATA_UINT64, 0 },
773         { "tcp_fusion_unfusable", KSTAT_DATA_UINT64, 0 },
774         { "tcp_fusion_aborted", KSTAT_DATA_UINT64, 0 },
775         { "tcp_fusion_unqualified", KSTAT_DATA_UINT64, 0 },
776         { "tcp_fusion_rrw_busy", KSTAT_DATA_UINT64, 0 },
777         { "tcp_fusion_rrw_msgcnt", KSTAT_DATA_UINT64, 0 },
778         { "tcp_fusion_rrw_plugged", KSTAT_DATA_UINT64, 0 },
779         { "tcp_in_ack_unsent_drop", KSTAT_DATA_UINT64, 0 },
780         { "tcp_sock_fallback", KSTAT_DATA_UINT64, 0 },
781         { "tcp_lso_enabled", KSTAT_DATA_UINT64, 0 },
782         { "tcp_lso_disabled", KSTAT_DATA_UINT64, 0 },
783         { "tcp_lso_times", KSTAT_DATA_UINT64, 0 },
784         { "tcp_lso_pkt_out", KSTAT_DATA_UINT64, 0 },
785         { "tcp_listen_cnt_drop", KSTAT_DATA_UINT64, 0 },
786         { "tcp_listen_mem_drop", KSTAT_DATA_UINT64, 0 }

```

```

877     { "tcp_zwin_mem_drop",      KSTAT_DATA_UINT64, 0 },
878     { "tcp_zwin_ack_syn",      KSTAT_DATA_UINT64, 0 },
879     { "tcp_rst_unsent",       KSTAT_DATA_UINT64, 0 },
890     { "tcp_reclaim_cnt",      KSTAT_DATA_UINT64, 0 },
891     { "tcp_reass_timeout",    KSTAT_DATA_UINT64, 0 },
892 #ifdef TCP_DEBUG_COUNTER
893     { "tcp_time_wait",        KSTAT_DATA_UINT64, 0 },
894     { "tcp_rput_time_wait",   KSTAT_DATA_UINT64, 0 },
895     { "tcp_detach_time_wait", KSTAT_DATA_UINT64, 0 },
896     { "tcp_timeout_calls",    KSTAT_DATA_UINT64, 0 },
897     { "tcp_timeout_cached_alloc", KSTAT_DATA_UINT64, 0 },
898     { "tcp_timeout_cancel_reqs", KSTAT_DATA_UINT64, 0 },
899     { "tcp_timeout_canceled", KSTAT_DATA_UINT64, 0 },
900     { "tcp_timermp_freed",    KSTAT_DATA_UINT64, 0 },
901     { "tcp_push_timer_cnt",   KSTAT_DATA_UINT64, 0 },
902     { "tcp_ack_timer_cnt",    KSTAT_DATA_UINT64, 0 },
903 #endif
904 };

906     ksp = kstat_create_netstack(TCP_MOD_NAME, stackid, "tcpstat", "net",
907                               KSTAT_TYPE_NAMED, sizeof(template) / sizeof(kstat_named_t), 0,
908                               stackid);

910     if (ksp == NULL)
911         return (NULL);

913     bcopy(&template, ksp->ks_data, sizeof(template));
914     ksp->ks_private = (void *) (uintptr_t) stackid;
915     ksp->ks_update = tcp_kstat2_update;

917     /*
918      * If this is an exclusive netstack for a local zone, the global zone
919      * should still be able to read the kstat.
920      */
921     if (stackid != GLOBAL_NETSTACKID)
922         kstat_zone_add(ksp, GLOBAL_ZONEID);

924     kstat_install(ksp);
925     return (ksp);
926 }

928 void
929 tcp_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
930 {
931     if (ksp != NULL) {
932         ASSERT(stackid == (netstackid_t) (uintptr_t) ksp->ks_private);
933         kstat_delete_netstack(ksp, stackid);
934     }
935 }

937 /*
938  * Sum up all per CPU tcp_stat_t kstat counters.
939  */
940 static int
941 tcp_kstat2_update(kstat_t *kp, int rw)
942 {
943     netstackid_t    stackid = (netstackid_t) (uintptr_t) kp->ks_private;
944     netstack_t      *ns;
945     tcp_stack_t     *tcps;
946     tcp_stat_t      *stats;
947     int             i;
948     int             cnt;

950     if (rw == KSTAT_WRITE)
951         return (EACCES);

```

```

853     ns = netstack_find_by_stackid(stackid);
854     if (ns == NULL)
855         return (-1);
856     tcps = ns->netstack_tcp;
857     if (tcps == NULL) {
858         netstack_rele(ns);
859         return (-1);
860     }

862     stats = (tcp_stat_t *) kp->ks_data;
863     tcp_clr_stats(stats);

865     /*
866      * tcps_sc_cnt may change in the middle of the loop. It is better
867      * to get its value first.
868      */
869     cnt = tcps->tcps_sc_cnt;
870     for (i = 0; i < cnt; i++)
871         tcp_add_stats(&tcps->tcps_sc[i]->tcp_sc_stats, stats);

873     netstack_rele(ns);
874     return (0);
875 }

877 /*
878  * To add stats from one mib2_tcp_t to another. Static fields are not added.
879  * The caller should set them up properly.
880  */
881 static void
882 tcp_add_mib(mib2_tcp_t *from, mib2_tcp_t *to)
883 {
884     to->tcpActiveOpens += from->tcpActiveOpens;
885     to->tcpPassiveOpens += from->tcpPassiveOpens;
886     to->tcpAttemptFails += from->tcpAttemptFails;
887     to->tcpEstabResets += from->tcpEstabResets;
888     to->tcpInSegs += from->tcpInSegs;
889     to->tcpOutSegs += from->tcpOutSegs;
890     to->tcpRetransSegs += from->tcpRetransSegs;
891     to->tcpOutRsts += from->tcpOutRsts;

893     to->tcpOutDataSegs += from->tcpOutDataSegs;
894     to->tcpOutDataBytes += from->tcpOutDataBytes;
895     to->tcpRetransBytes += from->tcpRetransBytes;
896     to->tcpOutAck += from->tcpOutAck;
897     to->tcpOutAckDelayed += from->tcpOutAckDelayed;
898     to->tcpOutUrg += from->tcpOutUrg;
899     to->tcpOutWinUpdate += from->tcpOutWinUpdate;
900     to->tcpOutWinProbe += from->tcpOutWinProbe;
901     to->tcpOutControl += from->tcpOutControl;
902     to->tcpOutFastRetrans += from->tcpOutFastRetrans;

904     to->tcpInAckBytes += from->tcpInAckBytes;
905     to->tcpInDupAck += from->tcpInDupAck;
906     to->tcpInAckUnsent += from->tcpInAckUnsent;
907     to->tcpInDataInorderSegs += from->tcpInDataInorderSegs;
908     to->tcpInDataInorderBytes += from->tcpInDataInorderBytes;
909     to->tcpInDataUnorderSegs += from->tcpInDataUnorderSegs;
910     to->tcpInDataUnorderBytes += from->tcpInDataUnorderBytes;
911     to->tcpInDataDupSegs += from->tcpInDataDupSegs;
912     to->tcpInDataDupBytes += from->tcpInDataDupBytes;
913     to->tcpInDataPartDupSegs += from->tcpInDataPartDupSegs;
914     to->tcpInDataPartDupBytes += from->tcpInDataPartDupBytes;
915     to->tcpInDataPastWinSegs += from->tcpInDataPastWinSegs;
916     to->tcpInDataPastWinBytes += from->tcpInDataPastWinBytes;
917     to->tcpInWinProbe += from->tcpInWinProbe;
918     to->tcpInWinUpdate += from->tcpInWinUpdate;

```

```

919     to->tcpInClosed += from->tcpInClosed;

921     to->tcpRttNoUpdate += from->tcpRttNoUpdate;
922     to->tcpRttUpdate += from->tcpRttUpdate;
923     to->tcpTimRetrans += from->tcpTimRetrans;
924     to->tcpTimRetransDrop += from->tcpTimRetransDrop;
925     to->tcpTimKeepalive += from->tcpTimKeepalive;
926     to->tcpTimKeepaliveProbe += from->tcpTimKeepaliveProbe;
927     to->tcpTimKeepaliveDrop += from->tcpTimKeepaliveDrop;
928     to->tcpListenDrop += from->tcpListenDrop;
929     to->tcpListenDropQ0 += from->tcpListenDropQ0;
930     to->tcpHalfOpenDrop += from->tcpHalfOpenDrop;
931     to->tcpOutSackRetransSegs += from->tcpOutSackRetransSegs;
932     to->tcpHCInSegs += from->tcpHCInSegs;
933     to->tcpHCOutSegs += from->tcpHCOutSegs;
934 }

936 /*
937  * To sum up all MIB2 stats for a tcp_stack_t from all per CPU stats. The
938  * caller should initialize the target mib2_tcp_t properly as this function
939  * just adds up all the per CPU stats.
940  */
941 static void
942 tcp_sum_mib(tcp_stack_t *tcps, mib2_tcp_t *tcp_mib)
943 {
944     int i;
945     int cnt;

947     /*
948      * tcps_sc_cnt may change in the middle of the loop. It is better
949      * to get its value first.
950      */
951     cnt = tcps->tcps_sc_cnt;
952     for (i = 0; i < cnt; i++)
953         tcp_add_mib(&tcps->tcps_sc[i]->tcp_sc_mib, tcp_mib);
954 }

956 /*
957  * To set all tcp_stat_t counters to 0.
958  */
959 static void
960 tcp_clr_stats(tcp_stat_t *stats)
961 {
962     stats->tcp_time_wait_syn_success.value.ui64 = 0;
963     stats->tcp_clean_death_nondetached.value.ui64 = 0;
964     stats->tcp_eager_blowoff_q.value.ui64 = 0;
965     stats->tcp_eager_blowoff_q0.value.ui64 = 0;
966     stats->tcp_no_listener.value.ui64 = 0;
967     stats->tcp_listendrop.value.ui64 = 0;
968     stats->tcp_listendropq0.value.ui64 = 0;
969     stats->tcp_wsrv_called.value.ui64 = 0;
970     stats->tcp_flwctl_on.value.ui64 = 0;
971     stats->tcp_timer_fire_early.value.ui64 = 0;
972     stats->tcp_timer_fire_miss.value.ui64 = 0;
973     stats->tcp_zcopy_on.value.ui64 = 0;
974     stats->tcp_zcopy_off.value.ui64 = 0;
975     stats->tcp_zcopy_backoff.value.ui64 = 0;
976     stats->tcp_fusion_flowctl.value.ui64 = 0;
977     stats->tcp_fusion_backenabled.value.ui64 = 0;
978     stats->tcp_fusion_urg.value.ui64 = 0;
979     stats->tcp_fusion_putnext.value.ui64 = 0;
980     stats->tcp_fusion_unfusable.value.ui64 = 0;
981     stats->tcp_fusion_aborted.value.ui64 = 0;
982     stats->tcp_fusion_unqualified.value.ui64 = 0;
983     stats->tcp_fusion_rrw_busy.value.ui64 = 0;
984     stats->tcp_fusion_rrw_msgcnt.value.ui64 = 0;

```

```

985     stats->tcp_fusion_rrw_plugged.value.ui64 = 0;
986     stats->tcp_in_ack_unsent_drop.value.ui64 = 0;
987     stats->tcp_sock_fallback.value.ui64 = 0;
988     stats->tcp_lso_enabled.value.ui64 = 0;
989     stats->tcp_lso_disabled.value.ui64 = 0;
990     stats->tcp_lso_times.value.ui64 = 0;
991     stats->tcp_lso_pkt_out.value.ui64 = 0;
992     stats->tcp_listen_cnt_drop.value.ui64 = 0;
993     stats->tcp_listen_mem_drop.value.ui64 = 0;
994     stats->tcp_zwin_mem_drop.value.ui64 = 0;
995     stats->tcp_zwin_ack_syn.value.ui64 = 0;
996     stats->tcp_rst_unsent.value.ui64 = 0;
997     stats->tcp_reclaim_cnt.value.ui64 = 0;
998     stats->tcp_reass_timeout.value.ui64 = 0;

1000 #ifdef TCP_DEBUG_COUNTER
1001     stats->tcp_time_wait.value.ui64 = 0;
1002     stats->tcp_rput_time_wait.value.ui64 = 0;
1003     stats->tcp_detach_time_wait.value.ui64 = 0;
1004     stats->tcp_timeout_calls.value.ui64 = 0;
1005     stats->tcp_timeout_cached_alloc.value.ui64 = 0;
1006     stats->tcp_timeout_cancel_reqs.value.ui64 = 0;
1007     stats->tcp_timeout_canceled.value.ui64 = 0;
1008     stats->tcp_timermp_freed.value.ui64 = 0;
1009     stats->tcp_push_timer_cnt.value.ui64 = 0;
1010     stats->tcp_ack_timer_cnt.value.ui64 = 0;
1011 #endif
1012 }

1014 /*
1015  * To add counters from the per CPU tcp_stat_counter_t to the stack
1016  * tcp_stat_t.
1017  */
1018 static void
1019 tcp_add_stats(tcp_stat_counter_t *from, tcp_stat_t *to)
1020 {
1021     to->tcp_time_wait_syn_success.value.ui64 +=
1022         from->tcp_time_wait_syn_success;
1023     to->tcp_clean_death_nondetached.value.ui64 +=
1024         from->tcp_clean_death_nondetached;
1025     to->tcp_eager_blowoff_q.value.ui64 +=
1026         from->tcp_eager_blowoff_q;
1027     to->tcp_eager_blowoff_q0.value.ui64 +=
1028         from->tcp_eager_blowoff_q0;
1029     to->tcp_no_listener.value.ui64 +=
1030         from->tcp_no_listener;
1031     to->tcp_listendrop.value.ui64 +=
1032         from->tcp_listendrop;
1033     to->tcp_listendropq0.value.ui64 +=
1034         from->tcp_listendropq0;
1035     to->tcp_wsrv_called.value.ui64 +=
1036         from->tcp_wsrv_called;
1037     to->tcp_flwctl_on.value.ui64 +=
1038         from->tcp_flwctl_on;
1039     to->tcp_timer_fire_early.value.ui64 +=
1040         from->tcp_timer_fire_early;
1041     to->tcp_timer_fire_miss.value.ui64 +=
1042         from->tcp_timer_fire_miss;
1043     to->tcp_zcopy_on.value.ui64 +=
1044         from->tcp_zcopy_on;
1045     to->tcp_zcopy_off.value.ui64 +=
1046         from->tcp_zcopy_off;
1047     to->tcp_zcopy_backoff.value.ui64 +=
1048         from->tcp_zcopy_backoff;
1049     to->tcp_fusion_flowctl.value.ui64 +=
1050         from->tcp_fusion_flowctl;

```

```

1051 to->tcp_fusion_backenabled.value.ui64 +=
1052 from->tcp_fusion_backenabled;
1053 to->tcp_fusion_urg.value.ui64 +=
1054 from->tcp_fusion_urg;
1055 to->tcp_fusion_putnext.value.ui64 +=
1056 from->tcp_fusion_putnext;
1057 to->tcp_fusion_unfusable.value.ui64 +=
1058 from->tcp_fusion_unfusable;
1059 to->tcp_fusion_aborted.value.ui64 +=
1060 from->tcp_fusion_aborted;
1061 to->tcp_fusion_unqualified.value.ui64 +=
1062 from->tcp_fusion_unqualified;
1063 to->tcp_fusion_rrw_busy.value.ui64 +=
1064 from->tcp_fusion_rrw_busy;
1065 to->tcp_fusion_rrw_msgcnt.value.ui64 +=
1066 from->tcp_fusion_rrw_msgcnt;
1067 to->tcp_fusion_rrw_plugged.value.ui64 +=
1068 from->tcp_fusion_rrw_plugged;
1069 to->tcp_in_ack_unsent_drop.value.ui64 +=
1070 from->tcp_in_ack_unsent_drop;
1071 to->tcp_sock_fallback.value.ui64 +=
1072 from->tcp_sock_fallback;
1073 to->tcp_lso_enabled.value.ui64 +=
1074 from->tcp_lso_enabled;
1075 to->tcp_lso_disabled.value.ui64 +=
1076 from->tcp_lso_disabled;
1077 to->tcp_lso_times.value.ui64 +=
1078 from->tcp_lso_times;
1079 to->tcp_lso_pkt_out.value.ui64 +=
1080 from->tcp_lso_pkt_out;
1081 to->tcp_listen_cnt_drop.value.ui64 +=
1082 from->tcp_listen_cnt_drop;
1083 to->tcp_listen_mem_drop.value.ui64 +=
1084 from->tcp_listen_mem_drop;
1085 to->tcp_zwin_mem_drop.value.ui64 +=
1086 from->tcp_zwin_mem_drop;
1087 to->tcp_zwin_ack_syn.value.ui64 +=
1088 from->tcp_zwin_ack_syn;
1089 to->tcp_rst_unsent.value.ui64 +=
1090 from->tcp_rst_unsent;
1091 to->tcp_reclaim_cnt.value.ui64 +=
1092 from->tcp_reclaim_cnt;
1093 to->tcp_reass_timeout.value.ui64 +=
1094 from->tcp_reass_timeout;

1096 #ifdef TCP_DEBUG_COUNTER
1097 to->tcp_time_wait.value.ui64 +=
1098 from->tcp_time_wait;
1099 to->tcp_rput_time_wait.value.ui64 +=
1100 from->tcp_rput_time_wait;
1101 to->tcp_detach_time_wait.value.ui64 +=
1102 from->tcp_detach_time_wait;
1103 to->tcp_timeout_calls.value.ui64 +=
1104 from->tcp_timeout_calls;
1105 to->tcp_timeout_cached_alloc.value.ui64 +=
1106 from->tcp_timeout_cached_alloc;
1107 to->tcp_timeout_cancel_reqs.value.ui64 +=
1108 from->tcp_timeout_cancel_reqs;
1109 to->tcp_timeout_canceled.value.ui64 +=
1110 from->tcp_timeout_canceled;
1111 to->tcp_timermp_freed.value.ui64 +=
1112 from->tcp_timermp_freed;
1113 to->tcp_push_timer_cnt.value.ui64 +=
1114 from->tcp_push_timer_cnt;
1115 to->tcp_ack_timer_cnt.value.ui64 +=
1116 from->tcp_ack_timer_cnt;

```

```

1117 #endif
1118 }

```

```

*****
17786 Wed Jul 13 01:32:39 2016
new/usr/src/uts/common/inet/udp/udp_stats.c
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

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

26 #include <sys/types.h>
27 #include <sys/tihdr.h>
28 #include <sys/policy.h>
29 #include <sys/tsol/tnet.h>

31 #include <inet/common.h>
32 #include <inet/kstatcom.h>
33 #include <inet/snmpcom.h>
34 #include <inet/mib2.h>
35 #include <inet/optcom.h>
36 #include <inet/snmpcom.h>
37 #include <inet/kstatcom.h>
38 #include <inet/udp_impl.h>

40 static int      udp_kstat_update(kstat_t *, int);
41 static int      udp_kstat2_update(kstat_t *, int);
42 static void      udp_sum_mib(udp_stack_t *, mib2_udp_t *);
43 static void      udp_clr_stats(udp_stat_t *);
44 static void      udp_add_stats(udp_stat_counter_t *, udp_stat_t *);
45 static void      udp_add_mib(mib2_udp_t *, mib2_udp_t *);
46 /*
47  * return SNMP stuff in buffer in mpdata. We don't hold any lock and report
48  * information that can be changing beneath us.
49  */
50 mblk_t *
51 udp_snmp_get(queue_t *q, mblk_t *mpctl, boolean_t legacy_req)
52 {
53     mblk_t      *mpdata;
54     mblk_t      *mp_conn_ctl;
55     mblk_t      *mp_attr_ctl;
56     mblk_t      *mp_pidnode_ctl;
57 #endif /* ! codereview */
58     mblk_t      *mp6_conn_ctl;
59     mblk_t      *mp6_attr_ctl;
60     mblk_t      *mp6_pidnode_ctl;
61 #endif /* ! codereview */

```

```

62     mblk_t      *mp_conn_tail;
63     mblk_t      *mp_attr_tail;
64     mblk_t      *mp_pidnode_tail;
65 #endif /* ! codereview */
66     mblk_t      *mp6_conn_tail;
67     mblk_t      *mp6_attr_tail;
68     mblk_t      *mp6_pidnode_tail;
69 #endif /* ! codereview */
70     struct ophdr *optp;
71     mib2_udpEntry_t ude;
72     mib2_udp6Entry_t ude6;
73     mib2_transportMLPEntry_t mlp;
74     int state;
75     zoneid_t zoneid;
76     int i;
77     connf_t *connfp;
78     conn_t *connp = Q_TO_CONN(q);
79     int v4_conn_idx;
80     int v6_conn_idx;
81     boolean_t needattr;
82     udp_t *udp;
83     ip_stack_t *ipst = connp->conn_netstack->netstack_ip;
84     udp_stack_t *us = connp->conn_netstack->netstack_udp;
85     mblk_t *mp2ctl;
86     mib2_udp_t udp_mib;
87     size_t udp_mib_size, ude_size, ude6_size;

89     /*
90      * make a copy of the original message
91      */
92     mp2ctl = copymsg(mpctl);

94     mp_conn_ctl = mp_attr_ctl = mp6_conn_ctl = NULL;
95     if (mpctl == NULL ||
96         (mpdata = mpctl->b_cont) == NULL ||
97         (mp_conn_ctl = copymsg(mpctl)) == NULL ||
98         (mp_attr_ctl = copymsg(mpctl)) == NULL ||
99         (mp_pidnode_ctl = copymsg(mpctl)) == NULL ||
100 #endif /* ! codereview */
101         (mp6_conn_ctl = copymsg(mpctl)) == NULL ||
102         (mp6_attr_ctl = copymsg(mpctl)) == NULL ||
103         (mp6_pidnode_ctl = copymsg(mpctl)) == NULL) {
104         (mp6_attr_ctl = copymsg(mpctl)) == NULL) {
104         freemsg(mp_conn_ctl);
105         freemsg(mp_attr_ctl);
106         freemsg(mp_pidnode_ctl);
107 #endif /* ! codereview */
108         freemsg(mp6_conn_ctl);
109         freemsg(mp6_attr_ctl);
110         freemsg(mp6_pidnode_ctl);
111 #endif /* ! codereview */
112         freemsg(mpctl);
113         freemsg(mp2ctl);
114         return (0);
115     }

117     zoneid = connp->conn_zoneid;

119     if (legacy_req) {
120         udp_mib_size = LEGACY_MIB_SIZE(&udp_mib, mib2_udp_t);
121         ude_size = LEGACY_MIB_SIZE(&ude, mib2_udpEntry_t);
122         ude6_size = LEGACY_MIB_SIZE(&ude6, mib2_udp6Entry_t);
123     } else {
124         udp_mib_size = sizeof (mib2_udp_t);
125         ude_size = sizeof (mib2_udpEntry_t);

```

```

126         ude6_size = sizeof (mib2_udp6Entry_t);
127     }

129     bzero(&udp_mib, sizeof (udp_mib));
130     /* fixed length structure for IPv4 and IPv6 counters */
131     SET_MIB(udp_mib.udpEntrySize, ude_size);
132     SET_MIB(udp_mib.udp6EntrySize, ude6_size);

134     udp_sum_mib(us, &udp_mib);

136     /*
137     * Synchronize 32- and 64-bit counters. Note that udpInDatagrams and
138     * udpOutDatagrams are not updated anywhere in UDP. The new 64 bits
139     * counters are used. Hence the old counters' values in us_sc_mib
140     * are always 0.
141     */
142     SYNC32_MIB(&udp_mib, udpInDatagrams, udpHCInDatagrams);
143     SYNC32_MIB(&udp_mib, udpOutDatagrams, udpHCOutDatagrams);

145     optp = (struct opthdr *)&mpctl->b_rptr[sizeof (struct T_optmgmt_ack)];
146     optp->level = MIB2_UDP;
147     optp->name = 0;
148     (void) snmp_append_data(mpdata, (char *)&udp_mib, udp_mib_size);
149     optp->len = msgdsz(mpdata);
150     qreply(q, mpctl);

152     mp_conn_tail = mp_attr_tail = mp6_conn_tail = mp6_attr_tail = NULL;
153     mp_pidnode_tail = mp6_pidnode_tail = NULL;
154 #endif /* ! codereview */
155     v4_conn_idx = v6_conn_idx = 0;

157     for (i = 0; i < CONN_G_HASH_SIZE; i++) {
158         connfp = &ipst->ips_ipcl_globalhash_fanout[i];
159         connp = NULL;

161         while ((connp = ipcl_get_next_conn(connfp, connp,
162             IPCL_UDPCONN))) {
163             udp = connp->conn_udp;
164             if (zoneid != connp->conn_zoneid)
165                 continue;

167             /*
168             * Note that the port numbers are sent in
169             * host byte order
170             */

172             if (udp->udp_state == TS_UNBND)
173                 state = MIB2_UDP_unbound;
174             else if (udp->udp_state == TS_IDLE)
175                 state = MIB2_UDP_idle;
176             else if (udp->udp_state == TS_DATA_XFER)
177                 state = MIB2_UDP_connected;
178             else
179                 state = MIB2_UDP_unknown;

181             needattr = B_FALSE;
182             bzero(&mlp, sizeof (mlp));
183             if (connp->conn_mlp_type != mlptSingle) {
184                 if (connp->conn_mlp_type == mlptShared ||
185                     connp->conn_mlp_type == mlptBoth)
186                     mlp.tme_flags |= MIB2_TMEF_SHARED;
187                 if (connp->conn_mlp_type == mlptPrivate ||
188                     connp->conn_mlp_type == mlptBoth)
189                     mlp.tme_flags |= MIB2_TMEF_PRIVATE;
190             }
191             needattr = B_TRUE;

```

```

192         if (connp->conn_anon_mlp) {
193             mlp.tme_flags |= MIB2_TMEF_ANONMLP;
194             needattr = B_TRUE;
195         }
196         switch (connp->conn_mac_mode) {
197             case CONN_MAC_DEFAULT:
198                 break;
199             case CONN_MAC_AWARE:
200                 mlp.tme_flags |= MIB2_TMEF_MACEXEMPT;
201                 needattr = B_TRUE;
202                 break;
203             case CONN_MAC_IMPLICIT:
204                 mlp.tme_flags |= MIB2_TMEF_MACIMPLICIT;
205                 needattr = B_TRUE;
206                 break;
207         }
208         mutex_enter(&connp->conn_lock);
209         if (udp->udp_state == TS_DATA_XFER &&
210             connp->conn_ixa->ixa_tsl != NULL) {
211             ts_label_t *tsl;

213             tsl = connp->conn_ixa->ixa_tsl;
214             mlp.tme_flags |= MIB2_TMEF_IS_LABELED;
215             mlp.tme_doi = label2doi(tsl);
216             mlp.tme_label = *label2bslabel(tsl);
217             needattr = B_TRUE;
218         }
219         mutex_exit(&connp->conn_lock);

221         /*
222         * Create an IPv4 table entry for IPv4 entries and also
223         * any IPv6 entries which are bound to in6addr_any
224         * (i.e. anything a IPv4 peer could connect/send to).
225         */
226         if (connp->conn_ipversion == IPV4_VERSION ||
227             (udp->udp_state <= TS_IDLE &&
228                 IN6_IS_ADDR_UNSPECIFIED(&connp->conn_laddr_v6))) {
229             ude.udpEntryInfo.ue_state = state;
230             /*
231             * If in6addr_any this will set it to
232             * INADDR_ANY
233             */
234             ude.udpLocalAddress = connp->conn_laddr_v4;
235             ude.udpLocalPort = ntohs(connp->conn_lport);
236             if (udp->udp_state == TS_DATA_XFER) {
237                 /*
238                 * Can potentially get here for
239                 * v6 socket if another process
240                 * (say, ping) has just done a
241                 * sendto(), changing the state
242                 * from the TS_IDLE above to
243                 * TS_DATA_XFER by the time we hit
244                 * this part of the code.
245                 */
246                 ude.udpEntryInfo.ue_RemoteAddress =
247                     connp->conn_faddr_v4;
248                 ude.udpEntryInfo.ue_RemotePort =
249                     ntohs(connp->conn_fport);
250             } else {
251                 ude.udpEntryInfo.ue_RemoteAddress = 0;
252                 ude.udpEntryInfo.ue_RemotePort = 0;
253             }

255             /*
256             * We make the assumption that all udp_t
257             * structs will be created within an address

```

```

258     * region no larger than 32-bits.
259     */
260     ude.udpInstance = (uint32_t)(uintptr_t)udp;
261     ude.udpCreationProcess =
262         (connp->conn_cpuid < 0) ?
263         MIB2_UNKNOWN_PROCESS :
264         connp->conn_cpuid;
265     ude.udpCreationTime = connp->conn_open_time;

267     (void) snmp_append_data2(mp_conn_ctl->b_cont,
268                             &mp_conn_tail, (char *)&ude, ude_size);

270     (void) snmp_append_data2(mp_pidnode_ctl->b_cont,
271                             &mp_pidnode_tail, (char *)&ude, ude_size);

273     (void) snmp_append_mblk2(mp_pidnode_ctl->b_cont,
274                             &mp_pidnode_tail, conn_get_pid_mblk(connp));

276 #endif /* ! codereview */
277     mlp.tme_connidx = v4_conn_idx++;
278     if (needattr)
279         (void) snmp_append_data2(
280             mp_attr_ctl->b_cont, &mp_attr_tail,
281             (char *)&mlp, sizeof (mlp));
282     }
283     if (connp->conn_ipversion == IPV6_VERSION) {
284         ude6.udp6EntryInfo.ue_state = state;
285         ude6.udp6LocalAddress = connp->conn_laddr_v6;
286         ude6.udp6LocalPort = ntohs(connp->conn_lport);
287         mutex_enter(&connp->conn_lock);
288         if (connp->conn_ixa->ixa_flags &
289             IXAF_SCOPEID_SET) {
290             ude6.udp6IfIndex =
291                 connp->conn_ixa->ixa_scopeid;
292         } else {
293             ude6.udp6IfIndex = connp->conn_bound_if;
294         }
295         mutex_exit(&connp->conn_lock);
296         if (udp->udp_state == TS_DATA_XFER) {
297             ude6.udp6EntryInfo.ue_RemoteAddress =
298                 connp->conn_faddr_v6;
299             ude6.udp6EntryInfo.ue_RemotePort =
300                 ntohs(connp->conn_fport);
301         } else {
302             ude6.udp6EntryInfo.ue_RemoteAddress =
303                 sin6_null.sin6_addr;
304             ude6.udp6EntryInfo.ue_RemotePort = 0;
305         }
306     }
307     /*
308     * We make the assumption that all udp_t
309     * structs will be created within an address
310     * region no larger than 32-bits.
311     */
312     ude6.udp6Instance = (uint32_t)(uintptr_t)udp;
313     ude6.udp6CreationProcess =
314         (connp->conn_cpuid < 0) ?
315         MIB2_UNKNOWN_PROCESS :
316         connp->conn_cpuid;
317     ude6.udp6CreationTime = connp->conn_open_time;

318     (void) snmp_append_data2(mp6_conn_ctl->b_cont,
319                             &mp6_conn_tail, (char *)&ude6, ude6_size);

321     (void) snmp_append_data2(
322         mp6_pidnode_ctl->b_cont, &mp6_pidnode_tail,
323         (char *)&ude6, ude6_size);

```

```

325     (void) snmp_append_mblk2(
326         mp6_pidnode_ctl->b_cont, &mp6_pidnode_tail,
327         conn_get_pid_mblk(connp));

329 #endif /* ! codereview */
330     mlp.tme_connidx = v6_conn_idx++;
331     if (needattr)
332         (void) snmp_append_data2(
333             mp6_attr_ctl->b_cont,
334             &mp6_attr_tail, (char *)&mlp,
335             sizeof (mlp));
336     }
337     }
338     }

340     /* IPv4 UDP endpoints */
341     optp = (struct ophdr *)&mp_conn_ctl->b_rptr[
342         sizeof (struct T_optmgmt_ack)];
343     optp->level = MIB2_UDP;
344     optp->name = MIB2_UDP_ENTRY;
345     optp->len = msgdsize(mp_conn_ctl->b_cont);
346     qreply(q, mp_conn_ctl);

348     /* table of MLP attributes... */
349     optp = (struct ophdr *)&mp_attr_ctl->b_rptr[
350         sizeof (struct T_optmgmt_ack)];
351     optp->level = MIB2_UDP;
352     optp->name = EXPER_XPORT_MLP;
353     optp->len = msgdsize(mp_attr_ctl->b_cont);
354     if (optp->len == 0)
355         freemsg(mp_attr_ctl);
356     else
357         qreply(q, mp_attr_ctl);

359     /* table of EXPER_XPORT_PROC_INFO ipv4 */
360     optp = (struct ophdr *)&mp_pidnode_ctl->b_rptr[
361         sizeof (struct T_optmgmt_ack)];
362     optp->level = MIB2_UDP;
363     optp->name = EXPER_XPORT_PROC_INFO;
364     optp->len = msgdsize(mp_pidnode_ctl->b_cont);
365     if (optp->len == 0)
366         freemsg(mp_pidnode_ctl);
367     else
368         qreply(q, mp_pidnode_ctl);

370 #endif /* ! codereview */
371     /* IPv6 UDP endpoints */
372     optp = (struct ophdr *)&mp6_conn_ctl->b_rptr[
373         sizeof (struct T_optmgmt_ack)];
374     optp->level = MIB2_UDP6;
375     optp->name = MIB2_UDP6_ENTRY;
376     optp->len = msgdsize(mp6_conn_ctl->b_cont);
377     qreply(q, mp6_conn_ctl);

379     /* table of MLP attributes... */
380     optp = (struct ophdr *)&mp6_attr_ctl->b_rptr[
381         sizeof (struct T_optmgmt_ack)];
382     optp->level = MIB2_UDP6;
383     optp->name = EXPER_XPORT_MLP;
384     optp->len = msgdsize(mp6_attr_ctl->b_cont);
385     if (optp->len == 0)
386         freemsg(mp6_attr_ctl);
387     else
388         qreply(q, mp6_attr_ctl);

```

```

390 /* table of EXPER_XPORT_PROC_INFO ipv6 */
391 optp = (struct ophdr *)&mp6_pidnode_ctl->b_rprtr[
392     sizeof(struct T_optmgmt_ack)];
393 optp->level = MIB2_UDP6;
394 optp->name = EXPER_XPORT_PROC_INFO;
395 optp->len = msgdsize(mp6_pidnode_ctl->b_cont);
396 if (optp->len == 0)
397     freemsg(mp6_pidnode_ctl);
398 else
399     qreply(q, mp6_pidnode_ctl);
400 #endif /* ! codereview */

402     return (mp2ctl);
403 }

405 /*
406 * Return 0 if invalid set request, 1 otherwise, including non-udp requests.
407 * NOTE: Per MIB-II, UDP has no writable data.
408 * TODO: If this ever actually tries to set anything, it needs to be
409 * to do the appropriate locking.
410 */
411 /* ARGSUSED */
412 int
413 udp_snmp_set(queue_t *q, t_scalar_t level, t_scalar_t name,
414     uchar_t *ptr, int len)
415 {
416     switch (level) {
417     case MIB2_UDP:
418         return (0);
419     default:
420         return (1);
421     }
422 }

424 void
425 udp_kstat_fini(netstackid_t stackid, kstat_t *ksp)
426 {
427     if (ksp != NULL) {
428         ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
429         kstat_delete_netstack(ksp, stackid);
430     }
431 }

433 /*
434 * To add stats from one mib2_udp_t to another. Static fields are not added.
435 * The caller should set them up properly.
436 */
437 static void
438 udp_add_mib(mib2_udp_t *from, mib2_udp_t *to)
439 {
440     to->udpHCInDatagrams += from->udpHCInDatagrams;
441     to->udpInErrors += from->udpInErrors;
442     to->udpHCOutDatagrams += from->udpHCOutDatagrams;
443     to->udpOutErrors += from->udpOutErrors;
444 }

447 void *
448 udp_kstat2_init(netstackid_t stackid)
449 {
450     kstat_t *ksp;

452     udp_stat_t template = {
453     { "udp_sock_fallback",          KSTAT_DATA_UINT64 },
454     { "udp_out_opt",                KSTAT_DATA_UINT64 },
455     { "udp_out_err_notconn",       KSTAT_DATA_UINT64 },

```

```

456     { "udp_out_err_output",        KSTAT_DATA_UINT64 },
457     { "udp_out_err_tudr",          KSTAT_DATA_UINT64 },
458 #ifdef DEBUG
459     { "udp_data_conn",             KSTAT_DATA_UINT64 },
460     { "udp_data_notconn",          KSTAT_DATA_UINT64 },
461     { "udp_out_lastdst",           KSTAT_DATA_UINT64 },
462     { "udp_out_diffdst",           KSTAT_DATA_UINT64 },
463     { "udp_out_ipv6",              KSTAT_DATA_UINT64 },
464     { "udp_out_mapped",            KSTAT_DATA_UINT64 },
465     { "udp_out_ipv4",              KSTAT_DATA_UINT64 },
466 #endif
467     };

469     ksp = kstat_create_netstack(UDP_MOD_NAME, 0, "udpstat", "net",
470         KSTAT_TYPE_NAMED, sizeof(template) / sizeof(kstat_named_t),
471         0, stackid);

473     if (ksp == NULL)
474         return (NULL);

476     bcopy(&template, ksp->ks_data, sizeof(template));
477     ksp->ks_update = udp_kstat2_update;
478     ksp->ks_private = (void *) (uintptr_t) stackid;

480     kstat_install(ksp);
481     return (ksp);
482 }

484 void
485 udp_kstat2_fini(netstackid_t stackid, kstat_t *ksp)
486 {
487     if (ksp != NULL) {
488         ASSERT(stackid == (netstackid_t)(uintptr_t)ksp->ks_private);
489         kstat_delete_netstack(ksp, stackid);
490     }
491 }

493 /*
494 * To copy counters from the per CPU udpp_stat_counter_t to the stack
495 * udp_stat_t.
496 */
497 static void
498 udp_add_stats(udp_stat_counter_t *from, udp_stat_t *to)
499 {
500     to->udp_sock_fallback.value.ui64 += from->udp_sock_fallback;
501     to->udp_out_opt.value.ui64 += from->udp_out_opt;
502     to->udp_out_err_notconn.value.ui64 += from->udp_out_err_notconn;
503     to->udp_out_err_output.value.ui64 += from->udp_out_err_output;
504     to->udp_out_err_tudr.value.ui64 += from->udp_out_err_tudr;
505 #ifdef DEBUG
506     to->udp_data_conn.value.ui64 += from->udp_data_conn;
507     to->udp_data_notconn.value.ui64 += from->udp_data_notconn;
508     to->udp_out_lastdst.value.ui64 += from->udp_out_lastdst;
509     to->udp_out_diffdst.value.ui64 += from->udp_out_diffdst;
510     to->udp_out_ipv6.value.ui64 += from->udp_out_ipv6;
511     to->udp_out_mapped.value.ui64 += from->udp_out_mapped;
512     to->udp_out_ipv4.value.ui64 += from->udp_out_ipv4;
513 #endif
514 }

516 /*
517 * To set all udp_stat_t counters to 0.
518 */
519 static void
520 udp_clr_stats(udp_stat_t *stats)
521 {

```

```

522 stats->udp_sock_fallback.value.ui64 = 0;
523 stats->udp_out_opt.value.ui64 = 0;
524 stats->udp_out_err_notconn.value.ui64 = 0;
525 stats->udp_out_err_output.value.ui64 = 0;
526 stats->udp_out_err_tudr.value.ui64 = 0;
527 #ifndef DEBUG
528 stats->udp_data_conn.value.ui64 = 0;
529 stats->udp_data_notconn.value.ui64 = 0;
530 stats->udp_out_lastdst.value.ui64 = 0;
531 stats->udp_out_diffdst.value.ui64 = 0;
532 stats->udp_out_ipv6.value.ui64 = 0;
533 stats->udp_out_mapped.value.ui64 = 0;
534 stats->udp_out_ipv4.value.ui64 = 0;
535 #endif
536 }

538 int
539 udp_kstat2_update(kstat_t *kp, int rw)
540 {
541     udp_stat_t      *stats;
542     netstackid_t    stackid = (netstackid_t)(uintptr_t)kp->ks_private;
543     netstack_t      *ns;
544     udp_stack_t     *us;
545     int              i;
546     int              cnt;

548     if (rw == KSTAT_WRITE)
549         return (EACCES);

551     ns = netstack_find_by_stackid(stackid);
552     if (ns == NULL)
553         return (-1);
554     us = ns->netstack_udp;
555     if (us == NULL) {
556         netstack_rele(ns);
557         return (-1);
558     }
559     stats = (udp_stat_t *)kp->ks_data;
560     udp_clr_stats(stats);

562     cnt = us->us_sc_cnt;
563     for (i = 0; i < cnt; i++)
564         udp_add_stats(&us->us_sc[i]->udp_sc_stats, stats);

566     netstack_rele(ns);
567     return (0);
568 }

570 void *
571 udp_kstat_init(netstackid_t stackid)
572 {
573     kstat_t *ksp;

575     udp_named_kstat_t template = {
576         { "inDatagrams",      KSTAT_DATA_UINT64, 0 },
577         { "inErrors",         KSTAT_DATA_UINT32, 0 },
578         { "outDatagrams",     KSTAT_DATA_UINT64, 0 },
579         { "entrySize",        KSTAT_DATA_INT32, 0 },
580         { "entry6Size",       KSTAT_DATA_INT32, 0 },
581         { "outErrors",        KSTAT_DATA_UINT32, 0 },
582     };

584     ksp = kstat_create_netstack(UDP_MOD_NAME, 0, UDP_MOD_NAME, "mib2",
585         KSTAT_TYPE_NAMED, NUM_OF_FIELDS(udp_named_kstat_t), 0, stackid);

587     if (ksp == NULL)

```

```

588         return (NULL);

590     template.entrySize.value.ui32 = sizeof (mib2_udpEntry_t);
591     template.entry6Size.value.ui32 = sizeof (mib2_udp6Entry_t);

593     bcopy(&template, ksp->ks_data, sizeof (template));
594     ksp->ks_update = udp_kstat_update;
595     ksp->ks_private = (void *) (uintptr_t)stackid;

597     kstat_install(ksp);
598     return (ksp);
599 }

601 /*
602  * To sum up all MIB2 stats for a udp_stack_t from all per CPU stats. The
603  * caller should initialize the target mib2_udp_t properly as this function
604  * just adds up all the per CPU stats.
605  */
606 static void
607 udp_sum_mib(udp_stack_t *us, mib2_udp_t *udp_mib)
608 {
609     int i;
610     int cnt;

612     cnt = us->us_sc_cnt;
613     for (i = 0; i < cnt; i++)
614         udp_add_mib(&us->us_sc[i]->udp_sc_mib, udp_mib);
615 }

617 static int
618 udp_kstat_update(kstat_t *kp, int rw)
619 {
620     udp_named_kstat_t *udpkp;
621     netstackid_t      stackid = (netstackid_t)(uintptr_t)kp->ks_private;
622     netstack_t        *ns;
623     udp_stack_t       *us;
624     mib2_udp_t        udp_mib;

626     if (rw == KSTAT_WRITE)
627         return (EACCES);

629     ns = netstack_find_by_stackid(stackid);
630     if (ns == NULL)
631         return (-1);
632     us = ns->netstack_udp;
633     if (us == NULL) {
634         netstack_rele(ns);
635         return (-1);
636     }
637     udpkp = (udp_named_kstat_t *)kp->ks_data;

639     bzero(&udp_mib, sizeof (udp_mib));
640     udp_sum_mib(us, &udp_mib);

642     udpkp->inDatagrams.value.ui64 = udp_mib.udpHCInDatagrams;
643     udpkp->inErrors.value.ui32 = udp_mib.udpInErrors;
644     udpkp->outDatagrams.value.ui64 = udp_mib.udpHCOutDatagrams;
645     udpkp->outErrors.value.ui32 = udp_mib.udpOutErrors;
646     netstack_rele(ns);
647     return (0);
648 }

```

```

*****
47248 Wed Jul 13 01:32:40 2016
new/usr/src/uts/common/os/fio.c
XXXX adding PID information to netstat output
*****
_____unchanged_portion_omitted_____

837 /*
838  * Duplicate all file descriptors across a fork.
839  */
840 void
841 flist_fork(proc_t *pp, proc_t *cp)
842 flist_fork(uf_info_t *pfip, uf_info_t *cfip)
843 {
844     int fd, nfiles;
845     uf_entry_t *pufp, *cufp;

846     uf_info_t *pfip = P_FINFO(pp);
847     uf_info_t *cfip = P_FINFO(cp);

849 #endif /* ! codereview */
850     mutex_init(&cfip->fi_lock, NULL, MUTEX_DEFAULT, NULL);
851     cfip->fi_rlist = NULL;

853     /*
854     * We don't need to hold fi_lock because all other lwp's in the
855     * parent have been held.
856     */
857     cfip->fi_nfiles = nfiles = flist_minsize(pfip);

859     cfip->fi_list = kmem_zalloc(nfiles * sizeof (uf_entry_t), KM_SLEEP);

861     for (fd = 0, pufp = pfip->fi_list, cufp = cfip->fi_list; fd < nfiles;
862          fd++, pufp++, cufp++) {
863         cufp->uf_file = pufp->uf_file;
864         cufp->uf_alloc = pufp->uf_alloc;
865         cufp->uf_flag = pufp->uf_flag;
866         cufp->uf_busy = pufp->uf_busy;

868         if (cufp->uf_file != NULL && cufp->uf_file->f_vnode != NULL) {
869             (void) VOP_IOCTL(cufp->uf_file->f_vnode, F_ASSOCI_PID,
870                             (intptr_t)cp->p_pidp->pid_id, FKIOCTL, kcred,
871                             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         (void) VOP_IOCTL(vp, F_DASSOC_PID,
962             (intptr_t)(ttoproc(curthread)->p_pidp->pid_id), FKIOCTL,
963             kcred, NULL, NULL);
964     }
965 #endif /* ! codereview */

967     error = VOP_CLOSE(vp, flag, count, offset, fp->f_cred, NULL);

969     if (count > 1) {
970         mutex_exit(&fp->f_tlock);
971         return (error);
972     }
973     ASSERT(fp->f_count == 0);
974     /* Last reference, remove any OFD style lock for the file_t */
975     ofdcleanlock(fp);
976     mutex_exit(&fp->f_tlock);

978     /*
979     * If DTrace has getf() subroutines active, it will set dtrace_closef
980     * to point to code that implements a barrier with respect to probe
981     * context. This must be called before the file_t is freed (and the
982     * vnode that it refers to is released) -- but it must be after the
983     * file_t has been removed from the uf_entry_t. That is, there must
984     * be no way for a racing getf() in probe context to yield the fp that
985     * we're operating upon.
986     */
987     if (dtrace_closef != NULL)
988         (*dtrace_closef)();

990     VN_RELE(vp);
991     /*
992     * deallocate resources to audit_data
993     */
994     if (audit_active)
995         audit_unfalloc(fp);
996     crfree(fp->f_cred);
997     kmem_cache_free(file_cache, fp);
998     return (error);
999 }

1001 /*
1002 * This is a combination of ufalloc() and setf().
1003 */
1004 int
1005 ufalloc_file(int start, file_t *fp)
1006 {
1007     proc_t *p = curproc;
1008     uf_info_t *fip = P_FINFO(p);
1009     int filelimit;
1010     uf_entry_t *ufp;
1011     int nfiles;
1012     int fd;

1014     /*
1015     * Assertion is to convince the correctness of the following
1016     * assignment for filelimit after casting to int.
1017     */
1018     ASSERT(p->p_fno_ctl <= INT_MAX);
1019     filelimit = (int)p->p_fno_ctl;

1021     for (;;) {
1022         mutex_enter(&fip->fi_lock);
1023         fd = fd_find(fip, start);
1024         if (fd >= 0 && fd == fip->fi_badfd) {
1025             start = fd + 1;
1026             mutex_exit(&fip->fi_lock);

```

```

1027         continue;
1028     }
1029     if ((uint_t)fd < filelimit)
1030         break;
1031     if (fd >= filelimit) {
1032         mutex_exit(&fip->fi_lock);
1033         mutex_enter(&p->p_lock);
1034         (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1035             p->p_rctlis, p, RCA_SAFE);
1036         mutex_exit(&p->p_lock);
1037         return (-1);
1038     }
1039     /* fd_find() returned -1 */
1040     nfiles = fip->fi_nfiles;
1041     mutex_exit(&fip->fi_lock);
1042     flist_grow(MAX(start, nfiles));
1043 }

1045     UF_ENTER(ufp, fip, fd);
1046     fd_reserve(fip, fd, 1);
1047     ASSERT(ufp->uf_file == NULL);
1048     ufp->uf_file = fp;
1049     UF_EXIT(ufp);
1050     mutex_exit(&fip->fi_lock);
1051     return (fd);
1052 }

1054 /*
1055 * Allocate a user file descriptor greater than or equal to "start".
1056 */
1057 int
1058 ufalloc(int start)
1059 {
1060     return (ufalloc_file(start, NULL));
1061 }

1063 /*
1064 * Check that a future allocation of count fds on proc p has a good
1065 * chance of succeeding. If not, do rctl processing as if we'd failed
1066 * the allocation.
1067 *
1068 * Our caller must guarantee that p cannot disappear underneath us.
1069 */
1070 int
1071 ufcanalloc(proc_t *p, uint_t count)
1072 {
1073     uf_info_t *fip = P_FINFO(p);
1074     int filelimit;
1075     int current;

1077     if (count == 0)
1078         return (1);

1080     ASSERT(p->p_fno_ctl <= INT_MAX);
1081     filelimit = (int)p->p_fno_ctl;

1083     mutex_enter(&fip->fi_lock);
1084     current = flist_nalloc(fip); /* # of in-use descriptors */
1085     mutex_exit(&fip->fi_lock);

1087     /*
1088     * If count is a positive integer, the worst that can happen is
1089     * an overflow to a negative value, which is caught by the >= 0 check.
1090     */
1091     current += count;
1092     if (count <= INT_MAX && current >= 0 && current <= filelimit)

```

```

1093         return (1);
1095     mutex_enter(&p->p_lock);
1096     (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1097         p->p_rctls, p, RCA_SAFE);
1098     mutex_exit(&p->p_lock);
1099     return (0);
1100 }
1102 /*
1103  * Allocate a user file descriptor and a file structure.
1104  * Initialize the descriptor to point at the file structure.
1105  * If fdp is NULL, the user file descriptor will not be allocated.
1106  */
1107 int
1108 falloc(vnode_t *vp, int flag, file_t **fpp, int *fdp)
1109 {
1110     file_t *fp;
1111     int fd;
1113     if (fdp) {
1114         if ((fd = ufalloc(0)) == -1)
1115             return (EMFILE);
1116     }
1117     fp = kmem_cache_alloc(file_cache, KM_SLEEP);
1118     /*
1119      * Note: falloc returns the fp locked
1120      */
1121     mutex_enter(&fp->f_tlock);
1122     fp->f_count = 1;
1123     fp->f_flag = (ushort_t)flag;
1124     fp->f_flag2 = (flag & (FSEARCH|FEXEC)) >> 16;
1125     fp->f_vnode = vp;
1126     fp->f_offset = 0;
1127     fp->f_audit_data = 0;
1128     crhold(fp->f_cred = CRED());
1129     /*
1130      * allocate resources to audit_data
1131      */
1132     if (audit_active)
1133         audit_falloc(fp);
1134     *fpp = fp;
1135     if (fdp)
1136         *fdp = fd;
1137     return (0);
1138 }
1140 /*ARGSUSED*/
1141 static int
1142 file_cache_constructor(void *buf, void *cdrarg, int kmflags)
1143 {
1144     file_t *fp = buf;
1146     mutex_init(&fp->f_tlock, NULL, MUTEX_DEFAULT, NULL);
1147     return (0);
1148 }
1150 /*ARGSUSED*/
1151 static void
1152 file_cache_destructor(void *buf, void *cdrarg)
1153 {
1154     file_t *fp = buf;
1156     mutex_destroy(&fp->f_tlock);
1157 }

```

```

1159 void
1160 finit()
1161 {
1162     file_cache = kmem_cache_create("file_cache", sizeof (file_t), 0,
1163     file_cache_constructor, file_cache_destructor, NULL, NULL, NULL, 0);
1164 }
1166 void
1167 unfalloc(file_t *fp)
1168 {
1169     ASSERT(MUTEX_HELD(&fp->f_tlock));
1170     if (--fp->f_count <= 0) {
1171         /*
1172          * deallocate resources to audit_data
1173          */
1174         if (audit_active)
1175             audit_unfalloc(fp);
1176         crfree(fp->f_cred);
1177         mutex_exit(&fp->f_tlock);
1178         kmem_cache_free(file_cache, fp);
1179     } else
1180         mutex_exit(&fp->f_tlock);
1181 }
1183 /*
1184  * Given a file descriptor, set the user's
1185  * file pointer to the given parameter.
1186  */
1187 void
1188 setf(int fd, file_t *fp)
1189 {
1190     uf_info_t *fip = P_FINFO(curproc);
1191     uf_entry_t *ufp;
1193     if (AU_AUDITING())
1194         audit_setf(fp, fd);
1196     if (fp == NULL) {
1197         mutex_enter(&fip->fi_lock);
1198         UF_ENTER(ufp, fip, fd);
1199         fd_reserve(fip, fd, -1);
1200         mutex_exit(&fip->fi_lock);
1201     } else {
1202         UF_ENTER(ufp, fip, fd);
1203         ASSERT(ufp->uf_busy);
1204     }
1205     ASSERT(ufp->uf_pollinfo == NULL);
1206     ASSERT(ufp->uf_flag == 0);
1207     ufp->uf_file = fp;
1208     cv_broadcast(&ufp->uf_wanted_cv);
1209     UF_EXIT(ufp);
1210 }
1212 /*
1213  * Given a file descriptor, return the file table flags, plus,
1214  * if this is a socket in asynchronous mode, the FASYNC flag.
1215  * getf() may or may not have been called before calling f_getfl().
1216  */
1217 int
1218 f_getfl(int fd, int *flagp)
1219 {
1220     uf_info_t *fip = P_FINFO(curproc);
1221     uf_entry_t *ufp;
1222     file_t *fp;
1223     int error;

```

```

1225     if ((uint_t)fd >= fip->fi_nfiles)
1226         error = EBADF;
1227     else {
1228         UF_ENTER(ufp, fip, fd);
1229         if ((fp = ufp->uf_file) == NULL)
1230             error = EBADF;
1231         else {
1232             vnode_t *vp = fp->f_vnode;
1233             int flag = fp->f_flag |
1234                 ((fp->f_flag2 & ~FEPOLLED) << 16);
1235
1236             /*
1237              * BSD fcntl() FASYNC compatibility.
1238              */
1239             if (vp->v_type == VSOCK)
1240                 flag |= sock_getfasync(vp);
1241             *flagp = flag;
1242             error = 0;
1243         }
1244         UF_EXIT(ufp);
1245     }
1246
1247     return (error);
1248 }
1249
1250 /*
1251  * Given a file descriptor, return the user's file flags.
1252  * Force the FD_CLOEXEC flag for writable self-open /proc files.
1253  * getf() may or may not have been called before calling f_getfd_error().
1254  */
1255 int
1256 f_getfd_error(int fd, int *flagp)
1257 {
1258     uf_info_t *fip = P_FINFO(curproc);
1259     uf_entry_t *ufp;
1260     file_t *fp;
1261     int flag;
1262     int error;
1263
1264     if ((uint_t)fd >= fip->fi_nfiles)
1265         error = EBADF;
1266     else {
1267         UF_ENTER(ufp, fip, fd);
1268         if ((fp = ufp->uf_file) == NULL)
1269             error = EBADF;
1270         else {
1271             flag = ufp->uf_flag;
1272             if ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode))
1273                 flag |= FD_CLOEXEC;
1274             *flagp = flag;
1275             error = 0;
1276         }
1277         UF_EXIT(ufp);
1278     }
1279
1280     return (error);
1281 }
1282
1283 /*
1284  * getf() must have been called before calling f_getfd().
1285  */
1286 char
1287 f_getfd(int fd)
1288 {
1289     int flag = 0;
1290     (void) f_getfd_error(fd, &flag);

```

```

1291         return ((char)flag);
1292     }
1293
1294 /*
1295  * Given a file descriptor and file flags, set the user's file flags.
1296  * At present, the only valid flag is FD_CLOEXEC.
1297  * getf() may or may not have been called before calling f_setfd_error().
1298  */
1299 int
1300 f_setfd_error(int fd, int flags)
1301 {
1302     uf_info_t *fip = P_FINFO(curproc);
1303     uf_entry_t *ufp;
1304     int error;
1305
1306     if ((uint_t)fd >= fip->fi_nfiles)
1307         error = EBADF;
1308     else {
1309         UF_ENTER(ufp, fip, fd);
1310         if (ufp->uf_file == NULL)
1311             error = EBADF;
1312         else {
1313             ufp->uf_flag = flags & FD_CLOEXEC;
1314             error = 0;
1315         }
1316         UF_EXIT(ufp);
1317     }
1318     return (error);
1319 }
1320
1321 void
1322 f_setfd(int fd, char flags)
1323 {
1324     (void) f_setfd_error(fd, flags);
1325 }
1326
1327 #define BADFD_MIN    3
1328 #define BADFD_MAX    255
1329
1330 /*
1331  * Attempt to allocate a file descriptor which is bad and which
1332  * is "poison" to the application. It cannot be closed (except
1333  * on exec), allocated for a different use, etc.
1334  */
1335 int
1336 f_badfd(int start, int *fdp, int action)
1337 {
1338     int fdr;
1339     int badfd;
1340     uf_info_t *fip = P_FINFO(curproc);
1341
1342 #ifdef _LP64
1343     /* No restrictions on 64 bit _file */
1344     if (get_umatamodel() != DATAMODEL_ILP32)
1345         return (EINVAL);
1346 #endif
1347
1348     if (start > BADFD_MAX || start < BADFD_MIN)
1349         return (EINVAL);
1350
1351     if (action >= NSIG || action < 0)
1352         return (EINVAL);
1353
1354     mutex_enter(&fip->fi_lock);
1355     badfd = fip->fi_badfd;
1356     mutex_exit(&fip->fi_lock);

```

```

1358     if (badfd != -1)
1359         return (EAGAIN);

1361     fdr = ufalloc(start);

1363     if (fdr > BADFD_MAX) {
1364         setf(fdr, NULL);
1365         return (EMFILE);
1366     }
1367     if (fdr < 0)
1368         return (EMFILE);

1370     mutex_enter(&fip->fi_lock);
1371     if (fip->fi_badfd != -1) {
1372         /* Lost race */
1373         mutex_exit(&fip->fi_lock);
1374         setf(fdr, NULL);
1375         return (EAGAIN);
1376     }
1377     fip->fi_action = action;
1378     fip->fi_badfd = fdr;
1379     mutex_exit(&fip->fi_lock);
1380     setf(fdr, NULL);

1382     *fdp = fdr;

1384     return (0);
1385 }

1387 /*
1388  * Allocate a file descriptor and assign it to the vnode "*vpp",
1389  * performing the usual open protocol upon it and returning the
1390  * file descriptor allocated. It is the responsibility of the
1391  * caller to dispose of "*vpp" if any error occurs.
1392  */
1393 int
1394 fassign(vnode_t **vpp, int mode, int *fdp)
1395 {
1396     file_t *fp;
1397     int error;
1398     int fd;

1400     if (error = falloc((vnode_t *)NULL, mode, &fp, &fd))
1401         return (error);
1402     if (error = VOP_OPEN(vpp, mode, fp->f_cred, NULL)) {
1403         setf(fd, NULL);
1404         unfalloc(fp);
1405         return (error);
1406     }
1407     fp->f_vnode = *vpp;
1408     mutex_exit(&fp->f_tlock);
1409     /*
1410      * Fill in the slot falloc reserved.
1411      */
1412     setf(fd, fp);
1413     *fdp = fd;
1414     return (0);
1415 }

1417 /*
1418  * When a process forks it must increment the f_count of all file pointers
1419  * since there is a new process pointing at them. fcnt_add(fip, 1) does this.
1420  * Since we are called when there is only 1 active lwp we don't need to
1421  * hold fi_lock or any uf_lock. If the fork fails, fork_fail() calls
1422  * fcnt_add(fip, -1) to restore the counts.

```

```

1423  */
1424 void
1425 fcnt_add(uf_info_t *fip, int incr)
1426 {
1427     int i;
1428     uf_entry_t *ufp;
1429     file_t *fp;

1431     ufp = fip->fi_list;
1432     for (i = 0; i < fip->fi_nfiles; i++, ufp++) {
1433         if ((fp = ufp->uf_file) != NULL) {
1434             mutex_enter(&fp->f_tlock);
1435             ASSERT((incr == 1 && fp->f_count >= 1) ||
1436                 (incr == -1 && fp->f_count >= 2));
1437             fp->f_count += incr;
1438             mutex_exit(&fp->f_tlock);
1439         }
1440     }
1441 }

1443 /*
1444  * This is called from exec to close all fd's that have the FD_CLOEXEC flag
1445  * set and also to close all self-open for write /proc file descriptors.
1446  */
1447 void
1448 close_exec(uf_info_t *fip)
1449 {
1450     int fd;
1451     file_t *fp;
1452     fpollinfo_t *fpip;
1453     uf_entry_t *ufp;
1454     portfd_t *pfd;

1456     ufp = fip->fi_list;
1457     for (fd = 0; fd < fip->fi_nfiles; fd++, ufp++) {
1458         if ((fp = ufp->uf_file) != NULL &&
1459             ((ufp->uf_flag & FD_CLOEXEC) ||
1460              ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode)))) {
1461             fpip = ufp->uf_fpollinfo;
1462             mutex_enter(&fip->fi_lock);
1463             mutex_enter(&ufp->uf_lock);
1464             fd_reserve(fip, fd, -1);
1465             mutex_exit(&fip->fi_lock);
1466             ufp->uf_file = NULL;
1467             ufp->uf_fpollinfo = NULL;
1468             ufp->uf_flag = 0;
1469             /*
1470              * We may need to cleanup some cached poll states
1471              * in t_pollstate before the fd can be reused. It
1472              * is important that we don't access a stale thread
1473              * structure. We will do the cleanup in two
1474              * phases to avoid deadlock and holding uf_lock for
1475              * too long. In phase 1, hold the uf_lock and call
1476              * pollblockexit() to set state in t_pollstate struct
1477              * so that a thread does not exit on us. In phase 2,
1478              * we drop the uf_lock and call pollcacheclean().
1479              */
1480             pfd = ufp->uf_portfd;
1481             ufp->uf_portfd = NULL;
1482             if (fpip != NULL)
1483                 pollblockexit(fpip);
1484             mutex_exit(&ufp->uf_lock);
1485             if (fpip != NULL)
1486                 pollcacheclean(fpip, fd);
1487             if (pfd)
1488                 port_close_fd(pfd);

```

```

1489         (void) closef(fp);
1490     }
1491 }

1493 /* Reset bad fd */
1494 fip->fi_badfd = -1;
1495 fip->fi_action = -1;
1496 }

1498 /*
1499 * Utility function called by most of the *at() system call interfaces.
1500 * Generate a starting vnode pointer for an (fd, path) pair where 'fd'
1501 * is an open file descriptor for a directory to be used as the starting
1502 * point for the lookup of the relative pathname 'path' (or, if path is
1503 * NULL, generate a vnode pointer for the direct target of the operation).
1504 * NULL, generate a vnode pointer for the direct target of the operation).
1505 * If we successfully return a non-NULL startvp, it has been the target
1506 * of VN_HOLD() and the caller must call VN_RELE() on it.
1507 */
1508 int
1509 fgetstartvp(int fd, char *path, vnode_t **startvpp)
1510 {
1511     vnode_t      *startvp;
1512     file_t       *startfp;
1513     char         startchar;

1516     if (fd == AT_FDCWD && path == NULL)
1517         return (EFAULT);

1519     if (fd == AT_FDCWD) {
1520         /*
1521          * Start from the current working directory.
1522          */
1523         startvp = NULL;
1524     } else {
1525         if (path == NULL)
1526             startchar = '\0';
1527         else if (copyin(path, &startchar, sizeof (char)))
1528             return (EFAULT);

1530         if (startchar == '/') {
1531             /*
1532              * 'path' is an absolute pathname.
1533              */
1534             startvp = NULL;
1535         } else {
1536             /*
1537              * 'path' is a relative pathname or we will
1538              * be applying the operation to 'fd' itself.
1539              */
1540             if ((startfp = getf(fd)) == NULL)
1541                 return (EBADF);
1542             startvp = startfp->f_vnode;
1543             VN_HOLD(startvp);
1544             releasef(fd);
1545         }
1546     }
1547     *startvpp = startvp;
1548     return (0);
1549 }

1551 /*
1552 * Called from fchownat() and fchmodat() to set ownership and mode.
1553 * The contents of *vap must be set before calling here.
1554 */

```

```

1555 int
1556 fsetattrat(int fd, char *path, int flags, struct vattr *vap)
1557 {
1558     vnode_t      *startvp;
1559     vnode_t      *vp;
1560     int          error;

1562     /*
1563      * Since we are never called to set the size of a file, we don't
1564      * need to check for non-blocking locks (via nbl_need_check(vp)).
1565      */
1566     ASSERT(!(vap->va_mask & AT_SIZE));

1568     if ((error = fgetstartvp(fd, path, &startvp)) != 0)
1569         return (error);
1570     if (AU_AUDITING() && startvp != NULL)
1571         audit_setfsat_path(1);

1573     /*
1574      * Do lookup for fchownat/fchmodat when path not NULL
1575      */
1576     if (path != NULL) {
1577         if (error = lookupnameat(path, UIO_USERSPACE,
1578             (flags == AT_SYMLINK_NOFOLLOW) ?
1579             NO_FOLLOW : FOLLOW,
1580             NULLVPP, &vp, startvp)) {
1581             if (startvp != NULL)
1582                 VN_RELE(startvp);
1583             return (error);
1584         }
1585     } else {
1586         vp = startvp;
1587         ASSERT(vp);
1588         VN_HOLD(vp);
1589     }

1591     if (vn_is_readonly(vp)) {
1592         error = EROFS;
1593     } else {
1594         error = VOP_SETATTR(vp, vap, 0, CRED(), NULL);
1595     }

1597     if (startvp != NULL)
1598         VN_RELE(startvp);
1599     VN_RELE(vp);

1601     return (error);
1602 }

1604 /*
1605 * Return true if the given vnode is referenced by any
1606 * entry in the current process's file descriptor table.
1607 */
1608 int
1609 fisopen(vnode_t *vp)
1610 {
1611     int fd;
1612     file_t *fp;
1613     vnode_t *ovp;
1614     uf_info_t *fip = P_FINFO(curproc);
1615     uf_entry_t *ufp;

1617     mutex_enter(&fip->fi_lock);
1618     for (fd = 0; fd < fip->fi_nfiles; fd++) {
1619         UF_ENTER(ufp, fip, fd);
1620         if ((fp = ufp->uf_file) != NULL &&

```

```

1621         (ovp = fp->f_vnode) != NULL && VN_CMP(vp, ovp)) {
1622             UF_EXIT(ufp);
1623             mutex_exit(&fip->fi_lock);
1624             return (1);
1625         }
1626         UF_EXIT(ufp);
1627     }
1628     mutex_exit(&fip->fi_lock);
1629     return (0);
1630 }

1632 /*
1633  * Return zero if at least one file currently open (by curproc) shouldn't be
1634  * allowed to change zones.
1635  */
1636 int
1637 files_can_change_zones(void)
1638 {
1639     int fd;
1640     file_t *fp;
1641     uf_info_t *fip = P_FINFO(curproc);
1642     uf_entry_t *ufp;

1644     mutex_enter(&fip->fi_lock);
1645     for (fd = 0; fd < fip->fi_nfiles; fd++) {
1646         UF_ENTER(ufp, fip, fd);
1647         if ((fp = ufp->uf_file) != NULL &&
1648             !vn_can_change_zones(fp->f_vnode)) {
1649             UF_EXIT(ufp);
1650             mutex_exit(&fip->fi_lock);
1651             return (0);
1652         }
1653         UF_EXIT(ufp);
1654     }
1655     mutex_exit(&fip->fi_lock);
1656     return (1);
1657 }

1659 #ifdef DEBUG

1661 /*
1662  * The following functions are only used in ASSERT()s elsewhere.
1663  * They do not modify the state of the system.
1664  */

1666 /*
1667  * Return true (1) if the current thread is in the fpollinfo
1668  * list for this file descriptor, else false (0).
1669  */
1670 static int
1671 curthread_in_plist(uf_entry_t *ufp)
1672 {
1673     fpollinfo_t *fpip;

1675     ASSERT(MUTEX_HELD(&ufp->uf_lock));
1676     for (fpip = ufp->uf_fpollinfo; fpip; fpip = fpip->fp_next)
1677         if (fpip->fp_thread == curthread)
1678             return (1);
1679     return (0);
1680 }

1682 /*
1683  * Sanity check to make sure that after lwp_exit(),
1684  * curthread does not appear on any fd's fpollinfo list.
1685  */
1686 void

```

```

1687 checkfpollinfo(void)
1688 {
1689     int fd;
1690     uf_info_t *fip = P_FINFO(curproc);
1691     uf_entry_t *ufp;

1693     mutex_enter(&fip->fi_lock);
1694     for (fd = 0; fd < fip->fi_nfiles; fd++) {
1695         UF_ENTER(ufp, fip, fd);
1696         ASSERT(!curthread_in_plist(ufp));
1697         UF_EXIT(ufp);
1698     }
1699     mutex_exit(&fip->fi_lock);
1700 }

1702 /*
1703  * Return true (1) if the current thread is in the fpollinfo
1704  * list for this file descriptor, else false (0).
1705  * This is the same as curthread_in_plist(),
1706  * but is called w/o holding uf_lock.
1707  */
1708 int
1709 infpollinfo(int fd)
1710 {
1711     uf_info_t *fip = P_FINFO(curproc);
1712     uf_entry_t *ufp;
1713     int rc;

1715     UF_ENTER(ufp, fip, fd);
1716     rc = curthread_in_plist(ufp);
1717     UF_EXIT(ufp);
1718     return (rc);
1719 }

1721 #endif /* DEBUG */

1723 /*
1724  * Add the curthread to fpollinfo list, meaning this fd is currently in the
1725  * thread's poll cache. Each lwp polling this file descriptor should call
1726  * this routine once.
1727  */
1728 void
1729 addfpollinfo(int fd)
1730 {
1731     struct uf_entry *ufp;
1732     fpollinfo_t *fpip;
1733     uf_info_t *fip = P_FINFO(curproc);

1735     fpip = kmem_zalloc(sizeof (fpollinfo_t), KM_SLEEP);
1736     fpip->fp_thread = curthread;
1737     UF_ENTER(ufp, fip, fd);
1738     /*
1739      * Assert we are not already on the list, that is, that
1740      * this lwp did not call addfpollinfo twice for the same fd.
1741      */
1742     ASSERT(!curthread_in_plist(ufp));
1743     /*
1744      * addfpollinfo is always done inside the getf/releasef pair.
1745      */
1746     ASSERT(ufp->uf_refcnt >= 1);
1747     fpip->fp_next = ufp->uf_fpollinfo;
1748     ufp->uf_fpollinfo = fpip;
1749     UF_EXIT(ufp);
1750 }

1752 /*

```

```

1753 * Delete curthread from fpollinfo list if it is there.
1754 */
1755 void
1756 delfpollinfo(int fd)
1757 {
1758     struct uf_entry *ufp;
1759     struct fpollinfo *fpip;
1760     struct fpollinfo **fpipp;
1761     uf_info_t *fip = P_FINFO(curproc);
1762
1763     UF_ENTER(ufp, fip, fd);
1764     for (fpipp = &ufp->uf_fpollinfo;
1765          (fpip = *fpipp) != NULL;
1766          fpipp = &fpip->fp_next) {
1767         if (fpip->fp_thread == curthread) {
1768             *fpipp = fpip->fp_next;
1769             kmem_free(fpip, sizeof (fpollinfo_t));
1770             break;
1771         }
1772     }
1773     /*
1774      * Assert that we are not still on the list, that is, that
1775      * this lwp did not call addfpollinfo twice for the same fd.
1776      */
1777     ASSERT(!curthread_in_plist(ufp));
1778     UF_EXIT(ufp);
1779 }
1780
1781 /*
1782  * fd is associated with a port. pfd is a pointer to the fd entry in the
1783  * cache of the port.
1784  */
1785
1786 void
1787 addfd_port(int fd, portfd_t *pfd)
1788 {
1789     struct uf_entry *ufp;
1790     uf_info_t *fip = P_FINFO(curproc);
1791
1792     UF_ENTER(ufp, fip, fd);
1793     /*
1794      * addfd_port is always done inside the getf/releasef pair.
1795      */
1796     ASSERT(ufp->uf_refcnt >= 1);
1797     if (ufp->uf_portfd == NULL) {
1798         /* first entry */
1799         ufp->uf_portfd = pfd;
1800         pfd->pfd_next = NULL;
1801     } else {
1802         pfd->pfd_next = ufp->uf_portfd;
1803         ufp->uf_portfd = pfd;
1804         pfd->pfd_next->pfd_prev = pfd;
1805     }
1806     UF_EXIT(ufp);
1807 }
1808
1809 void
1810 delfd_port(int fd, portfd_t *pfd)
1811 {
1812     struct uf_entry *ufp;
1813     uf_info_t *fip = P_FINFO(curproc);
1814
1815     UF_ENTER(ufp, fip, fd);
1816     /*
1817      * delfd_port is always done inside the getf/releasef pair.
1818      */

```

```

1819     ASSERT(ufp->uf_refcnt >= 1);
1820     if (ufp->uf_portfd == pfd) {
1821         /* remove first entry */
1822         ufp->uf_portfd = pfd->pfd_next;
1823     } else {
1824         pfd->pfd_prev->pfd_next = pfd->pfd_next;
1825         if (pfd->pfd_next != NULL)
1826             pfd->pfd_next->pfd_prev = pfd->pfd_prev;
1827     }
1828     UF_EXIT(ufp);
1829 }
1830
1831 static void
1832 port_close_fd(portfd_t *pfd)
1833 {
1834     portfd_t *pfdn;
1835
1836     /*
1837      * At this point, no other thread should access
1838      * the portfd_t list for this fd. The uf_file, uf_portfd
1839      * pointers in the uf_entry_t struct for this fd would
1840      * be set to NULL.
1841      */
1842     for (; pfd != NULL; pfd = pfdn) {
1843         pfdn = pfd->pfd_next;
1844         port_close_pfd(pfd);
1845     }
1846 }

```

```

*****
37047 Wed Jul 13 01:32:40 2016
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_tl_lgrpuid = LGRP_NONE;
1005     cp->p_tr_lgrpuid = 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, procvf, 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         zcomm_err(zoneid, CE_NOTE,
1053             "out of per-user processes for uid %d", ruid);
1054         goto bad;
1055     }
1056
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);
1064
1065     cp->p_next = practive;
1066     practive->p_prev = cp;
1067     practive = cp;
1068
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_procdta(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;
1090
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));
1096
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;
1104
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_sibling = cp;
1113
1114     cp->p_parent = pp;
1115     pp->p_child = cp;

```

```

1117     cp->p_child_ns = NULL;
1118     cp->p_sibling_ns = NULL;
1119
1120     cp->p_nextorph = pp->p_orphan;
1121     cp->p_nextofkin = pp;
1122     pp->p_orphan = cp;
1123
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;
1129
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);
1143
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);
1155
1156     avl_create(&cp->p_ct_held, contract_compar, sizeof (contract_t),
1157         offsetof(contract_t, ct_ctlist));
1158
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);
1164
1165     crhold(cp->p_cred = cr);
1166
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);
1172
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     getthretime(&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         premtypset(&cp->p_fltmask);
1209         uarea->u_systrap = 0;
1210         premtypset(&uarea->u_entrymask);
1211         premtypset(&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_plock);
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_ffnoprocs);

```

```

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

new/usr/src/uts/common/os/pidnode.c

1

849 Wed Jul 13 01:32:40 2016

new/usr/src/uts/common/os/pidnode.c

XXXX adding PID information to netstat output

```
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2015 Mohamed A. Khalfella <khalfella@gmail.com>
14 */

16 /*
17  * General pidnode routines are stored in this file.
18 */

20 #include <sys/pidnode.h>

23 /*
24  * Compare two pid_node_t elements. Used by AVL trees.
25 */

27 int
28 pid_node_comparator(const void *l, const void *r)
29 {
30     const pid_node_t *li = l;
31     const pid_node_t *ri = r;

33     if (li->pn_pid > ri->pn_pid)
34         return (1);
35     if (li->pn_pid < ri->pn_pid)
36         return (-1);
37     return (0);
38 }
39 #endif /* ! codereview */
```

new/usr/src/uts/common/os/streamio.c

1

```
*****
219093 Wed Jul 13 01:32:40 2016
new/usr/src/uts/common/os/streamio.c
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
22 /*      All Rights Reserved */

25 /*
26  * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
27  * Copyright (c) 2014, Joyent, Inc. All rights reserved.
28  */

30 #include <sys/types.h>
31 #include <sys/sysmacros.h>
32 #include <sys/param.h>
33 #include <sys/errno.h>
34 #include <sys/signal.h>
35 #include <sys/stat.h>
36 #include <sys/proc.h>
37 #include <sys/cred.h>
38 #include <sys/user.h>
39 #include <sys/vnode.h>
40 #include <sys/file.h>
41 #include <sys/stream.h>
42 #include <sys/strsubr.h>
43 #include <sys/stropts.h>
44 #include <sys/tihdr.h>
45 #include <sys/var.h>
46 #include <sys/poll.h>
47 #include <sys/termio.h>
48 #include <sys/ttold.h>
49 #include <sys/system.h>
50 #include <sys/uio.h>
51 #include <sys/cmn_err.h>
52 #include <sys/sad.h>
53 #include <sys/netstack.h>
54 #include <sys/priocntl.h>
55 #include <sys/jioctl.h>
56 #include <sys/procset.h>
57 #include <sys/session.h>
58 #include <sys/kmem.h>
59 #include <sys/filio.h>
60 #include <sys/vtrace.h>
61 #include <sys/debug.h>
```

new/usr/src/uts/common/os/streamio.c

2

```
62 #include <sys/stredir.h>
63 #include <sys/fs/fifonode.h>
64 #include <sys/fs/snnode.h>
65 #include <sys/strlog.h>
66 #include <sys/strsun.h>
67 #include <sys/project.h>
68 #include <sys/kbio.h>
69 #include <sys/msio.h>
70 #include <sys/tty.h>
71 #include <sys/ptyvar.h>
72 #include <sys/vuid_event.h>
73 #include <sys/modctl.h>
74 #include <sys/sunddi.h>
75 #include <sys/sunldi_impl.h>
76 #include <sys/autoconf.h>
77 #include <sys/policy.h>
78 #include <sys/dld.h>
79 #include <sys/zone.h>
80 #include <c2/audit.h>
81 #include <sys/fcntl.h>
82 #endif /* ! codereview */

84 /*
85  * This define helps improve the readability of streams code while
86  * still maintaining a very old streams performance enhancement. The
87  * performance enhancement basically involved having all callers
88  * of straccess() perform the first check that straccess() will do
89  * locally before actually calling straccess(). (There by reducing
90  * the number of unnecessary calls to straccess().)
91  */
92 #define i_straccess(x, y)      ((stp->sd_sidp == NULL) ? 0 : \
93                               (stp->sd_vnode->v_type == VFIFO) ? 0 : \
94                               straccess(x, (y)))

96 /*
97  * what is mblk_pull_len?
98  *
99  * If a streams message consists of many short messages,
100 * a performance degradation occurs from copyout overhead.
101 * To decrease the per mblk overhead, messages that are
102 * likely to consist of many small mblks are pulled up into
103 * one continuous chunk of memory.
104 *
105 * To avoid the processing overhead of examining every
106 * mblk, a quick heuristic is used. If the first mblk in
107 * the message is shorter than mblk_pull_len, it is likely
108 * that the rest of the mblk will be short.
109 *
110 * This heuristic was decided upon after performance tests
111 * indicated that anything more complex slowed down the main
112 * code path.
113 */
114 #define MBLK_PULL_LEN 64
115 uint32_t mblk_pull_len = MBLK_PULL_LEN;

117 /*
118  * The sgtyb_handling flag controls the handling of the old BSD
119  * TIOCGETP, TIOCSETP, and TIOCSETN ioctls as follows:
120  *
121  * 0 - Emit no warnings at all and retain old, broken behavior.
122  * 1 - Emit no warnings and silently handle new semantics.
123  * 2 - Send cmn_err(CE_NOTE) when either TIOCSETP or TIOCSETN is used
124  *     (once per system invocation). Handle with new semantics.
125  * 3 - Send SIGSYS when any TIOCGETP, TIOCSETP, or TIOCSETN call is
126  *     made (so that offenders drop core and are easy to debug).
127  */
```

```

128 * The "new semantics" are that TIOCGETP returns B38400 for
129 * sg_[io]speed if the corresponding value is over B38400, and that
130 * TIOCSSET[PN] accept B38400 in these cases to mean "retain current
131 * bit rate."
132 */
133 int sgtytb_handling = 1;
134 static boolean_t sgtytb_complaint;

136 /* don't push drcompat module by default on Style-2 streams */
137 static int push_drcompat = 0;

139 /*
140 * id value used to distinguish between different ioctl messages
141 */
142 static uint32_t ioc_id;

144 static void putback(struct stdata *, queue_t *, mblk_t *, int);
145 static void strcleanall(struct vnode *);
146 static int strwsrv(queue_t *);
147 static int strdocmd(struct stdata *, struct strcmd *, cred_t *);
148 static boolean_t is_xti_str(const struct stdata *);
149 #endif /* ! codereview */

151 /*
152 * qinit and module_info structures for stream head read and write queues
153 */
154 struct module_info strm_info = { 0, "strrhead", 0, INFPSZ, STRHIGH, STRLOW };
155 struct module_info stwm_info = { 0, "strwhead", 0, 0, 0, 0 };
156 struct qinit strdata = { strrput, NULL, NULL, NULL, NULL, &strm_info };
157 struct qinit stwdata = { NULL, strwsrv, NULL, NULL, NULL, &stwm_info };
158 struct module_info fiform_info = { 0, "fifostrhead", 0, PIPE_BUF, FIFOWAT,
159     FIFOWAT };
160 struct module_info fifowm_info = { 0, "fifowstrhead", 0, 0, 0, 0 };
161 struct qinit fifo_strdata = { strrput, NULL, NULL, NULL, NULL, &fiform_info };
162 struct qinit fifo_stwdata = { NULL, strwsrv, NULL, NULL, NULL, &fifowm_info };

164 extern kmutex_t strresources; /* protects global resources */
165 extern kmutex_t muxifier; /* single-threads multiplexor creation */

167 static boolean_t msghasdata(mblk_t *bp);
168 #define msgnodata(bp) (!msghasdata(bp))

170 /*
171 * Stream head locking notes:
172 * There are four monitors associated with the stream head:
173 * 1. v_stream monitor: in stropen() and strclose() v_lock
174 * is held while the association of vnode and stream
175 * head is established or tested for.
176 * 2. open/close/push/pop monitor: sd_lock is held while each
177 * thread bids for exclusive access to this monitor
178 * for opening or closing a stream. In addition, this
179 * monitor is entered during pushes and pops. This
180 * guarantees that during plumbing operations there
181 * is only one thread trying to change the plumbing.
182 * Any other threads present in the stream are only
183 * using the plumbing.
184 * 3. read/write monitor: in the case of read, a thread holds
185 * sd_lock while trying to get data from the stream
186 * head queue. if there is none to fulfill a read
187 * request, it sets RSLEEP and calls cv_wait_sig() down
188 * in strwaitq() to await the arrival of new data.
189 * when new data arrives in strput(), sd_lock is acquired
190 * before testing for RSLEEP and calling cv_broadcast().
191 * the behavior of strwrite(), strwsrv(), and WSLEEP
192 * mirror this.
193 * 4. ioctl monitor: sd_lock is gotten to ensure that only one

```

```

194 * thread is doing an ioctl at a time.
195 */

197 static int
198 push_mod(queue_t *qp, dev_t *devp, struct stdata *stp, const char *name,
199 int anchor, cred_t *crp, uint_t anchor_zoneid)
200 {
201     int error;
202     fmodsw_impl_t *fp;

204     if (stp->sd_flag & (STRHUP|STRDERR|STWRERR)) {
205         error = (stp->sd_flag & STRHUP) ? ENXIO : EIO;
206         return (error);
207     }
208     if (stp->sd_pushcnt >= nstrpush) {
209         return (EINVAL);
210     }

212     if ((fp = fmodsw_find(name, FMODSW_HOLD | FMODSW_LOAD)) == NULL) {
213         stp->sd_flag |= STREOPENFAIL;
214         return (EINVAL);
215     }

217     /*
218      * push new module and call its open routine via qattach
219      */
220     if ((error = qattach(qp, devp, 0, crp, fp, B_FALSE)) != 0)
221         return (error);

223     /*
224      * Check to see if caller wants a STREAMS anchor
225      * put at this place in the stream, and add if so.
226      */
227     mutex_enter(&stp->sd_lock);
228     if (anchor == stp->sd_pushcnt) {
229         stp->sd_anchor = stp->sd_pushcnt;
230         stp->sd_anchorzone = anchor_zoneid;
231     }
232     mutex_exit(&stp->sd_lock);

234     return (0);
235 }

237 /*
238 * Open a stream device.
239 */
240 int
241 stropen(vnode_t *vp, dev_t *devp, int flag, cred_t *crp)
242 {
243     struct stdata *stp;
244     queue_t *qp;
245     int s;
246     dev_t dummydev, savedev;
247     struct autopush *ap;
248     struct dautopush dlap;
249     int error = 0;
250     ssize_t rmin, rmax;
251     int cloneopen;
252     queue_t *brq;
253     major_t major;
254     str_stack_t *ss;
255     zoneid_t zoneid;
256     uint_t anchor;

258     /*
259      * If the stream already exists, wait for any open in progress

```

```

260     * to complete, then call the open function of each module and
261     * driver in the stream. Otherwise create the stream.
262     */
263     TRACE_1(TR_FAC_STREAMS_FR, TR_STROPEN, "stropen:%p", vp);
264     retry:
265     mutex_enter(&vp->v_lock);
266     if ((stp = vp->v_stream) != NULL) {
267
268         /*
269          * Waiting for stream to be created to device
270          * due to another open.
271          */
272         mutex_exit(&vp->v_lock);
273
274         if (STRMATED(stp)) {
275             struct stdata *strmatep = stp->sd_mate;
276
277             STRLOCKMATES(stp);
278             if (strmatep->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
279                 if (flag & (FNDELAY|FNONBLOCK)) {
280                     error = EAGAIN;
281                     mutex_exit(&strmatep->sd_lock);
282                     goto ckreturn;
283                 }
284                 mutex_exit(&stp->sd_lock);
285                 if (!cv_wait_sig(&strmatep->sd_monitor,
286                     &strmatep->sd_lock)) {
287                     error = EINTR;
288                     mutex_exit(&strmatep->sd_lock);
289                     mutex_enter(&stp->sd_lock);
290                     goto ckreturn;
291                 }
292                 mutex_exit(&strmatep->sd_lock);
293                 goto retry;
294             }
295             if (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
296                 if (flag & (FNDELAY|FNONBLOCK)) {
297                     error = EAGAIN;
298                     mutex_exit(&strmatep->sd_lock);
299                     goto ckreturn;
300                 }
301                 mutex_exit(&strmatep->sd_lock);
302                 if (!cv_wait_sig(&stp->sd_monitor,
303                     &stp->sd_lock)) {
304                     error = EINTR;
305                     goto ckreturn;
306                 }
307                 mutex_exit(&stp->sd_lock);
308                 goto retry;
309             }
310
311             if (stp->sd_flag & (STRDERR|STWRERR)) {
312                 error = EIO;
313                 mutex_exit(&strmatep->sd_lock);
314                 goto ckreturn;
315             }
316
317             stp->sd_flag |= STWOPEN;
318             STRUNLOCKMATES(stp);
319         } else {
320             mutex_enter(&stp->sd_lock);
321             if (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
322                 if (flag & (FNDELAY|FNONBLOCK)) {
323                     error = EAGAIN;
324                     goto ckreturn;
325                 }

```

```

326         if (!cv_wait_sig(&stp->sd_monitor,
327             &stp->sd_lock)) {
328             error = EINTR;
329             goto ckreturn;
330         }
331         mutex_exit(&stp->sd_lock);
332         goto retry; /* could be clone! */
333     }
334
335     if (stp->sd_flag & (STRDERR|STWRERR)) {
336         error = EIO;
337         goto ckreturn;
338     }
339
340     stp->sd_flag |= STWOPEN;
341     mutex_exit(&stp->sd_lock);
342 }
343
344 /*
345  * Open all modules and devices down stream to notify
346  * that another user is streaming. For modules, set the
347  * last argument to MODOPEN and do not pass any open flags.
348  * Ignore dummydev since this is not the first open.
349  */
350     claimstr(stp->sd_wrq);
351     qp = stp->sd_wrq;
352     while (_SAMESTR(qp)) {
353         qp = qp->q_next;
354         if ((error = greopen(_RD(qp), devp, flag, crp)) != 0)
355             break;
356     }
357     releasestr(stp->sd_wrq);
358     mutex_enter(&stp->sd_lock);
359     stp->sd_flag &= ~(STRHUP|STWOPEN|STRDERR|STWRERR);
360     stp->sd_rerror = 0;
361     stp->sd_werror = 0;
362     ckreturn:
363     cv_broadcast(&stp->sd_monitor);
364     mutex_exit(&stp->sd_lock);
365     return (error);
366 }
367
368 /*
369  * This vnode isn't streaming. SPECFS already
370  * checked for multiple vnodes pointing to the
371  * same stream, so create a stream to the driver.
372  */
373     qp = allocq();
374     stp = shalloc(qp);
375
376 /*
377  * Initialize stream head. shalloc() has given us
378  * exclusive access, and we have the vnode locked;
379  * we can do whatever we want with stp.
380  */
381     stp->sd_flag = STWOPEN;
382     stp->sd_siglist = NULL;
383     stp->sd_pollist.ph_list = NULL;
384     stp->sd_sigflags = 0;
385     stp->sd_mark = NULL;
386     stp->sd_closetime = STRTIMEOUT;
387     stp->sd_sidp = NULL;
388     stp->sd_pgidp = NULL;
389     stp->sd_vnode = vp;
390     stp->sd_rerror = 0;
391     stp->sd_werror = 0;

```

```

392     stp->sd_wroff = 0;
393     stp->sd_tail = 0;
394     stp->sd_iocblk = NULL;
395     stp->sd_cmdblk = NULL;
396     stp->sd_pushcnt = 0;
397     stp->sd_qn_minpsz = 0;
398     stp->sd_qn_maxpsz = INFPSSZ - 1; /* used to check for initialization */
399     stp->sd_maxblk = INFPSSZ;
400     qp->q_ptr = _WR(qp)->q_ptr = stp;
401     STREAM(qp) = STREAM(_WR(qp)) = stp;
402     vp->v_stream = stp;
403     mutex_exit(&vp->v_lock);

405     /*
406      * If this is not a system process, then add it to
407      * the list associated with the stream head.
408      */
409     if (!(curproc->p_flag & SSYS) && is_xti_str(stp))
410         sh_insert_pid(stp, curproc->p_pidp->pid_id);

412 #endif /* ! codereview */
413     if (vp->v_type == VFIFO) {
414         stp->sd_flag |= OLDNDelay;
415         /*
416          * This means, both for pipes and fifos
417          * strwrite will send SIGPIPE if the other
418          * end is closed. For putmsg it depends
419          * on whether it is a XPG4_2 application
420          * or not
421          */
422         stp->sd_wput_opt = SW_SIGPIPE;

424         /* setq might sleep in kmem_alloc - avoid holding locks. */
425         setq(qp, &fifo_strdata, &fifo_stwdata, NULL, QMTSAFE,
426             SQ_CI|SQ_CO, B_FALSE);

428         set_qend(qp);
429         stp->sd_strtab = fifo_getinfo();
430         _WR(qp)->q_nfsrv = _WR(qp);
431         qp->q_nfsrv = qp;
432         /*
433          * Wake up others that are waiting for stream to be created.
434          */
435         mutex_enter(&stp->sd_lock);
436         /*
437          * nothing is be pushed on stream yet, so
438          * optimized stream head packetsizes are just that
439          * of the read queue
440          */
441         stp->sd_qn_minpsz = qp->q_minpsz;
442         stp->sd_qn_maxpsz = qp->q_maxpsz;
443         stp->sd_flag &= ~STWOPEN;
444         goto fifo_opendone;
445     }
446     /* setq might sleep in kmem_alloc - avoid holding locks. */
447     setq(qp, &strdata, &stwdata, NULL, QMTSAFE, SQ_CI|SQ_CO, B_FALSE);

449     set_qend(qp);

451     /*
452      * Open driver and create stream to it (via qattach).
453      */
454     savedev = *devp;
455     cloneopen = (getmajor(*devp) == clone_major);
456     if ((error = qattach(qp, devp, flag, crp, NULL, B_FALSE)) != 0) {
457         mutex_enter(&vp->v_lock);

```

```

458         vp->v_stream = NULL;
459         mutex_exit(&vp->v_lock);
460         mutex_enter(&stp->sd_lock);
461         cv_broadcast(&stp->sd_monitor);
462         mutex_exit(&stp->sd_lock);
463         freeq(_RD(qp));
464         shfree(stp);
465         return (error);
466     }
467     /*
468      * Set sd_strtab after open in order to handle clonable drivers
469      */
470     stp->sd_strtab = STREAMSTAB(getmajor(*devp));

472     /*
473      * Historical note: dummydev used to be prior to the initial
474      * open (via qattach above), which made the value seen
475      * inconsistent between an I_PUSH and an autopush of a module.
476      */
477     dummydev = *devp;

479     /*
480      * For clone open of old style (Q not associated) network driver,
481      * push DRMODNAME module to handle DL_ATTACH/DL_DETACH
482      */
483     brq = _RD(_WR(qp)->q_next);
484     major = getmajor(*devp);
485     if (push_drcompat && cloneopen && NETWORK_DRV(major) &&
486         ((brq->q_flag & _QASSOCIATED) == 0)) {
487         if (push_mod(qp, &dummydev, stp, DRMODNAME, 0, crp, 0) != 0)
488             cmn_err(CE_WARN, "cannot push " DRMODNAME
489                 " streams module");
490     }

492     if (!NETWORK_DRV(major)) {
493         savedev = *devp;
494     } else {
495         /*
496          * For network devices, process differently based on the
497          * return value from dld_autopush():
498          *
499          * 0: the passed-in device points to a GLDv3 datalink with
500          * per-link autopush configuration; use that configuration
501          * and ignore any per-driver autopush configuration.
502          *
503          * 1: the passed-in device points to a physical GLDv3
504          * datalink without per-link autopush configuration. The
505          * passed in device was changed to refer to the actual
506          * physical device (if it's not already); we use that new
507          * device to look up any per-driver autopush configuration.
508          *
509          * -1: neither of the above cases applied; use the initial
510          * device to look up any per-driver autopush configuration.
511          */
512         switch (dld_autopush(&savedev, &dlap)) {
513             case 0:
514                 zoneid = crgetzoneid(crp);
515                 for (s = 0; s < dlap.dap_npush; s++) {
516                     error = push_mod(qp, &dummydev, stp,
517                         dlap.dap_aplist[s], dlap.dap_anchor, crp,
518                         zoneid);
519                     if (error != 0)
520                         break;
521                 }
522                 goto opendone;
523             case 1:

```

```

524         break;
525     case -1:
526         savedev = *devp;
527         break;
528     }
529 }
530 /*
531  * Find the autopush configuration based on "savedev". Start with the
532  * global zone. If not found check in the local zone.
533  */
534 zoneid = GLOBAL_ZONEID;
535 retryap:
536 ss = netstack_find_by_stackid(zoneid_to_netstackid(zoneid))->
537     netstack_str;
538 if ((ap = sad_ap_find_by_dev(savedev, ss)) == NULL) {
539     netstack_rele(ss->ss_netstack);
540     if (zoneid == GLOBAL_ZONEID) {
541         /*
542          * None found. Also look in the zone's autopush table.
543          */
544         zoneid = crgetzoneid(crp);
545         if (zoneid != GLOBAL_ZONEID)
546             goto retryap;
547     }
548     goto opendone;
549 }
550 anchor = ap->ap_anchor;
551 zoneid = crgetzoneid(crp);
552 for (s = 0; s < ap->ap_npush; s++) {
553     error = push_mod(qp, &dummydev, stp, ap->ap_list[s],
554         anchor, crp, zoneid);
555     if (error != 0)
556         break;
557 }
558 sad_ap_rele(ap, ss);
559 netstack_rele(ss->ss_netstack);

561 opendone:

563 /*
564  * let specfs know that open failed part way through
565  */
566 if (error) {
567     mutex_enter(&stp->sd_lock);
568     stp->sd_flag |= STREOPENFAIL;
569     mutex_exit(&stp->sd_lock);
570 }

572 /*
573  * Wake up others that are waiting for stream to be created.
574  */
575 mutex_enter(&stp->sd_lock);
576 stp->sd_flag &= ~STWOPEN;

578 /*
579  * As a performance concern we are caching the values of
580  * q_minpsz and q_maxpsz of the module below the stream
581  * head in the stream head.
582  */
583 mutex_enter(QLOCK(stp->sd_wrq->q_next));
584 rmin = stp->sd_wrq->q_next->q_minpsz;
585 rmax = stp->sd_wrq->q_next->q_maxpsz;
586 mutex_exit(QLOCK(stp->sd_wrq->q_next));

588 /* do this processing here as a performance concern */
589 if (strmsgsz != 0) {

```

```

590         if (rmax == INFP SZ)
591             rmax = strmsgsz;
592         else
593             rmax = MIN(strmsgsz, rmax);
594     }

596     mutex_enter(QLOCK(stp->sd_wrq));
597     stp->sd_qn_minpsz = rmin;
598     stp->sd_qn_maxpsz = rmax;
599     mutex_exit(QLOCK(stp->sd_wrq));

601 fifo_opendone:
602     cv_broadcast(&stp->sd_monitor);
603     mutex_exit(&stp->sd_lock);
604     return (error);
605 }

607 static int strsink(queue_t *, mblk_t *);
608 static struct qinit deadrend = {
609     strsink, NULL, NULL, NULL, NULL, &strm_info, NULL
610 };
611 static struct qinit deadwend = {
612     NULL, NULL, NULL, NULL, NULL, &stwm_info, NULL
613 };

615 /*
616  * Close a stream.
617  * This is called from closef() on the last close of an open stream.
618  * Strclean() will already have removed the siglist and pollist
619  * information, so all that remains is to remove all multiplexor links
620  * for the stream, pop all the modules (and the driver), and free the
621  * stream structure.
622  */

624 int
625 strclose(struct vnode *vp, int flag, cred_t *crp)
626 {
627     struct stdata *stp;
628     queue_t *qp;
629     int rval;
630     int freestp = 1;
631     queue_t *rmq;

633     TRACE_1(TR_FAC_STREAMS_FR,
634         TR_STRCLOSE, "strclose:%p", vp);
635     ASSERT(vp->v_stream);

637     stp = vp->v_stream;
638     ASSERT(!(stp->sd_flag & STPLEX));
639     qp = stp->sd_wrq;

641     /*
642      * Needed so that strpoll will return non-zero for this fd.
643      * Note that with POLLNOERR STRHUP does still cause POLLHUP.
644      */
645     mutex_enter(&stp->sd_lock);
646     stp->sd_flag |= STRHUP;
647     mutex_exit(&stp->sd_lock);

649     /*
650      * If the registered process or process group did not have an
651      * open instance of this stream then strclean would not be
652      * called. Thus at the time of closing all remaining siglist entries
653      * are removed.
654      */
655     if (stp->sd_siglist != NULL)

```

```

656         strcleanall(vp);
658     ASSERT(stp->sd_siglist == NULL);
659     ASSERT(stp->sd_sigflags == 0);
661     if (STRMATED(stp)) {
662         struct stdata *strmatep = stp->sd_mate;
663         int waited = 1;
665         STRLOCKMATES(stp);
666         while (waited) {
667             waited = 0;
668             while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
669                 mutex_exit(&strmatep->sd_lock);
670                 cv_wait(&stp->sd_monitor, &stp->sd_lock);
671                 mutex_exit(&stp->sd_lock);
672                 STRLOCKMATES(stp);
673                 waited = 1;
674             }
675             while (strmatep->sd_flag &
676                 (STWOPEN|STRCLOSE|STRPLUMB)) {
677                 mutex_exit(&stp->sd_lock);
678                 cv_wait(&strmatep->sd_monitor,
679                     &strmatep->sd_lock);
680                 mutex_exit(&strmatep->sd_lock);
681                 STRLOCKMATES(stp);
682                 waited = 1;
683             }
684         }
685         stp->sd_flag |= STRCLOSE;
686         STRUNLOCKMATES(stp);
687     } else {
688         mutex_enter(&stp->sd_lock);
689         stp->sd_flag |= STRCLOSE;
690         mutex_exit(&stp->sd_lock);
691     }
693     ASSERT(qp->q_first == NULL);    /* No more delayed write */
695     /* Check if an I_LINK was ever done on this stream */
696     if (stp->sd_flag & STRHASLINKS) {
697         netstack_t *ns;
698         str_stack_t *ss;
700         ns = netstack_find_by_cred(crp);
701         ASSERT(ns != NULL);
702         ss = ns->netstack_str;
703         ASSERT(ss != NULL);
705         (void) munlinkall(stp, LINKCLOSE|LINKNORMAL, crp, &rval, ss);
706         netstack_rele(ss->ss_netstack);
707     }
709     while (!_SAMESTR(qp)) {
710         /*
711          * Holding sd_lock prevents q_next from changing in
712          * this stream.
713          */
714         mutex_enter(&stp->sd_lock);
715         if (!(flag & (FNDELAY|FNONBLOCK)) && (stp->sd_closetime > 0)) {
717             /*
718              * sleep until awakened by strwsrv() or timeout
719              */
720             for (;;) {
721                 mutex_enter(QLOCK(qp->q_next));

```

```

722         if (!(qp->q_next->q_mblkcnt)) {
723             mutex_exit(QLOCK(qp->q_next));
724             break;
725         }
726         stp->sd_flag |= WSLEEP;
728         /* ensure strwsrv gets enabled */
729         qp->q_next->q_flag |= QWANTW;
730         mutex_exit(QLOCK(qp->q_next));
731         /* get out if we timed out or recv'd a signal */
732         if (str_cv_wait(&qp->q_wait, &stp->sd_lock,
733             stp->sd_closetime, 0) <= 0) {
734             break;
735         }
736     }
737     stp->sd_flag &= ~WSLEEP;
738 }
739 mutex_exit(&stp->sd_lock);
741     rmq = qp->q_next;
742     if (rmq->q_flag & QISDRV) {
743         ASSERT(!_SAMESTR(rmq));
744         wait_sq_svc(_RD(qp)->q_syncq);
745     }
747     qdetach(_RD(rmq), 1, flag, crp, B_FALSE);
748 }
750 /*
751  * Since we call pollwake up in close() now, the poll list should
752  * be empty in most cases. The only exception is the layered devices
753  * (e.g. the console drivers with redirection modules pushed on top
754  * of it). We have to do this after calling qdetach() because
755  * the redirection module won't have torn down the console
756  * redirection until after qdetach() has been invoked.
757  */
758     if (stp->sd_pollist.ph_list != NULL) {
759         pollwake up(&stp->sd_pollist, POLLERR);
760         pollhead_clean(&stp->sd_pollist);
761     }
762     ASSERT(stp->sd_pollist.ph_list == NULL);
763     ASSERT(stp->sd_sidp == NULL);
764     ASSERT(stp->sd_pgidp == NULL);
766     /* Prevent qenable from re-enabling the stream head queue */
767     disable_svc(_RD(qp));
769     /*
770      * Wait until service procedure of each queue is
771      * run, if QINSERVICE is set.
772      */
773     wait_svc(_RD(qp));
775     /*
776      * Now, flush both queues.
777      */
778     flushq(_RD(qp), FLUSHALL);
779     flushq(qp, FLUSHALL);
781     /*
782      * If the write queue of the stream head is pointing to a
783      * read queue, we have a twisted stream. If the read queue
784      * is alive, convert the stream head queues into a dead end.
785      * If the read queue is dead, free the dead pair.
786      */
787     if (qp->q_next && !_SAMESTR(qp)) {

```

```

788     if (qp->q_next->q_qinfo == &deadrend) { /* half-closed pipe */
789         flushq(qp->q_next, FLUSHALL); /* ensure no message */
790         shfree(qp->q_next->q_stream);
791         freeq(qp->q_next);
792         freeq(_RD(qp));
793     } else if (qp->q_next == _RD(qp)) { /* fifo */
794         freeq(_RD(qp));
795     } else { /* pipe */
796         freestp = 0;
797         /*
798          * The q_info pointers are never accessed when
799          * SLOCK is held.
800          */
801         ASSERT(qp->q_syncq == _RD(qp)->q_syncq);
802         mutex_enter(SLOCK(qp->q_syncq));
803         qp->q_qinfo = &deadrend;
804         _RD(qp)->q_qinfo = &deadrend;
805         mutex_exit(SLOCK(qp->q_syncq));
806     }
807 } else {
808     freeq(_RD(qp)); /* free stream head queue pair */
809 }

811 mutex_enter(&vp->v_lock);
812 if (stp->sd_iocblk) {
813     if (stp->sd_iocblk != (mblk_t *)-1) {
814         freemsg(stp->sd_iocblk);
815     }
816     stp->sd_iocblk = NULL;
817 }
818 stp->sd_vnode = NULL;
819 vp->v_stream = NULL;
820 mutex_exit(&vp->v_lock);
821 mutex_enter(&stp->sd_lock);
822 freemsg(stp->sd_cmdblk);
823 stp->sd_cmdblk = NULL;
824 stp->sd_flag &= ~STRCLOSE;
825 cv_broadcast(&stp->sd_monitor);
826 mutex_exit(&stp->sd_lock);

828 if (freestp)
829     shfree(stp);
830 return (0);
831 }

833 static int
834 strsink(queue_t *q, mblk_t *bp)
835 {
836     struct copyresp *resp;

838     switch (bp->b_datap->db_type) {
839     case M_FLUSH:
840         if ((*bp->b_rptr & FLUSHW) && !(bp->b_flag & MSGNOLOOP)) {
841             *bp->b_rptr &= ~FLUSHR;
842             bp->b_flag |= MSGNOLOOP;
843             /*
844              * Protect against the driver passing up
845              * messages after it has done a qprocsoff.
846              */
847             if (_OTHERQ(q)->q_next == NULL)
848                 freemsg(bp);
849             else
850                 qreply(q, bp);
851         } else {
852             freemsg(bp);
853         }

```

```

854         break;

856     case M_COPYIN:
857     case M_COPYOUT:
858         if (bp->b_cont) {
859             freemsg(bp->b_cont);
860             bp->b_cont = NULL;
861         }
862         bp->b_datap->db_type = M_IOCTLDATA;
863         bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
864         resp = (struct copyresp *)bp->b_rptr;
865         resp->cp_rval = (caddr_t)1; /* failure */
866         /*
867          * Protect against the driver passing up
868          * messages after it has done a qprocsoff.
869          */
870         if (_OTHERQ(q)->q_next == NULL)
871             freemsg(bp);
872         else
873             qreply(q, bp);
874         break;

876     case M_IOCTL:
877         if (bp->b_cont) {
878             freemsg(bp->b_cont);
879             bp->b_cont = NULL;
880         }
881         bp->b_datap->db_type = M_IOCNAK;
882         /*
883          * Protect against the driver passing up
884          * messages after it has done a qprocsoff.
885          */
886         if (_OTHERQ(q)->q_next == NULL)
887             freemsg(bp);
888         else
889             qreply(q, bp);
890         break;

892     default:
893         freemsg(bp);
894         break;
895     }

897     return (0);
898 }

900 /*
901  * Clean up after a process when it closes a stream. This is called
902  * from closef for all closes, whereas strclose is called only for the
903  * last close on a stream. The siglist is scanned for entries for the
904  * current process, and these are removed.
905  */
906 void
907 strclean(struct vnode *vp)
908 {
909     strsig_t *ssp, *pssp, *tssp;
910     stdata_t *stp;
911     int update = 0;

913     TRACE_1(TR_FAC_STREAMS_FR,
914            TR_STRCLEAN, "strclean:%p", vp);
915     stp = vp->v_stream;
916     pssp = NULL;
917     mutex_enter(&stp->sd_lock);
918     ssp = stp->sd_siglist;
919     while (ssp) {

```

```

920         if (ssp->ss_pidp == curproc->p_pidp) {
921             tssp = ssp->ss_next;
922             if (pssp)
923                 pssp->ss_next = tssp;
924             else
925                 stp->sd_siglist = tssp;
926             mutex_enter(&pidlock);
927             PID_RELE(ssp->ss_pidp);
928             mutex_exit(&pidlock);
929             kmem_free(ssp, sizeof (strsig_t));
930             update = 1;
931             ssp = tssp;
932         } else {
933             pssp = ssp;
934             ssp = ssp->ss_next;
935         }
936     }
937     if (update) {
938         stp->sd_sigflags = 0;
939         for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
940             stp->sd_sigflags |= ssp->ss_events;
941     }
942     mutex_exit(&stp->sd_lock);
943 }

945 /*
946  * Used on the last close to remove any remaining items on the siglist.
947  * These could be present on the siglist due to I_ESETSIG calls that
948  * use process groups or processed that do not have an open file descriptor
949  * for this stream (Such entries would not be removed by strclean).
950  */
951 static void
952 strcleanall(struct vnode *vp)
953 {
954     strsig_t *ssp, *nssp;
955     stdata_t *stp;

957     stp = vp->v_stream;
958     mutex_enter(&stp->sd_lock);
959     ssp = stp->sd_siglist;
960     stp->sd_siglist = NULL;
961     while (ssp) {
962         nssp = ssp->ss_next;
963         mutex_enter(&pidlock);
964         PID_RELE(ssp->ss_pidp);
965         mutex_exit(&pidlock);
966         kmem_free(ssp, sizeof (strsig_t));
967         ssp = nssp;
968     }
969     stp->sd_sigflags = 0;
970     mutex_exit(&stp->sd_lock);
971 }

973 /*
974  * Retrieve the next message from the logical stream head read queue
975  * using either rwnext (if sync stream) or getq_noenab.
976  * It is the callers responsibility to call qbackenable after
977  * it is finished with the message. The caller should not call
978  * qbackenable until after any putback calls to avoid spurious backenabling.
979  */
980 mblk_t *
981 strget(struct stdata *stp, queue_t *q, struct uio *uiop, int first,
982        int *errorp)
983 {
984     mblk_t *bp;
985     int error;

```

```

986     ssize_t rbytes = 0;

988     /* Holding sd_lock prevents the read queue from changing */
989     ASSERT(MUTEX_HELD(&stp->sd_lock));

991     if (uiop != NULL && stp->sd_struioirdq != NULL &&
992         q->q_first == NULL &&
993         (!first || (stp->sd_wakeq & RSLEEP))) {
994         /*
995          * Stream supports rwnext() for the read side.
996          * If this is the first time we're called by e.g. streadd
997          * only do the downcall if there is a deferred wakeup
998          * (registered in sd_wakeq).
999          */
1000        struioird_t uioid;

1002        if (first)
1003            stp->sd_wakeq &= ~RSLEEP;

1005        (void) uioidup(uiop, &uioid.d_uio, uioid.d_iov,
1006                    sizeof (uioid.d_iov) / sizeof (*uioid.d_iov));
1007        uioid.d_mp = 0;
1008        /*
1009         * Mark that a thread is in rwnext on the read side
1010         * to prevent strput from nacking ioctls immediately.
1011         * When the last concurrent rwnext returns
1012         * the ioctls are nack'ed.
1013         */
1014        ASSERT(MUTEX_HELD(&stp->sd_lock));
1015        stp->sd_struiodnak++;
1016        /*
1017         * Note: rwnext will drop sd_lock.
1018         */
1019        error = rwnext(q, &uioid);
1020        ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));
1021        mutex_enter(&stp->sd_lock);
1022        stp->sd_struiodnak--;
1023        while (stp->sd_struiodnak == 0 &&
1024            ((bp = stp->sd_struionak) != NULL)) {
1025            stp->sd_struionak = bp->b_next;
1026            bp->b_next = NULL;
1027            bp->b_datap->db_type = M_IOCNAK;
1028            /*
1029             * Protect against the driver passing up
1030             * messages after it has done a qprocsoff.
1031             */
1032            if (_OTHERQ(q)->q_next == NULL)
1033                freemsg(bp);
1034            else {
1035                mutex_exit(&stp->sd_lock);
1036                qreply(q, bp);
1037                mutex_enter(&stp->sd_lock);
1038            }
1039        }
1040        ASSERT(MUTEX_HELD(&stp->sd_lock));
1041        if (error == 0 || error == EWOULDBLOCK) {
1042            if ((bp = uioid.d_mp) != NULL) {
1043                *errorp = 0;
1044                ASSERT(MUTEX_HELD(&stp->sd_lock));
1045                return (bp);
1046            }
1047            error = 0;
1048        } else if (error == EINVAL) {
1049            /*
1050             * The stream plumbing must have
1051             * changed while we were away, so

```

```

1052         * just turn off rwnext().
1053         */
1054         error = 0;
1055     } else if (error == EBUSY) {
1056         /*
1057          * The module might have data in transit using putnext
1058          * Fall back on waiting + getq.
1059          */
1060         error = 0;
1061     } else {
1062         *errorp = error;
1063         ASSERT(MUTEX_HELD(&stp->sd_lock));
1064         return (NULL);
1065     }
1066     /*
1067     * Try a getq in case a rwnext() generated mblk
1068     * has bubbled up via strrput().
1069     */
1070 }
1071 *errorp = 0;
1072 ASSERT(MUTEX_HELD(&stp->sd_lock));

1074 /*
1075 * If we have a valid uiop, try and use this as a guide for how
1076 * many bytes to retrieve from the queue via getq_noenab().
1077 * Doing this can avoid unnecessary counting of overlong
1078 * messages in putback(). We currently only do this for sockets
1079 * and only if there is no sd_rputdatafunc hook.
1080 *
1081 * The sd_rputdatafunc hook transforms the entire message
1082 * before any bytes in it can be given to a client. So, rbytes
1083 * must be 0 if there is a hook.
1084 */
1085 if ((uiop != NULL) && (stp->sd_vnode->v_type == VSOCK) &&
1086     (stp->sd_rputdatafunc == NULL))
1087     rbytes = uiop->uio_resid;

1089     return (getq_noenab(q, rbytes));
1090 }

1092 /*
1093 * Copy out the message pointed to by 'bp' into the uio pointed to by 'uiop'.
1094 * If the message does not fit in the uio the remainder of it is returned;
1095 * otherwise NULL is returned. Any embedded zero-length mblk_t's are
1096 * consumed, even if uio_resid reaches zero. On error, '*errorp' is set to
1097 * the error code, the message is consumed, and NULL is returned.
1098 */
1099 static mblk_t *
1100 struiocopyout(mblk_t *bp, struct uio *uiop, int *errorp)
1101 {
1102     int error;
1103     ptrdiff_t n;
1104     mblk_t *nbp;

1106     ASSERT(bp->b_wptr >= bp->b_rptr);

1108     do {
1109         if ((n = MIN(uiop->uio_resid, MBLKL(bp))) != 0) {
1110             ASSERT(n > 0);

1112             error = uiomove(bp->b_rptr, n, UIO_READ, uiop);
1113             if (error != 0) {
1114                 freemsg(bp);
1115                 *errorp = error;
1116                 return (NULL);
1117             }

```

```

1118     }
1120     bp->b_rptr += n;
1121     while (bp != NULL && (bp->b_rptr >= bp->b_wptr)) {
1122         nbp = bp;
1123         bp = bp->b_cont;
1124         freeb(nbp);
1125     }
1126     } while (bp != NULL && uiop->uio_resid > 0);

1128     *errorp = 0;
1129     return (bp);
1130 }

1132 /*
1133 * Read a stream according to the mode flags in sd_flag:
1134 *
1135 * (default mode)           - Byte stream, msg boundaries are ignored
1136 * RD_MSGDIS (msg discard) - Read on msg boundaries and throw away
1137 *                          any data remaining in msg
1138 * RD_MSGNODIS (msg non-discard) - Read on msg boundaries and put back
1139 *                          any remaining data on head of read queue
1140 *
1141 * Consume readable messages on the front of the queue until
1142 * *ttolwp(curthread)->lwp_count
1143 * is satisfied, the readable messages are exhausted, or a message
1144 * boundary is reached in a message mode. If no data was read and
1145 * the stream was not opened with the NDELAY flag, block until data arrives.
1146 * Otherwise return the data read and update the count.
1147 *
1148 * In default mode a 0 length message signifies end-of-file and terminates
1149 * a read in progress. The 0 length message is removed from the queue
1150 * only if it is the only message read (no data is read).
1151 *
1152 * An attempt to read an M_PROTO or M_PCPROTO message results in an
1153 * EBADMSG error return, unless either RD_PROTDAT or RD_PROTDIS are set.
1154 * If RD_PROTDAT is set, M_PROTO and M_PCPROTO messages are read as data.
1155 * If RD_PROTDIS is set, the M_PROTO and M_PCPROTO parts of the message
1156 * are unlinked from and M_DATA blocks in the message, the protos are
1157 * thrown away, and the data is read.
1158 */
1159 /* ARGSUSED */
1160 int
1161 stread(struct vnode *vp, struct uio *uiop, cred_t *crp)
1162 {
1163     struct stdata *stp;
1164     mblk_t *bp, *nbp;
1165     queue_t *q;
1166     int error = 0;
1167     uint_t old_sd_flag;
1168     int first;
1169     char rflg;
1170     uint_t mark;
1171     #define _LASTMARK 0x8000 /* Contains MSG*MARK and _LASTMARK */
1172     /* Distinct from MSG*MARK */
1173     short delim;
1174     unsigned char pri = 0;
1175     char waitflag;
1176     unsigned char type;

1177     TRACE_1(TR_FAC_STREAMS_FR,
1178            TR_STRREAD_ENTER, "stread:%p", vp);
1179     ASSERT(vp->v_stream);
1180     stp = vp->v_stream;

1182     mutex_enter(&stp->sd_lock);

```

```

1184     if ((error = i_straccess(stp, JCREAD)) != 0) {
1185         mutex_exit(&stp->sd_lock);
1186         return (error);
1187     }

1189     if (stp->sd_flag & (STRDERR|STPLEX)) {
1190         error = strgeterr(stp, STRDERR|STPLEX, 0);
1191         if (error != 0) {
1192             mutex_exit(&stp->sd_lock);
1193             return (error);
1194         }
1195     }

1197     /*
1198      * Loop terminates when uiop->uio_resid == 0.
1199      */
1200     rflg = 0;
1201     waitflag = READWAIT;
1202     q = _RD(stp->sd_wrq);
1203     for (;;) {
1204         ASSERT(MUTEX_HELD(&stp->sd_lock));
1205         old_sd_flag = stp->sd_flag;
1206         mark = 0;
1207         delim = 0;
1208         first = 1;
1209         while ((bp = strget(stp, q, uiop, first, &error)) == NULL) {
1210             int done = 0;

1212             ASSERT(MUTEX_HELD(&stp->sd_lock));

1214             if (error != 0)
1215                 goto oops;

1217             if (stp->sd_flag & (STRHUP|STREOF)) {
1218                 goto oops;
1219             }
1220             if (rflg && !(stp->sd_flag & STRDELIM)) {
1221                 goto oops;
1222             }
1223             /*
1224              * If a read(fd,buf,0) has been done, there is no
1225              * need to sleep. We always have zero bytes to
1226              * return.
1227              */
1228             if (uiop->uio_resid == 0) {
1229                 goto oops;
1230             }
1232             qbackenable(q, 0);

1234             TRACE_3(TR_FAC_STREAMS_FR, TR_STREAD_WAIT,
1235                  "stread calls strwaitq:%p, %p, %p",
1236                  vp, uiop, crp);
1237             if ((error = strwaitq(stp, waitflag, uiop->uio_resid,
1238                  uiop->uio_fmode, -1, &done)) != 0 || done) {
1239                 TRACE_3(TR_FAC_STREAMS_FR, TR_STREAD_DONE,
1240                      "stread error or done:%p, %p, %p",
1241                      vp, uiop, crp);
1242                 if ((uiop->uio_fmode & FNDELAY) &&
1243                     (stp->sd_flag & OLDNDELAY) &&
1244                     (error == EAGAIN))
1245                     error = 0;
1246                 goto oops;
1247             }
1248             TRACE_3(TR_FAC_STREAMS_FR, TR_STREAD_AWAKE,
1249                  "stread awakes:%p, %p, %p", vp, uiop, crp);

```

```

1250         if ((error = i_straccess(stp, JCREAD)) != 0) {
1251             goto oops;
1252         }
1253         first = 0;
1254     }

1256     ASSERT(MUTEX_HELD(&stp->sd_lock));
1257     ASSERT(bp);
1258     pri = bp->b_band;
1259     /*
1260      * Extract any mark information. If the message is not
1261      * completely consumed this information will be put in the mblk
1262      * that is putback.
1263      * If MSGMARKNEXT is set and the message is completely consumed
1264      * the STRATMARK flag will be set below. Likewise, if
1265      * MSGNOTMARKNEXT is set and the message is
1266      * completely consumed STRNOTATMARK will be set.
1267      *
1268      * For some unknown reason strread only breaks the read at the
1269      * last mark.
1270      */
1271     mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
1272     ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
1273            (MSGMARKNEXT|MSGNOTMARKNEXT));
1274     if (mark != 0 && bp == stp->sd_mark) {
1275         if (rflg) {
1276             putback(stp, q, bp, pri);
1277             goto oops;
1278         }
1279         mark |= _LASTMARK;
1280         stp->sd_mark = NULL;
1281     }
1282     if ((stp->sd_flag & STRDELIM) && (bp->b_flag & MSGDELIM))
1283         delim = 1;
1284     mutex_exit(&stp->sd_lock);

1286     if (STREAM_NEEDSERVICE(stp))
1287         stream_runservice(stp);

1289     type = bp->b_datap->db_type;

1291     switch (type) {

1293     case M_DATA:
1294         ismdata:
1295             if (msgnodata(bp)) {
1296                 if (mark || delim) {
1297                     freemsg(bp);
1298                 } else if (rflg) {
1300                     /*
1301                      * If already read data put zero
1302                      * length message back on queue else
1303                      * free msg and return 0.
1304                      */
1305                     bp->b_band = pri;
1306                     mutex_enter(&stp->sd_lock);
1307                     putback(stp, q, bp, pri);
1308                     mutex_exit(&stp->sd_lock);
1309                 } else {
1310                     freemsg(bp);
1311                 }
1312             }
1313             error = 0;
1314             goto oops1;

```

```

1316         rflg = 1;
1317         waitflag |= NOINTR;
1318         bp = struiocopyout(bp, uiop, &error);
1319         if (error != 0)
1320             goto oops1;
1321
1322         mutex_enter(&stp->sd_lock);
1323         if (bp) {
1324             /*
1325              * Have remaining data in message.
1326              * Free msg if in discard mode.
1327              */
1328             if (stp->sd_read_opt & RD_MSGDIS) {
1329                 freemsg(bp);
1330             } else {
1331                 bp->b_band = pri;
1332                 if ((mark & _LASTMARK) &&
1333                     (stp->sd_mark == NULL))
1334                     stp->sd_mark = bp;
1335                 bp->b_flag |= mark & ~_LASTMARK;
1336                 if (delim)
1337                     bp->b_flag |= MSGDELIM;
1338                 if (msgnodata(bp))
1339                     freemsg(bp);
1340                 else
1341                     putback(stp, q, bp, pri);
1342             }
1343         } else {
1344             /*
1345              * Consumed the complete message.
1346              * Move the MSG*MARKNEXT information
1347              * to the stream head just in case
1348              * the read queue becomes empty.
1349              *
1350              * If the stream head was at the mark
1351              * (STRATMARK) before we dropped sd_lock above
1352              * and some data was consumed then we have
1353              * moved past the mark thus STRATMARK is
1354              * cleared. However, if a message arrived in
1355              * strrrput during the copyout above causing
1356              * STRATMARK to be set we can not clear that
1357              * flag.
1358              */
1359             if (mark &
1360                 (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
1361                 if (mark & MSGMARKNEXT) {
1362                     stp->sd_flag &= ~STRNOTATMARK;
1363                     stp->sd_flag |= STRATMARK;
1364                 } else if (mark & MSGNOTMARKNEXT) {
1365                     stp->sd_flag &= ~STRATMARK;
1366                     stp->sd_flag |= STRNOTATMARK;
1367                 } else {
1368                     stp->sd_flag &=
1369                         ~(STRATMARK|STRNOTATMARK);
1370                 }
1371             } else if (rflg && (old_sd_flag & STRATMARK)) {
1372                 stp->sd_flag &= ~STRATMARK;
1373             }
1374         }
1375
1376         /*
1377          * Check for signal messages at the front of the read
1378          * queue and generate the signal(s) if appropriate.
1379          * The only signal that can be on queue is M_SIG at
1380          * this point.
1381          */

```

```

1382         while (((bp = q->q_first) != NULL) &&
1383             (bp->b_datap->db_type == M_SIG)) {
1384             bp = getq_noenab(q, 0);
1385             /*
1386              * sd_lock is held so the content of the
1387              * read queue can not change.
1388              */
1389             ASSERT(bp != NULL && DB_TYPE(bp) == M_SIG);
1390             strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
1391             mutex_exit(&stp->sd_lock);
1392             freemsg(bp);
1393             if (STREAM_NEEDSERVICE(stp))
1394                 stream_runservice(stp);
1395             mutex_enter(&stp->sd_lock);
1396         }
1397
1398         if ((uiop->uio_resid == 0) || (mark & _LASTMARK) ||
1399             delim ||
1400             (stp->sd_read_opt & (RD_MSGDIS|RD_MSGNODIS))) {
1401             goto oops;
1402         }
1403         continue;
1404
1405     case M_SIG:
1406         strsignal(stp, *bp->b_rptr, (int32_t)bp->b_band);
1407         freemsg(bp);
1408         mutex_enter(&stp->sd_lock);
1409         continue;
1410
1411     case M_PROTO:
1412     case M_PCPROTO:
1413         /*
1414          * Only data messages are readable.
1415          * Any others generate an error, unless
1416          * RD_PROTDIS or RD_PROTDAT is set.
1417          */
1418         if (stp->sd_read_opt & RD_PROTDAT) {
1419             for (nbp = bp; nbp; nbp = nbp->b_next) {
1420                 if ((nbp->b_datap->db_type ==
1421                     M_PROTO) ||
1422                     (nbp->b_datap->db_type ==
1423                     M_PCPROTO)) {
1424                     nbp->b_datap->db_type = M_DATA;
1425                 } else {
1426                     break;
1427                 }
1428             }
1429             /*
1430              * clear stream head hi pri flag based on
1431              * first message
1432              */
1433             if (type == M_PCPROTO) {
1434                 mutex_enter(&stp->sd_lock);
1435                 stp->sd_flag &= ~STRPRI;
1436                 mutex_exit(&stp->sd_lock);
1437             }
1438             goto ismdata;
1439         } else if (stp->sd_read_opt & RD_PROTDIS) {
1440             /*
1441              * discard non-data messages
1442              */
1443             while (bp &&
1444                 ((bp->b_datap->db_type == M_PROTO) ||
1445                  (bp->b_datap->db_type == M_PCPROTO))) {
1446                 nbp = unlinkb(bp);
1447                 freeb(bp);

```

```

1448         bp = nbp;
1449     }
1450     /*
1451     * clear stream head hi pri flag based on
1452     * first message
1453     */
1454     if (type == M_PCPROTO) {
1455         mutex_enter(&stp->sd_lock);
1456         stp->sd_flag &= ~STRPRI;
1457         mutex_exit(&stp->sd_lock);
1458     }
1459     if (bp) {
1460         bp->b_band = pri;
1461         goto ismdata;
1462     } else {
1463         break;
1464     }
1465 }
1466 /* FALLTHRU */
1467 case M_PASSFP:
1468     if ((bp->b_datap->db_type == M_PASSFP) &&
1469         (stp->sd_read_opt & RD_PROTDIS)) {
1470         freemsg(bp);
1471         break;
1472     }
1473     mutex_enter(&stp->sd_lock);
1474     putback(stp, q, bp, pri);
1475     mutex_exit(&stp->sd_lock);
1476     if (rflg == 0)
1477         error = EBADMSG;
1478     goto oops1;
1479
1480 default:
1481     /*
1482     * Garbage on stream head read queue.
1483     */
1484     cmn_err(CE_WARN, "bad %x found at stream head\n",
1485            bp->b_datap->db_type);
1486     freemsg(bp);
1487     goto oops1;
1488 }
1489     mutex_enter(&stp->sd_lock);
1490 }
1491 oops:
1492     mutex_exit(&stp->sd_lock);
1493 oops1:
1494     qbackenable(q, pri);
1495     return (error);
1496 #undef _LASTMARK
1497 }
1498
1499 /*
1500 * Default processing of M_PROTO/M_PCPROTO messages.
1501 * Determine which wakeups and signals are needed.
1502 * This can be replaced by a user-specified procedure for kernel users
1503 * of STREAMS.
1504 */
1505 /* ARGSUSED */
1506 mblk_t *
1507 strrput_proto(vnode_t *vp, mblk_t *mp,
1508             strwakeupt_t *wakeups, strsigset_t *firstmsgsig,
1509             strsigset_t *allmsgsig, strpollset_t *pollwakeups)
1510 {
1511     *wakeups = RSLEEP;
1512     *allmsgsig = 0;

```

```

1514     switch (mp->b_datap->db_type) {
1515     case M_PROTO:
1516         if (mp->b_band == 0) {
1517             *firstmsgsig = S_INPUT | S_RDNORM;
1518             *pollwakeups = POLLIN | POLLRDNORM;
1519         } else {
1520             *firstmsgsig = S_INPUT | S_RDBAND;
1521             *pollwakeups = POLLIN | POLLRDBAND;
1522         }
1523         break;
1524     case M_PCPROTO:
1525         *firstmsgsig = S_HIPRI;
1526         *pollwakeups = POLLPRI;
1527         break;
1528     }
1529     return (mp);
1530 }
1531
1532 /*
1533 * Default processing of everything but M_DATA, M_PROTO, M_PCPROTO and
1534 * M_PASSFP messages.
1535 * Determine which wakeups and signals are needed.
1536 * This can be replaced by a user-specified procedure for kernel users
1537 * of STREAMS.
1538 */
1539 /* ARGSUSED */
1540 mblk_t *
1541 strrput_misc(vnode_t *vp, mblk_t *mp,
1542            strwakeupt_t *wakeups, strsigset_t *firstmsgsig,
1543            strsigset_t *allmsgsig, strpollset_t *pollwakeups)
1544 {
1545     *wakeups = 0;
1546     *firstmsgsig = 0;
1547     *allmsgsig = 0;
1548     *pollwakeups = 0;
1549     return (mp);
1550 }
1551
1552 /*
1553 * Stream read put procedure. Called from downstream driver/module
1554 * with messages for the stream head. Data, protocol, and in-stream
1555 * signal messages are placed on the queue, others are handled directly.
1556 */
1557 int
1558 strrput(queue_t *q, mblk_t *bp)
1559 {
1560     struct stdata *stp;
1561     ulong_t rput_opt;
1562     strwakeupt_t wakeups;
1563     strsigset_t firstmsgsig; /* Signals if first message on queue */
1564     strsigset_t allmsgsig; /* Signals for all messages */
1565     strsigset_t signals; /* Signals events to generate */
1566     strpollset_t pollwakeups;
1567     mblk_t *nextbp;
1568     uchar_t band = 0;
1569     int hipri_sig;
1570
1571     stp = (struct stdata *)q->q_ptr;
1572     /*
1573     * Use rput_opt for optimized access to the SR_flags except
1574     * SR_POLLIN. That flag has to be checked under sd_lock since it
1575     * is modified by strpoll().
1576     */
1577     rput_opt = stp->sd_rput_opt;
1578
1579     ASSERT(qclaimed(q));

```

```

1580 TRACE_2(TR_FAC_STREAMS_FR, TR_STRRPUT_ENTER,
1581 "strrput called with message type:q %p bp %p", q, bp);
1583 /*
1584  * Perform initial processing and pass to the parameterized functions.
1585  */
1586 ASSERT(bp->b_next == NULL);
1588 switch (bp->b_datap->db_type) {
1589 case M_DATA:
1590     /*
1591      * sockfs is the only consumer of STREOF and when it is set,
1592      * it implies that the receiver is not interested in receiving
1593      * any more data, hence the mblk is freed to prevent unnecessary
1594      * message queueing at the stream head.
1595      */
1596     if (stp->sd_flag == STREOF) {
1597         freemsg(bp);
1598         return (0);
1599     }
1600     if ((rput_opt & SR_IGN_ZEROLEN) &&
1601         bp->b_rptr == bp->b_wptr && msgnodata(bp)) {
1602         /*
1603          * Ignore zero-length M_DATA messages. These might be
1604          * generated by some transports.
1605          * The zero-length M_DATA messages, even if they
1606          * are ignored, should effect the atmark tracking and
1607          * should wake up a thread sleeping in strwaitmark.
1608          */
1609         mutex_enter(&stp->sd_lock);
1610         if (bp->b_flag & MSGMARKNEXT) {
1611             /*
1612              * Record the position of the mark either
1613              * in q_last or in STRATMARK.
1614              */
1615             if (q->q_last != NULL) {
1616                 q->q_last->b_flag &= ~MSGNOTMARKNEXT;
1617                 q->q_last->b_flag |= MSGMARKNEXT;
1618             } else {
1619                 stp->sd_flag &= ~STRNOTATMARK;
1620                 stp->sd_flag |= STRATMARK;
1621             }
1622         } else if (bp->b_flag & MSGNOTMARKNEXT) {
1623             /*
1624              * Record that this is not the position of
1625              * the mark either in q_last or in
1626              * STRNOTATMARK.
1627              */
1628             if (q->q_last != NULL) {
1629                 q->q_last->b_flag &= ~MSGMARKNEXT;
1630                 q->q_last->b_flag |= MSGNOTMARKNEXT;
1631             } else {
1632                 stp->sd_flag &= ~STRATMARK;
1633                 stp->sd_flag |= STRNOTATMARK;
1634             }
1635         }
1636         if (stp->sd_flag & RSLEEP) {
1637             stp->sd_flag &= ~RSLEEP;
1638             cv_broadcast(&q->q_wait);
1639         }
1640         mutex_exit(&stp->sd_lock);
1641         freemsg(bp);
1642         return (0);
1643     }
1644     wakeups = RSLEEP;
1645     if (bp->b_band == 0) {

```

```

1646         firstmsgsig = S_INPUT | S_RDNORM;
1647         pollwakeups = POLLIN | POLLRDNORM;
1648     } else {
1649         firstmsgsig = S_INPUT | S_RDBAND;
1650         pollwakeups = POLLIN | POLLRDBAND;
1651     }
1652     if (rput_opt & SR_SIGALLDATA)
1653         allmsgsig = firstmsgsig;
1654     else
1655         allmsgsig = 0;
1657     mutex_enter(&stp->sd_lock);
1658     if ((rput_opt & SR_CONSOL_DATA) &&
1659         (q->q_last != NULL) &&
1660         (bp->b_flag & (MSGMARK|MSGDELIM)) == 0) {
1661         /*
1662          * Consolidate an M_DATA message onto an M_DATA,
1663          * M_PROTO, or M_PCPROTO by merging it with q_last.
1664          * The consolidation does not take place if
1665          * the old message is marked with either of the
1666          * marks or the delim flag or if the new
1667          * message is marked with MSGMARK. The MSGMARK
1668          * check is needed to handle the odd semantics of
1669          * MSGMARK where essentially the whole message
1670          * is to be treated as marked.
1671          * Carry any MSGMARKNEXT and MSGNOTMARKNEXT from the
1672          * new message to the front of the b_cont chain.
1673          */
1674         mblk_t *lbp = q->q_last;
1675         unsigned char db_type = lbp->b_datap->db_type;
1677         if ((db_type == M_DATA || db_type == M_PROTO ||
1678             db_type == M_PCPROTO) &&
1679             !(lbp->b_flag & (MSGDELIM|MSGMARK|MSGMARKNEXT))) {
1680             rmvq_noenab(q, lbp);
1681             /*
1682              * The first message in the b_cont list
1683              * tracks MSGMARKNEXT and MSGNOTMARKNEXT.
1684              * We need to handle the case where we
1685              * are appending:
1686              *
1687              * 1) a MSGMARKNEXT to a MSGNOTMARKNEXT.
1688              * 2) a MSGMARKNEXT to a plain message.
1689              * 3) a MSGNOTMARKNEXT to a plain message
1690              * 4) a MSGNOTMARKNEXT to a MSGNOTMARKNEXT
1691              * message.
1692              *
1693              * Thus we never append a MSGMARKNEXT or
1694              * MSGNOTMARKNEXT to a MSGMARKNEXT message.
1695              */
1696             if (bp->b_flag & MSGMARKNEXT) {
1697                 lbp->b_flag |= MSGMARKNEXT;
1698                 lbp->b_flag &= ~MSGNOTMARKNEXT;
1699                 bp->b_flag &= ~MSGMARKNEXT;
1700             } else if (bp->b_flag & MSGNOTMARKNEXT) {
1701                 lbp->b_flag |= MSGNOTMARKNEXT;
1702                 bp->b_flag &= ~MSGNOTMARKNEXT;
1703             }
1705             linkb(lbp, bp);
1706             bp = lbp;
1707             /*
1708              * The new message logically isn't the first
1709              * even though the q_first check below thinks
1710              * it is. Clear the firstmsgsig to make it
1711              * not appear to be first.

```

```

1712         */
1713         firstmsgsig = 0;
1714     }
1715 }
1716 break;
1717
1718 case M_PASSFP:
1719     wakeups = RSLEEP;
1720     allmsgsig = 0;
1721     if (bp->b_band == 0) {
1722         firstmsgsig = S_INPUT | S_RDNORM;
1723         pollwakeups = POLLIN | POLLRDNORM;
1724     } else {
1725         firstmsgsig = S_INPUT | S_RDBAND;
1726         pollwakeups = POLLIN | POLLRDBAND;
1727     }
1728     mutex_enter(&stp->sd_lock);
1729     break;
1730
1731 case M_PROTO:
1732 case M_PCPROTO:
1733     ASSERT(stp->sd_rprotofunc != NULL);
1734     bp = (stp->sd_rprotofunc)(stp->sd_vnode, bp,
1735         &wakeups, &firstmsgsig, &allmsgsig, &pollwakeups);
1736 #define ALLSIG (S_INPUT|S_HIPRI|S_OUTPUT|S_MSG|S_ERROR|S_HANGUP|S_RDNORM|\
1737 S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)
1738 #define ALLPOLL (POLLIN|POLLPRI|POLLOUT|POLLRDNORM|POLLWRNORM|POLLRDBAND|\
1739 POLLWRBAND)
1740
1741     ASSERT((wakeups & ~(RSLEEP|WSLEEP)) == 0);
1742     ASSERT((firstmsgsig & ~ALLSIG) == 0);
1743     ASSERT((allmsgsig & ~ALLSIG) == 0);
1744     ASSERT((pollwakeups & ~ALLPOLL) == 0);
1745
1746     mutex_enter(&stp->sd_lock);
1747     break;
1748
1749 default:
1750     ASSERT(stp->sd_rmiscfunc != NULL);
1751     bp = (stp->sd_rmiscfunc)(stp->sd_vnode, bp,
1752         &wakeups, &firstmsgsig, &allmsgsig, &pollwakeups);
1753     ASSERT((wakeups & ~(RSLEEP|WSLEEP)) == 0);
1754     ASSERT((firstmsgsig & ~ALLSIG) == 0);
1755     ASSERT((allmsgsig & ~ALLSIG) == 0);
1756     ASSERT((pollwakeups & ~ALLPOLL) == 0);
1757 #undef ALLSIG
1758 #undef ALLPOLL
1759     mutex_enter(&stp->sd_lock);
1760     break;
1761 }
1762 ASSERT(MUTEX_HELD(&stp->sd_lock));
1763
1764 /* By default generate superset of signals */
1765 signals = (firstmsgsig | allmsgsig);
1766
1767 /*
1768 * The proto and misc functions can return multiple messages
1769 * as a b_next chain. Such messages are processed separately.
1770 */
1771 one_more:
1772     hipri_sig = 0;
1773     if (bp == NULL) {
1774         nextbp = NULL;
1775     } else {
1776         nextbp = bp->b_next;
1777         bp->b_next = NULL;

```

```

1778     switch (bp->b_datap->db_type) {
1779     case M_PCPROTO:
1780         /*
1781          * Only one priority protocol message is allowed at the
1782          * stream head at a time.
1783          */
1784         if (stp->sd_flag & STRPRI) {
1785             TRACE_0(TR_FAC_STREAMS_FR, TR_STRRPUT_PROTERR,
1786                 "M_PCPROTO already at head");
1787             freemsg(bp);
1788             mutex_exit(&stp->sd_lock);
1789             goto done;
1790         }
1791         stp->sd_flag |= STRPRI;
1792         hipri_sig = 1;
1793         /* FALLTHRU */
1794     case M_DATA:
1795     case M_PROTO:
1796     case M_PASSFP:
1797         band = bp->b_band;
1798         /*
1799          * Marking doesn't work well when messages
1800          * are marked in more than one band. We only
1801          * remember the last message received, even if
1802          * it is placed on the queue ahead of other
1803          * marked messages.
1804          */
1805         if (bp->b_flag & MSGMARK)
1806             stp->sd_mark = bp;
1807         (void) putq(q, bp);
1808
1809         /*
1810          * If message is a PCPROTO message, always use
1811          * firstmsgsig to determine if a signal should be
1812          * sent as strrrput is the only place to send
1813          * signals for PCPROTO. Other messages are based on
1814          * the STRGETINPROG flag. The flag determines if
1815          * strrrput or (k)strgetmsg will be responsible for
1816          * sending the signals, in the firstmsgsig case.
1817          */
1818         if ((hipri_sig == 1) ||
1819             ((stp->sd_flag & STRGETINPROG) == 0) &&
1820             (q->q_first == bp))
1821             signals = (firstmsgsig | allmsgsig);
1822         else
1823             signals = allmsgsig;
1824         break;
1825
1826     default:
1827         mutex_exit(&stp->sd_lock);
1828         (void) strrrput_nondata(q, bp);
1829         mutex_enter(&stp->sd_lock);
1830         break;
1831     }
1832 }
1833 ASSERT(MUTEX_HELD(&stp->sd_lock));
1834 /*
1835 * Wake sleeping read/getmsg and cancel deferred wakeup
1836 */
1837 if (wakeups & RSLEEP)
1838     stp->sd_wakeq &= ~RSLEEP;
1839
1840 wakeups &= stp->sd_flag;
1841 if (wakeups & RSLEEP) {
1842     stp->sd_flag &= ~RSLEEP;

```



```

1976         flushed_already |= FLUSHR;
1977         stp->sd_flag |= STRDERR;
1978         rw |= FLUSHR;
1979     } else {
1980         stp->sd_flag &= ~STRDERR;
1981     }
1982     stp->sd_rerror = *bp->b_rptr;
1983 }
1984 bp->b_rptr++;
1985 if (*bp->b_rptr != NOERROR) { /* write error */
1986     if (*bp->b_rptr != 0) {
1987         if (stp->sd_flag & STWRERR)
1988             flushed_already |= FLUSHW;
1989         stp->sd_flag |= STWRERR;
1990         rw |= FLUSHW;
1991     } else {
1992         stp->sd_flag &= ~STWRERR;
1993     }
1994     stp->sd_werror = *bp->b_rptr;
1995 }
1996 if (rw) {
1997     TRACE_2(TR_FAC_STREAMS_FR, TR_STRRPUT_WAKE,
1998         "strrput cv_broadcast:q %p, bp %p",
1999         q, bp);
2000     cv_broadcast(&q->q_wait); /* readers */
2001     cv_broadcast(&WR(q)->q_wait); /* writers */
2002     cv_broadcast(&stp->sd_monitor); /* ioctlers */
2003
2004     mutex_exit(&stp->sd_lock);
2005     pollwakep(&stp->sd_pollist, POLLERR);
2006     mutex_enter(&stp->sd_lock);
2007
2008     if (stp->sd_sigflags & S_ERROR)
2009         strsendsig(stp->sd_siglist, S_ERROR, 0,
2010             ((rw & FLUSHR) ? stp->sd_rerror :
2011             stp->sd_werror));
2012     mutex_exit(&stp->sd_lock);
2013     /*
2014     * Send the M_FLUSH only
2015     * for the first M_ERROR
2016     * message on the stream
2017     */
2018     if (flushed_already == rw) {
2019         freemsg(bp);
2020         return (0);
2021     }
2022
2023     bp->b_datap->db_type = M_FLUSH;
2024     *bp->b_rptr = rw;
2025     bp->b_wptr = bp->b_rptr + 1;
2026     /*
2027     * Protect against the driver
2028     * passing up messages after
2029     * it has done a qprocsoff
2030     */
2031     if (_OTHERQ(q)->q_next == NULL)
2032         freemsg(bp);
2033     else
2034         greply(q, bp);
2035     return (0);
2036 } else
2037     mutex_exit(&stp->sd_lock);
2038 } else if (*bp->b_rptr != 0) { /* Old flavor */
2039     if (stp->sd_flag & (STRDERR|STWRERR))
2040         flushed_already = FLUSHRW;
2041     mutex_enter(&stp->sd_lock);

```

```

2042     stp->sd_flag |= (STRDERR|STWRERR);
2043     stp->sd_rerror = *bp->b_rptr;
2044     stp->sd_werror = *bp->b_rptr;
2045     TRACE_2(TR_FAC_STREAMS_FR,
2046         TR_STRRPUT_WAKE2,
2047         "strrput wakeup #2:q %p, bp %p", q, bp);
2048     cv_broadcast(&q->q_wait); /* the readers */
2049     cv_broadcast(&WR(q)->q_wait); /* the writers */
2050     cv_broadcast(&stp->sd_monitor); /* ioctlers */
2051
2052     mutex_exit(&stp->sd_lock);
2053     pollwakep(&stp->sd_pollist, POLLERR);
2054     mutex_enter(&stp->sd_lock);
2055
2056     if (stp->sd_sigflags & S_ERROR)
2057         strsendsig(stp->sd_siglist, S_ERROR, 0,
2058             (stp->sd_werror ? stp->sd_werror :
2059             stp->sd_rerror));
2060     mutex_exit(&stp->sd_lock);
2061
2062     /*
2063     * Send the M_FLUSH only
2064     * for the first M_ERROR
2065     * message on the stream
2066     */
2067     if (flushed_already != FLUSHRW) {
2068         bp->b_datap->db_type = M_FLUSH;
2069         *bp->b_rptr = FLUSHRW;
2070         /*
2071         * Protect against the driver passing up
2072         * messages after it has done a
2073         * qprocsoff.
2074         */
2075         if (_OTHERQ(q)->q_next == NULL)
2076             freemsg(bp);
2077         else
2078             greply(q, bp);
2079         return (0);
2080     }
2081     freemsg(bp);
2082     return (0);
2083 }
2084
2085 case M_HANGUP:
2086
2087     freemsg(bp);
2088     mutex_enter(&stp->sd_lock);
2089     stp->sd_werror = ENXIO;
2090     stp->sd_flag |= STRHUP;
2091     stp->sd_flag &= ~(WSLEEP|RSLEEP);
2092
2093     /*
2094     * send signal if controlling tty
2095     */
2096
2097     if (stp->sd_sidp) {
2098         prsignal(stp->sd_sidp, SIGHUP);
2099         if (stp->sd_sidp != stp->sd_pgidp)
2100             pgsignal(stp->sd_pgidp, SIGTSTP);
2101     }
2102
2103     /*
2104     * wake up read, write, and exception pollers and
2105     * reset wakeup mechanism.
2106     */
2107     cv_broadcast(&q->q_wait); /* the readers */

```

```

2108     cv_broadcast(&WR(q)->q_wait); /* the writers */
2109     cv_broadcast(&stp->sd_monitor); /* the ioctllers */
2110     strhup(stp);
2111     mutex_exit(&stp->sd_lock);
2112     return (0);

2114 case M_UNHANGUP:
2115     freemsg(bp);
2116     mutex_enter(&stp->sd_lock);
2117     stp->sd_werror = 0;
2118     stp->sd_flag &= ~STRHUP;
2119     mutex_exit(&stp->sd_lock);
2120     return (0);

2122 case M_SIG:
2123     /*
2124     * Someone downstream wants to post a signal. The
2125     * signal to post is contained in the first byte of the
2126     * message. If the message would go on the front of
2127     * the queue, send a signal to the process group
2128     * (if not SIGPOLL) or to the siglist processes
2129     * (SIGPOLL). If something is already on the queue,
2130     * OR if we are delivering a delayed suspend (*sigh*
2131     * another "tty" hack) and there's no one sleeping already,
2132     * just enqueue the message.
2133     */
2134     mutex_enter(&stp->sd_lock);
2135     if (q->q_first || (*bp->b_rptr == SIGTSTP &&
2136         !(stp->sd_flag & RSLEEP))) {
2137         (void) putq(q, bp);
2138         mutex_exit(&stp->sd_lock);
2139         return (0);
2140     }
2141     mutex_exit(&stp->sd_lock);
2142     /* FALLTHRU */

2144 case M_PCSIG:
2145     /*
2146     * Don't enqueue, just post the signal.
2147     */
2148     strsignal(stp, *bp->b_rptr, 0L);
2149     freemsg(bp);
2150     return (0);

2152 case M_CMD:
2153     if (MBLKL(bp) != sizeof (cmdblk_t)) {
2154         freemsg(bp);
2155         return (0);
2156     }

2158     mutex_enter(&stp->sd_lock);
2159     if (stp->sd_flag & STRCMDWAIT) {
2160         ASSERT(stp->sd_cmdblk == NULL);
2161         stp->sd_cmdblk = bp;
2162         cv_broadcast(&stp->sd_monitor);
2163         mutex_exit(&stp->sd_lock);
2164     } else {
2165         mutex_exit(&stp->sd_lock);
2166         freemsg(bp);
2167     }
2168     return (0);

2170 case M_FLUSH:
2171     /*
2172     * Flush queues. The indication of which queues to flush
2173     * is in the first byte of the message. If the read queue

```

```

2174     * is specified, then flush it. If FLUSHBAND is set, just
2175     * flush the band specified by the second byte of the message.
2176     *
2177     * If a module has issued a M_SETOPT to not flush hi
2178     * priority messages off of the stream head, then pass this
2179     * flag into the flushq code to preserve such messages.
2180     */

2182 if (*bp->b_rptr & FLUSHR) {
2183     mutex_enter(&stp->sd_lock);
2184     if (*bp->b_rptr & FLUSHBAND) {
2185         ASSERT((bp->b_wptr - bp->b_rptr) >= 2);
2186         flushband(q, *(bp->b_rptr + 1), FLUSHALL);
2187     } else
2188         flushq_common(q, FLUSHALL,
2189             stp->sd_read_opt & RFLUSHPCPROT);
2190     if ((q->q_first == NULL) ||
2191         (q->q_first->b_datap->db_type < QPCTL))
2192         stp->sd_flag &= ~STRPRI;
2193     else {
2194         ASSERT(stp->sd_flag & STRPRI);
2195     }
2196     mutex_exit(&stp->sd_lock);
2197 }
2198 if ((*bp->b_rptr & FLUSHW) && !(bp->b_flag & MSGNOLOOP)) {
2199     *bp->b_rptr &= ~FLUSHR;
2200     bp->b_flag |= MSGNOLOOP;
2201     /*
2202     * Protect against the driver passing up
2203     * messages after it has done a qprocsoff.
2204     */
2205     if (_OTHERQ(q)->q_next == NULL)
2206         freemsg(bp);
2207     else
2208         qreply(q, bp);
2209     return (0);
2210 }
2211 freemsg(bp);
2212 return (0);

2214 case M_IOCACK:
2215 case M_IOCNAK:
2216     iocbp = (struct iocblk *)bp->b_rptr;
2217     /*
2218     * If not waiting for ACK or NAK then just free msg.
2219     * If incorrect id sequence number then just free msg.
2220     * If already have ACK or NAK for user then this is a
2221     * duplicate, display a warning and free the msg.
2222     */
2223     mutex_enter(&stp->sd_lock);
2224     if ((stp->sd_flag & IOCWAIT) == 0 || stp->sd_iocblk ||
2225         (stp->sd_iocid != iocbp->ioc_id)) {
2226         /*
2227         * If the ACK/NAK is a dup, display a message
2228         * Dup is when sd_iocid == ioc_id, and
2229         * sd_iocblk == <valid ptr> or -1 (the former
2230         * is when an ioctl has been put on the stream
2231         * head, but has not yet been consumed, the
2232         * later is when it has been consumed).
2233         */
2234         if ((stp->sd_iocid == iocbp->ioc_id) &&
2235             (stp->sd_iocblk != NULL)) {
2236             log_dupioc(q, bp);
2237         }
2238         freemsg(bp);
2239         mutex_exit(&stp->sd_lock);

```

```

2240         return (0);
2241     }
2242
2243     /*
2244     * Assign ACK or NAK to user and wake up.
2245     */
2246     stp->sd_iocblk = bp;
2247     cv_broadcast(&stp->sd_monitor);
2248     mutex_exit(&stp->sd_lock);
2249     return (0);
2250
2251 case M_COPYIN:
2252 case M_COPYOUT:
2253     reqp = (struct copyreq *)bp->b_rptr;
2254
2255     /*
2256     * If not waiting for ACK or NAK then just fail request.
2257     * If already have ACK, NAK, or copy request, then just
2258     * fail request.
2259     * If incorrect id sequence number then just fail request.
2260     */
2261     mutex_enter(&stp->sd_lock);
2262     if ((stp->sd_flag & IOCWAIT) == 0 || stp->sd_iocblk ||
2263         (stp->sd_iocid != reqp->cq_id)) {
2264         if (bp->b_cont) {
2265             freemsg(bp->b_cont);
2266             bp->b_cont = NULL;
2267         }
2268         bp->b_datap->db_type = M_IOCDATA;
2269         bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
2270         resp = (struct copyresp *)bp->b_rptr;
2271         resp->cp_rval = (caddr_t)1; /* failure */
2272         mutex_exit(&stp->sd_lock);
2273         putnext(stp->sd_wrq, bp);
2274         return (0);
2275     }
2276
2277     /*
2278     * Assign copy request to user and wake up.
2279     */
2280     stp->sd_iocblk = bp;
2281     cv_broadcast(&stp->sd_monitor);
2282     mutex_exit(&stp->sd_lock);
2283     return (0);
2284
2285 case M_SETOPTS:
2286     /*
2287     * Set stream head options (read option, write offset,
2288     * min/max packet size, and/or high/low water marks for
2289     * the read side only).
2290     */
2291
2292     bpri = 0;
2293     sop = (struct stroptions *)bp->b_rptr;
2294     mutex_enter(&stp->sd_lock);
2295     if (sop->so_flags & SO_READOPT) {
2296         switch (sop->so_readopt & RMODEMASK) {
2297             case RNORM:
2298                 stp->sd_read_opt &= ~(RD_MSGDIS | RD_MSGNODIS);
2299                 break;
2300
2301             case RMSGD:
2302                 stp->sd_read_opt =
2303                     ((stp->sd_read_opt & ~RD_MSGNODIS) |
2304                      RD_MSGDIS);
2305                 break;

```

```

2307         case RMSGN:
2308             stp->sd_read_opt =
2309                 ((stp->sd_read_opt & ~RD_MSGDIS) |
2310                  RD_MSGNODIS);
2311             break;
2312         }
2313     }
2314     switch (sop->so_readopt & RPROTMASK) {
2315     case RPROTNORM:
2316         stp->sd_read_opt &= ~(RD_PROTDAT | RD_PROTDIS);
2317         break;
2318
2319     case RPROTDAT:
2320         stp->sd_read_opt =
2321             ((stp->sd_read_opt & ~RD_PROTDIS) |
2322              RD_PROTDAT);
2323         break;
2324
2325     case RPROTDIS:
2326         stp->sd_read_opt =
2327             ((stp->sd_read_opt & ~RD_PROTDAT) |
2328              RD_PROTDIS);
2329         break;
2330     }
2331     switch (sop->so_readopt & RFLUSHMASK) {
2332     case RFLUSHPCPROT:
2333         /*
2334         * This sets the stream head to NOT flush
2335         * M_PCPROTO messages.
2336         */
2337         stp->sd_read_opt |= RFLUSHPCPROT;
2338         break;
2339     }
2340     if (sop->so_flags & SO_ERROPT) {
2341         switch (sop->so_erropt & RERRMASK) {
2342         case RERRNORM:
2343             stp->sd_flag &= ~STRDERRNONPERSIST;
2344             break;
2345         case RERRNONPERSIST:
2346             stp->sd_flag |= STRDERRNONPERSIST;
2347             break;
2348         }
2349     }
2350     switch (sop->so_erropt & WERRMASK) {
2351     case WERRNORM:
2352         stp->sd_flag &= ~STWRERRNONPERSIST;
2353         break;
2354     case WERRNONPERSIST:
2355         stp->sd_flag |= STWRERRNONPERSIST;
2356         break;
2357     }
2358     if (sop->so_flags & SO_COPYOPT) {
2359         if (sop->so_copyopt & ZCVMSAFE) {
2360             stp->sd_copyflag |= STZCVMSAFE;
2361             stp->sd_copyflag &= ~STZCVMUNSAFE;
2362         } else if (sop->so_copyopt & ZCVMUNSAFE) {
2363             stp->sd_copyflag |= STZCVMUNSAFE;
2364             stp->sd_copyflag &= ~STZCVMSAFE;
2365         }
2366     }
2367     if (sop->so_copyopt & COPYCACHED) {
2368         stp->sd_copyflag |= STRCOPYCACHED;
2369     }
2370     }
2371     if (sop->so_flags & SO_WROFF)

```

```

2372         stp->sd_wroff = sop->so_wroff;
2373     if (sop->so_flags & SO_TAIL)
2374         stp->sd_tail = sop->so_tail;
2375     if (sop->so_flags & SO_MINPSZ)
2376         q->q_minpsz = sop->so_minpsz;
2377     if (sop->so_flags & SO_MAXPSZ)
2378         q->q_maxpsz = sop->so_maxpsz;
2379     if (sop->so_flags & SO_MAXBLK)
2380         stp->sd_maxblk = sop->so_maxblk;
2381     if (sop->so_flags & SO_HIWAT) {
2382         if (sop->so_flags & SO_BAND) {
2383             if (strqset(q, QHIWAT,
2384                 sop->so_band, sop->so_hiwat)) {
2385                 cmn_err(CE_WARN, "strrput: could not "
2386                     "allocate qband\n");
2387             } else {
2388                 bpri = sop->so_band;
2389             }
2390         } else {
2391             q->q_hiwat = sop->so_hiwat;
2392         }
2393     }
2394     if (sop->so_flags & SO_LOWAT) {
2395         if (sop->so_flags & SO_BAND) {
2396             if (strqset(q, QLOWAT,
2397                 sop->so_band, sop->so_lowat)) {
2398                 cmn_err(CE_WARN, "strrput: could not "
2399                     "allocate qband\n");
2400             } else {
2401                 bpri = sop->so_band;
2402             }
2403         } else {
2404             q->q_lowat = sop->so_lowat;
2405         }
2406     }
2407     if (sop->so_flags & SO_MREADON)
2408         stp->sd_flag |= SNDMREAD;
2409     if (sop->so_flags & SO_MREADOFF)
2410         stp->sd_flag &= ~SNDMREAD;
2411     if (sop->so_flags & SO_NDELOK)
2412         stp->sd_flag |= OLDNDELAY;
2413     if (sop->so_flags & SO_NDELOFF)
2414         stp->sd_flag &= ~OLDNDELAY;
2415     if (sop->so_flags & SO_ISTTY)
2416         stp->sd_flag |= STRISTTY;
2417     if (sop->so_flags & SO_ISNTTY)
2418         stp->sd_flag &= ~STRISTTY;
2419     if (sop->so_flags & SO_TOSTOP)
2420         stp->sd_flag |= STRTOSTOP;
2421     if (sop->so_flags & SO_TONSTOP)
2422         stp->sd_flag &= ~STRTOSTOP;
2423     if (sop->so_flags & SO_DELMIM)
2424         stp->sd_flag |= STRDELIM;
2425     if (sop->so_flags & SO_NODELIM)
2426         stp->sd_flag &= ~STRDELIM;
2428     mutex_exit(&stp->sd_lock);
2429     freemsg(bp);
2431     /* Check backenable in case the water marks changed */
2432     qbackenable(q, bpri);
2433     return (0);
2435 /*
2436  * The following set of cases deal with situations where two stream
2437  * heads are connected to each other (twisted streams). These messages

```

```

2438     * have no meaning at the stream head.
2439     */
2440     case M_BREAK:
2441     case M_CTL:
2442     case M_DELAY:
2443     case M_START:
2444     case M_STOP:
2445     case M_IOCADATA:
2446     case M_STARTI:
2447     case M_STOPI:
2448         freemsg(bp);
2449         return (0);
2451     case M_IOCTL:
2452         /*
2453          * Always NAK this condition
2454          * (makes no sense)
2455          * If there is one or more threads in the read side
2456          * rwnext we have to defer the nacking until that thread
2457          * returns (in strget).
2458          */
2459         mutex_enter(&stp->sd_lock);
2460         if (stp->sd_struiodnak != 0) {
2461             /*
2462              * Defer NAK to the streamhead. Queue at the end
2463              * the list.
2464              */
2465             mblk_t *mp = stp->sd_struionak;
2467             while (mp && mp->b_next)
2468                 mp = mp->b_next;
2469             if (mp)
2470                 mp->b_next = bp;
2471             else
2472                 stp->sd_struionak = bp;
2473             bp->b_next = NULL;
2474             mutex_exit(&stp->sd_lock);
2475             return (0);
2476         }
2477         mutex_exit(&stp->sd_lock);
2479         bp->b_datap->db_type = M_IOCNAK;
2480         /*
2481          * Protect against the driver passing up
2482          * messages after it has done a qprocsoff.
2483          */
2484         if (_OTHERQ(q)->q_next == NULL)
2485             freemsg(bp);
2486         else
2487             qreply(q, bp);
2488         return (0);
2490     default:
2491     #ifdef DEBUG
2492         cmn_err(CE_WARN,
2493             "bad message type %x received at stream head\n",
2494             bp->b_datap->db_type);
2495     #endif
2496         freemsg(bp);
2497         return (0);
2498     }
2499 }
2500     /* NOTREACHED */
2501 }
2503 /*

```

```

2504 * Check if the stream pointed to by 'stp' can be written to, and return an
2505 * error code if not. If 'eiohup' is set, then return EIO if STRHUP is set.
2506 * If 'sigpipeok' is set and the SW_SIGPIPE option is enabled on the stream,
2507 * then always return EPIPE and send a SIGPIPE to the invoking thread.
2508 */
2509 static int
2510 strwriteable(struct stdata *stp, boolean_t eiohup, boolean_t sigpipeok)
2511 {
2512     int error;
2513
2514     ASSERT(MUTEX_HELD(&stp->sd_lock));
2515
2516     /*
2517      * For modem support, POSIX states that on writes, EIO should
2518      * be returned if the stream has been hung up.
2519      */
2520     if (eiohup && (stp->sd_flag & (STPLEX|STRHUP)) == STRHUP)
2521         error = EIO;
2522     else
2523         error = strgeterr(stp, STRHUP|STPLEX|STWRERR, 0);
2524
2525     if (error != 0) {
2526         if (!(stp->sd_flag & STPLEX) &&
2527             (stp->sd_wput_opt & SW_SIGPIPE) && sigpipeok) {
2528             tsignal(curthread, SIGPIPE);
2529             error = EPIPE;
2530         }
2531     }
2532
2533     return (error);
2534 }
2535
2536 /*
2537 * Copyin and send data down a stream.
2538 * The caller will allocate and copyin any control part that precedes the
2539 * message and pass that in as mctl.
2540 *
2541 * Caller should *not* hold sd_lock.
2542 * When EWOULDBLOCK is returned the caller has to redo the canputnext
2543 * under sd_lock in order to avoid missing a backenabling wakeup.
2544 *
2545 * Use iosize = -1 to not send any M_DATA. iosize = 0 sends zero-length M_DATA.
2546 *
2547 * Set MSG_IGNFLOW in flags to ignore flow control for hipri messages.
2548 * For sync streams we can only ignore flow control by reverting to using
2549 * putnext.
2550 *
2551 * If sd_maxblk is less than *iosize this routine might return without
2552 * transferring all of *iosize. In all cases, on return *iosize will contain
2553 * the amount of data that was transferred.
2554 */
2555 static int
2556 strput(struct stdata *stp, mblk_t *mctl, struct uiop *uiop, ssize_t *iosize,
2557        int b_flag, int pri, int flags)
2558 {
2559     struid_t uiod;
2560     mblk_t *mp;
2561     queue_t *wqp = stp->sd_wrq;
2562     int error = 0;
2563     ssize_t count = *iosize;
2564
2565     ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));
2566
2567     if (uiop != NULL && count >= 0)
2568         flags |= stp->sd_struiowrq ? STRUIO_POSTPONE : 0;

```

```

2570     if (!(flags & STRUIO_POSTPONE)) {
2571         /*
2572          * Use regular canputnext, strmkeddata, putnext sequence.
2573          */
2574         if (pri == 0) {
2575             if (!canputnext(wqp) && !(flags & MSG_IGNFLOW)) {
2576                 freemsg(mctl);
2577                 return (EWOULDBLOCK);
2578             }
2579         } else {
2580             if (!(flags & MSG_IGNFLOW) && !canputnext(wqp, pri)) {
2581                 freemsg(mctl);
2582                 return (EWOULDBLOCK);
2583             }
2584         }
2585
2586         if ((error = strmkeddata(iosize, uiop, stp, flags,
2587                                &mp)) != 0) {
2588             freemsg(mctl);
2589             /*
2590              * need to change return code to ENOMEM
2591              * so that this is not confused with
2592              * flow control, EAGAIN.
2593              */
2594
2595             if (error == EAGAIN)
2596                 return (ENOMEM);
2597             else
2598                 return (error);
2599         }
2600         if (mctl != NULL) {
2601             if (mctl->b_cont == NULL)
2602                 mctl->b_cont = mp;
2603             else if (mp != NULL)
2604                 linkb(mctl, mp);
2605             mp = mctl;
2606         } else if (mp == NULL)
2607             return (0);
2608
2609         mp->b_flag |= b_flag;
2610         mp->b_band = (uchar_t)pri;
2611
2612         if (flags & MSG_IGNFLOW) {
2613             /*
2614              * XXX Hack: Don't get stuck running service
2615              * procedures. This is needed for sockfs when
2616              * sending the unbind message out of the rput
2617              * procedure - we don't want a put procedure
2618              * to run service procedures.
2619              */
2620             putnext(wqp, mp);
2621         } else {
2622             stream_willservice(stp);
2623             putnext(wqp, mp);
2624             stream_runservice(stp);
2625         }
2626         return (0);
2627     }
2628     /*
2629      * Stream supports rwnext() for the write side.
2630      */
2631     if ((error = strmkeddata(iosize, uiop, stp, flags, &mp)) != 0) {
2632         freemsg(mctl);
2633         /*
2634          * map EAGAIN to ENOMEM since EAGAIN means "flow controlled".
2635          */

```

```

2636         return (error == EAGAIN ? ENOMEM : error);
2637     }
2638     if (mctl != NULL) {
2639         if (mctl->b_cont == NULL)
2640             mctl->b_cont = mp;
2641         else if (mp != NULL)
2642             linkb(mctl, mp);
2643         mp = mctl;
2644     } else if (mp == NULL) {
2645         return (0);
2646     }
2648     mp->b_flag |= b_flag;
2649     mp->b_band = (uchar_t)pri;
2651     (void) uiodup(uiop, &uiod.d_uio, uiod.d_iov,
2652         sizeof (uiod.d_iov) / sizeof (*uiod.d_iov));
2653     uiod.d_uio.uio_offset = 0;
2654     uiod.d_mp = mp;
2655     error = rwnext(wqp, &uiod);
2656     if (! uiod.d_mp) {
2657         uioskip(uiop, *iosize);
2658         return (error);
2659     }
2660     ASSERT(mp == uiod.d_mp);
2661     if (error == EINVAL) {
2662         /*
2663          * The stream plumbing must have changed while
2664          * we were away, so just turn off rwnext(s).
2665          */
2666         error = 0;
2667     } else if (error == EBUSY || error == EWOULDBLOCK) {
2668         /*
2669          * Couldn't enter a perimeter or took a page fault,
2670          * so fall-back to putnext().
2671          */
2672         error = 0;
2673     } else {
2674         freemsg(mp);
2675         return (error);
2676     }
2677     /* Have to check canput before consuming data from the uio */
2678     if (pri == 0) {
2679         if (!canputnext(wqp) && !(flags & MSG_IGNFLOW)) {
2680             freemsg(mp);
2681             return (EWOULDBLOCK);
2682         }
2683     } else {
2684         if (!bcanputnext(wqp, pri) && !(flags & MSG_IGNFLOW)) {
2685             freemsg(mp);
2686             return (EWOULDBLOCK);
2687         }
2688     }
2689     ASSERT(mp == uiod.d_mp);
2690     /* Copyin data from the uio */
2691     if ((error = struioget(wqp, mp, &uiod, 0)) != 0) {
2692         freemsg(mp);
2693         return (error);
2694     }
2695     uioskip(uiop, *iosize);
2696     if (flags & MSG_IGNFLOW) {
2697         /*
2698          * XXX Hack: Don't get stuck running service procedures.
2699          * This is needed for sockfs when sending the unbind message
2700          * out of the rput procedure - we don't want a put procedure
2701          * to run service procedures.

```

```

2702         */
2703         putnext(wqp, mp);
2704     } else {
2705         stream_willservice(stp);
2706         putnext(wqp, mp);
2707         stream_runservice(stp);
2708     }
2709     return (0);
2710 }
2712 /*
2713 * Write attempts to break the write request into messages conforming
2714 * with the minimum and maximum packet sizes set downstream.
2715 *
2716 * Write will not block if downstream queue is full and
2717 * O_NDELAY is set, otherwise it will block waiting for the queue to get room.
2718 *
2719 * A write of zero bytes gets packaged into a zero length message and sent
2720 * downstream like any other message.
2721 *
2722 * If buffers of the requested sizes are not available, the write will
2723 * sleep until the buffers become available.
2724 *
2725 * Write (if specified) will supply a write offset in a message if it
2726 * makes sense. This can be specified by downstream modules as part of
2727 * a M_SETOPTS message. Write will not supply the write offset if it
2728 * cannot supply any data in a buffer. In other words, write will never
2729 * send down an empty packet due to a write offset.
2730 */
2731 /* ARGSUSED2 */
2732 int
2733 strwrite(struct vnode *vp, struct uio *uiop, cred_t *crp)
2734 {
2735     return (strwrite_common(vp, uiop, crp, 0));
2736 }
2738 /* ARGSUSED2 */
2739 int
2740 strwrite_common(struct vnode *vp, struct uio *uiop, cred_t *crp, int wflag)
2741 {
2742     struct stdata *stp;
2743     struct queue *wqp;
2744     ssize_t rmin, rmax;
2745     ssize_t iosize;
2746     int waitflag;
2747     int tempmode;
2748     int error = 0;
2749     int b_flag;
2751     ASSERT(vp->v_stream);
2752     stp = vp->v_stream;
2754     mutex_enter(&stp->sd_lock);
2756     if ((error = i_straccess(stp, JCWRITE)) != 0) {
2757         mutex_exit(&stp->sd_lock);
2758         return (error);
2759     }
2761     if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
2762         error = strwriteable(stp, B_TRUE, B_TRUE);
2763         if (error != 0) {
2764             mutex_exit(&stp->sd_lock);
2765             return (error);
2766         }
2767     }

```

```

2769     mutex_exit(&stp->sd_lock);
2771     wqp = stp->sd_wrq;
2773     /* get these values from them cached in the stream head */
2774     rmin = stp->sd_qn_minpsz;
2775     rmax = stp->sd_qn_maxpsz;
2777     /*
2778     * Check the min/max packet size constraints.  If min packet size
2779     * is non-zero, the write cannot be split into multiple messages
2780     * and still guarantee the size constraints.
2781     */
2782     TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_IN, "strwrite in:q %p", wqp);
2784     ASSERT((rmax >= 0) || (rmax == INFPSZ));
2785     if (rmax == 0) {
2786         return (0);
2787     }
2788     if (rmin > 0) {
2789         if (uiop->uio_resid < rmin) {
2790             TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2791                 "strwrite out:q %p out %d error %d",
2792                 wqp, 0, ERANGE);
2793             return (ERANGE);
2794         }
2795         if ((rmax != INFPSZ) && (uiop->uio_resid > rmax)) {
2796             TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2797                 "strwrite out:q %p out %d error %d",
2798                 wqp, 1, ERANGE);
2799             return (ERANGE);
2800         }
2801     }
2803     /*
2804     * Do until count satisfied or error.
2805     */
2806     waitflag = WRITEWAIT | wflag;
2807     if (stp->sd_flag & OLDNDELAY)
2808         tempmode = uiop->uio_fmode & ~FNDELAY;
2809     else
2810         tempmode = uiop->uio_fmode;
2812     if (rmax == INFPSZ)
2813         rmax = uiop->uio_resid;
2815     /*
2816     * Note that tempmode does not get used in strput/strmakedata
2817     * but only in strwaitq.  The other routines use uio_fmode
2818     * unmodified.
2819     */
2821     /* LINTED: constant in conditional context */
2822     while (1) { /* breaks when uio_resid reaches zero */
2823         /*
2824         * Determine the size of the next message to be
2825         * packaged.  May have to break write into several
2826         * messages based on max packet size.
2827         */
2828         iosize = MIN(uiop->uio_resid, rmax);
2830         /*
2831         * Put block downstream when flow control allows it.
2832         */
2833         if ((stp->sd_flag & STRDELIM) && (uiop->uio_resid == iosize))

```

```

2834         b_flag = MSGDELIM;
2835     else
2836         b_flag = 0;
2838     for (;;) {
2839         int done = 0;
2841         error = strput(stp, NULL, uiop, &iosize, b_flag, 0, 0);
2842         if (error == 0)
2843             break;
2844         if (error != EWOULDBLOCK)
2845             goto out;
2847         mutex_enter(&stp->sd_lock);
2848         /*
2849         * Check for a missed wakeup.
2850         * Needed since strput did not hold sd_lock across
2851         * the canputnext.
2852         */
2853         if (canputnext(wqp)) {
2854             /* Try again */
2855             mutex_exit(&stp->sd_lock);
2856             continue;
2857         }
2858         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_WAIT,
2859             "strwrite wait:q %p wait", wqp);
2860         if ((error = strwaitq(stp, waitflag, (ssize_t)0,
2861             tempmode, -1, &done) != 0 || done) {
2862             mutex_exit(&stp->sd_lock);
2863             if ((vp->v_type == VFIFO) &&
2864                 (uiop->uio_fmode & FNDELAY) &&
2865                 (error == EAGAIN))
2866                 error = 0;
2867             goto out;
2868         }
2869         TRACE_1(TR_FAC_STREAMS_FR, TR_STRWRITE_WAKE,
2870             "strwrite wake:q %p awakes", wqp);
2871         if ((error = i_straccess(stp, JCWRITE)) != 0) {
2872             mutex_exit(&stp->sd_lock);
2873             goto out;
2874         }
2875         mutex_exit(&stp->sd_lock);
2876     }
2877     waitflag |= NOINTR;
2878     TRACE_2(TR_FAC_STREAMS_FR, TR_STRWRITE_RESID,
2879         "strwrite resid:q %p uiop %p", wqp, uiop);
2880     if (uiop->uio_resid) {
2881         /* Recheck for errors - needed for sockets */
2882         if ((stp->sd_wput_opt & SW_RECHECK_ERR) &&
2883             (stp->sd_flag & (STWRERR|STRHUP|STPLEX))) {
2884             mutex_enter(&stp->sd_lock);
2885             error = strwriteable(stp, B_FALSE, B_TRUE);
2886             mutex_exit(&stp->sd_lock);
2887             if (error != 0)
2888                 return (error);
2889         }
2890         continue;
2891     }
2892     break;
2893 }
2894 out:
2895     /*
2896     * For historical reasons, applications expect EAGAIN when a data
2897     * mblk_t cannot be allocated, so change ENOMEM back to EAGAIN.
2898     */
2899     if (error == ENOMEM)

```

```

2900         error = EAGAIN;
2901     TRACE_3(TR_FAC_STREAMS_FR, TR_STRWRITE_OUT,
2902         "strwrite out:q %p out %d error %d", wqp, 2, error);
2903     return (error);
2904 }

2906 /*
2907  * Stream head write service routine.
2908  * Its job is to wake up any sleeping writers when a queue
2909  * downstream needs data (part of the flow control in putq and getq).
2910  * It also must wake anyone sleeping on a poll().
2911  * For stream head right below mux module, it must also invoke put procedure
2912  * of next downstream module.
2913  */
2914 int
2915 strwsrv(queue_t *q)
2916 {
2917     struct stdata *stp;
2918     queue_t *tq;
2919     qband_t *qbp;
2920     int i;
2921     qband_t *myqbp;
2922     int isevent;
2923     unsigned char  qbf[NBAND];    /* band flushing backenable flags */

2925     TRACE_1(TR_FAC_STREAMS_FR,
2926         TR_STRWSRV, "strwsrv:q %p", q);
2927     stp = (struct stdata *)q->q_ptr;
2928     ASSERT(qclaimed(q));
2929     mutex_enter(&stp->sd_lock);
2930     ASSERT(!(stp->sd_flag & STPLEX));

2932     if (stp->sd_flag & WSLEEP) {
2933         stp->sd_flag &= ~WSLEEP;
2934         cv_broadcast(&q->q_wait);
2935     }
2936     mutex_exit(&stp->sd_lock);

2938     /* The other end of a stream pipe went away. */
2939     if ((tq = q->q_next) == NULL) {
2940         return (0);
2941     }

2943     /* Find the next module forward that has a service procedure */
2944     claimstr(q);
2945     tq = q->q_nfsrv;
2946     ASSERT(tq != NULL);

2948     if ((q->q_flag & QBACK) {
2949         if ((tq->q_flag & QFULL) {
2950             mutex_enter(QLOCK(tq));
2951             if (!(tq->q_flag & QFULL) {
2952                 mutex_exit(QLOCK(tq));
2953                 goto wakeup;
2954             }
2955             /*
2956              * The queue must have become full again. Set QWANTW
2957              * again so strwsrv will be back enabled when
2958              * the queue becomes non-full next time.
2959              */
2960             tq->q_flag |= QWANTW;
2961             mutex_exit(QLOCK(tq));
2962         } else {
2963             wakeup:
2964             pollwakeup(&stp->sd_pollist, POLLWRNORM);
2965             mutex_enter(&stp->sd_lock);

```

```

2966         if (stp->sd_sigflags & S_WRNORM)
2967             strsendsig(stp->sd_siglist, S_WRNORM, 0, 0);
2968         mutex_exit(&stp->sd_lock);
2969     }
2970 }

2972     isevent = 0;
2973     i = 1;
2974     bzero((caddr_t)qbf, NBAND);
2975     mutex_enter(QLOCK(tq));
2976     if ((myqbp = q->q_bandp) != NULL)
2977         for (qbp = tq->q_bandp; qbp && myqbp; qbp = qbp->qb_next) {
2978             ASSERT(myqbp);
2979             if ((myqbp->qb_flag & QB_BACK) {
2980                 if (qbp->qb_flag & QB_FULL) {
2981                     /*
2982                      * The band must have become full again.
2983                      * Set QB_WANTW again so strwsrv will
2984                      * be back enabled when the band becomes
2985                      * non-full next time.
2986                      */
2987                     qbp->qb_flag |= QB_WANTW;
2988                 } else {
2989                     isevent = 1;
2990                     qbf[i] = 1;
2991                 }
2992             }
2993             myqbp = myqbp->qb_next;
2994             i++;
2995         }
2996     mutex_exit(QLOCK(tq));

2998     if (isevent) {
2999         for (i = tq->q_nband; i; i--) {
3000             if (qbf[i]) {
3001                 pollwakeup(&stp->sd_pollist, POLLWRBAND);
3002                 mutex_enter(&stp->sd_lock);
3003                 if (stp->sd_sigflags & S_WRBAND)
3004                     strsendsig(stp->sd_siglist, S_WRBAND,
3005                         (uchar_t)i, 0);
3006                 mutex_exit(&stp->sd_lock);
3007             }
3008         }
3009     }

3011     releasestr(q);
3012     return (0);
3013 }

3015 /*
3016  * Special case of strcopyin/strcopyout for copying
3017  * struct striocctl that can deal with both data
3018  * models.
3019  */

3021 #ifdef _LP64

3023 static int
3024 strcopyin_striocctl(void *from, void *to, int flag, int copyflag)
3025 {
3026     struct striocctl32 strioc32;
3027     struct striocctl *striocp;

3029     if (copyflag & U_TO_K) {
3030         ASSERT((copyflag & K_TO_K) == 0);

```

```

3032         if ((flag & FMODELS) == DATAMODEL_ILP32) {
3033             if (copyin(from, &strioc32, sizeof (strioc32)))
3034                 return (EFAULT);
3035
3036             striocp = (struct striocctl *)to;
3037             striocp->ic_cmd = strioc32.ic_cmd;
3038             striocp->ic_timeout = strioc32.ic_timeout;
3039             striocp->ic_len = strioc32.ic_len;
3040             striocp->ic_dp = (char *) (uintptr_t)strioc32.ic_dp;
3041
3042         } else { /* NATIVE data model */
3043             if (copyin(from, to, sizeof (struct striocctl))) {
3044                 return (EFAULT);
3045             } else {
3046                 return (0);
3047             }
3048         }
3049     } else {
3050         ASSERT(copyflag & K_TO_K);
3051         bcopy(from, to, sizeof (struct striocctl));
3052     }
3053     return (0);
3054 }
3055
3056 static int
3057 strcopyout_striocctl(void *from, void *to, int flag, int copyflag)
3058 {
3059     struct striocctl32 strioc32;
3060     struct striocctl *striocp;
3061
3062     if (copyflag & U_TO_K) {
3063         ASSERT((copyflag & K_TO_K) == 0);
3064
3065         if ((flag & FMODELS) == DATAMODEL_ILP32) {
3066             striocp = (struct striocctl *)from;
3067             strioc32.ic_cmd = striocp->ic_cmd;
3068             strioc32.ic_timeout = striocp->ic_timeout;
3069             strioc32.ic_len = striocp->ic_len;
3070             strioc32.ic_dp = (caddr32_t)(uintptr_t)striocp->ic_dp;
3071             ASSERT((char *) (uintptr_t)strioc32.ic_dp ==
3072                 striocp->ic_dp);
3073
3074             if (copyout(&strioc32, to, sizeof (strioc32)))
3075                 return (EFAULT);
3076
3077         } else { /* NATIVE data model */
3078             if (copyout(from, to, sizeof (struct striocctl))) {
3079                 return (EFAULT);
3080             } else {
3081                 return (0);
3082             }
3083         }
3084     } else {
3085         ASSERT(copyflag & K_TO_K);
3086         bcopy(from, to, sizeof (struct striocctl));
3087     }
3088     return (0);
3089 }
3090
3091 #else /* !_LP64 */
3092
3093 /* ARGSUSED2 */
3094 static int
3095 strcopyin_striocctl(void *from, void *to, int flag, int copyflag)
3096 {
3097     return (strcopyin(from, to, sizeof (struct striocctl), copyflag));

```

```

3098 }
3099
3100 /* ARGSUSED2 */
3101 static int
3102 strcopyout_striocctl(void *from, void *to, int flag, int copyflag)
3103 {
3104     return (strcopyout(from, to, sizeof (struct striocctl), copyflag));
3105 }
3106
3107 #endif /* !_LP64 */
3108
3109 /*
3110  * Determine type of job control semantics expected by user. The
3111  * possibilities are:
3112  *   JCREAD - Behaves like read() on fd; send SIGTTIN
3113  *   JCWRITE - Behaves like write() on fd; send SIGTTOU if TOSTOP set
3114  *   JCSETP - Sets a value in the stream; send SIGTTOU, ignore TOSTOP
3115  *   JCGETP - Gets a value in the stream; no signals.
3116  * See straccess in strsubr.c for usage of these values.
3117  *
3118  * This routine also returns -1 for I_STR as a special case; the
3119  * caller must call again with the real ioctl number for
3120  * classification.
3121  */
3122 static int
3123 job_control_type(int cmd)
3124 {
3125     switch (cmd) {
3126     case I_STR:
3127         return (-1);
3128
3129     case I_RECVFD:
3130     case I_E_RECVFD:
3131         return (JCREAD);
3132
3133     case I_FDINSERT:
3134     case I_SENDFD:
3135         return (JCWRITE);
3136
3137     case TCSETA:
3138     case TCSETAW:
3139     case TCSETAF:
3140     case TCSETP:
3141     case TCXONC:
3142     case TCFLSH:
3143     case TCDSET: /* Obsolete */
3144     case TIOCSWINSZ:
3145     case TCSETS:
3146     case TCSETSW:
3147     case TCSETSF:
3148     case TIOCSETD:
3149     case TIOCHPCL:
3150     case TIOCSETP:
3151     case TIOCSETN:
3152     case TIOCEXCL:
3153     case TIOCNXCL:
3154     case TIOCFLUSH:
3155     case TIOCSETC:
3156     case TIOCLBIS:
3157     case TIOCLBIC:
3158     case TIOCLSET:
3159     case TIOCSBRK:
3160     case TIOCCBRK:
3161     case TIOCSDFR:
3162     case TIOCCDFR:
3163     case TIOCSLTC:

```

```

3164     case TIOCSTOP:
3165     case TIOCSTART:
3166     case TIOCSTI:
3167     case TIOCSGPRP:
3168     case TIOCMSSET:
3169     case TIOCMBIS:
3170     case TIOCMBIC:
3171     case TIOCREMOTE:
3172     case TIOCSIGNAL:
3173     case LDSETT:
3174     case LDSMAP:      /* Obsolete */
3175     case DIOCSETP:
3176     case I_FLUSH:
3177     case I_SRDOP:
3178     case I_SETSIG:
3179     case I_SWROPT:
3180     case I_FLUSHBAND:
3181     case I_SETCLTIME:
3182     case I_SERROPT:
3183     case I_ESETSIG:
3184     case FIONBIO:
3185     case FIOASYNC:
3186     case FIOSETOWN:
3187     case JBOOT:      /* Obsolete */
3188     case JTERM:      /* Obsolete */
3189     case JTIMOM:     /* Obsolete */
3190     case JZOMBOOT:  /* Obsolete */
3191     case JAGENT:    /* Obsolete */
3192     case JTRUN:     /* Obsolete */
3193     case JXTPROTO:  /* Obsolete */
3194         return (JCSETP);
3195     }
3197     return (JCGETP);
3198 }

3200 /*
3201  * ioctl for streams
3202  */
3203 int
3204 strioctl(struct vnode *vp, int cmd, intptr_t arg, int flag, int copyflag,
3205          cred_t *crp, int *rvalp)
3206 {
3207     struct stdata *stp;
3208     struct strcmd *scp;
3209     struct strioctl strioc;
3210     struct uio uio;
3211     struct iovec iov;
3212     int access;
3213     mblk_t *mp;
3214     int error = 0;
3215     int done = 0;
3216     ssize_t rmin, rmax;
3217     queue_t *wrq;
3218     queue_t *rdq;
3219     boolean_t kioctl = B_FALSE;
3220     uint32_t auditing = AU_AUDITING();

3222     if (flag & FKIOCTL) {
3223         copyflag = K_TO_K;
3224         kioctl = B_TRUE;
3225     }
3226     ASSERT(vp->v_stream);
3227     ASSERT(copyflag == U_TO_K || copyflag == K_TO_K);
3228     stp = vp->v_stream;

```

```

3230     TRACE_3(TR_FAC_STREAMS_FR, TR_IOCTL_ENTER,
3231            "strioctl:stp %p cmd %X arg %lX", stp, cmd, arg);

3233     /*
3234     * If the copy is kernel to kernel, make sure that the FNATIVE
3235     * flag is set. After this it would be a serious error to have
3236     * no model flag.
3237     */
3238     if (copyflag == K_TO_K)
3239         flag = (flag & ~FMODELS) | FNATIVE;

3241     ASSERT((flag & FMODELS) != 0);

3243     wrq = stp->sd_wrq;
3244     rdq = _RD(wrq);

3246     access = job_control_type(cmd);

3248     /* We should never see these here, should be handled by iwscn */
3249     if (cmd == SRIOCSREDIR || cmd == SRIOCISREDIR)
3250         return (EINVAL);

3252     mutex_enter(&stp->sd_lock);
3253     if ((access != -1) && ((error = i_straccess(stp, access)) != 0)) {
3254         mutex_exit(&stp->sd_lock);
3255         return (error);
3256     }
3257     mutex_exit(&stp->sd_lock);

3259     /*
3260     * Check for sgTTYb-related ioctls first, and complain as
3261     * necessary.
3262     */
3263     switch (cmd) {
3264     case TIOCGETP:
3265     case TIOCSETP:
3266     case TIOCSETN:
3267         if (sgTTYb_handling >= 2 && !sgTTYb_complaint) {
3268             sgTTYb_complaint = B_TRUE;
3269             cmn_err(CE_NOTE,
3270                  "application used obsolete TIOC[GS]ET");
3271         }
3272         if (sgTTYb_handling >= 3) {
3273             tsignal(curthread, SIGSYS);
3274             return (EIO);
3275         }
3276         break;
3277     }

3279     mutex_enter(&stp->sd_lock);

3281     switch (cmd) {
3282     case I_RECVFD:
3283     case I_E_RECVFD:
3284     case I_PEEK:
3285     case I_NREAD:
3286     case FIONREAD:
3287     case FIORDCHK:
3288     case I_ATMARK:
3289     case FIONBIO:
3290     case FIOASYNC:
3291         if (stp->sd_flag & (STRDERR|STPLEX)) {
3292             error = strgeterr(stp, STRDERR|STPLEX, 0);
3293             if (error != 0) {
3294                 mutex_exit(&stp->sd_lock);
3295                 return (error);

```

```

3296         }
3297     }
3298     break;

3300 default:
3301     if (stp->sd_flag & (STRDERR|STWRERR|STPLEX)) {
3302         error = strgeterr(stp, STRDERR|STWRERR|STPLEX, 0);
3303         if (error != 0) {
3304             mutex_exit(&stp->sd_lock);
3305             return (error);
3306         }
3307     }
3308 }

3310 mutex_exit(&stp->sd_lock);

3312 switch (cmd) {
3313 default:
3314     /*
3315      * The stream head has hardcoded knowledge of a
3316      * miscellaneous collection of terminal-, keyboard- and
3317      * mouse-related ioctls, enumerated below. This hardcoded
3318      * knowledge allows the stream head to automatically
3319      * convert transparent ioctl requests made by userland
3320      * programs into I_STR ioctls which many old STREAMS
3321      * modules and drivers require.
3322      *
3323      * No new ioctls should ever be added to this list.
3324      * Instead, the STREAMS module or driver should be written
3325      * to either handle transparent ioctls or require any
3326      * userland programs to use I_STR ioctls (by returning
3327      * EINVAL to any transparent ioctl requests).
3328      *
3329      * More importantly, removing ioctls from this list should
3330      * be done with the utmost care, since our STREAMS modules
3331      * and drivers *count* on the stream head performing this
3332      * conversion, and thus may panic while processing
3333      * transparent ioctl request for one of these ioctls (keep
3334      * in mind that third party modules and drivers may have
3335      * similar problems).
3336      */
3337     if (((cmd & IOCTYPE) == LDIOC) ||
3338         ((cmd & IOCTYPE) == tIOC) ||
3339         ((cmd & IOCTYPE) == TIOC) ||
3340         ((cmd & IOCTYPE) == KIOC) ||
3341         ((cmd & IOCTYPE) == MSIOC) ||
3342         ((cmd & IOCTYPE) == VUIOC)) {
3343         /*
3344          * The ioctl is a tty ioctl - set up strioc buffer
3345          * and call strdoioctl() to do the work.
3346          */
3347         if (stp->sd_flag & STRHUP)
3348             return (ENXIO);
3349         strioc.ic_cmd = cmd;
3350         strioc.ic_timeout = INFTIM;

3352         switch (cmd) {

3354         case TCXONC:
3355         case TCSBRK:
3356         case TCFLSH:
3357         case TCDSSET:
3358             {
3359                 int native_arg = (int)arg;
3360                 strioc.ic_len = sizeof (int);
3361                 strioc.ic_dp = (char *)&native_arg;

```

```

3362         return (strdoioctl(stp, &strioc, flag,
3363             K_TO_K, crp, rvalp));
3364     }

3366     case TCSETA:
3367     case TCSETAW:
3368     case TCSETAF:
3369         strioc.ic_len = sizeof (struct termio);
3370         strioc.ic_dp = (char *)arg;
3371         return (strdoioctl(stp, &strioc, flag,
3372             copyflag, crp, rvalp));

3374     case TCSETS:
3375     case TCSETSW:
3376     case TCSETSF:
3377         strioc.ic_len = sizeof (struct termios);
3378         strioc.ic_dp = (char *)arg;
3379         return (strdoioctl(stp, &strioc, flag,
3380             copyflag, crp, rvalp));

3382     case LDSETT:
3383         strioc.ic_len = sizeof (struct termcb);
3384         strioc.ic_dp = (char *)arg;
3385         return (strdoioctl(stp, &strioc, flag,
3386             copyflag, crp, rvalp));

3388     case TIOCSERP:
3389         strioc.ic_len = sizeof (struct sgttyb);
3390         strioc.ic_dp = (char *)arg;
3391         return (strdoioctl(stp, &strioc, flag,
3392             copyflag, crp, rvalp));

3394     case TIOCSTI:
3395         if ((flag & FREAD) == 0 &&
3396             secpolicy_sti(crp) != 0) {
3397             return (EPERM);
3398         }
3399         mutex_enter(&stp->sd_lock);
3400         mutex_enter(&curproc->p_spllock);
3401         if (stp->sd_sidp != curproc->p_sessp->s_sidp &&
3402             secpolicy_sti(crp) != 0) {
3403             mutex_exit(&curproc->p_spllock);
3404             mutex_exit(&stp->sd_lock);
3405             return (EACCES);
3406         }
3407         mutex_exit(&curproc->p_spllock);
3408         mutex_exit(&stp->sd_lock);

3410         strioc.ic_len = sizeof (char);
3411         strioc.ic_dp = (char *)arg;
3412         return (strdoioctl(stp, &strioc, flag,
3413             copyflag, crp, rvalp));

3415     case TIOCSWINSZ:
3416         strioc.ic_len = sizeof (struct winsize);
3417         strioc.ic_dp = (char *)arg;
3418         return (strdoioctl(stp, &strioc, flag,
3419             copyflag, crp, rvalp));

3421     case TIOCSSIZE:
3422         strioc.ic_len = sizeof (struct ttysize);
3423         strioc.ic_dp = (char *)arg;
3424         return (strdoioctl(stp, &strioc, flag,
3425             copyflag, crp, rvalp));

3427     case TIOCSOFTCAR:

```

```

3428     case KIOCTRANS:
3429     case KIOCTRANSABLE:
3430     case KIOCCMD:
3431     case KIOCSDIRECT:
3432     case KIOCSCOMPAT:
3433     case KIOCSKABORTEN:
3434     case KIOCSRPTDELAY:
3435     case KIOCSRPTRATE:
3436     case VUIDSFORMAT:
3437     case TIOCSPPS:
3438         strioc.ic_len = sizeof (int);
3439         strioc.ic_dp = (char *)arg;
3440         return (strdoioctl(stp, &strioc, flag,
3441             copyflag, crp, rvalp));
3442
3443     case KIOCSETKEY:
3444     case KIOCGETKEY:
3445         strioc.ic_len = sizeof (struct kiockey);
3446         strioc.ic_dp = (char *)arg;
3447         return (strdoioctl(stp, &strioc, flag,
3448             copyflag, crp, rvalp));
3449
3450     case KIOCSKEY:
3451     case KIOCGKEY:
3452         strioc.ic_len = sizeof (struct kiockeymap);
3453         strioc.ic_dp = (char *)arg;
3454         return (strdoioctl(stp, &strioc, flag,
3455             copyflag, crp, rvalp));
3456
3457     case KIOCSLED:
3458         /* arg is a pointer to char */
3459         strioc.ic_len = sizeof (char);
3460         strioc.ic_dp = (char *)arg;
3461         return (strdoioctl(stp, &strioc, flag,
3462             copyflag, crp, rvalp));
3463
3464     case MSIOSETPARMS:
3465         strioc.ic_len = sizeof (Ms_parms);
3466         strioc.ic_dp = (char *)arg;
3467         return (strdoioctl(stp, &strioc, flag,
3468             copyflag, crp, rvalp));
3469
3470     case VUIDSADDR:
3471     case VUIDGADDR:
3472         strioc.ic_len = sizeof (struct void_addr_probe);
3473         strioc.ic_dp = (char *)arg;
3474         return (strdoioctl(stp, &strioc, flag,
3475             copyflag, crp, rvalp));
3476
3477     /*
3478     * These M_IOCTL's don't require any data to be sent
3479     * downstream, and the driver will allocate and link
3480     * on its own mblk_t upon M_IOCACK -- thus we set
3481     * ic_len to zero and set ic_dp to arg so we know
3482     * where to copyout to later.
3483     */
3484     case TIOCGSOFTCAR:
3485     case TIOCGWINSZ:
3486     case TIOCGSIZE:
3487     case KIOCGTRANS:
3488     case KIOCGTRANSABLE:
3489     case KIOCTYPE:
3490     case KIOCGDIRECT:
3491     case KIOCGCOMPAT:
3492     case KIOCLAYOUT:
3493     case KIOCGLED:

```

```

3494     case MSIOGETPARMS:
3495     case MSIOBUTTONS:
3496     case VUIDGFORMAT:
3497     case TIOCGPPS:
3498     case TIOCGPPSEV:
3499     case TCGETA:
3500     case TCGETS:
3501     case LDGETT:
3502     case TIOCGETP:
3503     case KIOCGRPTDELAY:
3504     case KIOCGRPTRATE:
3505         strioc.ic_len = 0;
3506         strioc.ic_dp = (char *)arg;
3507         return (strdoioctl(stp, &strioc, flag,
3508             copyflag, crp, rvalp));
3509     }
3510 }
3511
3512 /*
3513  * Unknown cmd - send it down as a transparent ioctl.
3514  */
3515     strioc.ic_cmd = cmd;
3516     strioc.ic_timeout = INFTIM;
3517     strioc.ic_len = TRANSPARENT;
3518     strioc.ic_dp = (char *)&arg;
3519
3520     return (strdoioctl(stp, &strioc, flag, copyflag, crp, rvalp));
3521
3522     case I_STR:
3523         /*
3524          * Stream ioctl. Read in an strioc buffer from the user
3525          * along with any data specified and send it downstream.
3526          * Strdoioctl will wait allow only one ioctl message at
3527          * a time, and waits for the acknowledgement.
3528          */
3529         if (stp->sd_flag & STRHUP)
3530             return (ENXIO);
3531
3532         error = strcopyin_strioc((void *)arg, &strioc, flag,
3533             copyflag);
3534         if (error != 0)
3535             return (error);
3536
3537         if ((strioc.ic_len < 0) || (strioc.ic_timeout < -1))
3538             return (EINVAL);
3539
3540         access = job_control_type(strioc.ic_cmd);
3541         mutex_enter(&stp->sd_lock);
3542         if ((access != -1) &&
3543             ((error = i_straccess(stp, access)) != 0)) {
3544             mutex_exit(&stp->sd_lock);
3545             return (error);
3546         }
3547         mutex_exit(&stp->sd_lock);
3548
3549     /*
3550     * The I_STR facility provides a trap door for malicious
3551     * code to send down bogus streamio(7I) ioctl commands to
3552     * unsuspecting STREAMS modules and drivers which expect to
3553     * only get these messages from the stream head.
3554     * Explicitly prohibit any streamio ioctls which can be
3555     * passed downstream by the stream head. Note that we do
3556     * not block all streamio ioctls because the ioctl
3557     * numberspace is not well managed and thus it's possible
3558     * that a module or driver's ioctl numbers may accidentally

```

```

3560         * collide with them.
3561         */
3562         switch (strioc.ic_cmd) {
3563         case I_LINK:
3564         case I_PLINK:
3565         case I_UNLINK:
3566         case I_PUNLINK:
3567         case _I_GETPEERCRED:
3568         case _I_PLINK_LH:
3569             return (EINVAL);
3570         }
3572         error = strdoioctl(stp, &strioc, flag, copyflag, crp, rvalp);
3573         if (error == 0) {
3574             error = strcopyout_strioc(&strioc, (void *)arg,
3575                                     flag, copyflag);
3576         }
3577         return (error);
3579     case _I_CMD:
3580         /*
3581          * Like I_STR, but without using M_IOC* messages and without
3582          * copyins/copyouts beyond the passed-in argument.
3583          */
3584         if (stp->sd_flag & STRHUP)
3585             return (ENXIO);
3587         if ((scp = kmem_alloc(sizeof (strcmd_t), KM_NOSLEEP)) == NULL)
3588             return (ENOMEM);
3590         if (copyin((void *)arg, scp, sizeof (strcmd_t))) {
3591             kmem_free(scp, sizeof (strcmd_t));
3592             return (EFAULT);
3593         }
3595         access = job_control_type(scp->sc_cmd);
3596         mutex_enter(&stp->sd_lock);
3597         if (access != -1 && (error = i_straccess(stp, access)) != 0) {
3598             mutex_exit(&stp->sd_lock);
3599             kmem_free(scp, sizeof (strcmd_t));
3600             return (error);
3601         }
3602         mutex_exit(&stp->sd_lock);
3604         *rvalp = 0;
3605         if ((error = strdocmd(stp, scp, crp)) == 0) {
3606             if (copyout(scp, (void *)arg, sizeof (strcmd_t)))
3607                 error = EFAULT;
3608         }
3609         kmem_free(scp, sizeof (strcmd_t));
3610         return (error);
3612     case I_NREAD:
3613         /*
3614          * Return number of bytes of data in first message
3615          * in queue in "arg" and return the number of messages
3616          * in queue in return value.
3617          */
3618     {
3619         size_t size;
3620         int retval;
3621         int count = 0;
3623         mutex_enter(QLOCK(rdq));
3625         size = msgdsz(rdq->q_first);

```

```

3626         for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3627             count++;
3629         mutex_exit(QLOCK(rdq));
3630         if (stp->sd_struirdq) {
3631             infod_t infod;
3633             infod.d_cmd = INFOD_COUNT;
3634             infod.d_count = 0;
3635             if (count == 0) {
3636                 infod.d_cmd |= INFOD_FIRSTBYTES;
3637                 infod.d_bytes = 0;
3638             }
3639             infod.d_res = 0;
3640             (void) infonext(rdq, &infod);
3641             count += infod.d_count;
3642             if (infod.d_res & INFOD_FIRSTBYTES)
3643                 size = infod.d_bytes;
3644         }
3646         /*
3647          * Drop down from size_t to the "int" required by the
3648          * interface. Cap at INT_MAX.
3649          */
3650         retval = MIN(size, INT_MAX);
3651         error = strcopyout(&retval, (void *)arg, sizeof (retval),
3652                           copyflag);
3653         if (!error)
3654             *rvalp = count;
3655         return (error);
3656     }
3658     case FIONREAD:
3659         /*
3660          * Return number of bytes of data in all data messages
3661          * in queue in "arg".
3662          */
3663     {
3664         size_t size = 0;
3665         int retval;
3667         mutex_enter(QLOCK(rdq));
3668         for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3669             size += msgdsz(mp);
3670         mutex_exit(QLOCK(rdq));
3672         if (stp->sd_struirdq) {
3673             infod_t infod;
3675             infod.d_cmd = INFOD_BYTES;
3676             infod.d_res = 0;
3677             infod.d_bytes = 0;
3678             (void) infonext(rdq, &infod);
3679             size += infod.d_bytes;
3680         }
3682         /*
3683          * Drop down from size_t to the "int" required by the
3684          * interface. Cap at INT_MAX.
3685          */
3686         retval = MIN(size, INT_MAX);
3687         error = strcopyout(&retval, (void *)arg, sizeof (retval),
3688                           copyflag);
3690         *rvalp = 0;
3691         return (error);

```

```

3692     }
3693     case FIORDCHK:
3694         /*
3695          * FIORDCHK does not use arg value (like FIONREAD),
3696          * instead a count is returned. I_NREAD value may
3697          * not be accurate but safe. The real thing to do is
3698          * to add the msgdsizes of all data messages until
3699          * a non-data message.
3700          */
3701     {
3702         size_t size = 0;
3703
3704         mutex_enter(QLOCK(rdq));
3705         for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
3706             size += msgdsizes(mp);
3707         mutex_exit(QLOCK(rdq));
3708
3709         if (stp->sd_striordq) {
3710             infod_t infod;
3711
3712             infod.d_cmd = INFOD_BYTES;
3713             infod.d_res = 0;
3714             infod.d_bytes = 0;
3715             (void) infonext(rdq, &infod);
3716             size += infod.d_bytes;
3717         }
3718
3719         /*
3720          * Since ioctl returns an int, and memory sizes under
3721          * LP64 may not fit, we return INT_MAX if the count was
3722          * actually greater.
3723          */
3724         *rvalp = MIN(size, INT_MAX);
3725         return (0);
3726     }
3727
3728     case I_FIND:
3729         /*
3730          * Get module name.
3731          */
3732     {
3733         char mname[FMNAMESZ + 1];
3734         queue_t *q;
3735
3736         error = (copyflag & U_TO_K ? copyinstr : copystr)((void *)arg,
3737             mname, FMNAMESZ + 1, NULL);
3738         if (error)
3739             return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);
3740
3741         /*
3742          * Return EINVAL if we're handed a bogus module name.
3743          */
3744         if (fmodsw_find(mname, FMODSW_LOAD) == NULL) {
3745             TRACE_0(TR_FAC_STREAMS_FR,
3746                 TR_I_CANT_FIND, "couldn't I_FIND");
3747             return (EINVAL);
3748         }
3749
3750         *rvalp = 0;
3751
3752         /* Look downstream to see if module is there. */
3753         claimstr(stp->sd_wrq);
3754         for (q = stp->sd_wrq->q_next; q; q = q->q_next) {
3755             if (q->q_flag & QREADR) {
3756                 q = NULL;
3757                 break;

```

```

3758     }
3759         if (strcmp(mname, Q2NAME(q)) == 0)
3760             break;
3761     }
3762     releasestr(stp->sd_wrq);
3763
3764     *rvalp = (q ? 1 : 0);
3765     return (error);
3766 }
3767
3768     case I_PUSH:
3769     case __I_PUSH_NOCTTY:
3770         /*
3771          * Push a module.
3772          * For the case __I_PUSH_NOCTTY push a module but
3773          * do not allocate controlling tty. See bugid 4025044
3774          */
3775     {
3776         char mname[FMNAMESZ + 1];
3777         fmodsw_impl_t *fp;
3778         dev_t dummydev;
3779
3780         if (stp->sd_flag & STRHUP)
3781             return (ENXIO);
3782
3783         /*
3784          * Get module name and look up in fmodsw.
3785          */
3786         error = (copyflag & U_TO_K ? copyinstr : copystr)((void *)arg,
3787             mname, FMNAMESZ + 1, NULL);
3788         if (error)
3789             return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);
3790
3791         if ((fp = fmodsw_find(mname, FMODSW_HOLD | FMODSW_LOAD)) ==
3792             NULL)
3793             return (EINVAL);
3794
3795         TRACE_2(TR_FAC_STREAMS_FR, TR_I_PUSH,
3796             "I_PUSH:fp %p stp %p", fp, stp);
3797
3798         if (error = strstartplumb(stp, flag, cmd)) {
3799             fmodsw_rele(fp);
3800             return (error);
3801         }
3802
3803         /*
3804          * See if any more modules can be pushed on this stream.
3805          * Note that this check must be done after strstartplumb()
3806          * since otherwise multiple threads issuing I_PUSHes on
3807          * the same stream will be able to exceed nstrpush.
3808          */
3809         mutex_enter(&stp->sd_lock);
3810         if (stp->sd_pushcnt >= nstrpush) {
3811             fmodsw_rele(fp);
3812             strendplumb(stp);
3813             mutex_exit(&stp->sd_lock);
3814             return (EINVAL);
3815         }
3816         mutex_exit(&stp->sd_lock);
3817
3818         /*
3819          * Push new module and call its open routine
3820          * via qattach(). Modules don't change device
3821          * numbers, so just ignore dummydev here.
3822          */
3823

```

```

3824 dummydev = vp->v_rdev;
3825 if ((error = qattach(rdq, &dummydev, 0, crp, fp,
3826     B_FALSE)) == 0) {
3827     if (vp->v_type == VCHR && /* sorry, no pipes allowed */
3828         (cmd == I_PUSH) && (stp->sd_flag & STRISTTY)) {
3829         /*
3830          * try to allocate it as a controlling terminal
3831          */
3832         (void) strctty(stp);
3833     }
3834 }
3835
3836 mutex_enter(&stp->sd_lock);
3837
3838 /*
3839  * As a performance concern we are caching the values of
3840  * q_minpsz and q_maxpsz of the module below the stream
3841  * head in the stream head.
3842  */
3843 mutex_enter(QLOCK(stp->sd_wrq->q_next));
3844 rmin = stp->sd_wrq->q_next->q_minpsz;
3845 rmax = stp->sd_wrq->q_next->q_maxpsz;
3846 mutex_exit(QLOCK(stp->sd_wrq->q_next));
3847
3848 /* Do this processing here as a performance concern */
3849 if (strmsgsz != 0) {
3850     if (rmax == INFPSZ)
3851         rmax = strmsgsz;
3852     else {
3853         if (vp->v_type == VFIFO)
3854             rmax = MIN(PIPE_BUF, rmax);
3855         else
3856             rmax = MIN(strmsgsz, rmax);
3857     }
3858 }
3859
3860 mutex_enter(QLOCK(wrq));
3861 stp->sd_qn_minpsz = rmin;
3862 stp->sd_qn_maxpsz = rmax;
3863 mutex_exit(QLOCK(wrq));
3864
3865 strendplumb(stp);
3866 mutex_exit(&stp->sd_lock);
3867 return (error);
3868 }
3869
3870 case I_POP:
3871 {
3872     queue_t *q;
3873
3874     if (stp->sd_flag & STRHUP)
3875         return (ENXIO);
3876     if (!wrq->q_next) /* for broken pipes */
3877         return (EINVAL);
3878
3879     if (error = strstartplumb(stp, flag, cmd))
3880         return (error);
3881
3882     /*
3883      * If there is an anchor on this stream and popping
3884      * the current module would attempt to pop through the
3885      * anchor, then disallow the pop unless we have sufficient
3886      * privileges; take the cheapest (non-locking) check
3887      * first.
3888      */
3889     if (secpolicy_ip_config(crp, B_TRUE) != 0 ||
3890         (stp->sd_anchorzone != crgetzoneid(crp))) {

```

```

3890         mutex_enter(&stp->sd_lock);
3891         /*
3892          * Anchors only apply if there's at least one
3893          * module on the stream (sd_pushcnt > 0).
3894          */
3895         if (stp->sd_pushcnt > 0 &&
3896             stp->sd_pushcnt == stp->sd_anchor &&
3897             stp->sd_vnode->v_type != VFIFO) {
3898             strendplumb(stp);
3899             mutex_exit(&stp->sd_lock);
3900             if (stp->sd_anchorzone != crgetzoneid(crp))
3901                 return (EINVAL);
3902             /* Audit and report error */
3903             return (secpolicy_ip_config(crp, B_FALSE));
3904         }
3905         mutex_exit(&stp->sd_lock);
3906     }
3907
3908     q = wrq->q_next;
3909     TRACE_2(TR_FAC_STREAMS_FR, TR_I_POP,
3910         "I_POP:%p from %p", q, stp);
3911     if (q->q_next == NULL || (q->q_flag & (QREADR|QISDRV))) {
3912         error = EINVAL;
3913     } else {
3914         qdetach(_RD(q), 1, flag, crp, B_FALSE);
3915         error = 0;
3916     }
3917     mutex_enter(&stp->sd_lock);
3918
3919     /*
3920      * As a performance concern we are caching the values of
3921      * q_minpsz and q_maxpsz of the module below the stream
3922      * head in the stream head.
3923      */
3924     mutex_enter(QLOCK(wrq->q_next));
3925     rmin = wrq->q_next->q_minpsz;
3926     rmax = wrq->q_next->q_maxpsz;
3927     mutex_exit(QLOCK(wrq->q_next));
3928
3929     /* Do this processing here as a performance concern */
3930     if (strmsgsz != 0) {
3931         if (rmax == INFPSZ)
3932             rmax = strmsgsz;
3933         else {
3934             if (vp->v_type == VFIFO)
3935                 rmax = MIN(PIPE_BUF, rmax);
3936             else
3937                 rmax = MIN(strmsgsz, rmax);
3938         }
3939     }
3940
3941     mutex_enter(QLOCK(wrq));
3942     stp->sd_qn_minpsz = rmin;
3943     stp->sd_qn_maxpsz = rmax;
3944     mutex_exit(QLOCK(wrq));
3945
3946     /* If we popped through the anchor, then reset the anchor. */
3947     if (stp->sd_pushcnt < stp->sd_anchor) {
3948         stp->sd_anchor = 0;
3949         stp->sd_anchorzone = 0;
3950     }
3951     strendplumb(stp);
3952     mutex_exit(&stp->sd_lock);
3953     return (error);
3954 }
3955
3956 case _I_MUXID2FD:

```

```

3956     {
3957         /*
3958          * Create a fd for a I_PLINK'ed lower stream with a given
3959          * muxid. With the fd, application can send down ioctls,
3960          * like I_LIST, to the previously I_PLINK'ed stream. Note
3961          * that after getting the fd, the application has to do an
3962          * I_PUNLINK on the muxid before it can do any operation
3963          * on the lower stream. This is required by spec1170.
3964          *
3965          * The fd used to do this ioctl should point to the same
3966          * controlling device used to do the I_PLINK. If it uses
3967          * a different stream or an invalid muxid, I_MUXID2FD will
3968          * fail. The error code is set to EINVAL.
3969          *
3970          * The intended use of this interface is the following.
3971          * An application I_PLINK'ed a stream and exits. The fd
3972          * to the lower stream is gone. Another application
3973          * wants to get a fd to the lower stream, it uses I_MUXID2FD.
3974          */
3975         int muxid = (int)arg;
3976         int fd;
3977         linkinfo_t *linkp;
3978         struct file *fp;
3979         netstack_t *ns;
3980         str_stack_t *ss;
3981
3982         /*
3983          * Do not allow the wildcard muxid. This ioctl is not
3984          * intended to find arbitrary link.
3985          */
3986         if (muxid == 0) {
3987             return (EINVAL);
3988         }
3989
3990         ns = netstack_find_by_cred(crp);
3991         ASSERT(ns != NULL);
3992         ss = ns->netstack_str;
3993         ASSERT(ss != NULL);
3994
3995         mutex_enter(&muxifier);
3996         linkp = findlinks(vp->v_stream, muxid, LINKPERSIST, ss);
3997         if (linkp == NULL) {
3998             mutex_exit(&muxifier);
3999             netstack_rele(ss->ss_netstack);
4000             return (EINVAL);
4001         }
4002
4003         if ((fd = ufalloc(0)) == -1) {
4004             mutex_exit(&muxifier);
4005             netstack_rele(ss->ss_netstack);
4006             return (EMFILE);
4007         }
4008         fp = linkp->li_fpdwn;
4009         mutex_enter(&fp->f_tlock);
4010         fp->f_count++;
4011         mutex_exit(&fp->f_tlock);
4012         mutex_exit(&muxifier);
4013         setf(fd, fp);
4014         *rvalp = fd;
4015         netstack_rele(ss->ss_netstack);
4016         return (0);
4017     }
4018
4019     case _I_INSERT:
4020     {
4021         /*

```

```

4022          * To insert a module to a given position in a stream.
4023          * In the first release, only allow privileged user
4024          * to use this ioctl. Furthermore, the insert is only allowed
4025          * below an anchor if the zoneid is the same as the zoneid
4026          * which created the anchor.
4027          *
4028          * Note that we do not plan to support this ioctl
4029          * on pipes in the first release. We want to learn more
4030          * about the implications of these ioctls before extending
4031          * their support. And we do not think these features are
4032          * valuable for pipes.
4033          */
4034         STRUCT_DECL(strmodconf, strmodinsert);
4035         char mod_name[FMNAMESZ + 1];
4036         fmodsw_impl_t *fp;
4037         dev_t dummydev;
4038         queue_t *tmp_wrq;
4039         int pos;
4040         boolean_t is_insert;
4041
4042         STRUCT_INIT(strmodinsert, flag);
4043         if (stp->sd_flag & STRHUP)
4044             return (ENXIO);
4045         if (STRMATED(stp))
4046             return (EINVAL);
4047         if ((error = secpolicy_net_config(crp, B_FALSE)) != 0)
4048             return (error);
4049         if (stp->sd_anchor != 0 &&
4050             stp->sd_anchorzone != crgetzoneid(crp))
4051             return (EINVAL);
4052
4053         error = strcpyin((void *)arg, STRUCT_BUF(strmodinsert),
4054             STRUCT_SIZE(strmodinsert), copyflag);
4055         if (error)
4056             return (error);
4057
4058         /*
4059          * Get module name and look up in fmodsw.
4060          */
4061         error = (copyflag & U_TO_K ? copyinstr :
4062             copystr)(STRUCT_FGETP(strmodinsert, mod_name),
4063             mod_name, FMNAMESZ + 1, NULL);
4064         if (error)
4065             return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);
4066
4067         if ((fp = fmodsw_find(mod_name, FMODSW_HOLD | FMODSW_LOAD)) ==
4068             NULL)
4069             return (EINVAL);
4070
4071         if (error = strstartplumb(stp, flag, cmd)) {
4072             fmodsw_rele(fp);
4073             return (error);
4074         }
4075
4076         /*
4077          * Is this _I_INSERT just like an I_PUSH? We need to know
4078          * this because we do some optimizations if this is a
4079          * module being pushed.
4080          */
4081         pos = STRUCT_FGET(strmodinsert, pos);
4082         is_insert = (pos != 0);
4083
4084         /*
4085          * Make sure pos is valid. Even though it is not an I_PUSH,
4086          * we impose the same limit on the number of modules in a
4087          * stream.

```

```

4088     */
4089     mutex_enter(&stp->sd_lock);
4090     if (stp->sd_pushcnt >= nstrpush || pos < 0 ||
4091         pos > stp->sd_pushcnt) {
4092         fmodsw_rele(fp);
4093         strendplumb(stp);
4094         mutex_exit(&stp->sd_lock);
4095         return (EINVAL);
4096     }
4097     if (stp->sd_anchor != 0) {
4098         /*
4099          * Is this insert below the anchor?
4100          * Pushcnt hasn't been increased yet hence
4101          * we test for greater than here, and greater or
4102          * equal after qattach.
4103          */
4104         if (pos > (stp->sd_pushcnt - stp->sd_anchor) &&
4105             stp->sd_anchorzone != crgetzoneid(crp)) {
4106             fmodsw_rele(fp);
4107             strendplumb(stp);
4108             mutex_exit(&stp->sd_lock);
4109             return (EPERM);
4110         }
4111     }
4112
4113     mutex_exit(&stp->sd_lock);
4114
4115     /*
4116     * First find the correct position this module to
4117     * be inserted. We don't need to call claimstr()
4118     * as the stream should not be changing at this point.
4119     *
4120     * Insert new module and call its open routine
4121     * via qattach(). Modules don't change device
4122     * numbers, so just ignore dummydev here.
4123     */
4124     for (tmp_wrq = stp->sd_wrq; pos > 0;
4125         tmp_wrq = tmp_wrq->q_next, pos--) {
4126         ASSERT(SAMESTR(tmp_wrq));
4127     }
4128     dummydev = vp->v_rdev;
4129     if ((error = qattach(_RD(tmp_wrq), &dummydev, 0, crp,
4130         fp, is_insert)) != 0) {
4131         mutex_enter(&stp->sd_lock);
4132         strendplumb(stp);
4133         mutex_exit(&stp->sd_lock);
4134         return (error);
4135     }
4136
4137     mutex_enter(&stp->sd_lock);
4138
4139     /*
4140     * As a performance concern we are caching the values of
4141     * q_minpsz and q_maxpsz of the module below the stream
4142     * head in the stream head.
4143     */
4144     if (!is_insert) {
4145         mutex_enter(QLOCK(stp->sd_wrq->q_next));
4146         rmin = stp->sd_wrq->q_next->q_minpsz;
4147         rmax = stp->sd_wrq->q_next->q_maxpsz;
4148         mutex_exit(QLOCK(stp->sd_wrq->q_next));
4149
4150         /* Do this processing here as a performance concern */
4151         if (strmsgsz != 0) {
4152             if (rmax == INFP SZ) {
4153                 rmax = strmsgsz;

```

```

4154     } else {
4155         rmax = MIN(strmsgsz, rmax);
4156     }
4157 }
4158
4159     mutex_enter(QLOCK(wrq));
4160     stp->sd_qn_minpsz = rmin;
4161     stp->sd_qn_maxpsz = rmax;
4162     mutex_exit(QLOCK(wrq));
4163 }
4164
4165     /*
4166     * Need to update the anchor value if this module is
4167     * inserted below the anchor point.
4168     */
4169     if (stp->sd_anchor != 0) {
4170         pos = STRUCT_FGET(strmodinsert, pos);
4171         if (pos >= (stp->sd_pushcnt - stp->sd_anchor))
4172             stp->sd_anchor++;
4173     }
4174
4175     strendplumb(stp);
4176     mutex_exit(&stp->sd_lock);
4177     return (0);
4178 }
4179
4180     case _I_REMOVE:
4181     {
4182         /*
4183          * To remove a module with a given name in a stream. The
4184          * caller of this ioctl needs to provide both the name and
4185          * the position of the module to be removed. This eliminates
4186          * the ambiguity of removal if a module is inserted/pushed
4187          * multiple times in a stream. In the first release, only
4188          * allow privileged user to use this ioctl.
4189          * Furthermore, the remove is only allowed
4190          * below an anchor if the zoneid is the same as the zoneid
4191          * which created the anchor.
4192          *
4193          * Note that we do not plan to support this ioctl
4194          * on pipes in the first release. We want to learn more
4195          * about the implications of these ioctls before extending
4196          * their support. And we do not think these features are
4197          * valuable for pipes.
4198          *
4199          * Also note that _I_REMOVE cannot be used to remove a
4200          * driver or the stream head.
4201          */
4202         STRUCT_DECL(strmodconf, strmodremove);
4203         queue_t *q;
4204         int pos;
4205         char mod_name[FMNAMESZ + 1];
4206         boolean_t is_remove;
4207
4208         STRUCT_INIT(strmodremove, flag);
4209         if (stp->sd_flag & STRHUP)
4210             return (ENXIO);
4211         if (STRMATED(stp))
4212             return (EINVAL);
4213         if ((error = secpolicy_net_config(crp, B_FALSE)) != 0)
4214             return (error);
4215         if (stp->sd_anchor != 0 &&
4216             stp->sd_anchorzone != crgetzoneid(crp))
4217             return (EINVAL);
4218
4219         error = strcpyin((void *)arg, STRUCT_BUF(strmodremove),

```

```

4220     STRUCT_SIZE(strmodremove), copyflag);
4221     if (error)
4222         return (error);
4224     error = (copyflag & U_TO_K ? copyinstr :
4225             copystr)(STRUCT_FGETP(strmodremove, mod_name),
4226                    mod_name, FMNAMESZ + 1, NULL);
4227     if (error)
4228         return ((error == ENAMETOOLONG) ? EINVAL : EFAULT);
4230     if ((error = strstartplumb(stp, flag, cmd)) != 0)
4231         return (error);
4233     /*
4234      * Match the name of given module to the name of module at
4235      * the given position.
4236      */
4237     pos = STRUCT_FGET(strmodremove, pos);
4239     is_remove = (pos != 0);
4240     for (q = stp->sd_wrq->q_next; SAMESTR(q) && pos > 0;
4241          q = q->q_next, pos--)
4242     ;
4243     if (pos > 0 || !SAMESTR(q) ||
4244         strcmp(Q2NAME(q), mod_name) != 0) {
4245         mutex_enter(&stp->sd_lock);
4246         strendplumb(stp);
4247         mutex_exit(&stp->sd_lock);
4248         return (EINVAL);
4249     }
4251     /*
4252      * If the position is at or below an anchor, then the zoneid
4253      * must match the zoneid that created the anchor.
4254      */
4255     if (stp->sd_anchor != 0) {
4256         pos = STRUCT_FGET(strmodremove, pos);
4257         if (pos >= (stp->sd_pushcnt - stp->sd_anchor) &&
4258             stp->sd_anchorzone != crgetzoneid(crp)) {
4259             mutex_enter(&stp->sd_lock);
4260             strendplumb(stp);
4261             mutex_exit(&stp->sd_lock);
4262             return (EPERM);
4263         }
4264     }
4267     ASSERT(!(q->q_flag & QREADR));
4268     qdetach(_RD(q), 1, flag, crp, is_remove);
4270     mutex_enter(&stp->sd_lock);
4272     /*
4273      * As a performance concern we are caching the values of
4274      * q_minpsz and q_maxpsz of the module below the stream
4275      * head in the stream head.
4276      */
4277     if (!is_remove) {
4278         mutex_enter(QLOCK(wrq->q_next));
4279         rmin = wrq->q_next->q_minpsz;
4280         rmax = wrq->q_next->q_maxpsz;
4281         mutex_exit(QLOCK(wrq->q_next));
4283         /* Do this processing here as a performance concern */
4284         if (strmsgsz != 0) {
4285             if (rmax == INFP SZ)

```

```

4286         rmax = strmsgsz;
4287     else {
4288         if (vp->v_type == VFIFO)
4289             rmax = MIN(PIPE_BUF, rmax);
4290         else
4291             rmax = MIN(strmsgsz, rmax);
4292     }
4294     mutex_enter(QLOCK(wrq));
4295     stp->sd_qn_minpsz = rmin;
4296     stp->sd_qn_maxpsz = rmax;
4297     mutex_exit(QLOCK(wrq));
4298 }
4300 /*
4301  * Need to update the anchor value if this module is removed
4302  * at or below the anchor point. If the removed module is at
4303  * the anchor point, remove the anchor for this stream if
4304  * there is no module above the anchor point. Otherwise, if
4305  * the removed module is below the anchor point, decrement the
4306  * anchor point by 1.
4307  */
4308     if (stp->sd_anchor != 0) {
4309         pos = STRUCT_FGET(strmodremove, pos);
4310         if (pos == stp->sd_pushcnt - stp->sd_anchor + 1)
4311             stp->sd_anchor = 0;
4312         else if (pos > (stp->sd_pushcnt - stp->sd_anchor + 1))
4313             stp->sd_anchor--;
4314     }
4316     strendplumb(stp);
4317     mutex_exit(&stp->sd_lock);
4318     return (0);
4319 }
4321     case I_ANCHOR:
4322         /*
4323          * Set the anchor position on the stream to reside at
4324          * the top module (in other words, the top module
4325          * cannot be popped). Anchors with a FIFO make no
4326          * obvious sense, so they're not allowed.
4327          */
4328         mutex_enter(&stp->sd_lock);
4330         if (stp->sd_vnode->v_type == VFIFO) {
4331             mutex_exit(&stp->sd_lock);
4332             return (EINVAL);
4333         }
4334         /* Only allow the same zoneid to update the anchor */
4335         if (stp->sd_anchor != 0 &&
4336             stp->sd_anchorzone != crgetzoneid(crp)) {
4337             mutex_exit(&stp->sd_lock);
4338             return (EINVAL);
4339         }
4340         stp->sd_anchor = stp->sd_pushcnt;
4341         stp->sd_anchorzone = crgetzoneid(crp);
4342         mutex_exit(&stp->sd_lock);
4343         return (0);
4345     case I_LOOK:
4346         /*
4347          * Get name of first module downstream.
4348          * If no module, return an error.
4349          */
4350         claimstr(wrq);
4351         if (!_SAMESTR(wrq) && wrq->q_next->q_next != NULL) {

```

```

4352         char *name = Q2NAME(wrq->q_next);
4353
4354         error = strcpyout(name, (void *)arg, strlen(name) + 1,
4355             copyflag);
4356         releasestr(wrq);
4357         return (error);
4358     }
4359     releasestr(wrq);
4360     return (EINVAL);
4361
4362 case I_LINK:
4363 case I_PLINK:
4364     /*
4365      * Link a multiplexor.
4366      */
4367     return (mlink(vp, cmd, (int)arg, crp, rvalp, 0));
4368
4369 case I_PLINK_LH:
4370     /*
4371      * Link a multiplexor: Call must originate from kernel.
4372      */
4373     if (kiocctl)
4374         return (ldi_mlink_lh(vp, cmd, arg, crp, rvalp));
4375
4376     return (EINVAL);
4377 case I_UNLINK:
4378 case I_PUNLINK:
4379     /*
4380      * Unlink a multiplexor.
4381      * If arg is -1, unlink all links for which this is the
4382      * controlling stream. Otherwise, arg is an index number
4383      * for a link to be removed.
4384      */
4385     {
4386         struct linkinfo *linkp;
4387         int native_arg = (int)arg;
4388         int type;
4389         netstack_t *ns;
4390         str_stack_t *ss;
4391
4392         TRACE_1(TR_FAC_STREAMS_FR,
4393             TR_I_UNLINK, "I_UNLINK/I_PUNLINK:%p", stp);
4394         if (vp->v_type == VFIFO) {
4395             return (EINVAL);
4396         }
4397         if (cmd == I_UNLINK)
4398             type = LINKNORMAL;
4399         else /* I_PUNLINK */
4400             type = LINKPERSIST;
4401         if (native_arg == 0) {
4402             return (EINVAL);
4403         }
4404         ns = netstack_find_by_cred(crp);
4405         ASSERT(ns != NULL);
4406         ss = ns->netstack_str;
4407         ASSERT(ss != NULL);
4408
4409         if (native_arg == MUXID_ALL)
4410             error = munlinkall(stp, type, crp, rvalp, ss);
4411         else {
4412             mutex_enter(&muxifier);
4413             if (!(linkp = findlinks(stp, (int)arg, type, ss))) {
4414                 /* invalid user supplied index number */
4415                 mutex_exit(&muxifier);
4416                 netstack_rele(ss->ss_netstack);
4417                 return (EINVAL);

```

```

4418     }
4419     /* munlink drops the muxifier lock */
4420     error = munlink(stp, linkp, type, crp, rvalp, ss);
4421 }
4422 netstack_rele(ss->ss_netstack);
4423 return (error);
4424 }
4425
4426 case I_FLUSH:
4427     /*
4428      * send a flush message downstream
4429      * flush message can indicate
4430      * FLUSHR - flush read queue
4431      * FLUSHW - flush write queue
4432      * FLUSHRW - flush read/write queue
4433      */
4434     if (stp->sd_flag & STRHUP)
4435         return (ENXIO);
4436     if (arg & ~FLUSHRW)
4437         return (EINVAL);
4438
4439     for (;;) {
4440         if (putnextctl1(stp->sd_wrq, M_FLUSH, (int)arg)) {
4441             break;
4442         }
4443         if (error = strwaitbuf(1, BPRI_HI)) {
4444             return (error);
4445         }
4446     }
4447
4448     /*
4449      * Send down an unsupported ioctl and wait for the nack
4450      * in order to allow the M_FLUSH to propagate back
4451      * up to the stream head.
4452      * Replaces if (qready()) runqueues();
4453     */
4454     strioc.ic_cmd = -1; /* The unsupported ioctl */
4455     strioc.ic_timeout = 0;
4456     strioc.ic_len = 0;
4457     strioc.ic_dp = NULL;
4458     (void) strdoioctl(stp, &strioc, flag, K_TO_K, crp, rvalp);
4459     *rvalp = 0;
4460     return (0);
4461
4462 case I_FLUSHBAND:
4463     {
4464         struct bandinfo binfo;
4465
4466         error = strcpyin((void *)arg, &binfo, sizeof (binfo),
4467             copyflag);
4468         if (error)
4469             return (error);
4470         if (stp->sd_flag & STRHUP)
4471             return (ENXIO);
4472         if (binfo.bi_flag & ~FLUSHRW)
4473             return (EINVAL);
4474         while (!(mp = allocb(2, BPRI_HI))) {
4475             if (error = strwaitbuf(2, BPRI_HI))
4476                 return (error);
4477         }
4478         mp->b_datap->db_type = M_FLUSH;
4479         *mp->b_wptr++ = binfo.bi_flag | FLUSHBAND;
4480         *mp->b_wptr++ = binfo.bi_pri;
4481         putnext(stp->sd_wrq, mp);
4482         /*
4483          * Send down an unsupported ioctl and wait for the nack

```

```

4484     * in order to allow the M_FLUSH to propagate back
4485     * up to the stream head.
4486     * Replaces if (qready()) runqueues();
4487     */
4488     stioctl.ic_cmd = -1;    /* The unsupported ioctl */
4489     stioctl.ic_timeout = 0;
4490     stioctl.ic_len = 0;
4491     stioctl.ic_dp = NULL;
4492     (void) strdioctl(stp, &stioctl, flag, K_TO_K, crp, rvalp);
4493     *rvalp = 0;
4494     return (0);
4495 }

4497 case I_SRDOPT:
4498     /*
4499     * Set read options
4500     *
4501     * RNORM - default stream mode
4502     * RMSGN - message no discard
4503     * RMSGD - message discard
4504     * RPROTNORM - fail read with EBADMSG for M_[PC]PROTOS
4505     * RPROTDAT - convert M_[PC]PROTOS to M_DATAS
4506     * RPROTDIS - discard M_[PC]PROTOS and retain M_DATAS
4507     */
4508     if (arg & ~(RMODEMASK | RPROTMASK))
4509         return (EINVAL);

4511     if ((arg & (RMSGD|RMSGN)) == (RMSGD|RMSGN))
4512         return (EINVAL);

4514     mutex_enter(&stp->sd_lock);
4515     switch (arg & RMODEMASK) {
4516     case RNORM:
4517         stp->sd_read_opt &= ~(RD_MSGDIS | RD_MSGNODIS);
4518         break;
4519     case RMSGD:
4520         stp->sd_read_opt = (stp->sd_read_opt & ~RD_MSGNODIS) |
4521             RD_MSGDIS;
4522         break;
4523     case RMSGN:
4524         stp->sd_read_opt = (stp->sd_read_opt & ~RD_MSGDIS) |
4525             RD_MSGNODIS;
4526         break;
4527     }

4529     switch (arg & RPROTMASK) {
4530     case RPROTNORM:
4531         stp->sd_read_opt &= ~(RD_PROTDAT | RD_PROTDIS);
4532         break;

4534     case RPROTDAT:
4535         stp->sd_read_opt = ((stp->sd_read_opt & ~RD_PROTDIS) |
4536             RD_PROTDAT);
4537         break;

4539     case RPROTDIS:
4540         stp->sd_read_opt = ((stp->sd_read_opt & ~RD_PROTDAT) |
4541             RD_PROTDIS);
4542         break;
4543     }
4544     mutex_exit(&stp->sd_lock);
4545     return (0);

4547 case I_GRDOPT:
4548     /*
4549     * Get read option and return the value

```

```

4550     * to spot pointed to by arg
4551     */
4552     {
4553         int rdopt;

4555         rdopt = ((stp->sd_read_opt & RD_MSGDIS) ? RMSGD :
4556             ((stp->sd_read_opt & RD_MSGNODIS) ? RMSGN : RNORM));
4557         rdopt |= ((stp->sd_read_opt & RD_PROTDAT) ? RPROTDAT :
4558             ((stp->sd_read_opt & RD_PROTDIS) ? RPROTDIS : RPROTNORM));

4560         return (strcpyout(&rdopt, (void *)arg, sizeof (int),
4561             copyflag));
4562     }

4564 case I_SERROPT:
4565     /*
4566     * Set error options
4567     *
4568     * RERRNORM - persistent read errors
4569     * RERRNONPERSIST - non-persistent read errors
4570     * WERRNORM - persistent write errors
4571     * WERRNONPERSIST - non-persistent write errors
4572     */
4573     if (arg & ~(RERRMASK | WERRMASK))
4574         return (EINVAL);

4576     mutex_enter(&stp->sd_lock);
4577     switch (arg & RERRMASK) {
4578     case RERRNORM:
4579         stp->sd_flag &= ~STRDERRNONPERSIST;
4580         break;
4581     case RERRNONPERSIST:
4582         stp->sd_flag |= STRDERRNONPERSIST;
4583         break;
4584     }
4585     switch (arg & WERRMASK) {
4586     case WERRNORM:
4587         stp->sd_flag &= ~STWRERRNONPERSIST;
4588         break;
4589     case WERRNONPERSIST:
4590         stp->sd_flag |= STWRERRNONPERSIST;
4591         break;
4592     }
4593     mutex_exit(&stp->sd_lock);
4594     return (0);

4596 case I_GERROPT:
4597     /*
4598     * Get error option and return the value
4599     * to spot pointed to by arg
4600     */
4601     {
4602         int erropt = 0;

4604         erropt |= (stp->sd_flag & STRDERRNONPERSIST) ? RERRNONPERSIST :
4605             RERRNORM;
4606         erropt |= (stp->sd_flag & STWRERRNONPERSIST) ? WERRNONPERSIST :
4607             WERRNORM;
4608         return (strcpyout(&erropt, (void *)arg, sizeof (int),
4609             copyflag));
4610     }

4612 case I_SETSIG:
4613     /*
4614     * Register the calling proc to receive the SIGPOLL
4615     * signal based on the events given in arg.  If

```

```

4616     * arg is zero, remove the proc from register list.
4617     */
4618     {
4619         strsig_t *ssp, *pssp;
4620         struct pid *pidp;
4621
4622         pssp = NULL;
4623         pidp = curproc->p_pidp;
4624         /*
4625          * Hold sd_lock to prevent traversal of sd_siglist while
4626          * it is modified.
4627          */
4628         mutex_enter(&stp->sd_lock);
4629         for (ssp = stp->sd_siglist; ssp && (ssp->ss_pidp != pidp);
4630              pssp = ssp, ssp = ssp->ss_next)
4631             ;
4632
4633         if (arg) {
4634             if (arg & ~(S_INPUT|S_HIPRI|S_MSG|S_HANGUP|S_ERROR|
4635                      S_RDNORM|S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)) {
4636                 mutex_exit(&stp->sd_lock);
4637                 return (EINVAL);
4638             }
4639             if ((arg & S_BANDURG) && !(arg & S_RDBAND)) {
4640                 mutex_exit(&stp->sd_lock);
4641                 return (EINVAL);
4642             }
4643
4644             /*
4645              * If proc not already registered, add it
4646              * to list.
4647              */
4648             if (!ssp) {
4649                 ssp = kmem_alloc(sizeof (strsig_t), KM_SLEEP);
4650                 ssp->ss_pidp = pidp;
4651                 ssp->ss_pid = pidp->pid_id;
4652                 ssp->ss_next = NULL;
4653                 if (pssp)
4654                     pssp->ss_next = ssp;
4655                 else
4656                     stp->sd_siglist = ssp;
4657                 mutex_enter(&pidlock);
4658                 PID_HOLD(pidp);
4659                 mutex_exit(&pidlock);
4660             }
4661
4662             /*
4663              * Set events.
4664              */
4665             ssp->ss_events = (int)arg;
4666         } else {
4667             /*
4668              * Remove proc from register list.
4669              */
4670             if (ssp) {
4671                 mutex_enter(&pidlock);
4672                 PID_RELE(pidp);
4673                 mutex_exit(&pidlock);
4674                 if (pssp)
4675                     pssp->ss_next = ssp->ss_next;
4676                 else
4677                     stp->sd_siglist = ssp->ss_next;
4678                 kmem_free(ssp, sizeof (strsig_t));
4679             } else {
4680                 mutex_exit(&stp->sd_lock);
4681                 return (EINVAL);

```

```

4682     }
4683
4684     /*
4685     * Recalculate OR of sig events.
4686     */
4687     stp->sd_sigflags = 0;
4688     for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4689         stp->sd_sigflags |= ssp->ss_events;
4690     mutex_exit(&stp->sd_lock);
4691     return (0);
4692
4693 }
4694
4695 case I_GETSIG:
4696     /*
4697     * Return (in arg) the current registration of events
4698     * for which the calling proc is to be signaled.
4699     */
4700     {
4701         struct strsig *ssp;
4702         struct pid *pidp;
4703
4704         pidp = curproc->p_pidp;
4705         mutex_enter(&stp->sd_lock);
4706         for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4707             if (ssp->ss_pidp == pidp) {
4708                 error = strcpyout(&ssp->ss_events, (void *)arg,
4709                                 sizeof (int), copyflag);
4710                 mutex_exit(&stp->sd_lock);
4711                 return (error);
4712             }
4713         mutex_exit(&stp->sd_lock);
4714         return (EINVAL);
4715     }
4716
4717 case I_ESETSIG:
4718     /*
4719     * Register the ss_pid to receive the SIGPOLL
4720     * signal based on the events is ss_events arg. If
4721     * ss_events is zero, remove the proc from register list.
4722     */
4723     {
4724         struct strsig *ssp, *pssp;
4725         struct proc *proc;
4726         struct pid *pidp;
4727         pid_t pid;
4728         struct strsigset ss;
4729
4730         error = strcpyin((void *)arg, &ss, sizeof (ss), copyflag);
4731         if (error)
4732             return (error);
4733
4734         pid = ss.ss_pid;
4735
4736         if (ss.ss_events != 0) {
4737             /*
4738              * Permissions check by sending signal 0.
4739              * Note that when kill fails it does a set_errno
4740              * causing the system call to fail.
4741              */
4742             error = kill(pid, 0);
4743             if (error) {
4744                 return (error);
4745             }
4746         }
4747         mutex_enter(&pidlock);

```

```

4748     if (pid == 0)
4749         proc = curproc;
4750     else if (pid < 0)
4751         proc = pgfind(-pid);
4752     else
4753         proc = prfind(pid);
4754     if (proc == NULL) {
4755         mutex_exit(&pidlock);
4756         return (ESRCH);
4757     }
4758     if (pid < 0)
4759         pidp = proc->p_pidp;
4760     else
4761         pidp = proc->p_pidp;
4762     ASSERT(pidp);
4763     /*
4764     * Get a hold on the pid structure while referencing it.
4765     * There is a separate PID_HOLD should it be inserted
4766     * in the list below.
4767     */
4768     PID_HOLD(pidp);
4769     mutex_exit(&pidlock);

4771     pssp = NULL;
4772     /*
4773     * Hold sd_lock to prevent traversal of sd_siglist while
4774     * it is modified.
4775     */
4776     mutex_enter(&stp->sd_lock);
4777     for (ssp = stp->sd_siglist; ssp && (ssp->ss_pid != pid);
4778         pssp = ssp, ssp = ssp->ss_next)
4779         ;

4781     if (ss.ss_events) {
4782         if (ss.ss_events &
4783             ~(S_INPUT|S_HIPRI|S_MSG|S_HANGUP|S_ERROR|
4784               S_RDNORM|S_WRNORM|S_RDBAND|S_WRBAND|S_BANDURG)) {
4785             mutex_exit(&stp->sd_lock);
4786             mutex_enter(&pidlock);
4787             PID_RELE(pidp);
4788             mutex_exit(&pidlock);
4789             return (EINVAL);
4790         }
4791         if ((ss.ss_events & S_BANDURG) &&
4792             !(ss.ss_events & S_RDBAND)) {
4793             mutex_exit(&stp->sd_lock);
4794             mutex_enter(&pidlock);
4795             PID_RELE(pidp);
4796             mutex_exit(&pidlock);
4797             return (EINVAL);
4798         }
4799     }

4800     /*
4801     * If proc not already registered, add it
4802     * to list.
4803     */
4804     if (!ssp) {
4805         ssp = kmem_alloc(sizeof (strsig_t), KM_SLEEP);
4806         ssp->ss_pidp = pidp;
4807         ssp->ss_pid = pid;
4808         ssp->ss_next = NULL;
4809         if (pssp)
4810             pssp->ss_next = ssp;
4811         else
4812             stp->sd_siglist = ssp;
4813         mutex_enter(&pidlock);

```

```

4814         PID_HOLD(pidp);
4815         mutex_exit(&pidlock);
4816     }

4818     /*
4819     * Set events.
4820     */
4821     ssp->ss_events = ss.ss_events;
4822 } else {
4823     /*
4824     * Remove proc from register list.
4825     */
4826     if (ssp) {
4827         mutex_enter(&pidlock);
4828         PID_RELE(pidp);
4829         mutex_exit(&pidlock);
4830         if (pssp)
4831             pssp->ss_next = ssp->ss_next;
4832         else
4833             stp->sd_siglist = ssp->ss_next;
4834         kmem_free(ssp, sizeof (strsig_t));
4835     } else {
4836         mutex_exit(&stp->sd_lock);
4837         mutex_enter(&pidlock);
4838         PID_RELE(pidp);
4839         mutex_exit(&pidlock);
4840         return (EINVAL);
4841     }
4842 }

4844     /*
4845     * Recalculate OR of sig events.
4846     */
4847     stp->sd_sigflags = 0;
4848     for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4849         stp->sd_sigflags |= ssp->ss_events;
4850     mutex_exit(&stp->sd_lock);
4851     mutex_enter(&pidlock);
4852     PID_RELE(pidp);
4853     mutex_exit(&pidlock);
4854     return (0);
4855 }

4857     case I_EGETSIG:
4858         /*
4859         * Return (in arg) the current registration of events
4860         * for which the calling proc is to be signaled.
4861         */
4862     {
4863         struct strsig *ssp;
4864         struct proc *proc;
4865         pid_t pid;
4866         struct pid *pidp;
4867         struct strsigset ss;

4869         error = strcopyin((void *)arg, &ss, sizeof (ss), copyflag);
4870         if (error)
4871             return (error);

4873         pid = ss.ss_pid;
4874         mutex_enter(&pidlock);
4875         if (pid == 0)
4876             proc = curproc;
4877         else if (pid < 0)
4878             proc = pgfind(-pid);
4879         else

```

```

4880         proc = prfind(pid);
4881     if (proc == NULL) {
4882         mutex_exit(&pidlock);
4883         return (ESRCH);
4884     }
4885     if (pid < 0)
4886         pidp = proc->p_pidp;
4887     else
4888         pidp = proc->p_pidp;
4890
4891     /* Prevent the pidp from being reassigned */
4892     PID_HOLD(pidp);
4893     mutex_exit(&pidlock);
4894
4895     mutex_enter(&stp->sd_lock);
4896     for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
4897         if (ssp->ss_pid == pid) {
4898             ss.ss_pid = ssp->ss_pid;
4899             ss.ss_events = ssp->ss_events;
4900             error = strcpyout(&ss, (void *)arg,
4901                 sizeof (struct strsigset), copyflag);
4902             mutex_exit(&stp->sd_lock);
4903             mutex_enter(&pidlock);
4904             PID_RELE(pidp);
4905             mutex_exit(&pidlock);
4906             return (error);
4907         }
4908     mutex_exit(&stp->sd_lock);
4909     mutex_enter(&pidlock);
4910     PID_RELE(pidp);
4911     mutex_exit(&pidlock);
4912     return (EINVAL);
4913 }
4914
4915 case I_PEEK:
4916 {
4917     STRUCT_DECL(strpeek, strpeek);
4918     size_t n;
4919     mblk_t *fmp, *tmp_mp = NULL;
4920
4921     STRUCT_INIT(strpeek, flag);
4922
4923     error = strcpyin((void *)arg, STRUCT_BUF(strpeek),
4924         STRUCT_SIZE(strpeek), copyflag);
4925     if (error)
4926         return (error);
4927
4928     mutex_enter(QLOCK(rdq));
4929     /*
4930      * Skip the invalid messages
4931      */
4932     for (mp = rdq->q_first; mp != NULL; mp = mp->b_next)
4933         if (mp->b_datap->db_type != M_SIG)
4934             break;
4935
4936     /*
4937      * If user has requested to peek at a high priority message
4938      * and first message is not, return 0
4939      */
4940     if (mp != NULL) {
4941         if ((STRUCT_FGET(strpeek, flags) & RS_HIPRI) &&
4942             queclass(mp) == QNORM) {
4943             *rvalp = 0;
4944             mutex_exit(QLOCK(rdq));
4945             return (0);
4946         }
4947     }

```

```

4946     } else if (stp->sd_struioirdq == NULL ||
4947         (STRUCT_FGET(strpeek, flags) & RS_HIPRI)) {
4948         /*
4949          * No mblks to look at at the streamhead and
4950          * 1). This isn't a synch stream or
4951          * 2). This is a synch stream but caller wants high
4952          * priority messages which is not supported by
4953          * the synch stream. (it only supports QNORM)
4954          */
4955         *rvalp = 0;
4956         mutex_exit(QLOCK(rdq));
4957         return (0);
4958     }
4959
4960     fmp = mp;
4961
4962     if (mp && mp->b_datap->db_type == M_PASSFP) {
4963         mutex_exit(QLOCK(rdq));
4964         return (EBADMSG);
4965     }
4966
4967     ASSERT(mp == NULL || mp->b_datap->db_type == M_PCPROTO ||
4968         mp->b_datap->db_type == M_PROTO ||
4969         mp->b_datap->db_type == M_DATA);
4970
4971     if (mp && mp->b_datap->db_type == M_PCPROTO) {
4972         STRUCT_FSET(strpeek, flags, RS_HIPRI);
4973     } else {
4974         STRUCT_FSET(strpeek, flags, 0);
4975     }
4976
4977     if (mp && ((tmp_mp = dupmsg(mp)) == NULL)) {
4978         mutex_exit(QLOCK(rdq));
4979         return (ENOSR);
4980     }
4981     mutex_exit(QLOCK(rdq));
4982
4983     /*
4984      * set mp = tmp_mp, so that I_PEEK processing can continue.
4985      * tmp_mp is used to free the dup'd message.
4986      */
4987     mp = tmp_mp;
4988
4989     uio.uio_fmode = 0;
4990     uio.uio_extflg = UIO_COPY_CACHED;
4991     uio.uio_segflg = (copyflag == U_TO_K) ? UIO_USERSPACE :
4992         UIO_SYSSPACE;
4993     uio.uio_limit = 0;
4994     /*
4995      * First process PROTO blocks, if any.
4996      * If user doesn't want to get ctl info by setting maxlen <= 0,
4997      * then set len to -1/0 and skip control blocks part.
4998      */
4999     if (STRUCT_FGET(strpeek, ctlbuf.maxlen) < 0)
5000         STRUCT_FSET(strpeek, ctlbuf.len, -1);
5001     else if (STRUCT_FGET(strpeek, ctlbuf.maxlen) == 0)
5002         STRUCT_FSET(strpeek, ctlbuf.len, 0);
5003     else {
5004         int ctl_part = 0;
5005
5006         iov.iov_base = STRUCT_FGETP(strpeek, ctlbuf.buf);
5007         iov.iov_len = STRUCT_FGET(strpeek, ctlbuf.maxlen);
5008         uio.uio_iov = &iov;
5009         uio.uio_resid = iov.iov_len;
5010         uio.uio_loffset = 0;

```

```

5012     uio.uio_iovcnt = 1;
5013     while (mp && mp->b_datap->db_type != M_DATA &&
5014           uio.uio_resid >= 0) {
5015         ASSERT(STRUCT_FGET(strpeek, flags) == 0 ?
5016              mp->b_datap->db_type == M_PROTO :
5017              mp->b_datap->db_type == M_PCPROTO);
5018
5019         if ((n = MIN(uio.uio_resid,
5020                    mp->b_wptr - mp->b_rptr)) != 0 &&
5021             (error = uiomove((char *)mp->b_rptr, n,
5022                             UIO_READ, &uio)) != 0) {
5023             freemsg(tmp_mp);
5024             return (error);
5025         }
5026         ctl_part = 1;
5027         mp = mp->b_cont;
5028     }
5029     /* No ctl message */
5030     if (ctl_part == 0)
5031         STRUCT_FSET(strpeek, ctlbuf.len, -1);
5032     else
5033         STRUCT_FSET(strpeek, ctlbuf.len,
5034                    STRUCT_FGET(strpeek, ctlbuf.maxlen) -
5035                    uio.uio_resid);
5036 }
5037
5038 /*
5039  * Now process DATA blocks, if any.
5040  * If user doesn't want to get data info by setting maxlen <= 0,
5041  * then set len to -1/0 and skip data blocks part.
5042  */
5043 if (STRUCT_FGET(strpeek, databuf.maxlen) < 0)
5044     STRUCT_FSET(strpeek, databuf.len, -1);
5045 else if (STRUCT_FGET(strpeek, databuf.maxlen) == 0)
5046     STRUCT_FSET(strpeek, databuf.len, 0);
5047 else {
5048     int    data_part = 0;
5049
5050     iov.iov_base = STRUCT_FGETP(strpeek, databuf.buf);
5051     iov.iov_len = STRUCT_FGET(strpeek, databuf.maxlen);
5052     uio.uio_iov = &iov;
5053     uio.uio_resid = iov.iov_len;
5054     uio.uio_loffset = 0;
5055     uio.uio_iovcnt = 1;
5056     while (mp && uio.uio_resid) {
5057         if (mp->b_datap->db_type == M_DATA) {
5058             if ((n = MIN(uio.uio_resid,
5059                        mp->b_wptr - mp->b_rptr)) != 0 &&
5060                 (error = uiomove((char *)mp->b_rptr,
5061                                 n, UIO_READ, &uio)) != 0) {
5062                 freemsg(tmp_mp);
5063                 return (error);
5064             }
5065             data_part = 1;
5066         }
5067         ASSERT(data_part == 0 ||
5068              mp->b_datap->db_type == M_DATA);
5069         mp = mp->b_cont;
5070     }
5071     /* No data message */
5072     if (data_part == 0)
5073         STRUCT_FSET(strpeek, databuf.len, -1);
5074     else
5075         STRUCT_FSET(strpeek, databuf.len,
5076                    STRUCT_FGET(strpeek, databuf.maxlen) -
5077                    uio.uio_resid);

```

```

5078     }
5079     freemsg(tmp_mp);
5080
5081     /*
5082     * It is a synch stream and user wants to get
5083     * data (maxlen > 0).
5084     * uio setup is done by the codes that process DATA
5085     * blocks above.
5086     */
5087     if ((fmp == NULL) && STRUCT_FGET(strpeek, databuf.maxlen) > 0) {
5088         infod_t infod;
5089
5090         infod.d_cmd = INFOD_COPYOUT;
5091         infod.d_res = 0;
5092         infod.d_uiop = &uio;
5093         error = infonext(rdq, &infod);
5094         if (error == EINVAL || error == EBUSY)
5095             error = 0;
5096         if (error)
5097             return (error);
5098         STRUCT_FSET(strpeek, databuf.len, STRUCT_FGET(strpeek,
5099                                                         databuf.maxlen) -
5100                                                         uio.uio_resid);
5101         if (STRUCT_FGET(strpeek, databuf.len) == 0) {
5102             /*
5103              * No data found by the infonext().
5104              */
5105             STRUCT_FSET(strpeek, databuf.len, -1);
5106         }
5107         error = strcopyout(STRUCT_BUF(strpeek), (void *)arg,
5108                           STRUCT_SIZE(strpeek), copyflag);
5109         if (error) {
5110             return (error);
5111         }
5112         /*
5113          * If there is no message retrieved, set return code to 0
5114          * otherwise, set it to 1.
5115          */
5116         if (STRUCT_FGET(strpeek, ctlbuf.len) == -1 &&
5117             STRUCT_FGET(strpeek, databuf.len) == -1)
5118             *rvalp = 0;
5119         else
5120             *rvalp = 1;
5121         return (0);
5122     }
5123
5124     case I_FDINSERT:
5125     {
5126         STRUCT_DECL(strfdinsert, strfdinsert);
5127         struct file *resftp;
5128         struct stdata *resstp;
5129         t_uscalar_t ival;
5130         ssize_t msgsize;
5131         struct strbuf mctl;
5132
5133         STRUCT_INIT(strfdinsert, flag);
5134         if (stp->sd_flag & STRHUP)
5135             return (ENXIO);
5136         /*
5137          * STRDERR, STWRERR and STPLEX tested above.
5138          */
5139         error = strcopyin((void *)arg, STRUCT_BUF(strfdinsert),
5140                           STRUCT_SIZE(strfdinsert), copyflag);
5141         if (error)
5142             return (error);

```

```

5144     if (STRUCT_FGET(strfdinsert, offset) < 0 ||
5145         (STRUCT_FGET(strfdinsert, offset) %
5146          sizeof (t_uscalar_t)) != 0)
5147         return (EINVAL);
5148     if ((resftp = getf(STRUCT_FGET(strfdinsert, fildes))) != NULL) {
5149         if ((resstp = resftp->f_vnode->v_stream) == NULL) {
5150             releasef(STRUCT_FGET(strfdinsert, fildes));
5151             return (EINVAL);
5152         }
5153     } else
5154         return (EINVAL);

5156     mutex_enter(&resstp->sd_lock);
5157     if (resstp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
5158         error = strgeterr(resstp,
5159             STRDERR|STWRERR|STRHUP|STPLEX, 0);
5160         if (error != 0) {
5161             mutex_exit(&resstp->sd_lock);
5162             releasef(STRUCT_FGET(strfdinsert, fildes));
5163             return (error);
5164         }
5165     }
5166     mutex_exit(&resstp->sd_lock);

5168 #ifdef _ILP32
5169 {
5170     queue_t *q;
5171     queue_t *mate = NULL;

5173     /* get read queue of stream terminus */
5174     claimstr(resstp->sd_wrq);
5175     for (q = resstp->sd_wrq->q_next; q->q_next != NULL;
5176          q = q->q_next)
5177         if (!STREAMED(resstp) && STREAM(q) != resstp &&
5178             mate == NULL) {
5179             ASSERT(q->q_qinfo->qi_srvp);
5180             ASSERT(_OTHERQ(q)->q_qinfo->qi_srvp);
5181             claimstr(q);
5182             mate = q;
5183         }
5184     q = _RD(q);
5185     if (mate)
5186         releasestr(mate);
5187     releasestr(resstp->sd_wrq);
5188     ival = (t_uscalar_t)q;
5189 }
5190 #else
5191 ival = (t_uscalar_t)getminor(resftp->f_vnode->v_rdev);
5192 #endif /* _ILP32 */

5194     if (STRUCT_FGET(strfdinsert, ctlbuf.len) <
5195         STRUCT_FGET(strfdinsert, offset) + sizeof (t_uscalar_t)) {
5196         releasef(STRUCT_FGET(strfdinsert, fildes));
5197         return (EINVAL);
5198     }

5200     /*
5201     * Check for legal flag value.
5202     */
5203     if (STRUCT_FGET(strfdinsert, flags) & ~RS_HIPRI) {
5204         releasef(STRUCT_FGET(strfdinsert, fildes));
5205         return (EINVAL);
5206     }

5208     /* get these values from those cached in the stream head */
5209     mutex_enter(QLOCK(stp->sd_wrq));

```

```

5210         rmin = stp->sd_qn_minpsz;
5211         rmax = stp->sd_qn_maxpsz;
5212         mutex_exit(QLOCK(stp->sd_wrq));

5214     /*
5215     * Make sure ctl and data sizes together fall within
5216     * the limits of the max and min receive packet sizes
5217     * and do not exceed system limit. A negative data
5218     * length means that no data part is to be sent.
5219     */
5220     ASSERT((rmax >= 0) || (rmax == INFPSZ));
5221     if (rmax == 0) {
5222         releasef(STRUCT_FGET(strfdinsert, fildes));
5223         return (ERANGE);
5224     }
5225     if ((msgsize = STRUCT_FGET(strfdinsert, databuf.len) < 0)
5226         msgsize = 0;
5227     if ((msgsize < rmin) ||
5228         (msgsize > rmax) && (rmax != INFPSZ)) ||
5229         (STRUCT_FGET(strfdinsert, ctlbuf.len) > strctlsz)) {
5230         releasef(STRUCT_FGET(strfdinsert, fildes));
5231         return (ERANGE);
5232     }

5234     mutex_enter(&stp->sd_lock);
5235     while (!(STRUCT_FGET(strfdinsert, flags) & RS_HIPRI) &&
5236            !canputnext(stp->sd_wrq)) {
5237         if ((error = strwaitq(stp, WRITEWAIT, (ssize_t)0,
5238             flag, -1, &done) != 0 || done) {
5239             mutex_exit(&stp->sd_lock);
5240             releasef(STRUCT_FGET(strfdinsert, fildes));
5241             return (error);
5242         }
5243         if ((error = i_straccess(stp, access)) != 0) {
5244             mutex_exit(&stp->sd_lock);
5245             releasef(
5246                 STRUCT_FGET(strfdinsert, fildes));
5247             return (error);
5248         }
5249     }
5250     mutex_exit(&stp->sd_lock);

5252     /*
5253     * Copy strfdinsert.ctlbuf into native form of
5254     * ctlbuf to pass down into strmakemsg().
5255     */
5256     mctl.maxlen = STRUCT_FGET(strfdinsert, ctlbuf.maxlen);
5257     mctl.len = STRUCT_FGET(strfdinsert, ctlbuf.len);
5258     mctl.buf = STRUCT_FGETP(strfdinsert, ctlbuf.buf);

5260     iov.iov_base = STRUCT_FGETP(strfdinsert, databuf.buf);
5261     iov.iov_len = STRUCT_FGET(strfdinsert, databuf.len);
5262     uio.uio_iov = &iov;
5263     uio.uio_iovcnt = 1;
5264     uio.uio_loffset = 0;
5265     uio.uio_segflg = (copyflag == U_TO_K) ? UIO_USERSPACE :
5266         UIO_SYSSPACE;
5267     uio.uio_fmode = 0;
5268     uio.uio_extflg = UIO_COPY_CACHED;
5269     uio.uio_resid = iov.iov_len;
5270     if ((error = strmakemsg(&mctl,
5271         &msgsize, &uio, stp,
5272         STRUCT_FGET(strfdinsert, flags), &mp)) != 0 || !mp) {
5273         STRUCT_FSET(strfdinsert, databuf.len, msgsize);
5274         releasef(STRUCT_FGET(strfdinsert, fildes));
5275         return (error);

```

```

5276     }
5278     STRUCT_FSET(strfdinsert, databuf.len, msgsize);
5280     /*
5281     * Place the possibly reencoded queue pointer 'offset' bytes
5282     * from the start of the control portion of the message.
5283     */
5284     *((t_uscalar_t *) (mp->b_rptr +
5285     STRUCT_FGET(strfdinsert, offset))) = ival;
5287     /*
5288     * Put message downstream.
5289     */
5290     stream_willservice(stp);
5291     putnext(stp->sd_wrq, mp);
5292     stream_runservice(stp);
5293     releasef(STRUCT_FGET(strfdinsert, fildes));
5294     return (error);
5295 }
5297 case I_SENDFD:
5298 {
5299     struct file *fp;
5301     if ((fp = getf((int)arg)) == NULL)
5302         return (EBADF);
5303     error = do_sendfp(stp, fp, crp);
5304     if (auditing) {
5305         audit_fdsend((int)arg, fp, error);
5306     }
5307     releasef((int)arg);
5308     return (error);
5309 }
5311 case I_RECVFD:
5312 case I_E_RECVFD:
5313 {
5314     struct k_strrecvfd *srf;
5315     int i, fd;
5317     mutex_enter(&stp->sd_lock);
5318     while (!(mp = getq(rdq))) {
5319         if (stp->sd_flag & (STRHUP|STREOF)) {
5320             mutex_exit(&stp->sd_lock);
5321             return (ENXIO);
5322         }
5323         if ((error = strwaitq(stp, GETWAIT, (ssize_t)0,
5324         flag, -1, &done)) != 0 || done) {
5325             mutex_exit(&stp->sd_lock);
5326             return (error);
5327         }
5328         if ((error = i_straccess(stp, access)) != 0) {
5329             mutex_exit(&stp->sd_lock);
5330             return (error);
5331         }
5332     }
5333     if (mp->b_datap->db_type != M_PASSFP) {
5334         putback(stp, rdq, mp, mp->b_band);
5335         mutex_exit(&stp->sd_lock);
5336         return (EBADMSG);
5337     }
5338     mutex_exit(&stp->sd_lock);
5340     srf = (struct k_strrecvfd *) mp->b_rptr;
5341     if ((fd = ufalloc(0)) == -1) {

```

```

5342         mutex_enter(&stp->sd_lock);
5343         putback(stp, rdq, mp, mp->b_band);
5344         mutex_exit(&stp->sd_lock);
5345         return (EMFILE);
5346     }
5347     if (cmd == I_RECVFD) {
5348         struct o_strrecvfd ostrfd;
5350         /* check to see if uid/gid values are too large. */
5352         if (srf->uid > (o_uid_t)USHRT_MAX ||
5353             srf->gid > (o_gid_t)USHRT_MAX) {
5354             mutex_enter(&stp->sd_lock);
5355             putback(stp, rdq, mp, mp->b_band);
5356             mutex_exit(&stp->sd_lock);
5357             setf(fd, NULL); /* release fd entry */
5358             return (Eoverflow);
5359         }
5361         ostrfd.fd = fd;
5362         ostrfd.uid = (o_uid_t)srf->uid;
5363         ostrfd.gid = (o_gid_t)srf->gid;
5365         /* Null the filler bits */
5366         for (i = 0; i < 8; i++)
5367             ostrfd.fill[i] = 0;
5369         error = strcopyout(&ostrfd, (void *)arg,
5370         sizeof (struct o_strrecvfd), copyflag);
5371     } else { /* I_E_RECVFD */
5372         struct strrecvfd strfd;
5374         strfd.fd = fd;
5375         strfd.uid = srf->uid;
5376         strfd.gid = srf->gid;
5378         /* null the filler bits */
5379         for (i = 0; i < 8; i++)
5380             strfd.fill[i] = 0;
5382         error = strcopyout(&strfd, (void *)arg,
5383         sizeof (struct strrecvfd), copyflag);
5384     }
5386     if (error) {
5387         setf(fd, NULL); /* release fd entry */
5388         mutex_enter(&stp->sd_lock);
5389         putback(stp, rdq, mp, mp->b_band);
5390         mutex_exit(&stp->sd_lock);
5391         return (error);
5392     }
5393     if (auditing) {
5394         audit_fdrecv(fd, srf->fp);
5395     }
5397     /*
5398     * Always increment f_count since the freemsg() below will
5399     * always call free_passfp() which performs a closef().
5400     */
5401     mutex_enter(&srf->fp->f_tlock);
5402     srf->fp->f_count++;
5403     mutex_exit(&srf->fp->f_tlock);
5404     setf(fd, srf->fp);
5405     freemsg(mp);
5406     return (0);
5407 }

```

```

5409     case I_SWROPT:
5410         /*
5411          * Set/clear the write options. arg is a bit
5412          * mask with any of the following bits set...
5413          * SNDZERO - send zero length message
5414          * SNDPIPE - send sigpipe to process if
5415          *           sd_werror is set and process is
5416          *           doing a write or putmsg.
5417          * The new stream head write options should reflect
5418          * what is in arg.
5419          */
5420         if (arg & ~(SNDZERO|SNDPIPE))
5421             return (EINVAL);

5423         mutex_enter(&stp->sd_lock);
5424         stp->sd_wput_opt &= ~(SW_SIGPIPE|SW_SNDZERO);
5425         if (arg & SNDZERO)
5426             stp->sd_wput_opt |= SW_SNDZERO;
5427         if (arg & SNDPIPE)
5428             stp->sd_wput_opt |= SW_SIGPIPE;
5429         mutex_exit(&stp->sd_lock);
5430         return (0);

5432     case I_GWROPT:
5433     {
5434         int wropt = 0;

5436         if (stp->sd_wput_opt & SW_SNDZERO)
5437             wropt |= SNDZERO;
5438         if (stp->sd_wput_opt & SW_SIGPIPE)
5439             wropt |= SNDPIPE;
5440         return (strncpyout(&wropt, (void *)arg, sizeof (wropt),
5441             copyflag));
5442     }

5444     case I_LIST:
5445         /*
5446          * Returns all the modules found on this stream,
5447          * upto the driver. If argument is NULL, return the
5448          * number of modules (including driver). If argument
5449          * is not NULL, copy the names into the structure
5450          * provided.
5451          */

5453     {
5454         queue_t *q;
5455         char *qname;
5456         int i, nmods;
5457         struct str_mlist *mlist;
5458         STRUCT_DECL(str_list, strlist);

5460         if (arg == NULL) { /* Return number of modules plus driver */
5461             if (stp->sd_vnode->v_type == VFIFO)
5462                 *rvalp = stp->sd_pushcnt;
5463             else
5464                 *rvalp = stp->sd_pushcnt + 1;
5465             return (0);
5466         }

5468         STRUCT_INIT(strlist, flag);

5470         error = strncpyin((void *)arg, STRUCT_BUF(strlist),
5471             STRUCT_SIZE(strlist), copyflag);
5472         if (error != 0)
5473             return (error);

```

```

5475         mlist = STRUCT_FGETP(strlist, sl_modlist);
5476         nmods = STRUCT_FGET(strlist, sl_nmods);
5477         if (nmods <= 0)
5478             return (EINVAL);

5480         claimstr(stp->sd_wrq);
5481         q = stp->sd_wrq;
5482         for (i = 0; i < nmods && _SAMESTR(q); i++, q = q->q_next) {
5483             qname = Q2NAME(q->q_next);
5484             error = strncpyout(qname, &mlist[i], strlen(qname) + 1,
5485                 copyflag);
5486             if (error != 0) {
5487                 releasestr(stp->sd_wrq);
5488                 return (error);
5489             }
5490         }
5491         releasestr(stp->sd_wrq);
5492         return (strncpyout(&i, (void *)arg, sizeof (int), copyflag));
5493     }

5495     case I_CKBAND:
5496     {
5497         queue_t *q;
5498         qband_t *qbp;

5500         if ((arg < 0) || (arg >= NBAND))
5501             return (EINVAL);
5502         q = _RD(stp->sd_wrq);
5503         mutex_enter(QLOCK(q));
5504         if (arg > (int)q->q_nband) {
5505             *rvalp = 0;
5506         } else {
5507             if (arg == 0) {
5508                 if (q->q_first)
5509                     *rvalp = 1;
5510                 else
5511                     *rvalp = 0;
5512             } else {
5513                 qbp = q->q_bandp;
5514                 while (--arg > 0)
5515                     qbp = qbp->qband_next;
5516                 if (qbp->qband_first)
5517                     *rvalp = 1;
5518                 else
5519                     *rvalp = 0;
5520             }
5521         }
5522         mutex_exit(QLOCK(q));
5523         return (0);
5524     }

5526     case I_GETBAND:
5527     {
5528         int intpri;
5529         queue_t *q;

5531         q = _RD(stp->sd_wrq);
5532         mutex_enter(QLOCK(q));
5533         mp = q->q_first;
5534         if (!mp) {
5535             mutex_exit(QLOCK(q));
5536             return (ENODATA);
5537         }
5538         intpri = (int)mp->b_band;
5539         error = strncpyout(&intpri, (void *)arg, sizeof (int),

```

```

5540         copyflag);
5541         mutex_exit(QLOCK(q));
5542         return (error);
5543     }
5544
5545     case I_ATMARK:
5546     {
5547         queue_t *q;
5548
5549         if (arg & ~(ANYMARK|LASTMARK))
5550             return (EINVAL);
5551         q = _RD(stp->sd_wrq);
5552         mutex_enter(&stp->sd_lock);
5553         if ((stp->sd_flag & STRATMARK) && (arg == ANYMARK)) {
5554             *rvalp = 1;
5555         } else {
5556             mutex_enter(QLOCK(q));
5557             mp = q->q_first;
5558
5559             if (mp == NULL)
5560                 *rvalp = 0;
5561             else if ((arg == ANYMARK) && (mp->b_flag & MSGMARK))
5562                 *rvalp = 1;
5563             else if ((arg == LASTMARK) && (mp == stp->sd_mark))
5564                 *rvalp = 1;
5565             else
5566                 *rvalp = 0;
5567             mutex_exit(QLOCK(q));
5568         }
5569         mutex_exit(&stp->sd_lock);
5570         return (0);
5571     }
5572
5573     case I_CANPUT:
5574     {
5575         char band;
5576
5577         if ((arg < 0) || (arg >= NBAND))
5578             return (EINVAL);
5579         band = (char)arg;
5580         *rvalp = bcanputnext(stp->sd_wrq, band);
5581         return (0);
5582     }
5583
5584     case I_SETCLTIME:
5585     {
5586         int closetime;
5587
5588         error = strcpyin((void *)arg, &closetime, sizeof (int),
5589             copyflag);
5590         if (error)
5591             return (error);
5592         if (closetime < 0)
5593             return (EINVAL);
5594
5595         stp->sd_closetime = closetime;
5596         return (0);
5597     }
5598
5599     case I_GETCLTIME:
5600     {
5601         int closetime;
5602
5603         closetime = stp->sd_closetime;
5604         return (strcpyout(&closetime, (void *)arg, sizeof (int),
5605             copyflag));

```

```

5606     }
5607
5608     case TIOCGSID:
5609     {
5610         pid_t sid;
5611
5612         mutex_enter(&stp->sd_lock);
5613         if (stp->sd_sidp == NULL) {
5614             mutex_exit(&stp->sd_lock);
5615             return (ENOTTY);
5616         }
5617         sid = stp->sd_sidp->pid_id;
5618         mutex_exit(&stp->sd_lock);
5619         return (strcpyout(&sid, (void *)arg, sizeof (pid_t),
5620             copyflag));
5621     }
5622
5623     case TIOCSPGRP:
5624     {
5625         pid_t pgrp;
5626         proc_t *q;
5627         pid_t sid, fg_pgid, bg_pgid;
5628
5629         if (error = strcpyin((void *)arg, &pgrp, sizeof (pid_t),
5630             copyflag))
5631             return (error);
5632         mutex_enter(&stp->sd_lock);
5633         mutex_enter(&pidlock);
5634         if (stp->sd_sidp != ttoproc(curthread)->p_sessp->s_sidp) {
5635             mutex_exit(&pidlock);
5636             mutex_exit(&stp->sd_lock);
5637             return (ENOTTY);
5638         }
5639         if (pgrp == stp->sd_pggrp->pid_id) {
5640             mutex_exit(&pidlock);
5641             mutex_exit(&stp->sd_lock);
5642             return (0);
5643         }
5644         if (pgrp <= 0 || pgrp >= maxpid) {
5645             mutex_exit(&pidlock);
5646             mutex_exit(&stp->sd_lock);
5647             return (EINVAL);
5648         }
5649         if ((q = pgfind(pgrp)) == NULL ||
5650             q->p_sessp != ttoproc(curthread)->p_sessp) {
5651             mutex_exit(&pidlock);
5652             mutex_exit(&stp->sd_lock);
5653             return (EPERM);
5654         }
5655         sid = stp->sd_sidp->pid_id;
5656         fg_pgid = q->p_pgrp;
5657         bg_pgid = stp->sd_pggrp->pid_id;
5658         CL_SET_PROCESS_GROUP(curthread, sid, bg_pgid, fg_pgid);
5659         PID_RELE(stp->sd_pggrp);
5660         ctty_clear_sighuped();
5661         stp->sd_pggrp = q->p_pggrp;
5662         PID_HOLD(stp->sd_pggrp);
5663         mutex_exit(&pidlock);
5664         mutex_exit(&stp->sd_lock);
5665         return (0);
5666     }
5667
5668     case TIOCGPGRP:
5669     {
5670         pid_t pgrp;

```

```

5672     mutex_enter(&stp->sd_lock);
5673     if (stp->sd_sidp == NULL) {
5674         mutex_exit(&stp->sd_lock);
5675         return (ENOTTY);
5676     }
5677     pgrp = stp->sd_pgidp->pid_id;
5678     mutex_exit(&stp->sd_lock);
5679     return (strncpyout(&pgrp, (void *)arg, sizeof (pid_t),
5680         copyflag));
5681 }

5683 case TIOCSCTTY:
5684 {
5685     return (strctty(stp));
5686 }

5688 case TIOCNOTTY:
5689 {
5690     /* freectty() always assumes curproc. */
5691     if (freectty(B_FALSE) != 0)
5692         return (0);
5693     return (ENOTTY);
5694 }

5696 case FIONBIO:
5697 case FIOASYNC:
5698     return (0); /* handled by the upper layer */
5699 case F_ASSOCI_PID:
5700 {
5701     if (crp != kcred)
5702         return (EPERM);
5703     if (is_xti_str(stp))
5704         sh_insert_pid(stp, (pid_t)arg);
5705     return (0);
5706 }
5707 case F_DASSOC_PID:
5708 {
5709     if (crp != kcred)
5710         return (EPERM);
5711     if (is_xti_str(stp))
5712         sh_remove_pid(stp, (pid_t)arg);
5713     return (0);
5714 }
5715 #endif /* ! codereview */
5716 }
5717 }

5719 /*
5720  * Custom free routine used for M_PASSFP messages.
5721  */
5722 static void
5723 free_passfp(struct k_strrecvfd *srf)
5724 {
5725     (void) closef(srf->fp);
5726     kmem_free(srf, sizeof (struct k_strrecvfd) + sizeof (frtn_t));
5727 }

5729 /* ARGSUSED */
5730 int
5731 do_sendfp(struct stdata *stp, struct file *fp, struct cred *cr)
5732 {
5733     queue_t *qp, *nextqp;
5734     struct k_strrecvfd *srf;
5735     mblk_t *mp;
5736     frtn_t *frtnp;
5737     size_t bufsize;

```

```

5738     queue_t *mate = NULL;
5739     syncq_t *sq = NULL;
5740     int retval = 0;

5742     if (stp->sd_flag & STRHUP)
5743         return (ENXIO);

5745     claimstr(stp->sd_wrq);

5747     /* Fastpath, we have a pipe, and we are already mated, use it. */
5748     if (STRMATED(stp)) {
5749         qp = _RD(stp->sd_mate->sd_wrq);
5750         claimstr(qp);
5751         mate = qp;
5752     } else { /* Not already mated. */

5754         /*
5755          * Walk the stream to the end of this one.
5756          * assumes that the claimstr() will prevent
5757          * plumbing between the stream head and the
5758          * driver from changing
5759          */
5760         qp = stp->sd_wrq;

5762         /*
5763          * Loop until we reach the end of this stream.
5764          * On completion, qp points to the write queue
5765          * at the end of the stream, or the read queue
5766          * at the stream head if this is a fifo.
5767          */
5768         while (((qp = qp->q_next) != NULL) && !_SAMESTR(qp))
5769             ;

5771         /*
5772          * Just in case we get a q_next which is NULL, but
5773          * not at the end of the stream. This is actually
5774          * broken, so we set an assert to catch it in
5775          * debug, and set an error and return if not debug.
5776          */
5777         ASSERT(qp);
5778         if (qp == NULL) {
5779             releasestr(stp->sd_wrq);
5780             return (EINVAL);
5781         }

5783         /*
5784          * Enter the syncq for the driver, so (hopefully)
5785          * the queue values will not change on us.
5786          * XXXX - This will only prevent the race IFF only
5787          * the write side modifies the q_next member, and
5788          * the put procedure is protected by at least
5789          * MT_PERQ.
5790          */
5791         if ((sq = qp->q_syncq) != NULL)
5792             entersq(sq, SQ_PUT);

5794         /* Now get the q_next value from this qp. */
5795         nextqp = qp->q_next;

5797         /*
5798          * If nextqp exists and the other stream is different
5799          * from this one claim the stream, set the mate, and
5800          * get the read queue at the stream head of the other
5801          * stream. Assumes that nextqp was at least valid when
5802          * we got it. Hopefully the entersq of the driver
5803          * will prevent it from changing on us.

```

```

5804     */
5805     if ((nextqp != NULL) && (STREAM(nextqp) != stp)) {
5806         ASSERT(qp->q_qinfo->q_i_srvp);
5807         ASSERT(_OTHERQ(qp)->q_qinfo->q_i_srvp);
5808         ASSERT(_OTHERQ(qp->q_next)->q_qinfo->q_i_srvp);
5809         claimstr(nextqp);
5811
5812         /* Make sure we still have a q_next */
5813         if (nextqp != qp->q_next) {
5814             releasestr(stp->sd_wrq);
5815             releasestr(nextqp);
5816             return (EINVAL);
5817         }
5818
5819         qp = _RD(STREAM(nextqp)->sd_wrq);
5820         mate = qp;
5821     }
5822     /* If we entered the synq above, leave it. */
5823     if (sq != NULL)
5824         leavesq(sq, SQ_PUT);
5825 } /* STRMATED(STP) */
5826
5827 /* XXX prevents substitution of the ops vector */
5828 if (qp->q_qinfo != &strdata && qp->q_qinfo != &fifo_strdata) {
5829     retval = EINVAL;
5830     goto out;
5831 }
5832
5833 if (qp->q_flag & QFULL) {
5834     retval = EAGAIN;
5835     goto out;
5836 }
5837
5838 /*
5839  * Since M_PASSFP messages include a file descriptor, we use
5840  * esballoc() and specify a custom free routine (free_passfp()) that
5841  * will close the descriptor as part of freeing the message. For
5842  * convenience, we stash the frtn_t right after the data block.
5843  */
5844 bufsize = sizeof (struct k_strrecvfd) + sizeof (frtn_t);
5845 srf = kmem_alloc(bufsize, KM_NOSLEEP);
5846 if (srf == NULL) {
5847     retval = EAGAIN;
5848     goto out;
5849 }
5850
5851 frtnp = (frtn_t *) (srf + 1);
5852 frtnp->free_arg = (caddr_t) srf;
5853 frtnp->free_func = free_passfp;
5854
5855 mp = esballoc((uchar_t *) srf, bufsize, BPRI_MED, frtnp);
5856 if (mp == NULL) {
5857     kmem_free(srf, bufsize);
5858     retval = EAGAIN;
5859     goto out;
5860 }
5861 mp->b_wptr += sizeof (struct k_strrecvfd);
5862 mp->b_datap->db_type = M_PASSFP;
5863
5864 srf->fp = fp;
5865 srf->uid = crgetuid(curthread->t_cred);
5866 srf->gid = crgetgid(curthread->t_cred);
5867 mutex_enter(&fp->f_tlock);
5868 fp->f_count++;
5869 mutex_exit(&fp->f_tlock);

```

```

5870     put(qp, mp);
5871 out:
5872     releasestr(stp->sd_wrq);
5873     if (mate)
5874         releasestr(mate);
5875     return (retval);
5876 }
5877
5878 /*
5879  * Send an ioctl message downstream and wait for acknowledgement.
5880  * flags may be set to either U_TO_K or K_TO_K and a combination
5881  * of STR_NOERROR or STR_NOSIG
5882  * STR_NOSIG: Signals are essentially ignored or held and have
5883  * no effect for the duration of the call.
5884  * STR_NOERROR: Ignores stream head read, write and hup errors.
5885  * Additionally, if an existing ioctl times out, it is assumed
5886  * lost and and this ioctl will continue as if the previous ioctl had
5887  * finished. ETIME may be returned if this ioctl times out (i.e.
5888  * ic_timeout is not INFTIM). Non-stream head errors may be returned if
5889  * the ioc_error indicates that the driver/module had problems,
5890  * an EFAULT was found when accessing user data, a lack of
5891  * resources, etc.
5892  */
5893 int
5894 strdoioctl(
5895     struct stdata *stp,
5896     struct striocctl *strioc,
5897     int fflags, /* file flags with model info */
5898     int flag,
5899     cred_t *crp,
5900     int *rvalp)
5901 {
5902     mblk_t *bp;
5903     struct iocblk *iocbp;
5904     struct copyreq *reqp;
5905     struct copyresp *resp;
5906     int id;
5907     int transparent = 0;
5908     int error = 0;
5909     int len = 0;
5910     caddr_t taddr;
5911     int copyflag = (flag & (U_TO_K | K_TO_K));
5912     int sigflag = (flag & STR_NOSIG);
5913     int errs;
5914     uint_t waitflags;
5915     boolean_t set_iocwaitne = B_FALSE;
5916
5917     ASSERT(copyflag == U_TO_K || copyflag == K_TO_K);
5918     ASSERT((fflags & FMODELS) != 0);
5919
5920     TRACE_2(TR_FAC_STREAMS_FR,
5921            TR_STRDOIOCTL,
5922            "strdoioctl:stp %p strioc %p", stp, strioc);
5923     if (strioc->ic_len == TRANSPARENT) { /* send arg in M_DATA block */
5924         transparent = 1;
5925         strioc->ic_len = sizeof (intptr_t);
5926     }
5927
5928     if (strioc->ic_len < 0 || (strmsgsz > 0 && strioc->ic_len > strmsgsz))
5929         return (EINVAL);
5930
5931     if ((bp = allocb_cred_wait(sizeof (union ioctypes), sigflag, &error,
5932                                crp, curproc->p_pid)) == NULL)
5933         return (error);
5934
5935     bzero(bp->b_wptr, sizeof (union ioctypes));

```

```

5937     iocbp = (struct iocblk *)bp->b_wptr;
5938     iocbp->ioc_count = strioc->ic_len;
5939     iocbp->ioc_cmd = strioc->ic_cmd;
5940     iocbp->ioc_flag = (fflags & FMODELS);

5942     crhold(crp);
5943     iocbp->ioc_cr = crp;
5944     DB_TYPE(bp) = M_IOCTL;
5945     bp->b_wptr += sizeof (struct iocblk);

5947     if (flag & STR_NOERROR)
5948         errs = STPLEX;
5949     else
5950         errs = STRHUP|STRDERR|STWRERR|STPLEX;

5952     /*
5953     * If there is data to copy into ioctl block, do so.
5954     */
5955     if (iocbp->ioc_count > 0) {
5956         if (transparent)
5957             /*
5958             * Note: STR_NOERROR does not have an effect
5959             * in putiocd()
5960             */
5961             id = K_TO_K | sigflag;
5962         else
5963             id = flag;
5964         if ((error = putiocd(bp, strioc->ic_dp, id, crp)) != 0) {
5965             freemsg(bp);
5966             crfree(crp);
5967             return (error);
5968         }
5970     /*
5971     * We could have slept copying in user pages.
5972     * Recheck the stream head state (the other end
5973     * of a pipe could have gone away).
5974     */
5975     if (stp->sd_flag & errs) {
5976         mutex_enter(&stp->sd_lock);
5977         error = strgeterr(stp, errs, 0);
5978         mutex_exit(&stp->sd_lock);
5979         if (error != 0) {
5980             freemsg(bp);
5981             crfree(crp);
5982             return (error);
5983         }
5984     }
5985 }
5986 if (transparent)
5987     iocbp->ioc_count = TRANSPARENT;

5989 /*
5990 * Block for up to STRTIMOUT milliseconds if there is an outstanding
5991 * ioctl for this stream already running. All processes
5992 * sleeping here will be awakened as a result of an ACK
5993 * or NAK being received for the outstanding ioctl, or
5994 * as a result of the timer expiring on the outstanding
5995 * ioctl (a failure), or as a result of any waiting
5996 * process's timer expiring (also a failure).
5997 */

5999     error = 0;
6000     mutex_enter(&stp->sd_lock);
6001     while ((stp->sd_flag & IOCWAIT) ||

```

```

6002         (!set_iocwaitne && (stp->sd_flag & IOCWAITNE))) {
6003             clock_t cv_rval;

6005             TRACE_0(TR_FAC_STREAMS_FR,
6006                 TR_STRDIOCTL_WAIT,
6007                 "strdioioctl sleeps - IOCWAIT");
6008             cv_rval = str_cv_wait(&stp->sd_iocmonitor, &stp->sd_lock,
6009                 STRTIMOUT, sigflag);
6010             if (cv_rval <= 0) {
6011                 if (cv_rval == 0) {
6012                     error = EINTR;
6013                 } else {
6014                     if (flag & STR_NOERROR) {
6015                         /*
6016                         * Terminating current ioctl in
6017                         * progress -- assume it got lost and
6018                         * wake up the other thread so that the
6019                         * operation completes.
6020                         */
6021                         if (!(stp->sd_flag & IOCWAITNE)) {
6022                             set_iocwaitne = B_TRUE;
6023                             stp->sd_flag |= IOCWAITNE;
6024                             cv_broadcast(&stp->sd_monitor);
6025                         }
6026                         /*
6027                         * Otherwise, there's a running
6028                         * STR_NOERROR -- we have no choice
6029                         * here but to wait forever (or until
6030                         * interrupted).
6031                         */
6032                     } else {
6033                         /*
6034                         * pending ioctl has caused
6035                         * us to time out
6036                         */
6037                         error = ETIME;
6038                     }
6039                 }
6040             } else if ((stp->sd_flag & errs)) {
6041                 error = strgeterr(stp, errs, 0);
6042             }
6043             if (error) {
6044                 mutex_exit(&stp->sd_lock);
6045                 freemsg(bp);
6046                 crfree(crp);
6047                 return (error);
6048             }
6049         }

6051     /*
6052     * Have control of ioctl mechanism.
6053     * Send down ioctl packet and wait for response.
6054     */
6055     if (stp->sd_iocblk != (mblk_t *)-1) {
6056         freemsg(stp->sd_iocblk);
6057     }
6058     stp->sd_iocblk = NULL;

6060     /*
6061     * If this is marked with 'noerror' (internal; mostly
6062     * I_{P,}{UN,}LINK), then make sure nobody else is able to get
6063     * in here by setting IOCWAITNE.
6064     */
6065     waitflags = IOCWAIT;
6066     if (flag & STR_NOERROR)
6067         waitflags |= IOCWAITNE;

```

```

6069     stp->sd_flag |= waitflags;
6071     /*
6072      * Assign sequence number.
6073      */
6074     iocbp->ioc_id = stp->sd_iocid = getiocseqno();

6076     mutex_exit(&stp->sd_lock);

6078     TRACE_1(TR_FAC_STREAMS_FR,
6079            TR_STRDOIOCTL_PUT, "strdoioctl put: stp %p", stp);
6080     stream_willservice(stp);
6081     putnext(stp->sd_wrq, bp);
6082     stream_runservice(stp);

6084     /*
6085      * Timed wait for acknowledgment. The wait time is limited by the
6086      * timeout value, which must be a positive integer (number of
6087      * milliseconds) to wait, or 0 (use default value of STRTIMEOUT
6088      * milliseconds), or -1 (wait forever). This will be awakened
6089      * either by an ACK/NAK message arriving, the timer expiring, or
6090      * the timer expiring on another ioctl waiting for control of the
6091      * mechanism.
6092      */
6093     waitioc:
6094     mutex_enter(&stp->sd_lock);

6097     /*
6098      * If the reply has already arrived, don't sleep. If awakened from
6099      * the sleep, fail only if the reply has not arrived by then.
6100      * Otherwise, process the reply.
6101      */
6102     while (!stp->sd_iocblk) {
6103         clock_t cv_rval;

6105         if (stp->sd_flag & errs) {
6106             error = strgeterr(stp, errs, 0);
6107             if (error != 0) {
6108                 stp->sd_flag &= ~waitflags;
6109                 cv_broadcast(&stp->sd_iocmonitor);
6110                 mutex_exit(&stp->sd_lock);
6111                 crfree(crp);
6112                 return (error);
6113             }
6114         }

6116         TRACE_0(TR_FAC_STREAMS_FR,
6117                TR_STRDOIOCTL_WAIT2,
6118                "strdoioctl sleeps awaiting reply");
6119         ASSERT(error == 0);

6121         cv_rval = str_cv_wait(&stp->sd_monitor, &stp->sd_lock,
6122                             (strioc->ic_timeout ?
6123                              strioc->ic_timeout * 1000 : STRTIMEOUT), sigflag);

6125         /*
6126          * There are four possible cases here: interrupt, timeout,
6127          * wakeup by IOCWAITNE (above), or wakeup by strrput_nondata (a
6128          * valid M_IOCTL reply).
6129          *
6130          * If we've been awakened by a STR_NOERROR ioctl on some other
6131          * thread, then sd_iocblk will still be NULL, and IOCWAITNE
6132          * will be set. Pretend as if we just timed out. Note that
6133          * this other thread waited at least STRTIMEOUT before trying to

```

```

6134         * awakened our thread, so this is indistinguishable (even for
6135         * INFTIM) from the case where we failed with ETIME waiting on
6136         * IOCWAIT in the prior loop.
6137         */
6138         if (cv_rval > 0 && !(flag & STR_NOERROR) &&
6139             stp->sd_iocblk == NULL && (stp->sd_flag & IOCWAITNE)) {
6140             cv_rval = -1;
6141         }

6143         /*
6144          * note: STR_NOERROR does not protect
6145          * us here.. use ic_timeout < 0
6146          */
6147         if (cv_rval <= 0) {
6148             if (cv_rval == 0) {
6149                 error = EINTR;
6150             } else {
6151                 error = ETIME;
6152             }
6153             /*
6154              * A message could have come in after we were scheduled
6155              * but before we were actually run.
6156              */
6157             bp = stp->sd_iocblk;
6158             stp->sd_iocblk = NULL;
6159             if (bp != NULL) {
6160                 if ((bp->b_datap->db_type == M_COPYIN) ||
6161                     (bp->b_datap->db_type == M_COPYOUT)) {
6162                     mutex_exit(&stp->sd_lock);
6163                     if (bp->b_cont) {
6164                         freemsg(bp->b_cont);
6165                         bp->b_cont = NULL;
6166                     }
6167                     bp->b_datap->db_type = M_IOCTLDATA;
6168                     bp->b_wptr = bp->b_rptr +
6169                         sizeof (struct copyresp);
6170                     resp = (struct copyresp *)bp->b_rptr;
6171                     resp->cp_rval =
6172                         (caddr_t)1; /* failure */
6173                     stream_willservice(stp);
6174                     putnext(stp->sd_wrq, bp);
6175                     stream_runservice(stp);
6176                     mutex_enter(&stp->sd_lock);
6177                 } else {
6178                     freemsg(bp);
6179                 }
6180             }
6181             stp->sd_flag &= ~waitflags;
6182             cv_broadcast(&stp->sd_iocmonitor);
6183             mutex_exit(&stp->sd_lock);
6184             crfree(crp);
6185             return (error);
6186         }
6187     }
6188     bp = stp->sd_iocblk;
6189     /*
6190      * Note: it is strictly impossible to get here with sd_iocblk set to
6191      * -1. This is because the initial loop above doesn't allow any new
6192      * ioctls into the fray until all others have passed this point.
6193      */
6194     ASSERT(bp != NULL && bp != (mblock_t *)-1);
6195     TRACE_1(TR_FAC_STREAMS_FR,
6196            TR_STRDOIOCTL_ACK, "strdoioctl got reply: bp %p", bp);
6197     if ((bp->b_datap->db_type == M_IOCACK) ||
6198         (bp->b_datap->db_type == M_IOCNAK)) {
6199         /* for detection of duplicate ioctl replies */

```

```

6200         stp->sd_iocblk = (mblk_t *)-1;
6201         stp->sd_flag &= ~waitflags;
6202         cv_broadcast(&stp->sd_iocmonitor);
6203         mutex_exit(&stp->sd_lock);
6204     } else {
6205         /*
6206          * flags not cleared here because we're still doing
6207          * copy in/out for ioctl.
6208          */
6209         stp->sd_iocblk = NULL;
6210         mutex_exit(&stp->sd_lock);
6211     }

6214     /*
6215      * Have received acknowledgment.
6216     */

6218     switch (bp->b_datap->db_type) {
6219     case M_IOCACK:
6220         /*
6221          * Positive ack.
6222          */
6223         iocbp = (struct iocblk *)bp->b_rptr;

6225         /*
6226          * Set error if indicated.
6227          */
6228         if (iocbp->ioc_error) {
6229             error = iocbp->ioc_error;
6230             break;
6231         }

6233         /*
6234          * Set return value.
6235          */
6236         *rvalp = iocbp->ioc_rval;

6238         /*
6239          * Data may have been returned in ACK message (ioc_count > 0).
6240          * If so, copy it out to the user's buffer.
6241          */
6242         if (iocbp->ioc_count && !transparent) {
6243             if (error = getiocd(bp, strioc->ic_dp, copyflag))
6244                 break;
6245         }
6246         if (!transparent) {
6247             if (len) /* an M_COPYOUT was used with I_STR */
6248                 strioc->ic_len = len;
6249             else
6250                 strioc->ic_len = (int)iocbp->ioc_count;
6251         }
6252         break;

6254     case M_IOCNAK:
6255         /*
6256          * Negative ack.
6257          *
6258          * The only thing to do is set error as specified
6259          * in neg ack packet.
6260          */
6261         iocbp = (struct iocblk *)bp->b_rptr;

6263         error = (iocbp->ioc_error ? iocbp->ioc_error : EINVAL);
6264         break;

```

```

6266     case M_COPYIN:
6267         /*
6268          * Driver or module has requested user ioctl data.
6269          */
6270         reqp = (struct copyreq *)bp->b_rptr;

6272         /*
6273          * M_COPYIN should *never* have a message attached, though
6274          * it's harmless if it does -- thus, panic on a DEBUG
6275          * kernel and just free it on a non-DEBUG build.
6276          */
6277         ASSERT(bp->b_cont == NULL);
6278         if (bp->b_cont != NULL) {
6279             freemsg(bp->b_cont);
6280             bp->b_cont = NULL;
6281         }

6283         error = putiocd(bp, reqp->cq_addr, flag, crp);
6284         if (error && bp->b_cont) {
6285             freemsg(bp->b_cont);
6286             bp->b_cont = NULL;
6287         }

6289         bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
6290         bp->b_datap->db_type = M_IOCADATA;

6292         mblk_setcred(bp, crp, curproc->p_pid);
6293         resp = (struct copyresp *)bp->b_rptr;
6294         resp->cp_rval = (caddr_t)(uintptr_t)error;
6295         resp->cp_flag = (fflags & FMODELS);

6297         stream_willservice(stp);
6298         putnext(stp->sd_wrq, bp);
6299         stream_runservice(stp);

6301         if (error) {
6302             mutex_enter(&stp->sd_lock);
6303             stp->sd_flag &= ~waitflags;
6304             cv_broadcast(&stp->sd_iocmonitor);
6305             mutex_exit(&stp->sd_lock);
6306             crfree(crp);
6307             return (error);
6308         }

6310         goto waitioc;

6312     case M_COPYOUT:
6313         /*
6314          * Driver or module has ioctl data for a user.
6315          */
6316         reqp = (struct copyreq *)bp->b_rptr;
6317         ASSERT(bp->b_cont != NULL);

6319         /*
6320          * Always (transparent or non-transparent)
6321          * use the address specified in the request
6322          */
6323         taddr = reqp->cq_addr;
6324         if (!transparent)
6325             len = (int)reqp->cq_size;

6327         /* copyout data to the provided address */
6328         error = getiocd(bp, taddr, copyflag);

6330         freemsg(bp->b_cont);
6331         bp->b_cont = NULL;

```

```

6333         bp->b_wptr = bp->b_rptr + sizeof (struct copyresp);
6334         bp->b_datap->db_type = M_IOCTLDATA;

6336         mblk_setcred(bp, crp, curproc->p_pid);
6337         resp = (struct copyresp *)bp->b_rptr;
6338         resp->cp_rval = (caddr_t)(uintptr_t)error;
6339         resp->cp_flag = (fflags & FMODELS);

6341         stream_willservice(stp);
6342         putnext(stp->sd_wrq, bp);
6343         stream_runservice(stp);

6345         if (error) {
6346             mutex_enter(&stp->sd_lock);
6347             stp->sd_flag &= ~waitflags;
6348             cv_broadcast(&stp->sd_iocmonitor);
6349             mutex_exit(&stp->sd_lock);
6350             crfree(crp);
6351             return (error);
6352         }
6353         goto waitioc;

6355     default:
6356         ASSERT(0);
6357         mutex_enter(&stp->sd_lock);
6358         stp->sd_flag &= ~waitflags;
6359         cv_broadcast(&stp->sd_iocmonitor);
6360         mutex_exit(&stp->sd_lock);
6361         break;
6362     }

6364     freemsg(bp);
6365     crfree(crp);
6366     return (error);
6367 }

6369 /*
6370  * Send an M_CMD message downstream and wait for a reply. This is a ptools
6371  * special used to retrieve information from modules/drivers a stream without
6372  * being subjected to flow control or interfering with pending messages on the
6373  * stream (e.g. an ioctl in flight).
6374  */
6375 int
6376 strdocmd(struct stdata *stp, struct strcmd *scp, cred_t *crp)
6377 {
6378     mblk_t *mp;
6379     struct cmdblk *cmdp;
6380     int error = 0;
6381     int errs = STRHUP|STRDERR|STWRERR|STPLEX;
6382     clock_t rval, timeout = STRTIMOUT;

6384     if (scp->sc_len < 0 || scp->sc_len > sizeof (scp->sc_buf) ||
6385         scp->sc_timeout < -1)
6386         return (EINVAL);

6388     if (scp->sc_timeout > 0)
6389         timeout = scp->sc_timeout * MILLISEC;

6391     if ((mp = allocb_cred(sizeof (struct cmdblk), crp,
6392         curproc->p_pid)) == NULL)
6393         return (ENOMEM);

6395     crhold(crp);

6397     cmdp = (struct cmdblk *)mp->b_wptr;

```

```

6398     cmdp->cb_cr = crp;
6399     cmdp->cb_cmd = scp->sc_cmd;
6400     cmdp->cb_len = scp->sc_len;
6401     cmdp->cb_error = 0;
6402     mp->b_wptr += sizeof (struct cmdblk);

6404     DB_TYPE(mp) = M_CMD;
6405     DB_CPID(mp) = curproc->p_pid;

6407     /*
6408      * Copy in the payload.
6409      */
6410     if (cmdp->cb_len > 0) {
6411         mp->b_cont = allocb_cred(sizeof (scp->sc_buf), crp,
6412             curproc->p_pid);
6413         if (mp->b_cont == NULL) {
6414             error = ENOMEM;
6415             goto out;
6416         }

6418         /* cb_len comes from sc_len, which has already been checked */
6419         ASSERT(cmdp->cb_len <= sizeof (scp->sc_buf));
6420         (void) bcopy(scp->sc_buf, mp->b_cont->b_wptr, cmdp->cb_len);
6421         mp->b_cont->b_wptr += cmdp->cb_len;
6422         DB_CPID(mp->b_cont) = curproc->p_pid;
6423     }

6425     /*
6426      * Since this mechanism is strictly for ptools, and since only one
6427      * process can be grabbed at a time, we simply fail if there's
6428      * currently an operation pending.
6429      */
6430     mutex_enter(&stp->sd_lock);
6431     if (stp->sd_flag & STRCMDWAIT) {
6432         mutex_exit(&stp->sd_lock);
6433         error = EBUSY;
6434         goto out;
6435     }
6436     stp->sd_flag |= STRCMDWAIT;
6437     ASSERT(stp->sd_cmdblk == NULL);
6438     mutex_exit(&stp->sd_lock);

6440     putnext(stp->sd_wrq, mp);
6441     mp = NULL;

6443     /*
6444      * Timed wait for acknowledgment. If the reply has already arrived,
6445      * don't sleep. If awakened from the sleep, fail only if the reply
6446      * has not arrived by then. Otherwise, process the reply.
6447      */
6448     mutex_enter(&stp->sd_lock);
6449     while (stp->sd_cmdblk == NULL) {
6450         if (stp->sd_flag & errs) {
6451             if ((error = strgeterr(stp, errs, 0)) != 0)
6452                 goto waitout;
6453         }

6455         rval = str_cv_wait(&stp->sd_monitor, &stp->sd_lock, timeout, 0);
6456         if (stp->sd_cmdblk != NULL)
6457             break;

6459         if (rval <= 0) {
6460             error = (rval == 0) ? EINTR : ETIME;
6461             goto waitout;
6462         }
6463     }

```

```

6465  /*
6466   * We received a reply.
6467   */
6468  mp = stp->sd_cmdblk;
6469  stp->sd_cmdblk = NULL;
6470  ASSERT(mp != NULL && DB_TYPE(mp) == M_CMD);
6471  ASSERT(stp->sd_flag & STRCMDWAIT);
6472  stp->sd_flag &= ~STRCMDWAIT;
6473  mutex_exit(&stp->sd_lock);

6475  cmdp = (struct cmdblk *)mp->b_rptr;
6476  if ((error = cmdp->cb_error) != 0)
6477      goto out;

6479  /*
6480   * Data may have been returned in the reply (cb_len > 0).
6481   * If so, copy it out to the user's buffer.
6482   */
6483  if (cmdp->cb_len > 0) {
6484      if (mp->b_cont == NULL || MBLKL(mp->b_cont) < cmdp->cb_len) {
6485          error = EPROTO;
6486          goto out;
6487      }

6489      cmdp->cb_len = MIN(cmdp->cb_len, sizeof (scp->sc_buf));
6490      (void) bcopy(mp->b_cont->b_rptr, scp->sc_buf, cmdp->cb_len);
6491  }
6492  scp->sc_len = cmdp->cb_len;
6493  out:
6494  freemsg(mp);
6495  crfree(crp);
6496  return (error);
6497  waitout:
6498  ASSERT(stp->sd_cmdblk == NULL);
6499  stp->sd_flag &= ~STRCMDWAIT;
6500  mutex_exit(&stp->sd_lock);
6501  crfree(crp);
6502  return (error);
6503  }

6505  /*
6506   * For the SunOS keyboard driver.
6507   * Return the next available "ioctl" sequence number.
6508   * Exported, so that streams modules can send "ioctl" messages
6509   * downstream from their open routine.
6510   */
6511  int
6512  getiocseqno(void)
6513  {
6514      int    i;

6516      mutex_enter(&strresources);
6517      i = ++ioc_id;
6518      mutex_exit(&strresources);
6519      return (i);
6520  }

6522  /*
6523   * Get the next message from the read queue.  If the message is
6524   * priority, STRPRI will have been set by strrrpt().  This flag
6525   * should be reset only when the entire message at the front of the
6526   * queue as been consumed.
6527   *
6528   * NOTE: strgetmsg and kstrgetmsg have much of the logic in common.
6529   */

```

```

6530  int
6531  strgetmsg(
6532      struct vnode *vp,
6533      struct strbuf *mctl,
6534      struct strbuf *mdata,
6535      unsigned char *prip,
6536      int *flagsp,
6537      int fmode,
6538      rval_t *rvp)
6539  {
6540      struct stdata *stp;
6541      mblk_t *bp, *nbp;
6542      mblk_t *savemp = NULL;
6543      mblk_t *savemptail = NULL;
6544      uint_t old_sd_flag;
6545      int flg;
6546      int more = 0;
6547      int error = 0;
6548      char first = 1;
6549      uint_t mark;
6550  #define _LASTMARK      0x8000 /* Contains MSG*MARK and _LASTMARK */
6551      unsigned char pri = 0; /* Distinct from MSG*MARK */
6552      queue_t *q;
6553      int pr = 0; /* Partial read successful */
6554      struct uio uiops;
6555      struct uio *uiop = &uiops;
6556      struct iovec iovs;
6557      unsigned char type;

6559      TRACE_1(TR_FAC_STREAMS_FR, TR_STRGETMSG_ENTER,
6560             "strgetmsg:%p", vp);

6562      ASSERT(vp->v_stream);
6563      stp = vp->v_stream;
6564      rvp->r_vall = 0;

6566      mutex_enter(&stp->sd_lock);

6568      if ((error = i_straccess(stp, JCREAD)) != 0) {
6569          mutex_exit(&stp->sd_lock);
6570          return (error);
6571      }

6573      if (stp->sd_flag & (STRDERR|STPLEX)) {
6574          error = strgeterr(stp, STRDERR|STPLEX, 0);
6575          if (error != 0) {
6576              mutex_exit(&stp->sd_lock);
6577              return (error);
6578          }
6579      }
6580      mutex_exit(&stp->sd_lock);

6582      switch (*flagsp) {
6583      case MSG_HIPRI:
6584          if (*prip != 0)
6585              return (EINVAL);
6586          break;

6588      case MSG_ANY:
6589      case MSG_BAND:
6590          break;

6592      default:
6593          return (EINVAL);
6594      }
6595      /*

```

```

6596     * Setup uio and iov for data part
6597     */
6598     iovs.iov_base = mdata->buf;
6599     iovs.iov_len = mdata->maxlen;
6600     uios.uio_iov = &iovs;
6601     uios.uio_iovcnt = 1;
6602     uios.uio_loffset = 0;
6603     uios.uio_segflg = UIO_USERSPACE;
6604     uios.uio_fmode = 0;
6605     uios.uio_extflg = UIO_COPY_CACHED;
6606     uios.uio_resid = mdata->maxlen;
6607     uios.uio_offset = 0;

6609     q = _RD(stp->sd_wrq);
6610     mutex_enter(&stp->sd_lock);
6611     old_sd_flag = stp->sd_flag;
6612     mark = 0;
6613     for (;;) {
6614         int done = 0;
6615         mblk_t *q_first = q->q_first;

6617         /*
6618          * Get the next message of appropriate priority
6619          * from the stream head.  If the caller is interested
6620          * in band or hipri messages, then they should already
6621          * be enqueued at the stream head.  On the other hand
6622          * if the caller wants normal (band 0) messages, they
6623          * might be deferred in a synchronous stream and they
6624          * will need to be pulled up.
6625          *
6626          * After we have dequeued a message, we might find that
6627          * it was a deferred M_SIG that was enqueued at the
6628          * stream head.  It must now be posted as part of the
6629          * read by calling strsignal_nolock().
6630          *
6631          * Also note that strrput does not enqueue an M_PCSIG,
6632          * and there cannot be more than one hipri message,
6633          * so there was no need to have the M_PCSIG case.
6634          *
6635          * At some time it might be nice to try and wrap the
6636          * functionality of kstrgetmsg() and strgetmsg() into
6637          * a common routine so to reduce the amount of replicated
6638          * code (since they are extremely similar).
6639          */
6640         if (!(*flagsp & (MSG_HIPRI|MSG_BAND))) {
6641             /* Asking for normal, band0 data */
6642             bp = strget(stp, q, uiop, first, &error);
6643             ASSERT(MUTEX_HELD(&stp->sd_lock));
6644             if (bp != NULL) {
6645                 if (DB_TYPE(bp) == M_SIG) {
6646                     strsignal_nolock(stp, *bp->b_rptr,
6647                                     bp->b_band);
6648                     freemsg(bp);
6649                     continue;
6650                 } else {
6651                     break;
6652                 }
6653             }
6654             if (error != 0)
6655                 goto getmout;

6657         /*
6658          * We can't depend on the value of STRPRI here because
6659          * the stream head may be in transit.  Therefore, we
6660          * must look at the type of the first message to
6661          * determine if a high priority messages is waiting

```

```

6662         */
6663     } else if ((*flagsp & MSG_HIPRI) && q_first != NULL &&
6664              DB_TYPE(q_first) >= QPCTL &&
6665              (bp = getq_noenab(q, 0)) != NULL) {
6666         /* Asked for HIPRI and got one */
6667         ASSERT(DB_TYPE(bp) >= QPCTL);
6668         break;
6669     } else if ((*flagsp & MSG_BAND) && q_first != NULL &&
6670              ((q_first->b_band >= *prip) || DB_TYPE(q_first) >= QPCTL) &&
6671              (bp = getq_noenab(q, 0)) != NULL) {
6672         /*
6673          * Asked for at least band "prip" and got either at
6674          * least that band or a hipri message.
6675          */
6676         ASSERT(bp->b_band >= *prip || DB_TYPE(bp) >= QPCTL);
6677         if (DB_TYPE(bp) == M_SIG) {
6678             strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
6679             freemsg(bp);
6680             continue;
6681         } else {
6682             break;
6683         }
6684     }

6686     /* No data. Time to sleep? */
6687     qbackenable(q, 0);

6689     /*
6690     * If STRHUP or STREOF, return 0 length control and data.
6691     * If resid is 0, then a read(fd,buf,0) was done. Do not
6692     * sleep to satisfy this request because by default we have
6693     * zero bytes to return.
6694     */
6695     if ((stp->sd_flag & (STRHUP|STREOF)) || (mctl->maxlen == 0 &&
6696         mdata->maxlen == 0)) {
6697         mctl->len = mdata->len = 0;
6698         *flagsp = 0;
6699         mutex_exit(&stp->sd_lock);
6700         return (0);
6701     }
6702     TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_WAIT,
6703            "strgetmsg calls strwaitq:%p, %p",
6704            vp, uiop);
6705     if ((error = strwaitq(stp, GETWAIT, (ssize_t)0, fmode, -1,
6706                          &done)) != 0 || done) {
6707         TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_DONE,
6708            "strgetmsg error or done:%p, %p",
6709            vp, uiop);
6710         mutex_exit(&stp->sd_lock);
6711         return (error);
6712     }
6713     TRACE_2(TR_FAC_STREAMS_FR, TR_STRGETMSG_AWAKE,
6714            "strgetmsg awakes:%p, %p", vp, uiop);
6715     if ((error = i_straccess(stp, JCREAD)) != 0) {
6716         mutex_exit(&stp->sd_lock);
6717         return (error);
6718     }
6719     first = 0;
6720 }
6721 ASSERT(bp != NULL);
6722 /*
6723  * Extract any mark information.  If the message is not completely
6724  * consumed this information will be put in the mblk
6725  * that is putback.
6726  * If MSGMARKNEXT is set and the message is completely consumed
6727  * the STRATMARK flag will be set below.  Likewise, if

```

```

6728     * MSGNOTMARKNEXT is set and the message is
6729     * completely consumed STRNOTATMARK will be set.
6730     */
6731     mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
6732     ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
6733            (MSGMARKNEXT|MSGNOTMARKNEXT));
6734     if (mark != 0 && bp == stp->sd_mark) {
6735         mark |= _LASTMARK;
6736         stp->sd_mark = NULL;
6737     }
6738     /*
6739     * keep track of the original message type and priority
6740     */
6741     pri = bp->b_band;
6742     type = bp->b_datap->db_type;
6743     if (type == M_PASSFP) {
6744         if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
6745             stp->sd_mark = bp;
6746         bp->b_flag |= mark & ~_LASTMARK;
6747         putback(stp, q, bp, pri);
6748         qbackenable(q, pri);
6749         mutex_exit(&stp->sd_lock);
6750         return (EBADMSG);
6751     }
6752     ASSERT(type != M_SIG);

6754     /*
6755     * Set this flag so strrput will not generate signals. Need to
6756     * make sure this flag is cleared before leaving this routine
6757     * else signals will stop being sent.
6758     */
6759     stp->sd_flag |= STRGETINPROG;
6760     mutex_exit(&stp->sd_lock);

6762     if (STREAM_NEEDSERVICE(stp))
6763         stream_runservice(stp);

6765     /*
6766     * Set HIPRI flag if message is priority.
6767     */
6768     if (type >= QPCTL)
6769         flg = MSG_HIPRI;
6770     else
6771         flg = MSG_BAND;

6773     /*
6774     * First process PROTO or PCPROTO blocks, if any.
6775     */
6776     if (mctl->maxlen >= 0 && type != M_DATA) {
6777         size_t n, bcnt;
6778         char *ubuf;

6780         bcnt = mctl->maxlen;
6781         ubuf = mctl->buf;
6782         while (bp != NULL && bp->b_datap->db_type != M_DATA) {
6783             if ((n = MIN(bcnt, bp->b_wptra - bp->b_rptr)) != 0 &&
6784                 copyout(bp->b_rptr, ubuf, n)) {
6785                 error = EFAULT;
6786                 mutex_enter(&stp->sd_lock);
6787                 /*
6788                 * clear stream head pri flag based on
6789                 * first message type
6790                 */
6791                 if (type >= QPCTL) {
6792                     ASSERT(type == M_PCPROTO);
6793                     stp->sd_flag &= ~STRPRI;

```

```

6794         }
6795         more = 0;
6796         freemsg(bp);
6797         goto getmout;
6798     }
6799     ubuf += n;
6800     bp->b_rptr += n;
6801     if (bp->b_rptr >= bp->b_wptra) {
6802         nbp = bp;
6803         bp = bp->b_cont;
6804         freeb(nbp);
6805     }
6806     ASSERT(n <= bcnt);
6807     bcnt -= n;
6808     if (bcnt == 0)
6809         break;
6810     }
6811     mctl->len = mctl->maxlen - bcnt;
6812 } else
6813     mctl->len = -1;

6815     if (bp && bp->b_datap->db_type != M_DATA) {
6816         /*
6817         * More PROTO blocks in msg.
6818         */
6819         more |= MORECTL;
6820         savemp = bp;
6821         while (bp && bp->b_datap->db_type != M_DATA) {
6822             savemtail = bp;
6823             bp = bp->b_cont;
6824         }
6825         savemtail->b_cont = NULL;
6826     }

6828     /*
6829     * Now process DATA blocks, if any.
6830     */
6831     if (mdata->maxlen >= 0 && bp) {
6832         /*
6833         * struiocopyout will consume a potential zero-length
6834         * M_DATA even if uiop_resid is zero.
6835         */
6836         size_t oldresid = uiop->uiop_resid;

6838         bp = struiocopyout(bp, uiop, &error);
6839         if (error != 0) {
6840             mutex_enter(&stp->sd_lock);
6841             /*
6842             * clear stream head hi pri flag based on
6843             * first message
6844             */
6845             if (type >= QPCTL) {
6846                 ASSERT(type == M_PCPROTO);
6847                 stp->sd_flag &= ~STRPRI;
6848             }
6849             more = 0;
6850             freemsg(savemp);
6851             goto getmout;
6852         }
6853         /*
6854         * (pr == 1) indicates a partial read.
6855         */
6856         if (oldresid > uiop->uiop_resid)
6857             pr = 1;
6858         mdata->len = mdata->maxlen - uiop->uiop_resid;
6859     } else

```

```

6860         mdata->len = -1;
6862     if (bp) {
6863         more |= MOREDATA;
6864         if (savemp)
6865             savemtail->b_cont = bp;
6866         else
6867             savemp = bp;
6868     }
6870     mutex_enter(&stp->sd_lock);
6871     if (savemp) {
6872         if (pr && (savemp->b_datap->db_type == M_DATA) &&
6873             msgnodata(savemp)) {
6874             /*
6875              * Avoid queuing a zero-length tail part of
6876              * a message. pr=1 indicates that we read some of
6877              * the message.
6878              */
6879             freemsg(savemp);
6880             more &= ~MOREDATA;
6881             /*
6882              * clear stream head hi pri flag based on
6883              * first message
6884              */
6885             if (type >= QPCTL) {
6886                 ASSERT(type == M_PCPROTO);
6887                 stp->sd_flag &= ~STRPRI;
6888             }
6889         } else {
6890             savemp->b_band = pri;
6891             /*
6892              * If the first message was HIPRI and the one we're
6893              * putting back isn't, then clear STRPRI, otherwise
6894              * set STRPRI again. Note that we must set STRPRI
6895              * again since the flush logic in strrrput_nondata()
6896              * may have cleared it while we had sd_lock dropped.
6897              */
6898             if (type >= QPCTL) {
6899                 ASSERT(type == M_PCPROTO);
6900                 if (queclass(savemp) < QPCTL)
6901                     stp->sd_flag &= ~STRPRI;
6902                 else
6903                     stp->sd_flag |= STRPRI;
6904             } else if (queclass(savemp) >= QPCTL) {
6905                 /*
6906                  * The first message was not a HIPRI message,
6907                  * but the one we are about to putback is.
6908                  * For simplicity, we do not allow for HIPRI
6909                  * messages to be embedded in the message
6910                  * body, so just force it to same type as
6911                  * first message.
6912                  */
6913                 ASSERT(type == M_DATA || type == M_PROTO);
6914                 ASSERT(savemp->b_datap->db_type == M_PCPROTO);
6915                 savemp->b_datap->db_type = type;
6916             }
6917             if (mark != 0) {
6918                 savemp->b_flag |= mark & ~LASTMARK;
6919                 if ((mark & _LASTMARK) &&
6920                     (stp->sd_mark == NULL)) {
6921                     /*
6922                      * If another marked message arrived
6923                      * while sd_lock was not held sd_mark
6924                      * would be non-NULL.
6925                      */

```

```

6926         stp->sd_mark = savemp;
6927     }
6928     }
6929     putback(stp, q, savemp, pri);
6930 }
6931 } else {
6932     /*
6933     * The complete message was consumed.
6934     *
6935     * If another M_PCPROTO arrived while sd_lock was not held
6936     * it would have been discarded since STRPRI was still set.
6937     *
6938     * Move the MSG*MARKNEXT information
6939     * to the stream head just in case
6940     * the read queue becomes empty.
6941     * clear stream head hi pri flag based on
6942     * first message
6943     *
6944     * If the stream head was at the mark
6945     * (STRATMARK) before we dropped sd_lock above
6946     * and some data was consumed then we have
6947     * moved past the mark thus STRATMARK is
6948     * cleared. However, if a message arrived in
6949     * strrrput during the copyout above causing
6950     * STRATMARK to be set we can not clear that
6951     * flag.
6952     */
6953     if (type >= QPCTL) {
6954         ASSERT(type == M_PCPROTO);
6955         stp->sd_flag &= ~STRPRI;
6956     }
6957     if (mark & (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
6958         if (mark & MSGMARKNEXT) {
6959             stp->sd_flag &= ~STRNOTATMARK;
6960             stp->sd_flag |= STRATMARK;
6961         } else if (mark & MSGNOTMARKNEXT) {
6962             stp->sd_flag &= ~STRATMARK;
6963             stp->sd_flag |= STRNOTATMARK;
6964         } else {
6965             stp->sd_flag &= ~(STRATMARK|STRNOTATMARK);
6966         }
6967     } else if (pr && (old_sd_flag & STRATMARK)) {
6968         stp->sd_flag &= ~STRATMARK;
6969     }
6970 }
6972     *flagsp = flg;
6973     *prip = pri;
6975     /*
6976     * Getmsg cleanup processing - if the state of the queue has changed
6977     * some signals may need to be sent and/or poll awakened.
6978     */
6979     getmout:
6980     qbackenable(q, pri);
6982     /*
6983     * We dropped the stream head lock above. Send all M_SIG messages
6984     * before processing stream head for SIGPOLL messages.
6985     */
6986     ASSERT(MUTEX_HELD(&stp->sd_lock));
6987     while ((bp = q->q_first) != NULL &&
6988         (bp->b_datap->db_type == M_SIG)) {
6989         /*
6990         * sd_lock is held so the content of the read queue can not
6991         * change.

```

```

6992     */
6993     bp = getq(q);
6994     ASSERT(bp != NULL && bp->b_datap->db_type == M_SIG);

6996     strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
6997     mutex_exit(&stp->sd_lock);
6998     freemsg(bp);
6999     if (STREAM_NEEDSERVICE(stp))
7000         stream_runservice(stp);
7001     mutex_enter(&stp->sd_lock);
7002 }

7004 /*
7005  * stream head cannot change while we make the determination
7006  * whether or not to send a signal. Drop the flag to allow strrrput
7007  * to send firstmsgsig again.
7008  */
7009 stp->sd_flag &= ~STRGETINPROG;

7011 /*
7012  * If the type of message at the front of the queue changed
7013  * due to the receive the appropriate signals and pollwakeups events
7014  * are generated. The type of changes are:
7015  *     Processed a hipri message, q_first is not hipri.
7016  *     Processed a band X message, and q_first is band Y.
7017  * The generated signals and pollwakeups are identical to what
7018  * strrrput() generates should the message that is now on q_first
7019  * arrive to an empty read queue.
7020  *
7021  * Note: only strrrput will send a signal for a hipri message.
7022  */
7023 if ((bp = q->q_first) != NULL && !(stp->sd_flag & STRPRI)) {
7024     strsigset_t signals = 0;
7025     strpollset_t pollwakeups = 0;

7027     if (flg & MSG_HIPRI) {
7028         /*
7029          * Removed a hipri message. Regular data at
7030          * the front of the queue.
7031          */
7032         if (bp->b_band == 0) {
7033             signals = S_INPUT | S_RDNORM;
7034             pollwakeups = POLLIN | POLLRDNORM;
7035         } else {
7036             signals = S_INPUT | S_RDBAND;
7037             pollwakeups = POLLIN | POLLRDBAND;
7038         }
7039     } else if (pri != bp->b_band) {
7040         /*
7041          * The band is different for the new q_first.
7042          */
7043         if (bp->b_band == 0) {
7044             signals = S_RDNORM;
7045             pollwakeups = POLLIN | POLLRDNORM;
7046         } else {
7047             signals = S_RDBAND;
7048             pollwakeups = POLLIN | POLLRDBAND;
7049         }
7050     }

7052     if (pollwakeups != 0) {
7053         if (pollwakeups == (POLLIN | POLLRDNORM)) {
7054             if (!(stp->sd_rput_opt & SR_POLLIN))
7055                 goto no_pollwake;
7056             stp->sd_rput_opt &= ~SR_POLLIN;
7057         }

```

```

7058         mutex_exit(&stp->sd_lock);
7059         pollwakeups(&stp->sd_polllist, pollwakeups);
7060         mutex_enter(&stp->sd_lock);
7061     }
7062 no_pollwake:

7064         if (stp->sd_sigflags & signals)
7065             strsendsig(stp->sd_siglist, signals, bp->b_band, 0);
7066     }
7067     mutex_exit(&stp->sd_lock);

7069     rvp->r_vall = more;
7070     return (error);
7071 #undef _LASTMARK
7072 }

7074 /*
7075  * Get the next message from the read queue. If the message is
7076  * priority, STRPRI will have been set by strrrput(). This flag
7077  * should be reset only when the entire message at the front of the
7078  * queue as been consumed.
7079  *
7080  * If uiop is NULL all data is returned in mctlp.
7081  * Note that a NULL uiop implies that FNDELAY and FNONBLOCK are assumed
7082  * not enabled.
7083  * The timeout parameter is in milliseconds; -1 for infinity.
7084  * This routine handles the consolidation private flags:
7085  *     MSG_IGNERROR Ignore any stream head error except STPLEX.
7086  *     MSG_DELAYERROR Defer the error check until the queue is empty.
7087  *     MSG_HOLD SIG Hold signals while waiting for data.
7088  *     MSG_IPEEK Only peek at messages.
7089  *     MSG_DISCARDTAIL Discard the tail M_DATA part of the message
7090  *     that doesn't fit.
7091  *     MSG_NOMARK If the message is marked leave it on the queue.
7092  *
7093  * NOTE: strgetmsg and kstrgetmsg have much of the logic in common.
7094  */
7095 int
7096 kstrgetmsg(
7097     struct vnode *vp,
7098     mblk_t **mctlp,
7099     struct uio *uiop,
7100     unsigned char *prip,
7101     int *flagsp,
7102     clock_t timeout,
7103     rval_t *rvp)
7104 {
7105     struct stdata *stp;
7106     mblk_t *bp, *nbp;
7107     mblk_t *savemp = NULL;
7108     mblk_t *savemptail = NULL;
7109     int flags;
7110     uint_t old_sd_flag;
7111     int flg;
7112     int more = 0;
7113     int error = 0;
7114     char first = 1;
7115     uint_t mark; /* Contains MSG*MARK and _LASTMARK */
7116 #define _LASTMARK 0x8000 /* Distinct from MSG*MARK */
7117     unsigned char pri = 0;
7118     queue_t *q;
7119     int pr = 0; /* Partial read successful */
7120     unsigned char type;

7122     TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_ENTER,
7123         "kstrgetmsg:%p", vp);

```

```

7125     ASSERT(vp->v_stream);
7126     stp = vp->v_stream;
7127     rvp->r_vall = 0;

7129     mutex_enter(&stp->sd_lock);

7131     if ((error = i_straccess(stp, JCREAD)) != 0) {
7132         mutex_exit(&stp->sd_lock);
7133         return (error);
7134     }

7136     flags = *flagsp;
7137     if (stp->sd_flag & (STRDERR|STPLEX)) {
7138         if ((stp->sd_flag & STPLEX) ||
7139             (flags & (MSG_IGNERROR|MSG_DELAYERROR)) == 0) {
7140             error = strgeterr(stp, STRDERR|STPLEX,
7141                             (flags & MSG_IPEEK));
7142             if (error != 0) {
7143                 mutex_exit(&stp->sd_lock);
7144                 return (error);
7145             }
7146         }
7147     }
7148     mutex_exit(&stp->sd_lock);

7150     switch (flags & (MSG_HIPRI|MSG_ANY|MSG_BAND)) {
7151     case MSG_HIPRI:
7152         if (*prip != 0)
7153             return (EINVAL);
7154         break;

7156     case MSG_ANY:
7157     case MSG_BAND:
7158         break;

7160     default:
7161         return (EINVAL);
7162     }

7164     retry:
7165     q = _RD(stp->sd_wrq);
7166     mutex_enter(&stp->sd_lock);
7167     old_sd_flag = stp->sd_flag;
7168     mark = 0;
7169     for (;;) {
7170         int done = 0;
7171         int waitflag;
7172         int fmode;
7173         mblk_t *q_first = q->q_first;

7175         /*
7176          * This section of the code operates just like the code
7177          * in strgetmsg(). There is a comment there about what
7178          * is going on here.
7179          */
7180         if (!(flags & (MSG_HIPRI|MSG_BAND))) {
7181             /* Asking for normal, band0 data */
7182             bp = strget(stp, q, uiop, first, &error);
7183             ASSERT(MUTEX_HELD(&stp->sd_lock));
7184             if (bp != NULL) {
7185                 if (DB_TYPE(bp) == M_SIG) {
7186                     strsignal_nolock(stp, *bp->b_rptr,
7187                                     bp->b_band);
7188                     freemsg(bp);
7189                     continue;

```

```

7190     } else {
7191         break;
7192     }
7193     }
7194     if (error != 0) {
7195         goto getmout;
7196     }
7197     /*
7198     * We can't depend on the value of STRPRI here because
7199     * the stream head may be in transit. Therefore, we
7200     * must look at the type of the first message to
7201     * determine if a high priority messages is waiting
7202     */
7203     } else if ((flags & MSG_HIPRI) && q_first != NULL &&
7204              DB_TYPE(q_first) >= QPCTL &&
7205              (bp = getq_noenab(q, 0)) != NULL) {
7206         ASSERT(DB_TYPE(bp) >= QPCTL);
7207         break;
7208     } else if ((flags & MSG_BAND) && q_first != NULL &&
7209              ((q_first->b_band >= *prip) || DB_TYPE(q_first) >= QPCTL) &&
7210              (bp = getq_noenab(q, 0)) != NULL) {
7211         /*
7212          * Asked for at least band "prip" and got either at
7213          * least that band or a hipri message.
7214          */
7215         ASSERT(bp->b_band >= *prip || DB_TYPE(bp) >= QPCTL);
7216         if (DB_TYPE(bp) == M_SIG) {
7217             strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
7218             freemsg(bp);
7219             continue;
7220         } else {
7221             break;
7222         }
7223     }

7225     /* No data. Time to sleep? */
7226     qbackenable(q, 0);

7228     /*
7229     * Delayed error notification?
7230     */
7231     if ((stp->sd_flag & (STRDERR|STPLEX)) &&
7232         (flags & (MSG_IGNERROR|MSG_DELAYERROR)) == MSG_DELAYERROR) {
7233         error = strgeterr(stp, STRDERR|STPLEX,
7234                         (flags & MSG_IPEEK));
7235         if (error != 0) {
7236             mutex_exit(&stp->sd_lock);
7237             return (error);
7238         }
7239     }

7241     /*
7242     * If STRHUP or STREOF, return 0 length control and data.
7243     * If a read(fd,buf,0) has been done, do not sleep, just
7244     * return.
7245     *
7246     * If mctlp == NULL and uiop == NULL, then the code will
7247     * do the strwaitq. This is an understood way of saying
7248     * sleep "polling" until a message is received.
7249     */
7250     if ((stp->sd_flag & (STRHUP|STREOF)) ||
7251         (uiop != NULL && uiop->uio_resid == 0)) {
7252         if (mctlp != NULL)
7253             *mctlp = NULL;
7254         *flagsp = 0;
7255         mutex_exit(&stp->sd_lock);

```

```

7256         return (0);
7257     }

7259     waitflag = GETWAIT;
7260     if (flags &
7261         (MSG_HOLDMSG|MSG_IGNORE|MSG_IPEEK|MSG_DELAYERR)) {
7262         if (flags & MSG_HOLDMSG)
7263             waitflag |= STR_NOSIG;
7264         if (flags & MSG_IGNORE)
7265             waitflag |= STR_NOERROR;
7266         if (flags & MSG_IPEEK)
7267             waitflag |= STR_PEEK;
7268         if (flags & MSG_DELAYERR)
7269             waitflag |= STR_DELAYERR;
7270     }
7271     if (uiop != NULL)
7272         fmode = uiop->uio_fmode;
7273     else
7274         fmode = 0;

7276     TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_WAIT,
7277            "kstrgetmsg calls strwaitq:%p, %p",
7278            vp, uiop);
7279     if (((error = strwaitq(stp, waitflag, (ssize_t)0,
7280                          fmode, timeout, &done)) != 0 || done) {
7281         TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_DONE,
7282            "kstrgetmsg error or done:%p, %p",
7283            vp, uiop);
7284         mutex_exit(&stp->sd_lock);
7285         return (error);
7286     }
7287     TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRGETMSG_AWAKE,
7288            "kstrgetmsg awakes:%p, %p", vp, uiop);
7289     if ((error = i_straccess(stp, JCREAD)) != 0) {
7290         mutex_exit(&stp->sd_lock);
7291         return (error);
7292     }
7293     first = 0;
7294 }
7295 ASSERT(bp != NULL);
7296 /*
7297  * Extract any mark information. If the message is not completely
7298  * consumed this information will be put in the mblk
7299  * that is putback.
7300  * If MSGMARKNEXT is set and the message is completely consumed
7301  * the STRATMARK flag will be set below. Likewise, if
7302  * MSGNOTMARKNEXT is set and the message is
7303  * completely consumed STRNOTATMARK will be set.
7304  */
7305 mark = bp->b_flag & (MSGMARK | MSGMARKNEXT | MSGNOTMARKNEXT);
7306 ASSERT((mark & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
7307        (MSGMARKNEXT|MSGNOTMARKNEXT));
7308 pri = bp->b_band;
7309 if (mark != 0) {
7310     /*
7311      * If the caller doesn't want the mark return.
7312      * Used to implement MSG_WAITALL in sockets.
7313      */
7314     if (flags & MSG_NOMARK) {
7315         putback(stp, q, bp, pri);
7316         qbackenable(q, pri);
7317         mutex_exit(&stp->sd_lock);
7318         return (EWOULDBLOCK);
7319     }
7320     if (bp == stp->sd_mark) {
7321         mark |= _LASTMARK;

```

```

7322         stp->sd_mark = NULL;
7323     }
7324 }

7326 /*
7327  * keep track of the first message type
7328  */
7329 type = bp->b_datap->db_type;

7331 if (bp->b_datap->db_type == M_PASSFP) {
7332     if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7333         stp->sd_mark = bp;
7334     bp->b_flag |= mark & ~_LASTMARK;
7335     putback(stp, q, bp, pri);
7336     qbackenable(q, pri);
7337     mutex_exit(&stp->sd_lock);
7338     return (EBADMSG);
7339 }
7340 ASSERT(type != M_SIG);

7342 if (flags & MSG_IPEEK) {
7343     /*
7344      * Clear any struioflag - we do the uiomove over again
7345      * when peeking since it simplifies the code.
7346      *
7347      * Dup the message and put the original back on the queue.
7348      * If dupmsg() fails, try again with copymsg() to see if
7349      * there is indeed a shortage of memory. dupmsg() may fail
7350      * if db_ref in any of the messages reaches its limit.
7351      */
7353     if ((nbp = dupmsg(bp)) == NULL && (nbp = copymsg(bp)) == NULL) {
7354         /*
7355          * Restore the state of the stream head since we
7356          * need to drop sd_lock (strwaitbuf is sleeping).
7357          */
7358         size_t size = msgdsize(bp);

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         mutex_exit(&stp->sd_lock);
7365         error = strwaitbuf(size, BPRI_HI);
7366         if (error) {
7367             /*
7368              * There is no net change to the queue thus
7369              * no need to qbackenable.
7370              */
7371             return (error);
7372         }
7373         goto retry;
7374     }

7376     if ((mark & _LASTMARK) && (stp->sd_mark == NULL))
7377         stp->sd_mark = bp;
7378     bp->b_flag |= mark & ~_LASTMARK;
7379     putback(stp, q, bp, pri);
7380     bp = nbp;
7381 }

7383 /*
7384  * Set this flag so strputc will not generate signals. Need to
7385  * make sure this flag is cleared before leaving this routine
7386  * else signals will stop being sent.
7387  */

```

```

7388     stp->sd_flag |= STRGETINPROG;
7389     mutex_exit(&stp->sd_lock);

7391     if ((stp->sd_rputdatafunc != NULL) && (DB_TYPE(bp) == M_DATA)) {
7392         mblk_t *tmp, *prevmp;

7394         /*
7395          * Put first non-data mblk back to stream head and
7396          * cut the mblk chain so sd_rputdatafunc only sees
7397          * M_DATA mblks. We can skip the first mblk since it
7398          * is M_DATA according to the condition above.
7399          */
7400         for (prevmp = bp, tmp = bp->b_cont; tmp != NULL;
7401              prevmp = tmp, tmp = tmp->b_cont) {
7402             if (DB_TYPE(tmp) != M_DATA) {
7403                 prevmp->b_cont = NULL;
7404                 mutex_enter(&stp->sd_lock);
7405                 putback(stp, q, tmp, tmp->b_band);
7406                 mutex_exit(&stp->sd_lock);
7407                 break;
7408             }
7409         }

7411         bp = (stp->sd_rputdatafunc)(stp->sd_vnode, bp,
7412             NULL, NULL, NULL, NULL);

7414         if (bp == NULL)
7415             goto retry;
7416     }

7418     if (STREAM_NEEDSERVICE(stp))
7419         stream_runservice(stp);

7421     /*
7422      * Set HIPRI flag if message is priority.
7423      */
7424     if (type >= QPCTL)
7425         flg = MSG_HIPRI;
7426     else
7427         flg = MSG_BAND;

7429     /*
7430      * First process PROTO or PCPROTO blocks, if any.
7431      */
7432     if (mctlp != NULL && type != M_DATA) {
7433         mblk_t *nbp;

7435         *mctlp = bp;
7436         while (bp->b_cont && bp->b_cont->b_datap->db_type != M_DATA)
7437             bp = bp->b_cont;
7438         nbp = bp->b_cont;
7439         bp->b_cont = NULL;
7440         bp = nbp;
7441     }

7443     if (bp && bp->b_datap->db_type != M_DATA) {
7444         /*
7445          * More PROTO blocks in msg. Will only happen if mctlp is NULL.
7446          */
7447         more |= MORECTL;
7448         savemp = bp;
7449         while (bp && bp->b_datap->db_type != M_DATA) {
7450             savemptail = bp;
7451             bp = bp->b_cont;
7452         }
7453         savemptail->b_cont = NULL;

```

```

7454     }

7456     /*
7457      * Now process DATA blocks, if any.
7458      */
7459     if (uiop == NULL) {
7460         /* Append data to tail of mctlp */

7462         if (mctlp != NULL) {
7463             mblk_t **mpp = mctlp;

7465             while (*mpp != NULL)
7466                 mpp = &((*mpp)->b_cont);
7467             *mpp = bp;
7468             bp = NULL;
7469         }
7470     } else if (uiop->uio_resid >= 0 && bp) {
7471         size_t oldresid = uiop->uio_resid;

7473         /*
7474          * If a streams message is likely to consist
7475          * of many small mblks, it is pulled up into
7476          * one continuous chunk of memory.
7477          * The size of the first mblk may be bogus because
7478          * successive read() calls on the socket reduce
7479          * the size of this mblk until it is exhausted
7480          * and then the code walks on to the next. Thus
7481          * the size of the mblk may not be the original size
7482          * that was passed up, it's simply a remainder
7483          * and hence can be very small without any
7484          * implication that the packet is badly fragmented.
7485          * So the size of the possible second mblk is
7486          * used to spot a badly fragmented packet.
7487          * see longer comment at top of page
7488          * by mblk_pull_len declaration.
7489          */

7491         if (bp->b_cont != NULL && MBLKL(bp->b_cont) < mblk_pull_len) {
7492             (void) pullupmsg(bp, -1);
7493         }

7495         bp = struiocopyout(bp, uiop, &error);
7496         if (error != 0) {
7497             if (mctlp != NULL) {
7498                 freemsg(*mctlp);
7499                 *mctlp = NULL;
7500             } else
7501                 freemsg(savemp);
7502             mutex_enter(&stp->sd_lock);
7503             /*
7504              * clear stream head hi pri flag based on
7505              * first message
7506              */
7507             if (!(flg & MSG_IPEEK) && (type >= QPCTL)) {
7508                 ASSERT(type == M_PCPROTO);
7509                 stp->sd_flag &= ~STRPRI;
7510             }
7511             more = 0;
7512             goto getmout;
7513         }
7514         /*
7515          * (pr == 1) indicates a partial read.
7516          */
7517         if (oldresid > uiop->uio_resid)
7518             pr = 1;
7519     }

```

```

7521     if (bp) {                               /* more data blocks in msg */
7522         more |= MOREDATA;
7523         if (savemp)
7524             savemtail->b_cont = bp;
7525         else
7526             savemp = bp;
7527     }

7529     mutex_enter(&stp->sd_lock);
7530     if (savemp) {
7531         if (flags & (MSG_IPEEK|MSG_DISCARDTAIL)) {
7532             /*
7533              * When MSG_DISCARDTAIL is set or
7534              * when peeking discard any tail. When peeking this
7535              * is the tail of the dup that was copied out - the
7536              * message has already been putback on the queue.
7537              * Return MOREDATA to the caller even though the data
7538              * is discarded. This is used by sockets (to
7539              * set MSG_TRUNC).
7540              */
7541             freemsg(savemp);
7542             if (!(flags & MSG_IPEEK) && (type >= QPCTL)) {
7543                 ASSERT(type == M_PCPROTO);
7544                 stp->sd_flag &= ~STRPRI;
7545             }
7546         } else if (pr && (savemp->b_datap->db_type == M_DATA) &&
7547             msgnodata(savemp)) {
7548             /*
7549              * Avoid queuing a zero-length tail part of
7550              * a message. pr=1 indicates that we read some of
7551              * the message.
7552              */
7553             freemsg(savemp);
7554             more &= ~MOREDATA;
7555             if (type >= QPCTL) {
7556                 ASSERT(type == M_PCPROTO);
7557                 stp->sd_flag &= ~STRPRI;
7558             }
7559         } else {
7560             savemp->b_band = pri;
7561             /*
7562              * If the first message was HIPRI and the one we're
7563              * putting back isn't, then clear STRPRI, otherwise
7564              * set STRPRI again. Note that we must set STRPRI
7565              * again since the flush logic in strputc_nodata()
7566              * may have cleared it while we had sd_lock dropped.
7567              */
7569             if (type >= QPCTL) {
7570                 ASSERT(type == M_PCPROTO);
7571                 if (queclass(savemp) < QPCTL)
7572                     stp->sd_flag &= ~STRPRI;
7573                 else
7574                     stp->sd_flag |= STRPRI;
7575             } else if (queclass(savemp) >= QPCTL) {
7576                 /*
7577                  * The first message was not a HIPRI message,
7578                  * but the one we are about to putback is.
7579                  * For simplicity, we do not allow for HIPRI
7580                  * messages to be embedded in the message
7581                  * body, so just force it to same type as
7582                  * first message.
7583                  */
7584                 ASSERT(type == M_DATA || type == M_PROTO);
7585                 ASSERT(savemp->b_datap->db_type == M_PCPROTO);

```

```

7586         savemp->b_datap->db_type = type;
7587     }
7588     if (mark != 0) {
7589         if ((mark & _LASTMARK) &&
7590             (stp->sd_mark == NULL)) {
7591             /*
7592              * If another marked message arrived
7593              * while sd_lock was not held sd_mark
7594              * would be non-NULL.
7595              */
7596             stp->sd_mark = savemp;
7597         }
7598         savemp->b_flag |= mark & ~_LASTMARK;
7599     }
7600     putback(stp, q, savemp, pri);
7601 }
7602 } else if (!(flags & MSG_IPEEK)) {
7603     /*
7604      * The complete message was consumed.
7605      *
7606      * If another M_PCPROTO arrived while sd_lock was not held
7607      * it would have been discarded since STRPRI was still set.
7608      *
7609      * Move the MSG*MARKNEXT information
7610      * to the stream head just in case
7611      * the read queue becomes empty.
7612      * clear stream head hi pri flag based on
7613      * first message
7614      *
7615      * If the stream head was at the mark
7616      * (STRATMARK) before we dropped sd_lock above
7617      * and some data was consumed then we have
7618      * moved past the mark thus STRATMARK is
7619      * cleared. However, if a message arrived in
7620      * strputc during the copyout above causing
7621      * STRATMARK to be set we can not clear that
7622      * flag.
7623      * XXX A "perimeter" would help by single-threading strputc,
7624      * strread, strgetmsg and kstrgetmsg.
7625      */
7626     if (type >= QPCTL) {
7627         ASSERT(type == M_PCPROTO);
7628         stp->sd_flag &= ~STRPRI;
7629     }
7630     if (mark & (MSGMARKNEXT|MSGNOTMARKNEXT|MSGMARK)) {
7631         if (mark & MSGMARKNEXT) {
7632             stp->sd_flag &= ~STRNOTATMARK;
7633             stp->sd_flag |= STRATMARK;
7634         } else if (mark & MSGNOTMARKNEXT) {
7635             stp->sd_flag &= ~STRATMARK;
7636             stp->sd_flag |= STRNOTATMARK;
7637         } else {
7638             stp->sd_flag &= ~(STRATMARK|STRNOTATMARK);
7639         }
7640     } else if (pr && (old_sd_flag & STRATMARK)) {
7641         stp->sd_flag &= ~STRATMARK;
7642     }
7643 }
7644
7645 *flagsp = flg;
7646 *prip = pri;

7648 /*
7649 * Getmsg cleanup processing - if the state of the queue has changed
7650 * some signals may need to be sent and/or poll awakened.
7651 */

```

```

7652 getmout:
7653     qbackenable(q, pri);

7655     /*
7656     * We dropped the stream head lock above. Send all M_SIG messages
7657     * before processing stream head for SIGPOLL messages.
7658     */
7659     ASSERT(MUTEX_HELD(&stp->sd_lock));
7660     while ((bp = q->q_first) != NULL &&
7661            (bp->b_datap->db_type == M_SIG)) {
7662         /*
7663         * sd_lock is held so the content of the read queue can not
7664         * change.
7665         */
7666         bp = getq(q);
7667         ASSERT(bp != NULL && bp->b_datap->db_type == M_SIG);

7669         strsignal_nolock(stp, *bp->b_rptr, bp->b_band);
7670         mutex_exit(&stp->sd_lock);
7671         freemsg(bp);
7672         if (STREAM_NEEDSERVICE(stp))
7673             stream_runservice(stp);
7674         mutex_enter(&stp->sd_lock);
7675     }

7677     /*
7678     * stream head cannot change while we make the determination
7679     * whether or not to send a signal. Drop the flag to allow strputc
7680     * to send firstmsgsig again.
7681     */
7682     stp->sd_flag &= ~STRGETINPROG;

7684     /*
7685     * If the type of message at the front of the queue changed
7686     * due to the receive the appropriate signals and pollwakeups events
7687     * are generated. The type of changes are:
7688     *     Processed a hipri message, q_first is not hipri.
7689     *     Processed a band X message, and q_first is band Y.
7690     * The generated signals and pollwakeups are identical to what
7691     * strputc() generates should the message that is now on q_first
7692     * arrive to an empty read queue.
7693     *
7694     * Note: only strputc will send a signal for a hipri message.
7695     */
7696     if ((bp = q->q_first) != NULL && !(stp->sd_flag & STRPRI)) {
7697         strsigset_t signals = 0;
7698         strpollset_t pollwakeups = 0;

7700         if (flg & MSG_HIPRI) {
7701             /*
7702             * Removed a hipri message. Regular data at
7703             * the front of the queue.
7704             */
7705             if (bp->b_band == 0) {
7706                 signals = S_INPUT | S_RDNORM;
7707                 pollwakeups = POLLIN | POLLRDNORM;
7708             } else {
7709                 signals = S_INPUT | S_RDBAND;
7710                 pollwakeups = POLLIN | POLLRDBAND;
7711             }
7712         } else if (pri != bp->b_band) {
7713             /*
7714             * The band is different for the new q_first.
7715             */
7716             if (bp->b_band == 0) {
7717                 signals = S_RDNORM;

```

```

7718         pollwakeups = POLLIN | POLLRDNORM;
7719     } else {
7720         signals = S_RDBAND;
7721         pollwakeups = POLLIN | POLLRDBAND;
7722     }
7723 }

7725     if (pollwakeups != 0) {
7726         if (pollwakeups == (POLLIN | POLLRDNORM)) {
7727             if (!(stp->sd_rput_opt & SR_POLLIN))
7728                 goto no_pollwake;
7729             stp->sd_rput_opt &= ~SR_POLLIN;
7730         }
7731         mutex_exit(&stp->sd_lock);
7732         pollwakeups(&stp->sd_polllist, pollwakeups);
7733         mutex_enter(&stp->sd_lock);
7734     }
7735 no_pollwake:

7737     if (stp->sd_sigflags & signals)
7738         strsendsig(stp->sd_siglist, signals, bp->b_band, 0);
7739 }
7740 mutex_exit(&stp->sd_lock);

7742     rvp->r_vall = more;
7743     return (error);
7744 #undef _LASTMARK
7745 }

7747 /*
7748 * Put a message downstream.
7749 *
7750 * NOTE: strputcmsg and kstrputcmsg have much of the logic in common.
7751 */
7752 int
7753 strputcmsg(
7754     struct vnode *vp,
7755     struct strbuf *mctl,
7756     struct strbuf *mdata,
7757     unsigned char pri,
7758     int flag,
7759     int fmode)
7760 {
7761     struct stdata *stp;
7762     queue_t *wqp;
7763     mblk_t *mp;
7764     ssize_t msgsize;
7765     ssize_t rmin, rmax;
7766     int error;
7767     struct uio uios;
7768     struct uio *uiop = &uios;
7769     struct iovec iovs;
7770     int xpg4 = 0;

7772     ASSERT(vp->v_stream);
7773     stp = vp->v_stream;
7774     wqp = stp->sd_wrq;

7776     /*
7777     * If it is an XPG4 application, we need to send
7778     * SIGPIPE below
7779     */

7781     xpg4 = (flag & MSG_XPG4) ? 1 : 0;
7782     flag &= ~MSG_XPG4;

```

```

7784     if (AU_AUDITING())
7785         audit_strputmsg(vp, mctl, mdata, pri, flag, fmode);
7787     mutex_enter(&stp->sd_lock);
7789     if ((error = i_straccess(stp, JCWRITE)) != 0) {
7790         mutex_exit(&stp->sd_lock);
7791         return (error);
7792     }
7794     if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
7795         error = strwriteable(stp, B_FALSE, xpg4);
7796         if (error != 0) {
7797             mutex_exit(&stp->sd_lock);
7798             return (error);
7799         }
7800     }
7802     mutex_exit(&stp->sd_lock);
7804     /*
7805      * Check for legal flag value.
7806      */
7807     switch (flag) {
7808     case MSG_HIPRI:
7809         if ((mctl->len < 0) || (pri != 0))
7810             return (EINVAL);
7811         break;
7812     case MSG_BAND:
7813         break;
7815     default:
7816         return (EINVAL);
7817     }
7819     TRACE_1(TR_FAC_STREAMS_FR, TR_STRPUTMSG_IN,
7820            "strputmsg in:stp %p", stp);
7822     /* get these values from those cached in the stream head */
7823     rmin = stp->sd_qn_minpsz;
7824     rmax = stp->sd_qn_maxpsz;
7826     /*
7827      * Make sure ctl and data sizes together fall within the
7828      * limits of the max and min receive packet sizes and do
7829      * not exceed system limit.
7830      */
7831     ASSERT((rmax >= 0) || (rmax == INFP SZ));
7832     if (rmax == 0) {
7833         return (ERANGE);
7834     }
7835     /*
7836      * Use the MAXIMUM of sd_maxblk and q_maxpsz.
7837      * Needed to prevent partial failures in the strmakedata loop.
7838      */
7839     if (stp->sd_maxblk != INFP SZ && rmax != INFP SZ && rmax < stp->sd_maxblk)
7840         rmax = stp->sd_maxblk;
7842     if ((msgsize = mdata->len) < 0) {
7843         msgsize = 0;
7844         rmin = 0;        /* no range check for NULL data part */
7845     }
7846     if ((msgsize < rmin) ||
7847         (msgsize > rmax) && (rmax != INFP SZ) ||
7848         (mctl->len > strctlsz)) {
7849         return (ERANGE);

```

```

7850     }
7852     /*
7853      * Setup uio and iov for data part
7854      */
7855     iovs.iov_base = mdata->buf;
7856     iovs.iov_len = msgsize;
7857     uios.uio_iov = &iovs;
7858     uios.uio_iovcnt = 1;
7859     uios.uio_loffset = 0;
7860     uios.uio_segflg = UIO_USERSPACE;
7861     uios.uio_fmode = fmode;
7862     uios.uio_extflg = UIO_COPY_DEFAULT;
7863     uios.uio_resid = msgsize;
7864     uios.uio_offset = 0;
7866     /* Ignore flow control in strput for HIPRI */
7867     if (flag & MSG_HIPRI)
7868         flag |= MSG_IGNFLOW;
7870     for (;;) {
7871         int done = 0;
7873         /*
7874          * strput will always free the ctl mblk - even when strput
7875          * fails.
7876          */
7877         if ((error = strmakectl(mctl, flag, fmode, &mp)) != 0) {
7878             TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7879                "strputmsg out:stp %p out %d error %d",
7880                stp, 1, error);
7881             return (error);
7882         }
7883         /*
7884          * Verify that the whole message can be transferred by
7885          * strput.
7886          */
7887         ASSERT(stp->sd_maxblk == INFP SZ ||
7888            stp->sd_maxblk >= mdata->len);
7890         msgsize = mdata->len;
7891         error = strput(stp, mp, uiop, &msgsize, 0, pri, flag);
7892         mdata->len = msgsize;
7894         if (error == 0)
7895             break;
7897         if (error != EWOULDBLOCK)
7898             goto out;
7900         mutex_enter(&stp->sd_lock);
7901         /*
7902          * Check for a missed wakeup.
7903          * Needed since strput did not hold sd_lock across
7904          * the canputnext.
7905          */
7906         if (bcanputnext(wqp, pri)) {
7907             /* Try again */
7908             mutex_exit(&stp->sd_lock);
7909             continue;
7910         }
7911         TRACE_2(TR_FAC_STREAMS_FR, TR_STRPUTMSG_WAIT,
7912            "strputmsg wait:stp %p waits pri %d", stp, pri);
7913         if (((error = strwaitq(stp, WRITEWAIT, (ssize_t)0, fmode, -1,
7914            &done)) != 0) || done) {
7915             mutex_exit(&stp->sd_lock);

```

```

7916         TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7917                "strputmsg out:q %p out %d error %d",
7918                stp, 0, error);
7919         return (error);
7920     }
7921     TRACE_1(TR_FAC_STREAMS_FR, TR_STRPUTMSG_WAKE,
7922            "strputmsg wake:stp %p wakes", stp);
7923     if ((error = i_straccess(stp, JCWRITE)) != 0) {
7924         mutex_exit(&stp->sd_lock);
7925         return (error);
7926     }
7927     mutex_exit(&stp->sd_lock);
7928 }
7929 out:
7930 /*
7931  * For historic reasons, applications expect EAGAIN
7932  * when data mblk could not be allocated. so change
7933  * ENOMEM back to EAGAIN
7934  */
7935     if (error == ENOMEM)
7936         error = EAGAIN;
7937     TRACE_3(TR_FAC_STREAMS_FR, TR_STRPUTMSG_OUT,
7938            "strputmsg out:stp %p out %d error %d", stp, 2, error);
7939     return (error);
7940 }
7942 /*
7943  * Put a message downstream.
7944  * Can send only an M_PROTO/M_PCPROTO by passing in a NULL uiop.
7945  * The fmode flag (NDELAY, NONBLOCK) is the or of the flags in the uio
7946  * and the fmode parameter.
7947  *
7948  * This routine handles the consolidation private flags:
7949  * MSG_IGNERROR Ignore any stream head error except STPLEX.
7950  * MSG_HOLD SIG Hold signals while waiting for data.
7951  * MSG_IGNFLOW Don't check streams flow control.
7952  *
7953  * NOTE: strputmsg and kstrputmsg have much of the logic in common.
7954  */
7955     int
7956     kstrputmsg(
7957         struct vnode *vp,
7958         mblk_t *mctl,
7959         struct uio *uiop,
7960         ssize_t msgsize,
7961         unsigned char pri,
7962         int flag,
7963         int fmode)
7964     {
7965         struct stdata *stp;
7966         queue_t *wqp;
7967         ssize_t rmin, rmax;
7968         int error;
7970         ASSERT(vp->v_stream);
7971         stp = vp->v_stream;
7972         wqp = stp->sd_wrq;
7973         if (AU_AUDITING())
7974             audit_strputmsg(vp, NULL, NULL, pri, flag, fmode);
7975         if (mctl == NULL)
7976             return (EINVAL);
7978         mutex_enter(&stp->sd_lock);
7980         if ((error = i_straccess(stp, JCWRITE)) != 0) {
7981             mutex_exit(&stp->sd_lock);

```

```

7982         freemsg(mctl);
7983         return (error);
7984     }
7986     if ((stp->sd_flag & STPLEX) || !(flag & MSG_IGNERROR)) {
7987         if (stp->sd_flag & (STWRERR|STRHUP|STPLEX)) {
7988             error = strwriteable(stp, B_FALSE, B_TRUE);
7989             if (error != 0) {
7990                 mutex_exit(&stp->sd_lock);
7991                 freemsg(mctl);
7992                 return (error);
7993             }
7994         }
7995     }
7997     mutex_exit(&stp->sd_lock);
7999     /*
8000     * Check for legal flag value.
8001     */
8002     switch (flag & (MSG_HIPRI|MSG_BAND|MSG_ANY)) {
8003     case MSG_HIPRI:
8004         if (pri != 0) {
8005             freemsg(mctl);
8006             return (EINVAL);
8007         }
8008         break;
8009     case MSG_BAND:
8010         break;
8011     default:
8012         freemsg(mctl);
8013         return (EINVAL);
8014     }
8016     TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_IN,
8017            "kstrputmsg in:stp %p", stp);
8019     /* get these values from those cached in the stream head */
8020     rmin = stp->sd_qn_minpsz;
8021     rmax = stp->sd_qn_maxpsz;
8023     /*
8024     * Make sure ctl and data sizes together fall within the
8025     * limits of the max and min receive packet sizes and do
8026     * not exceed system limit.
8027     */
8028     ASSERT((rmax >= 0) || (rmax == INFPSZ));
8029     if (rmax == 0) {
8030         freemsg(mctl);
8031         return (ERANGE);
8032     }
8033     /*
8034     * Use the MAXIMUM of sd_maxblk and q_maxpsz.
8035     * Needed to prevent partial failures in the strmakedata loop.
8036     */
8037     if (stp->sd_maxblk != INFPSZ && rmax != INFPSZ && rmax < stp->sd_maxblk)
8038         rmax = stp->sd_maxblk;
8040     if (uiop == NULL) {
8041         msgsize = -1;
8042         rmin = -1; /* no range check for NULL data part */
8043     } else {
8044         /* Use uio flags as well as the fmode parameter flags */
8045         fmode |= uiop->uio_fmode;
8047         if ((msgsize < rmin) ||

```

```

8048         ((msgsize > rmax) && (rmax != INFPSZ))) {
8049             freemsg(mctl);
8050             return (ERANGE);
8051         }
8052     }

8054     /* Ignore flow control in strput for HIPRI */
8055     if (flag & MSG_HIPRI)
8056         flag |= MSG_IGNFLOW;

8058     for (;;) {
8059         int done = 0;
8060         int waitflag;
8061         mblk_t *mp;

8063         /*
8064          * strput will always free the ctl mblk - even when strput
8065          * fails. If MSG_IGNFLOW is set then any error returned
8066          * will cause us to break the loop, so we don't need a copy
8067          * of the message. If MSG_IGNFLOW is not set, then we can
8068          * get hit by flow control and be forced to try again. In
8069          * this case we need to have a copy of the message. We
8070          * do this using copymsg since the message may get modified
8071          * by something below us.
8072          *
8073          * We've observed that many TPI providers do not check db_ref
8074          * on the control messages but blindly reuse them for the
8075          * T_OK_ACK/T_ERROR_ACK. Thus using copymsg is more
8076          * friendly to such providers than using dupmsg. Also, note
8077          * that sockfs uses MSG_IGNFLOW for all TPI control messages.
8078          * Only data messages are subject to flow control, hence
8079          * subject to this copymsg.
8080          */
8081         if (flag & MSG_IGNFLOW) {
8082             mp = mctl;
8083             mctl = NULL;
8084         } else {
8085             do {
8086                 /*
8087                  * If a message has a free pointer, the message
8088                  * must be dupmsg to maintain this pointer.
8089                  * Code using this facility must be sure
8090                  * that modules below will not change the
8091                  * contents of the dblk without checking db_ref
8092                  * first. If db_ref is > 1, then the module
8093                  * needs to do a copymsg first. Otherwise,
8094                  * the contents of the dblk may become
8095                  * inconsistent because the freesmg/freeb below
8096                  * may end up calling atomic_add_32_nv.
8097                  * The atomic_add_32_nv in freeb (accessing
8098                  * all of db_ref, db_type, db_flags, and
8099                  * db_struioflag) does not prevent other threads
8100                  * from concurrently trying to modify e.g.
8101                  * db_type.
8102                  */
8103                 if (mctl->b_datap->db_frtnp != NULL)
8104                     mp = dupmsg(mctl);
8105                 else
8106                     mp = copymsg(mctl);

8108                 if (mp != NULL)
8109                     break;

8111                 error = strwaitbuf(msgdsize(mctl), BPRI_MED);
8112                 if (error) {
8113                     freemsg(mctl);

```

```

8114             return (error);
8115         } while (mp == NULL);
8116     }
8117 }
8118 /*
8119  * Verify that all of msgsize can be transferred by
8120  * strput.
8121  */
8122 ASSERT(stp->sd_maxblk == INFPSZ || stp->sd_maxblk >= msgsize);
8123 error = strput(stp, mp, uiop, &msgsize, 0, pri, flag);
8124 if (error == 0)
8125     break;

8127     if (error != EWOULDBLOCK)
8128         goto out;

8130     /*
8131      * IF MSG_IGNFLOW is set we should have broken out of loop
8132      * above.
8133      */
8134     ASSERT(!(flag & MSG_IGNFLOW));
8135     mutex_enter(&stp->sd_lock);
8136     /*
8137      * Check for a missed wakeup.
8138      * Needed since strput did not hold sd_lock across
8139      * the canputnext.
8140      */
8141     if (bcanputnext(wqp, pri)) {
8142         /* Try again */
8143         mutex_exit(&stp->sd_lock);
8144         continue;
8145     }
8146     TRACE_2(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_WAIT,
8147            "kstrputmsg wait:stp %p waits pri %d", stp, pri);

8149     waitflag = WRITEWAIT;
8150     if (flag & (MSG_HOLD SIG|MSG_IGNERROR)) {
8151         if (flag & MSG_HOLD SIG)
8152             waitflag |= STR_NOSIG;
8153         if (flag & MSG_IGNERROR)
8154             waitflag |= STR_NOERROR;
8155     }
8156     if (((error = strwaitq(stp, waitflag,
8157        (ssize_t)0, fmode, -1, &done)) != 0) || done) {
8158         mutex_exit(&stp->sd_lock);
8159         TRACE_3(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_OUT,
8160            "kstrputmsg out:stp %p out %d error %d",
8161            stp, 0, error);
8162         freemsg(mctl);
8163         return (error);
8164     }
8165     TRACE_1(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_WAKE,
8166            "kstrputmsg wake:stp %p wakes", stp);
8167     if (((error = i_straccess(stp, JWRITE)) != 0) {
8168         mutex_exit(&stp->sd_lock);
8169         freemsg(mctl);
8170         return (error);
8171     }
8172     mutex_exit(&stp->sd_lock);
8173 }
8174 out:
8175     freemsg(mctl);
8176     /*
8177      * For historic reasons, applications expect EAGAIN
8178      * when data mblk could not be allocated. so change
8179      * ENOMEM back to EAGAIN

```

```

8180  */
8181  if (error == ENOMEM)
8182      error = EAGAIN;
8183  TRACE_3(TR_FAC_STREAMS_FR, TR_KSTRPUTMSG_OUT,
8184          "kstrputmsg out:stp %p out %d error %d", stp, 2, error);
8185  return (error);
8186 }

8188 /*
8189  * Determines whether the necessary conditions are set on a stream
8190  * for it to be readable, writeable, or have exceptions.
8191  *
8192  * strpoll handles the consolidation private events:
8193  * POLLNOERR      Do not return POLLERR even if there are stream
8194  *                head errors.
8195  *                Used by sockfs.
8196  * POLLRDDATA    Do not return POLLIN unless at least one message on
8197  *                the queue contains one or more M_DATA mblks. Thus
8198  *                when this flag is set a queue with only
8199  *                M_PROTO/M_PCPROTO mblks does not return POLLIN.
8200  *                Used by sockfs to ignore T_EXDATA_IND messages.
8201  *
8202  * Note: POLLRDDATA assumes that synch streams only return messages with
8203  * an M_DATA attached (i.e. not messages consisting of only
8204  * an M_PROTO/M_PCPROTO part).
8205  */
8206 int
8207 strpoll(
8208     struct stdata *stp,
8209     short events_arg,
8210     int anyyet,
8211     short *reventsp,
8212     struct pollhead **phpp)
8213 {
8214     int events = (ushort_t)events_arg;
8215     int revents = 0;
8216     mblk_t *mp;
8217     qband_t *qbp;
8218     long sd_flags = stp->sd_flag;
8219     int headlocked = 0;

8221     /*
8222      * For performance, a single 'if' tests for most possible edge
8223      * conditions in one shot
8224      */
8225     if (sd_flags & (STPLEX | STRDERR | STWRERR)) {
8226         if (sd_flags & STPLEX) {
8227             *reventsp = POLLNVAL;
8228             return (EINVAL);
8229         }
8230         if (((events & (POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI)) &&
8231             (sd_flags & STRDERR)) ||
8232             ((events & (POLLOUT | POLLWRNORM | POLLWRBAND)) &&
8233             (sd_flags & STWRERR))) {
8234             if (!(events & POLLNOERR)) {
8235                 *reventsp = POLLERR;
8236                 return (0);
8237             }
8238         }
8239     }
8240     if (sd_flags & STRHUP) {
8241         revents |= POLLHUP;
8242     } else if (events & (POLLWRNORM | POLLWRBAND)) {
8243         queue_t *tq;
8244         queue_t *qp = stp->sd_wrq;

```

```

8246     claimstr(qp);
8247     /* Find next module forward that has a service procedure */
8248     tq = qp->q_next->q_nfsrv;
8249     ASSERT(tq != NULL);

8251     if (polllock(&stp->sd_pollist, QLOCK(tq)) != 0) {
8252         releasestr(qp);
8253         *reventsp = POLLNVAL;
8254         return (0);
8255     }
8256     if (events & POLLWRNORM) {
8257         queue_t *sqp;

8259         if (tq->q_flag & QFULL)
8260             /* ensure backq svc procedure runs */
8261             tq->q_flag |= QWANTW;
8262         else if ((sqp = stp->sd_struiowrq) != NULL) {
8263             /* Check sync stream barrier write q */
8264             mutex_exit(QLOCK(tq));
8265             if (polllock(&stp->sd_pollist,
8266                         QLOCK(sqp)) != 0) {
8267                 releasestr(qp);
8268                 *reventsp = POLLNVAL;
8269                 return (0);
8270             }
8271             if (sqp->q_flag & QFULL)
8272                 /* ensure pollwakep() is done */
8273                 sqp->q_flag |= QWANTWSYNC;
8274             else
8275                 revents |= POLLOUT;
8276             /* More write events to process ??? */
8277             if (!(events & POLLWRBAND)) {
8278                 mutex_exit(QLOCK(sqp));
8279                 releasestr(qp);
8280                 goto chkrd;
8281             }
8282             mutex_exit(QLOCK(sqp));
8283             if (polllock(&stp->sd_pollist,
8284                         QLOCK(tq)) != 0) {
8285                 releasestr(qp);
8286                 *reventsp = POLLNVAL;
8287                 return (0);
8288             }
8289             } else
8290                 revents |= POLLOUT;
8291         }
8292     if (events & POLLWRBAND) {
8293         qbp = tq->q_bandp;
8294         if (qbp) {
8295             while (qbp) {
8296                 if (qbp->qb_flag & QB_FULL)
8297                     qbp->qb_flag |= QB_WANTW;
8298                 else
8299                     revents |= POLLWRBAND;
8300                 qbp = qbp->qb_next;
8301             }
8302             } else {
8303                 revents |= POLLWRBAND;
8304             }
8305         }
8306     mutex_exit(QLOCK(tq));
8307     releasestr(qp);
8308 }
8309 chkrd:
8310 if (sd_flags & STRPRI) {
8311     revents |= (events & POLLPRI);

```

```

8312     } else if (events & (POLLRDNORM | POLLRDBAND | POLLIN)) {
8313         queue_t *qp = _RD(stp->sd_wrq);
8314         int normevents = (events & (POLLIN | POLLRDNORM));

8316         /*
8317          * Note: Need to do polllock() here since ps_lock may be
8318          * held. See bug 4191544.
8319          */
8320         if (polllock(&stp->sd_pollist, &stp->sd_lock) != 0) {
8321             *reventsp = POLLNVAL;
8322             return (0);
8323         }
8324         headlocked = 1;
8325         mp = qp->q_first;
8326         while (mp) {
8327             /*
8328              * For POLLRDDATA we scan b_cont and b_next until we
8329              * find an M_DATA.
8330              */
8331             if ((events & POLLRDDATA) &&
8332                 mp->b_datap->db_type != M_DATA) {
8333                 mblk_t *nmp = mp->b_cont;

8335                 while (nmp != NULL &&
8336                     nmp->b_datap->db_type != M_DATA)
8337                     nmp = nmp->b_cont;
8338                 if (nmp == NULL) {
8339                     mp = mp->b_next;
8340                     continue;
8341                 }
8342             }
8343             if (mp->b_band == 0)
8344                 retevents |= normevents;
8345             else
8346                 retevents |= (events & (POLLIN | POLLRDBAND));
8347             break;
8348         }
8349         if (!(retevents & normevents) &&
8350             (stp->sd_wakeq & RSLEEP)) {
8351             /*
8352              * Sync stream barrier read queue has data.
8353              */
8354             retevents |= normevents;
8355         }
8356         /* Treat eof as normal data */
8357         if (sd_flags & STREOF)
8358             retevents |= normevents;
8359     }

8361     *reventsp = (short)retevents;
8362     if (retevents && !(events & POLLET)) {
8363         if (headlocked)
8364             mutex_exit(&stp->sd_lock);
8365         return (0);
8366     }

8368     /*
8369     * If poll() has not found any events yet, set up event cell
8370     * to wake up the poll if a requested event occurs on this
8371     * stream. Check for collisions with outstanding poll requests.
8372     */
8373     if (!anyyet) {
8374         *phpp = &stp->sd_pollist;
8375         if (headlocked == 0) {
8376             if (polllock(&stp->sd_pollist, &stp->sd_lock) != 0) {
8377                 *reventsp = POLLNVAL;

```

```

8378         return (0);
8379     }
8380     headlocked = 1;
8381     }
8382     stp->sd_rput_opt |= SR_POLLIN;
8383     }
8384     if (headlocked)
8385         mutex_exit(&stp->sd_lock);
8386     return (0);
8387 }

8389 /*
8390 * The purpose of putback() is to assure sleeping polls/reads
8391 * are awakened when there are no new messages arriving at the,
8392 * stream head, and a message is placed back on the read queue.
8393 *
8394 * sd_lock must be held when messages are placed back on stream
8395 * head. (getq()) holds sd_lock when it removes messages from
8396 * the queue)
8397 */

8399 static void
8400 putback(struct stdata *stp, queue_t *q, mblk_t *bp, int band)
8401 {
8402     mblk_t *qfirst;
8403     ASSERT(MUTEX_HELD(&stp->sd_lock));

8405     /*
8406     * As a result of lock-step ordering around q_lock and sd_lock,
8407     * it's possible for function calls like putnext() and
8408     * canputnext() to get an inaccurate picture of how much
8409     * data is really being processed at the stream head.
8410     * We only consolidate with existing messages on the queue
8411     * if the length of the message we want to put back is smaller
8412     * than the queue hiwater mark.
8413     */
8414     if ((stp->sd_rput_opt & SR_CONSOL_DATA) &&
8415         (DB_TYPE(bp) == M_DATA) && ((qfirst = q->q_first) != NULL) &&
8416         (DB_TYPE(qfirst) == M_DATA) &&
8417         ((qfirst->b_flag & (MSGMARK|MSGDELIM)) == 0) &&
8418         ((bp->b_flag & (MSGMARK|MSGDELIM|MSGMARKNEXT)) == 0) &&
8419         (mp_cont_len(bp, NULL) < q->q_hiwat)) {
8420         /*
8421          * We use the same logic as defined in strrrput()
8422          * but in reverse as we are putting back onto the
8423          * queue and want to retain byte ordering.
8424          * Consolidate M_DATA messages with M_DATA ONLY.
8425          * strrrput() allows the consolidation of M_DATA onto
8426          * M_PROTO | M_PCPROTO but not the other way round.
8427          *
8428          * The consolidation does not take place if the message
8429          * we are returning to the queue is marked with either
8430          * of the marks or the delim flag or if q_first
8431          * is marked with MSGMARK. The MSGMARK check is needed to
8432          * handle the odd semantics of MSGMARK where essentially
8433          * the whole message is to be treated as marked.
8434          * Carry any MSGMARKNEXT and MSGNOTMARKNEXT from q_first
8435          * to the front of the b_cont chain.
8436          */
8437         rmvq_noenab(q, qfirst);

8439         /*
8440         * The first message in the b_cont list
8441         * tracks MSGMARKNEXT and MSGNOTMARKNEXT.
8442         * We need to handle the case where we
8443         * are appending:

```

```

8444      *
8445      * 1) a MSGMARKNEXT to a MSGNOTMARKNEXT.
8446      * 2) a MSGMARKNEXT to a plain message.
8447      * 3) a MSGNOTMARKNEXT to a plain message
8448      * 4) a MSGNOTMARKNEXT to a MSGNOTMARKNEXT
8449      *    message.
8450      *
8451      * Thus we never append a MSGMARKNEXT or
8452      * MSGNOTMARKNEXT to a MSGMARKNEXT message.
8453      */
8454      if (qfirst->b_flag & MSGMARKNEXT) {
8455          bp->b_flag |= MSGMARKNEXT;
8456          bp->b_flag &= ~MSGNOTMARKNEXT;
8457          qfirst->b_flag &= ~MSGMARKNEXT;
8458      } else if (qfirst->b_flag & MSGNOTMARKNEXT) {
8459          bp->b_flag |= MSGNOTMARKNEXT;
8460          qfirst->b_flag &= ~MSGNOTMARKNEXT;
8461      }
8463      linkb(bp, qfirst);
8464      }
8465      (void) putbq(q, bp);
8467      /*
8468      * A message may have come in when the sd_lock was dropped in the
8469      * calling routine. If this is the case and STR*ATMARK info was
8470      * received, need to move that from the stream head to the q_last
8471      * so that SIOCATMARK can return the proper value.
8472      */
8473      if (stp->sd_flag & (STRATMARK | STRNOTATMARK)) {
8474          unsigned short *flagp = &q->q_last->b_flag;
8475          uint_t b_flag = (uint_t)*flagp;
8477          if (stp->sd_flag & STRATMARK) {
8478              b_flag &= ~MSGNOTMARKNEXT;
8479              b_flag |= MSGMARKNEXT;
8480              stp->sd_flag &= ~STRATMARK;
8481          } else {
8482              b_flag &= ~MSGMARKNEXT;
8483              b_flag |= MSGNOTMARKNEXT;
8484              stp->sd_flag &= ~STRNOTATMARK;
8485          }
8486          *flagp = (unsigned short) b_flag;
8487      }
8489      #ifdef DEBUG
8490      /*
8491      * Make sure that the flags are not messed up.
8492      */
8493      {
8494          mblk_t *mp;
8495          mp = q->q_last;
8496          while (mp != NULL) {
8497              ASSERT((mp->b_flag & (MSGMARKNEXT|MSGNOTMARKNEXT)) !=
8498                  (MSGMARKNEXT|MSGNOTMARKNEXT));
8499              mp = mp->b_cont;
8500          }
8501      }
8502      #endif
8503      if (q->q_first == bp) {
8504          short pollevents;
8506          if (stp->sd_flag & RSLEEP) {
8507              stp->sd_flag &= ~RSLEEP;
8508              cv_broadcast(&q->q_wait);
8509          }

```

```

8510          if (stp->sd_flag & STRPRI) {
8511              pollevents = POLLPRI;
8512          } else {
8513              if (band == 0) {
8514                  if (!(stp->sd_rput_opt & SR_POLLIN))
8515                      return;
8516                  stp->sd_rput_opt &= ~SR_POLLIN;
8517                  pollevents = POLLIN | POLLRDNORM;
8518              } else {
8519                  pollevents = POLLIN | POLLRDBAND;
8520              }
8521          }
8522          mutex_exit(&stp->sd_lock);
8523          pollwakeup(&stp->sd_pollist, pollevents);
8524          mutex_enter(&stp->sd_lock);
8525      }
8526      }
8528      /*
8529      * Return the held vnode attached to the stream head of a
8530      * given queue
8531      * It is the responsibility of the calling routine to ensure
8532      * that the queue does not go away (e.g. pop).
8533      */
8534      vnode_t *
8535      strq2vp(queue_t *qp)
8536      {
8537          vnode_t *vp;
8538          vp = STREAM(qp)->sd_vnode;
8539          ASSERT(vp != NULL);
8540          VN_HOLD(vp);
8541          return (vp);
8542      }
8544      /*
8545      * return the stream head write queue for the given vp
8546      * It is the responsibility of the calling routine to ensure
8547      * that the stream or vnode do not close.
8548      */
8549      queue_t *
8550      strvp2wq(vnode_t *vp)
8551      {
8552          ASSERT(vp->v_stream != NULL);
8553          return (vp->v_stream->sd_wrq);
8554      }
8556      /*
8557      * pollwakeup stream head
8558      * It is the responsibility of the calling routine to ensure
8559      * that the stream or vnode do not close.
8560      */
8561      void
8562      strpollwakeup(vnode_t *vp, short event)
8563      {
8564          ASSERT(vp->v_stream);
8565          pollwakeup(&vp->v_stream->sd_pollist, event);
8566      }
8568      /*
8569      * Mate the stream heads of two vnodes together. If the two vnodes are the
8570      * same, we just make the write-side point at the read-side -- otherwise,
8571      * we do a full mate. Only works on vnodes associated with streams that are
8572      * still being built and thus have only a stream head.
8573      */
8574      void
8575      strmate(vnode_t *vp1, vnode_t *vp2)

```

```

8576 {
8577     queue_t *wrq1 = strvp2wq(vp1);
8578     queue_t *wrq2 = strvp2wq(vp2);

8580     /*
8581     * Verify that there are no modules on the stream yet. We also
8582     * rely on the stream head always having a service procedure to
8583     * avoid tweaking q_nfsrv.
8584     */
8585     ASSERT(wrq1->q_next == NULL && wrq2->q_next == NULL);
8586     ASSERT(wrq1->q_qinfo->q_i_srvp != NULL);
8587     ASSERT(wrq2->q_qinfo->q_i_srvp != NULL);

8589     /*
8590     * If the queues are the same, just twist; otherwise do a full mate.
8591     */
8592     if (wrq1 == wrq2) {
8593         wrq1->q_next = _RD(wrq1);
8594     } else {
8595         wrq1->q_next = _RD(wrq2);
8596         wrq2->q_next = _RD(wrq1);
8597         STREAM(wrq1)->sd_mate = STREAM(wrq2);
8598         STREAM(wrq1)->sd_flag |= STRMATE;
8599         STREAM(wrq2)->sd_mate = STREAM(wrq1);
8600         STREAM(wrq2)->sd_flag |= STRMATE;
8601     }
8602 }

8604 /*
8605 * XXX will go away when console is correctly fixed.
8606 * Clean up the console PIDS, from previous I_SETSIG.
8607 * called only for cnopen which never calls strclean().
8608 */
8609 void
8610 str_cn_clean(struct vnode *vp)
8611 {
8612     strsig_t *ssp, *pssp, *tssp;
8613     struct stdata *stp;
8614     struct pid *pidp;
8615     int update = 0;

8617     ASSERT(vp->v_stream);
8618     stp = vp->v_stream;
8619     pssp = NULL;
8620     mutex_enter(&stp->sd_lock);
8621     ssp = stp->sd_siglist;
8622     while (ssp) {
8623         mutex_enter(&pidlock);
8624         pidp = ssp->ss_pidp;
8625         /*
8626          * Get rid of PID if the proc is gone.
8627          */
8628         if (pidp->pid_prinactive) {
8629             tssp = ssp->ss_next;
8630             if (pssp)
8631                 pssp->ss_next = tssp;
8632             else
8633                 stp->sd_siglist = tssp;
8634             ASSERT(pidp->pid_ref <= 1);
8635             PID_RELE(ssp->ss_pidp);
8636             mutex_exit(&pidlock);
8637             kmem_free(ssp, sizeof (strsig_t));
8638             update = 1;
8639             ssp = tssp;
8640             continue;
8641         } else

```

```

8642         mutex_exit(&pidlock);
8643         pssp = ssp;
8644         ssp = ssp->ss_next;
8645     }
8646     if (update) {
8647         stp->sd_sigflags = 0;
8648         for (ssp = stp->sd_siglist; ssp; ssp = ssp->ss_next)
8649             stp->sd_sigflags |= ssp->ss_events;
8650     }
8651     mutex_exit(&stp->sd_lock);
8652 }

8654 /*
8655 * Return B_TRUE if there is data in the message, B_FALSE otherwise.
8656 */
8657 static boolean_t
8658 msghasdata(mblk_t *bp)
8659 {
8660     for (; bp; bp = bp->b_cont)
8661         if (bp->b_datap->db_type == M_DATA) {
8662             ASSERT(bp->b_wptr >= bp->b_rptr);
8663             if (bp->b_wptr > bp->b_rptr)
8664                 return (B_TRUE);
8665         }
8666     return (B_FALSE);
8667 }

8669 /*
8670 * Check whether a stream is an XTI stream or not.
8671 */
8672 static boolean_t
8673 is_xti_str(const struct stdata *stp)
8674 {
8675     struct devnames *dnp;
8676     vnode_t *vn;
8677     major_t major;
8678     if ((vn = stp->sd_vnode) != NULL && vn->v_type == VCHR &&
8679         vn->v_rdev != 0) {
8680         major = getmajor(vn->v_rdev);
8681         dnp = (major != DDI_MAJOR_T_NONE && major >= 0 &&
8682             major < devcnt) ? &devnames[major] : NULL;
8683         if (dnp != NULL && dnp->dn_name != NULL &&
8684             (strcmp(dnp->dn_name, "ip") == 0 ||
8685              strcmp(dnp->dn_name, "tcp") == 0 ||
8686              strcmp(dnp->dn_name, "udp") == 0 ||
8687              strcmp(dnp->dn_name, "icmp") == 0 ||
8688              strcmp(dnp->dn_name, "t1") == 0 ||
8689              strcmp(dnp->dn_name, "ip6") == 0 ||
8690              strcmp(dnp->dn_name, "tcp6") == 0 ||
8691              strcmp(dnp->dn_name, "udp6") == 0 ||
8692              strcmp(dnp->dn_name, "icmp6") == 0)) {
8693             return (B_TRUE);
8694         }
8695     }
8696     #endif /* ! codereview */
8697     return (B_FALSE);
8698 }

```

```
*****
```

```
232177 Wed Jul 13 01:32:41 2016
```

```
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_tree_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     avl_create(&stp->sd_pid_tree, pid_node_comparator, 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_tree_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     avl_destroy(&stp->sd_pid_tree);
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_rvcnt == 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_rmcount == 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(queueinfo_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         str_sendsig(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 }

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->q_i_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->q_i_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->q_i_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  * Wait for any pending service processing to complete.
1387  * The removal of queues from the runlist is not atomic with the
1388  * clearing of the QENABLED flag and setting the INSERVICE flag.
1389  * consequently it is possible for remove_runlist in strclose
1390  * to not find the queue on the runlist but for it to be QENABLED
1391  * and not yet INSERVICE -> hence wait_svc needs to check QENABLED
1392  * as well as INSERVICE.
1393  */
1394 void
1395 wait_svc(queue_t *qp)
1396 {
1397     queue_t *wqp = _WR(qp);
1398
1399     ASSERT(qp->q_flag & QREADR);
1400
1401     /*
1402      * Try to remove queues from qhead/qtail list.
1403      */
1404     if (qhead != NULL) {
1405         remove_runlist(qp);
1406         remove_runlist(wqp);
1407     }
1408     /*
1409      * Wait till the syncqs associated with the queue disappear from the
1410      * background processing list.
1411      * This only needs to be done for non-PERMOD perimeters since
1412      * for PERMOD perimeters the syncq may be shared and will only be freed
1413      * when the last module/driver is unloaded.
1414      * If for PERMOD perimeters queue was on the syncq list, removeq()
1415      * should call propagate_syncq() or drain_syncq() for it. Both of these
1416      * functions remove the queue from its syncq list, so sqthread will not
1417      * try to access the queue.
1418      */
1419     if (!(qp->q_flag & QPERMOD)) {
1420         syncq_t *rsq = qp->q_syncq;
1421         syncq_t *wsq = wqp->q_syncq;
1422
1423         /*
1424          * Disable rsq and wsq and wait for any background processing of
1425          * 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     * strdioctl 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);

1562         kmem_cache_free(linkinfo_cache, linkp);
1563     }

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);

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              * If ep->me_nodep is a FIFO (me_nodep == NULL),
1621              * ignore the edge and move on. ep->me_nodep gets
1622              * set to NULL in mux_addedge() if it is a FIFO.
1623              */
1624             ep = np->mn_startp;
1625             np->mn_startp = ep->me_nextp;
1626             if (ep->me_nodep == NULL)
1627                 continue;
1628             ep->me_nodep->mn_originp = np;

```

```

1625         np = ep->me_nodep;
1626     }
1627 }
1628 }

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;

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 }

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 }

1689 /*
1690  * Wait for the syncq count to drop to zero.

```

```

1691 * sq could be either outer or inner.
1692 */
1694 static void
1695 wait_syncq(syncq_t *sq)
1696 {
1697     uint16_t count;
1699     mutex_enter(SQLOCK(sq));
1700     count = sq->sq_count;
1701     SQ_PUTLOCKS_ENTER(sq);
1702     SUM_SQ_PUTCOUNTS(sq, count);
1703     while (count != 0) {
1704         sq->sq_flags |= SQ_WANTWAKEUP;
1705         SQ_PUTLOCKS_EXIT(sq);
1706         cv_wait(&sq->sq_wait, SQLOCK(sq));
1707         count = sq->sq_count;
1708         SQ_PUTLOCKS_ENTER(sq);
1709         SUM_SQ_PUTCOUNTS(sq, count);
1710     }
1711     SQ_PUTLOCKS_EXIT(sq);
1712     mutex_exit(SQLOCK(sq));
1713 }
1715 /*
1716  * Wait while there are any messages for the queue in its syncq.
1717  */
1718 static void
1719 wait_q_syncq(queue_t *q)
1720 {
1721     if ((q->q_sqflags & Q_SQQUEUED) || (q->q_syncqmsgs > 0)) {
1722         syncq_t *sq = q->q_syncq;
1724         mutex_enter(SQLOCK(sq));
1725         while ((q->q_sqflags & Q_SQQUEUED) || (q->q_syncqmsgs > 0)) {
1726             sq->sq_flags |= SQ_WANTWAKEUP;
1727             cv_wait(&sq->sq_wait, SQLOCK(sq));
1728         }
1729         mutex_exit(SQLOCK(sq));
1730     }
1731 }
1734 int
1735 mlink_file(vnode_t *vp, int cmd, struct file *fpdown, cred_t *crp, int *rvalp,
1736           int lmlink)
1737 {
1738     struct stdata *stp;
1739     struct striocctl strioc;
1740     struct linkinfo *linkp;
1741     struct stdata *stpdwn;
1742     struct streamtab *str;
1743     queue_t *passq;
1744     syncq_t *passyncq;
1745     queue_t *rq;
1746     cdevsw_impl_t *dp;
1747     uint32_t qflag;
1748     uint32_t sqtype;
1749     perdm_t *dmp;
1750     int error = 0;
1751     netstack_t *ns;
1752     str_stack_t *ss;
1754     stp = vp->v_stream;
1755     TRACE_1(TR_FAC_STREAMS_FR,
1756           TR_I_LINK, "I_LINK/I_PLINK:stp %p", stp);

```

```

1757 /*
1758  * Test for invalid upper stream
1759  */
1760 if (stp->sd_flag & STRHUP) {
1761     return (ENXIO);
1762 }
1763 if (vp->v_type == VFIFO) {
1764     return (EINVAL);
1765 }
1766 if (stp->sd_strtab == NULL) {
1767     return (EINVAL);
1768 }
1769 if (!stp->sd_strtab->st_muxwinit) {
1770     return (EINVAL);
1771 }
1772 if (fpdown == NULL) {
1773     return (EBADF);
1774 }
1775 ns = netstack_find_by_cred(crp);
1776 ASSERT(ns != NULL);
1777 ss = ns->netstack_str;
1778 ASSERT(ss != NULL);
1780 if (getmajor(stp->sd_vnode->v_rdev) >= ss->ss_devcnt) {
1781     netstack_rele(ss->ss_netstack);
1782     return (EINVAL);
1783 }
1784 mutex_enter(&muxifier);
1785 if (stp->sd_flag & STPLEX) {
1786     mutex_exit(&muxifier);
1787     netstack_rele(ss->ss_netstack);
1788     return (ENXIO);
1789 }
1791 /*
1792  * Test for invalid lower stream.
1793  * The check for the v_type != VFIFO and having a major
1794  * number not >= devcnt is done to avoid problems with
1795  * adding mux_node entry past the end of mux_nodes[].
1796  * For FIFO's we don't add an entry so this isn't a
1797  * problem.
1798  */
1799 if (((stpdwn = fpdown->f_vnode->v_stream) == NULL) ||
1800     (stpdwn == stp) || (stpdwn->sd_flag &
1801     (STPLEX|STRHUP|STRDERR|STWRERR|IOCWAIT|STRPLUMB)) ||
1802     ((stpdwn->sd_vnode->v_type != VFIFO) &&
1803     (getmajor(stpdwn->sd_vnode->v_rdev) >= ss->ss_devcnt)) ||
1804     linkcycle(stp, stpdwn, ss)) {
1805     mutex_exit(&muxifier);
1806     netstack_rele(ss->ss_netstack);
1807     return (EINVAL);
1808 }
1809 TRACE_1(TR_FAC_STREAMS_FR,
1810       TR_STPDOWN, "stpdwn:%p", stpdwn);
1811 rq = getendq(stp->sd_wrq);
1812 if (cmd == I_PLINK)
1813     rq = NULL;
1815 linkp = alloclink(rq, stpdwn->sd_wrq, fpdown);
1817 strioc.ic_cmd = cmd;
1818 strioc.ic_timeout = INFTIM;
1819 strioc.ic_len = sizeof (struct linkblk);
1820 strioc.ic_dp = (char *)&linkp->li_lblk;
1822 /*

```

```

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(&stpdn->sd_lock);
1855 stpdn->sd_flag |= STRPLUMB;
1856 mutex_exit(&stpdn->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(stpdn);

1863 mutex_enter(&stpdn->sd_lock);
1864 stpdn->sd_flag |= STPLEX;
1865 mutex_exit(&stpdn->sd_lock);

1867 rq = _RD(stpdn->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 = strdioctl(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 = stpdn;
1920     setq(rq, &strdata, &stwddata, NULL, QMTSAFE, SQ_CI|SQ_CO,
1921         B_TRUE);

1923     mutex_enter(&stpdn->sd_lock);
1924     stpdn->sd_flag &= ~STPLEX;
1925     mutex_exit(&stpdn->sd_lock);

1927     link_rempassthru(passq);

1929     mutex_enter(&stpdn->sd_lock);
1930     stpdn->sd_flag &= ~STRPLUMB;
1931     /* Wakeup anyone waiting for STRPLUMB to clear. */
1932     cv_broadcast(&stpdn->sd_monitor);
1933     mutex_exit(&stpdn->sd_lock);

1935     mutex_exit(&muxifier);
1936     netstack_rele(ss->ss_netstack);
1937     return (error);
1938 }
1939 mutex_enter(&fpdn->f_tlock);
1940 fpdn->f_count++;
1941 mutex_exit(&fpdn->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, fpdn, lhlink, LINKNORMAL);
1951 } else {
1952     ldi_mlink_fp(stp, fpdn, lhlink, LINKPERSIST);
1953 }

```

```

1955     link_rempassthru(passq);
1957     mux_adddedge(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 & LINKYPEMASK) == LINKNORMAL)
2047         strioc.ic_cmd = I_UNLINK;
2048     else
2049         strioc.ic_cmd = I_PUNLINK;
2050     strioc.ic_timeout = 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)));

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(&stpdn->sd_lock);
2232     stpdn->sd_flag &= ~STPLEX;
2233     mutex_exit(&stpdn->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(&stpdn->sd_lock);
2252     stpdn->sd_flag &= ~STRPLUMB;
2253     cv_broadcast(&stpdn->sd_monitor);
2254     mutex_exit(&stpdn->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 specs 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_MTCBSSHARED) {
2423         /*
2424          * The code in putnext assumes that it has the
2425          * highest concurrency by not checking sq_count.
2426          * Thus _D_MTCBSSHARED 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->q_i_mininfo->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_TYEMASK) != 0);
2503     IMPLY((qflag & (QPERMOD | QMTOUTPERIM)), dmp != NULL);

2505     wq = _WR(rq);
2506     rq->q_qinfo = rinit;
2507     rq->q_hiwat = rinit->q_i_mininfo->mi_hiwat;
2508     rq->q_lowat = rinit->q_i_mininfo->mi_lowat;
2509     rq->q_minpsz = rinit->q_i_mininfo->mi_minpsz;
2510     rq->q_maxpsz = rinit->q_i_mininfo->mi_maxpsz;
2511     wq->q_qinfo = winit;
2512     wq->q_hiwat = winit->q_i_mininfo->mi_hiwat;
2513     wq->q_lowat = winit->q_i_mininfo->mi_lowat;
2514     wq->q_minpsz = winit->q_i_mininfo->mi_minpsz;
2515     wq->q_maxpsz = winit->q_i_mininfo->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_TYEMASK) | QWANTR | qflag;
2599         mutex_exit(QLOCK(rq));
2600         mutex_enter(QLOCK(wq));
2601         wq->q_flag = (wq->q_flag & ~QMT_TYEMASK) | QWANTR | qflag;
2602         mutex_exit(QLOCK(wq));
2603     } else {
2604         rq->q_flag = (rq->q_flag & ~QMT_TYEMASK) | QWANTR | qflag;
2605         wq->q_flag = (wq->q_flag & ~QMT_TYEMASK) | 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;
2619
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;
2630
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));
2641
2642 /*
2643  * Initialize struiot() 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 }
2650
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;
2658
2659     ASSERT(str != NULL);
2660     ASSERT(qflag & (QPERMOD | QMTOUTPERIM));
2661
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);
2671
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;
2686
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     }
2697
2698     *pp = dmp;
2699     rw_exit(&perdm_rwlock);
2700     return (dmp);
2701 }
2702
2703 void
2704 rele_dm(perdm_t *dmp)
2705 {
2706     perdm_t **pp;
2707     perdm_t *p;
2708
2709     rw_enter(&perdm_rwlock, RW_WRITER);
2710     ASSERT(dmp->dm_ref > 0);
2711
2712     if (--dmp->dm_ref > 0) {
2713         rw_exit(&perdm_rwlock);
2714         return;
2715     }
2716
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);
2723
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 }
2732
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;
2754
2755     ASSERT(uiop != NULL);
2756
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 }
2781
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();
2797
2798     /* We do not support interrupt threads using the stream head to send */
2799     ASSERT(cr != NULL);
2800
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;
2809
2810         if (flag & RS_HIPRI)
2811             msgtype = M_PCPROTO;
2812         else

```

```

2813             msgtype = M_PROTO;
2814
2815             ctlcount = mctl->len;
2816             base = mctl->buf;
2817
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);
2823
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             }
2835
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     }
2846
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;
2873
2874         *mpp = NULL;
2875         if (count < 0)
2876             return (0);
2877
2878         /* We do not support interrupt threads using the stream head to send */

```

```

2879     cr = CRED();
2880     ASSERT(cr != NULL);

2882     maxblk = stp->sd_maxblk;
2883     if (maxblk == INFPSZ)
2884         maxblk = count;

2886     /*
2887      * Create data part of message, if any.
2888      */
2889     do {
2890         ssize_t size;
2891         dblk_t *dp;

2893         ASSERT(uiop);

2895         size = MIN(count, maxblk);

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_cpuid = curproc->p_pid;
2914         ASSERT(wroff <= dp->db_lim - bp->b_wptr);
2915         bp->b_wptr = bp->b_rptr = bp->b_rptr + wroff;

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_struiooun.data = 0ll;
2927             bp->b_wptr += size;
2928         } else {
2929             if (stp->sd_copyflag & STRCOPYCACHED)
2930                 uiop->uio_extflg |= UIO_COPY_CACHED;

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;

2943             if (stp->sd_wputdatafunc != NULL) {
2944                 mblk_t *newbp;

```

```

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         }
2957         count -= size;

2959         if (mp == NULL)
2960             mp = bp;
2961         else
2962             linkb(mp, bp);
2963     } while (count > 0);

2965     *mpp = mp;
2966     return (0);
2967 }

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;

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 }

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  * WRIWAIT       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 timeout,
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, timeout, 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 jaccess 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_spllock); /* 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_spllock);
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_spllock);
3198             mutex_exit(&p->p_lock);
3199             return (EIO);
3200         }

3202         mutex_exit(&sp->s_lock);
3203         mutex_exit(&p->p_spllock);

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 }

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_locid = 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 = strrrput_proto;
3307     stp->sd_rmiscfunc = strrrput_misc;
3308     stp->sd_rdrerrfunc = stp->sd_wrererrfunc = NULL;
3309     stp->sd_rputdatafunc = stp->sd_wputdatafunc = NULL;
3310     stp->sd_ciputctrl = NULL;
3311     stp->sd_nciputctrl = 0;
3312     stp->sd_qhead = NULL;
3313     stp->sd_qtail = NULL;
3314     stp->sd_servid = NULL;
3315     stp->sd_nqueues = 0;
3316     stp->sd_svcflags = 0;
3317     stp->sd_copyflag = 0;

3319     return (stp);
3320 }

3322 /*
3323  * Free a stream head.
3324  */
3325 void
3326 shfree(stdata_t *stp)
3327 {
3328     pid_node_t *pn;

3330 #endif /* ! codereview */
3331     ASSERT(MUTEX_NOT_HELD(&stp->sd_lock));

3333     stp->sd_wrq = NULL;

3335     mutex_enter(&stp->sd_qlock);
3336     while (stp->sd_svcflags & STRS_SCHEDULED) {
3337         STRSTAT(strwaits);
3338         cv_wait(&stp->sd_qcv, &stp->sd_qlock);
3339     }
3340     mutex_exit(&stp->sd_qlock);

```

```

3342     if (stp->sd_ciputctrl != NULL) {
3343         ASSERT(stp->sd_nciputctrl == n_ciputctrl - 1);
3344         SUMCHECK_CIPUTCTRL_COUNTS(stp->sd_ciputctrl,
3345             stp->sd_nciputctrl, 0);
3346         ASSERT(ciputctrl_cache != NULL);
3347         kmem_cache_free(ciputctrl_cache, stp->sd_ciputctrl);
3348         stp->sd_ciputctrl = NULL;
3349         stp->sd_nciputctrl = 0;
3350     }
3351     ASSERT(stp->sd_qhead == NULL);
3352     ASSERT(stp->sd_qtail == NULL);
3353     ASSERT(stp->sd_nqueues == 0);

3355     mutex_enter(&stp->sd_pid_tree_lock);
3356     while ((pn = avl_first(&stp->sd_pid_tree)) != NULL) {
3357         avl_remove(&stp->sd_pid_tree, pn);
3358         kmem_free(pn, sizeof (*pn));
3359     }
3360     mutex_exit(&stp->sd_pid_tree_lock);

3362 #endif /* ! codereview */
3363     kmem_cache_free(stream_head_cache, stp);
3364 }

3366 void
3367 sh_insert_pid(struct stdata *stp, pid_t pid)
3368 {
3369     pid_node_t *pn, lookup_pn;
3370     avl_index_t idx_pn;

3372     lookup_pn.pn_pid = pid;
3373     mutex_enter(&stp->sd_pid_tree_lock);
3374     pn = avl_find(&stp->sd_pid_tree, &lookup_pn, &idx_pn);

3376     if (pn != NULL) {
3377         pn->pn_count++;
3378     } else {
3379         pn = kmem_zalloc(sizeof (*pn), KM_SLEEP);
3380         pn->pn_pid = pid;
3381         pn->pn_count = 1;
3382         avl_insert(&stp->sd_pid_tree, pn, idx_pn);
3383     }
3384     mutex_exit(&stp->sd_pid_tree_lock);
3385 }

3387 void
3388 sh_remove_pid(struct stdata *stp, pid_t pid)
3389 {
3390     pid_node_t *pn, lookup_pn;

3392     lookup_pn.pn_pid = pid;
3393     mutex_enter(&stp->sd_pid_tree_lock);
3394     pn = avl_find(&stp->sd_pid_tree, &lookup_pn, NULL);

3396     if (pn != NULL) {
3397         if (pn->pn_count > 1) {
3398             pn->pn_count--;
3399         } else {
3400             avl_remove(&stp->sd_pid_tree, pn);
3401             kmem_free(pn, sizeof (*pn));
3402         }
3403     }
3404     mutex_exit(&stp->sd_pid_tree_lock);
3405 }

```

```

3407 mblk_t *
3408 sh_get_pid_mblk(struct stdata *stp)
3409 {
3410     mblk_t *mblk;
3411     ulong_t sz, n;
3412     pid_t *pids;
3413     pid_node_t *pn;
3414     conn_pid_info_t *cpi;
3415
3416     mutex_enter(&stp->sd_pid_tree_lock);
3417
3418     n = avl_numnodes(&stp->sd_pid_tree);
3419     sz = sizeof (conn_pid_info_t);
3420     sz += (n > 1) ? ((n - 1) * sizeof (pid_t)) : 0;
3421     if ((mblk = allocb(sz, BPRI_HI)) == NULL) {
3422         mutex_exit(&stp->sd_pid_tree_lock);
3423         return (NULL);
3424     }
3425     mblk->b_wptr += sz;
3426     cpi = (conn_pid_info_t *)mblk->b_datap->db_base;
3427     cpi->cpi_contents = CONN_PID_INFO_XTI;
3428     cpi->cpi_pids_cnt = n;
3429     cpi->cpi_tot_size = sz;
3430     cpi->cpi_pids[0] = 0;
3431
3432     if (cpi->cpi_pids_cnt > 0) {
3433         pids = cpi->cpi_pids;
3434         for (pn = avl_first(&stp->sd_pid_tree); pn != NULL;
3435             pids++, pn = AVL_NEXT(&stp->sd_pid_tree, pn))
3436             *pids = pn->pn_pid;
3437     }
3438     mutex_exit(&stp->sd_pid_tree_lock);
3439     return (mblk);
3440 }
3441
3442 #endif /* ! codereview */
3443 /*
3444  * Allocate a pair of queues and a syncq for the pair
3445  */
3446 queue_t *
3447 allocq(void)
3448 {
3449     queinfo_t *qip;
3450     queue_t *qp, *wqp;
3451     syncq_t *sq;
3452
3453     qip = kmem_cache_alloc(queue_cache, KM_SLEEP);
3454
3455     qp = &qip->qu_rqueue;
3456     wqp = &qip->qu_wqueue;
3457     sq = &qip->qu_syncq;
3458
3459     qp->q_last = NULL;
3460     qp->q_next = NULL;
3461     qp->q_ptr = NULL;
3462     qp->q_flag = QUSE | QREADR;
3463     qp->q_bandp = NULL;
3464     qp->q_stream = NULL;
3465     qp->q_syncq = sq;
3466     qp->q_nband = 0;
3467     qp->q_nfsrv = NULL;
3468     qp->q_draining = 0;
3469     qp->q_syncqmsgs = 0;
3470     qp->q_spri = 0;
3471     qp->q_qtstamp = 0;
3472     qp->q_sqtstamp = 0;

```

```

3473     qp->q_fp = NULL;
3474
3475     wqp->q_last = NULL;
3476     wqp->q_next = NULL;
3477     wqp->q_ptr = NULL;
3478     wqp->q_flag = QUSE;
3479     wqp->q_bandp = NULL;
3480     wqp->q_stream = NULL;
3481     wqp->q_syncq = sq;
3482     wqp->q_nband = 0;
3483     wqp->q_nfsrv = NULL;
3484     wqp->q_draining = 0;
3485     wqp->q_syncqmsgs = 0;
3486     wqp->q_qtstamp = 0;
3487     wqp->q_sqtstamp = 0;
3488     wqp->q_spri = 0;
3489
3490     sq->sq_count = 0;
3491     sq->sq_rmcount = 0;
3492     sq->sq_flags = 0;
3493     sq->sq_type = 0;
3494     sq->sq_callbflags = 0;
3495     sq->sq_cancelid = 0;
3496     sq->sq_ciputctrl = NULL;
3497     sq->sq_nciputctrl = 0;
3498     sq->sq_needexcl = 0;
3499     sq->sq_svcflags = 0;
3500
3501     return (qp);
3502 }
3503
3504 /*
3505  * Free a pair of queues and the "attached" syncq.
3506  * Discard any messages left on the syncq(s), remove the syncq(s) from the
3507  * outer perimeter, and free the syncq(s) if they are not the "attached" syncq.
3508  */
3509 void
3510 freeq(queue_t *qp)
3511 {
3512     qband_t *qbp, *nqbp;
3513     syncq_t *sq, *outer;
3514     queue_t *wqp = _WR(qp);
3515
3516     ASSERT(qp->q_flag & QREADR);
3517
3518     /*
3519      * If a previously dispatched taskq job is scheduled to run
3520      * sync_service() or a service routine is scheduled for the
3521      * queues about to be freed, wait here until all service is
3522      * done on the queue and all associated queues and syncqs.
3523      */
3524     wait_svc(qp);
3525
3526     (void) flush_syncq(qp->q_syncq, qp);
3527     (void) flush_syncq(wqp->q_syncq, wqp);
3528     ASSERT(qp->q_syncqmsgs == 0 && wqp->q_syncqmsgs == 0);
3529
3530     /*
3531      * Flush the queues before q_next is set to NULL This is needed
3532      * in order to backenable any downstream queue before we go away.
3533      * Note: we are already removed from the stream so that the
3534      * backenabling will not cause any messages to be delivered to our
3535      * put procedures.
3536      */
3537     flushq(qp, FLUSHALL);
3538     flushq(wqp, FLUSHALL);

```

```

3540 /* Tidy up - removeq only does a half-remove from stream */
3541 qp->q_next = wqp->q_next = NULL;
3542 ASSERT(!(qp->q_flag & QENAB));
3543 ASSERT(!(wqp->q_flag & QENAB));

3545 outer = qp->q_syncq->sq_outer;
3546 if (outer != NULL) {
3547     outer_remove(outer, qp->q_syncq);
3548     if (wqp->q_syncq != qp->q_syncq)
3549         outer_remove(outer, wqp->q_syncq);
3550 }
3551 /*
3552  * Free any syncqs that are outside what allocq returned.
3553  */
3554 if (qp->q_syncq != SQ(qp) && !(qp->q_flag & QPERMOD))
3555     free_syncq(qp->q_syncq);
3556 if (qp->q_syncq != wqp->q_syncq && wqp->q_syncq != SQ(qp))
3557     free_syncq(wqp->q_syncq);

3559 ASSERT((qp->q_sqflags & (Q_SQQUEUED | Q_SQDRAINING)) == 0);
3560 ASSERT((wqp->q_sqflags & (Q_SQQUEUED | Q_SQDRAINING)) == 0);
3561 ASSERT(MUTEX_NOT_HELD(QLOCK(qp)));
3562 ASSERT(MUTEX_NOT_HELD(QLOCK(wqp)));
3563 sq = SQ(qp);
3564 ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
3565 ASSERT(sq->sq_head == NULL && sq->sq_tail == NULL);
3566 ASSERT(sq->sq_outer == NULL);
3567 ASSERT(sq->sq_onext == NULL && sq->sq_oprev == NULL);
3568 ASSERT(sq->sq_callbpend == NULL);
3569 ASSERT(sq->sq_needexcl == 0);

3571 if (sq->sq_ciputctrl != NULL) {
3572     ASSERT(sq->sq_nciputctrl == n_ciputctrl - 1);
3573     SUMCHECK_CIPUTCTRL_COUNTS(sq->sq_ciputctrl,
3574         sq->sq_nciputctrl, 0);
3575     ASSERT(ciputctrl_cache != NULL);
3576     kmem_cache_free(ciputctrl_cache, sq->sq_ciputctrl);
3577     sq->sq_ciputctrl = NULL;
3578     sq->sq_nciputctrl = 0;
3579 }

3581 ASSERT(qp->q_first == NULL && wqp->q_first == NULL);
3582 ASSERT(qp->q_count == 0 && wqp->q_count == 0);
3583 ASSERT(qp->q_mblkcnt == 0 && wqp->q_mblkcnt == 0);

3585 qp->q_flag &= ~QUSE;
3586 wqp->q_flag &= ~QUSE;

3588 /* NOTE: Uncomment the assert below once bugid 1159635 is fixed. */
3589 /* ASSERT((qp->q_flag & QWANTW) == 0 && (wqp->q_flag & QWANTW) == 0); */

3591 qbp = qp->q_bandp;
3592 while (qbp) {
3593     nqbp = qbp->q_bnext;
3594     freeband(qbp);
3595     qbp = nqbp;
3596 }
3597 qbp = wqp->q_bandp;
3598 while (qbp) {
3599     nqbp = qbp->q_bnext;
3600     freeband(qbp);
3601     qbp = nqbp;
3602 }
3603 kmem_cache_free(queue_cache, qp);
3604 }

```

```

3606 /*
3607  * Allocate a qband structure.
3608  */
3609 qband_t *
3610 allocband(void)
3611 {
3612     qband_t *qbp;

3614     qbp = kmem_cache_alloc(qband_cache, KM_NOSLEEP);
3615     if (qbp == NULL)
3616         return (NULL);

3618     qbp->q_bnext = NULL;
3619     qbp->q_bcount = 0;
3620     qbp->q_bmlkcnt = 0;
3621     qbp->q_bfirst = NULL;
3622     qbp->q_blast = NULL;
3623     qbp->q_bflag = 0;

3625     return (qbp);
3626 }

3628 /*
3629  * Free a qband structure.
3630  */
3631 void
3632 freeband(qband_t *qbp)
3633 {
3634     kmem_cache_free(qband_cache, qbp);
3635 }

3637 /*
3638  * Just like putnextctl(9F), except that allocb_wait() is used.
3639  * Consolidation Private, and of course only callable from the stream head or
3640  * routines that may block.
3641  */
3642 int
3643 putnextctl_wait(queue_t *q, int type)
3644 {
3645     mblk_t *bp;
3646     int error;

3649     if ((datamsg(type) && (type != M_DELAY)) ||
3650         (bp = allocb_wait(0, BPRI_HI, 0, &error)) == NULL)
3651         return (0);

3653     bp->b_datap->db_type = (unsigned char)type;
3654     putnext(q, bp);
3655     return (1);
3656 }

3658 /*
3659  * Run any possible bufcalls.
3660  */
3661 void
3662 runbufcalls(void)
3663 {
3664     strbufcall_t *bcp;

3666     mutex_enter(&bcall_monitor);
3667     mutex_enter(&strbcall_lock);

3669     if (strbcalls.bc_head) {
3670         size_t count;

```

```

3671     int nevent;
3672
3673     /*
3674     * count how many events are on the list
3675     * now so we can check to avoid looping
3676     * in low memory situations
3677     */
3678     nevent = 0;
3679     for (bcp = strbcalls.bc_head; bcp; bcp = bcp->bc_next)
3680         nevent++;
3681
3682     /*
3683     * get estimate of available memory from kmem_avail().
3684     * awake all bufcall functions waiting for
3685     * memory whose request could be satisfied
3686     * by 'count' memory and let 'em fight for it.
3687     */
3688     count = kmem_avail();
3689     while ((bcp = strbcalls.bc_head) != NULL && nevent) {
3690         STRSTAT(bufcalls);
3691         --nevent;
3692         if (bcp->bc_size <= count) {
3693             bcp->bc_executor = curthread;
3694             mutex_exit(&strbcall_lock);
3695             (*bcp->bc_func)(bcp->bc_arg);
3696             mutex_enter(&strbcall_lock);
3697             bcp->bc_executor = NULL;
3698             cv_broadcast(&bcall_cv);
3699             strbcalls.bc_head = bcp->bc_next;
3700             kmem_free(bcp, sizeof(strbufcall_t));
3701         } else {
3702             /*
3703             * too big, try again later - note
3704             * that nevent was decremented above
3705             * so we won't retry this one on this
3706             * iteration of the loop
3707             */
3708             if (bcp->bc_next != NULL) {
3709                 strbcalls.bc_head = bcp->bc_next;
3710                 bcp->bc_next = NULL;
3711                 strbcalls.bc_tail->bc_next = bcp;
3712                 strbcalls.bc_tail = bcp;
3713             }
3714         }
3715     }
3716     if (strbcalls.bc_head == NULL)
3717         strbcalls.bc_tail = NULL;
3718 }
3719
3720 mutex_exit(&strbcall_lock);
3721 mutex_exit(&bcall_monitor);
3722 }
3723
3725 /*
3726 * Actually run queue's service routine.
3727 */
3728 static void
3729 runservice(queue_t *q)
3730 {
3731     qband_t *qbp;
3732
3733     ASSERT(q->q_qinfo->q_i_srvp);
3734     again:
3735     entersq(q->q_syncq, SQ_SVC);
3736     TRACE_1(TR_FAC_STREAMS_FR, TR_QRUNSERVICE_START,

```

```

3737         "runservice starts:%p", q);
3738
3739     if (!(q->q_flag & QWCLOSE))
3740         (*q->q_qinfo->q_i_srvp)(q);
3741
3742     TRACE_1(TR_FAC_STREAMS_FR, TR_QRUNSERVICE_END,
3743         "runservice ends:%p", q);
3744
3745     leavesq(q->q_syncq, SQ_SVC);
3746
3747     mutex_enter(QLOCK(q));
3748     if (q->q_flag & QENAB) {
3749         q->q_flag &= ~QENAB;
3750         mutex_exit(QLOCK(q));
3751         goto again;
3752     }
3753     q->q_flag &= ~QINSERVICE;
3754     q->q_flag &= ~QBACK;
3755     for (qbp = q->q_bandp; qbp; qbp = qbp->qb_next)
3756         qbp->qb_flag &= ~QB_BACK;
3757     /*
3758     * Wakeup thread waiting for the service procedure
3759     * to be run (strclose and qdetach).
3760     */
3761     cv_broadcast(&q->q_wait);
3762
3763     mutex_exit(QLOCK(q));
3764 }
3765
3766 /*
3767 * Background processing of bufcalls.
3768 */
3769 void
3770 streams_bufcall_service(void)
3771 {
3772     callb_cpr_t    cprinfo;
3773
3774     CALLB_CPR_INIT(&cprinfo, &strbcall_lock, callb_generic_cpr,
3775         "streams_bufcall_service");
3776
3777     mutex_enter(&strbcall_lock);
3778
3779     for (;;) {
3780         if (strbcalls.bc_head != NULL && kmem_avail() > 0) {
3781             mutex_exit(&strbcall_lock);
3782             runbufcalls();
3783             mutex_enter(&strbcall_lock);
3784         }
3785         if (strbcalls.bc_head != NULL) {
3786             STRSTAT(bcwaits);
3787             /* Wait for memory to become available */
3788             CALLB_CPR_SAFE_BEGIN(&cprinfo);
3789             (void) cv_reltimedwait(&memavail_cv, &strbcall_lock,
3790                 SEC_TO_TICK(60), TR_CLOCK_TICK);
3791             CALLB_CPR_SAFE_END(&cprinfo, &strbcall_lock);
3792         }
3793     }
3794
3795     /* Wait for new work to arrive */
3796     if (strbcalls.bc_head == NULL) {
3797         CALLB_CPR_SAFE_BEGIN(&cprinfo);
3798         cv_wait(&strbcall_cv, &strbcall_lock);
3799         CALLB_CPR_SAFE_END(&cprinfo, &strbcall_lock);
3800     }
3801 }

```

```

3803 /*
3804  * Background processing of streams background tasks which failed
3805  * taskq_dispatch.
3806  */
3807 static void
3808 streams_qbkgnd_service(void)
3809 {
3810     callb_cpr_t cprinfo;
3811     queue_t *q;

3813     CALLB_CPR_INIT(&cprinfo, &service_queue, callb_generic_cpr,
3814                  "streams_bkgnd_service");

3816     mutex_enter(&service_queue);

3818     for (;;) {
3819         /*
3820          * Wait for work to arrive.
3821          */
3822         while ((freebs_list == NULL) && (qhead == NULL)) {
3823             CALLB_CPR_SAFE_BEGIN(&cprinfo);
3824             cv_wait(&services_to_run, &service_queue);
3825             CALLB_CPR_SAFE_END(&cprinfo, &service_queue);
3826         }
3827         /*
3828          * Handle all pending freebs requests to free memory.
3829          */
3830         while (freebs_list != NULL) {
3831             mblk_t *mp = freebs_list;
3832             freebs_list = mp->b_next;
3833             mutex_exit(&service_queue);
3834             mblk_free(mp);
3835             mutex_enter(&service_queue);
3836         }
3837         /*
3838          * Run pending queues.
3839          */
3840         while (qhead != NULL) {
3841             DQ(q, qhead, qtail, q_link);
3842             ASSERT(q != NULL);
3843             mutex_exit(&service_queue);
3844             queue_service(q);
3845             mutex_enter(&service_queue);
3846         }
3847         ASSERT(qhead == NULL && qtail == NULL);
3848     }
3849 }

3851 /*
3852  * Background processing of streams background tasks which failed
3853  * taskq_dispatch.
3854  */
3855 static void
3856 streams_sqbkgnd_service(void)
3857 {
3858     callb_cpr_t cprinfo;
3859     syncq_t *sq;

3861     CALLB_CPR_INIT(&cprinfo, &service_queue, callb_generic_cpr,
3862                  "streams_sqbkgnd_service");

3864     mutex_enter(&service_queue);

3866     for (;;) {
3867         /*
3868          * Wait for work to arrive.

```

```

3869     */
3870     while (sqhead == NULL) {
3871         CALLB_CPR_SAFE_BEGIN(&cprinfo);
3872         cv_wait(&syncqs_to_run, &service_queue);
3873         CALLB_CPR_SAFE_END(&cprinfo, &service_queue);
3874     }

3876     /*
3877      * Run pending syncqs.
3878      */
3879     while (sqhead != NULL) {
3880         DQ(sq, sqhead, sqtail, sq_next);
3881         ASSERT(sq != NULL);
3882         ASSERT(sq->sq_svcflags & SQ_BGTHREAD);
3883         mutex_exit(&service_queue);
3884         syncq_service(sq);
3885         mutex_enter(&service_queue);
3886     }
3887 }
3888 }

3890 /*
3891  * Disable the syncq and wait for background syncq processing to complete.
3892  * If the syncq is placed on the sqhead/sqtail queue, try to remove it from the
3893  * list.
3894  */
3895 void
3896 wait_sq_svc(syncq_t *sq)
3897 {
3898     mutex_enter(SQLOCK(sq));
3899     sq->sq_svcflags |= SQ_DISABLED;
3900     if (sq->sq_svcflags & SQ_BGTHREAD) {
3901         syncq_t *sq_chase;
3902         syncq_t *sq_curr;
3903         int removed;

3905         ASSERT(sq->sq_servcount == 1);
3906         mutex_enter(&service_queue);
3907         RMQ(sq, sqhead, sqtail, sq_next, sq_chase, sq_curr, removed);
3908         mutex_exit(&service_queue);
3909         if (removed) {
3910             sq->sq_svcflags &= ~SQ_BGTHREAD;
3911             sq->sq_servcount = 0;
3912             STRSTAT(sqremoved);
3913             goto done;
3914         }
3915     }
3916     while (sq->sq_servcount != 0) {
3917         sq->sq_flags |= SQ_WANTWAKEUP;
3918         cv_wait(&sq->sq_wait, SQLOCK(sq));
3919     }
3920 done:
3921     mutex_exit(SQLOCK(sq));
3922 }

3924 /*
3925  * Put a syncq on the list of syncq's to be serviced by the sqthread.
3926  * Add the argument to the end of the sqhead list and set the flag
3927  * indicating this syncq has been enabled. If it has already been
3928  * enabled, don't do anything.
3929  * This routine assumes that SQLOCK is held.
3930  * NOTE that the lock order is to have the SQLOCK first,
3931  * so if the service_syncq lock is held, we need to release it
3932  * before acquiring the SQLOCK (mostly relevant for the background
3933  * thread, and this seems to be common among the STREAMS global locks).
3934  * Note that the sq_svcflags are protected by the SQLOCK.

```

```

3935 */
3936 void
3937 sqenable(syncq_t *sq)
3938 {
3939     /*
3940      * This is probably not important except for where I believe it
3941      * is being called. At that point, it should be held (and it
3942      * is a pain to release it just for this routine, so don't do
3943      * it).
3944      */
3945     ASSERT(MUTEX_HELD(SQLOCK(sq)));
3947     IMPLY(sq->sq_servcount == 0, sq->sq_next == NULL);
3948     IMPLY(sq->sq_next != NULL, sq->sq_svcflags & SQ_BGTHREAD);
3950     /*
3951      * Do not put on list if background thread is scheduled or
3952      * syncq is disabled.
3953      */
3954     if (sq->sq_svcflags & (SQ_DISABLED | SQ_BGTHREAD))
3955         return;
3957     /*
3958      * Check whether we should enable sq at all.
3959      * Non PERMOD syncqs may be drained by at most one thread.
3960      * PERMOD syncqs may be drained by several threads but we limit the
3961      * total amount to the lesser of
3962      *   Number of queues on the squeue and
3963      *   Number of CPUs.
3964      */
3965     if (sq->sq_servcount != 0) {
3966         if (((sq->sq_type & SQ_PERMOD) == 0) ||
3967             (sq->sq_servcount >= MIN(sq->sq_nqueues, ncpus_online))) {
3968             STRSTAT(sqtoomany);
3969             return;
3970         }
3971     }
3973     sq->sq_tstamp = ddi_get_lbolt();
3974     STRSTAT(sqenables);
3976     /* Attempt a taskq dispatch */
3977     sq->sq_servid = (void *)taskq_dispatch(streams_taskq,
3978         (task_func_t *)syncq_service, sq, TQ_NOSLEEP | TQ_NOQUEUE);
3979     if (sq->sq_servid != NULL) {
3980         sq->sq_servcount++;
3981         return;
3982     }
3984     /*
3985      * This taskq dispatch failed, but a previous one may have succeeded.
3986      * Don't try to schedule on the background thread whilst there is
3987      * outstanding taskq processing.
3988      */
3989     if (sq->sq_servcount != 0)
3990         return;
3992     /*
3993      * System is low on resources and can't perform a non-sleeping
3994      * dispatch. Schedule the syncq for a background thread and mark the
3995      * syncq to avoid any further taskq dispatch attempts.
3996      */
3997     mutex_enter(&service_queue);
3998     STRSTAT(taskqfails);
3999     ENQUEUE(sq, sqhead, sqtail, sq_next);
4000     sq->sq_svcflags |= SQ_BGTHREAD;

```

```

4001     sq->sq_servcount = 1;
4002     cv_signal(&syncqs_to_run);
4003     mutex_exit(&service_queue);
4004 }
4006 /*
4007 * Note: fifo_close() depends on the mblk_t on the queue being freed
4008 * asynchronously. The asynchronous freeing of messages breaks the
4009 * recursive call chain of fifo_close() while there are I_SENDFD type of
4010 * messages referring to other file pointers on the queue. Then when
4011 * closing pipes it can avoid stack overflow in case of daisy-chained
4012 * pipes, and also avoid deadlock in case of fifonode_t pairs (which
4013 * share the same fifolock_t).
4014 *
4015 * No need to kpreempt_disable to access cpu_seqid. If we migrate and
4016 * the esb queue does not match the new CPU, that is OK.
4017 */
4018 void
4019 freebs_enqueue(mblk_t *mp, dblk_t *dbp)
4020 {
4021     int qindex = CPU->cpu_seqid >> esbq_log2_cpus_per_q;
4022     esb_queue_t *eqp;
4024     ASSERT(dbp->db_mblk == mp);
4025     ASSERT(qindex < esbq_nelem);
4027     eqp = system_esbq_array;
4028     if (eqp != NULL) {
4029         eqp += qindex;
4030     } else {
4031         mutex_enter(&esbq_lock);
4032         if (kmem_ready && system_esbq_array == NULL)
4033             system_esbq_array = (esb_queue_t *)kmem_zalloc(
4034                 esbq_nelem * sizeof(esb_queue_t), KM_NOSLEEP);
4035         mutex_exit(&esbq_lock);
4036         eqp = system_esbq_array;
4037         if (eqp != NULL)
4038             eqp += qindex;
4039         else
4040             eqp = &system_esbq;
4041     }
4043     /*
4044      * Check data sanity. The dblock should have non-empty free function.
4045      * It is better to panic here than later when the dblock is freed
4046      * asynchronously when the context is lost.
4047      */
4048     if (dbp->db_frtnp->free_func == NULL) {
4049         panic("freebs_enqueue: dblock %p has a NULL free callback",
4050             (void *)dbp);
4051     }
4053     mutex_enter(&eqp->eq_lock);
4054     /* queue the new mblk on the esballoc queue */
4055     if (eqp->eq_head == NULL) {
4056         eqp->eq_head = eqp->eq_tail = mp;
4057     } else {
4058         eqp->eq_tail->b_next = mp;
4059         eqp->eq_tail = mp;
4060     }
4061     eqp->eq_len++;
4063     /* If we're the first thread to reach the threshold, process */
4064     if (eqp->eq_len >= esbq_max_qlen &&
4065         !(eqp->eq_flags & ESBQ_PROCESSING))
4066         esballoc_process_queue(eqp);

```

```

4068     esballoc_set_timer(eqp, esbq_timeout);
4069     mutex_exit(&eqp->eq_lock);
4070 }

4072 static void
4073 esballoc_process_queue(esb_queue_t *eqp)
4074 {
4075     mblk_t *mp;

4077     ASSERT(MUTEX_HELD(&eqp->eq_lock));

4079     eqp->eq_flags |= ESBQ_PROCESSING;

4081     do {
4082         /*
4083          * Detach the message chain for processing.
4084          */
4085         mp = eqp->eq_head;
4086         eqp->eq_tail->b_next = NULL;
4087         eqp->eq_head = eqp->eq_tail = NULL;
4088         eqp->eq_len = 0;
4089         mutex_exit(&eqp->eq_lock);

4091         /*
4092          * Process the message chain.
4093          */
4094         esballoc_enqueue_mblk(mp);
4095         mutex_enter(&eqp->eq_lock);
4096     } while ((eqp->eq_len >= esbq_max_qlen) && (eqp->eq_len > 0));

4098     eqp->eq_flags &= ~ESBQ_PROCESSING;
4099 }

4101 /*
4102 * taskq callback routine to free esballocated mblk's
4103 */
4104 static void
4105 esballoc_mblk_free(mblk_t *mp)
4106 {
4107     mblk_t *nextmp;

4109     for (; mp != NULL; mp = nextmp) {
4110         nextmp = mp->b_next;
4111         mp->b_next = NULL;
4112         mblk_free(mp);
4113     }
4114 }

4116 static void
4117 esballoc_enqueue_mblk(mblk_t *mp)
4118 {
4120     if (taskq_dispatch(system_taskq, (task_func_t *)esballoc_mblk_free, mp,
4121         TQ_NOSLEEP) == NULL) {
4122         mblk_t *first_mp = mp;
4123         /*
4124          * System is low on resources and can't perform a non-sleeping
4125          * dispatch. Schedule for a background thread.
4126          */
4127         mutex_enter(&service_queue);
4128         STRSTAT(taskqfails);

4130         while (mp->b_next != NULL)
4131             mp = mp->b_next;

```

```

4133         mp->b_next = freebs_list;
4134         freebs_list = first_mp;
4135         cv_signal(&services_to_run);
4136         mutex_exit(&service_queue);
4137     }
4138 }

4140 static void
4141 esballoc_timer(void *arg)
4142 {
4143     esb_queue_t *eqp = arg;

4145     mutex_enter(&eqp->eq_lock);
4146     eqp->eq_flags &= ~ESBQ_TIMER;

4148     if (!(eqp->eq_flags & ESBQ_PROCESSING) &&
4149         eqp->eq_len > 0)
4150         esballoc_process_queue(eqp);

4152     esballoc_set_timer(eqp, esbq_timeout);
4153     mutex_exit(&eqp->eq_lock);
4154 }

4156 static void
4157 esballoc_set_timer(esb_queue_t *eqp, clock_t eq_timeout)
4158 {
4159     ASSERT(MUTEX_HELD(&eqp->eq_lock));

4161     if (eqp->eq_len > 0 && !(eqp->eq_flags & ESBQ_TIMER)) {
4162         (void) timeout(esballoc_timer, eqp, eq_timeout);
4163         eqp->eq_flags |= ESBQ_TIMER;
4164     }
4165 }

4167 /*
4168 * Setup esbq array length based upon NCPU scaled by CPUs per
4169 * queue. Use static system_esbq until kmem_ready and we can
4170 * create an array in freebs_enqueue().
4171 */
4172 void
4173 esballoc_queue_init(void)
4174 {
4175     esbq_log2_cpus_per_q = highbit(esbq_cpus_per_q - 1);
4176     esbq_cpus_per_q = 1 << esbq_log2_cpus_per_q;
4177     esbq_nelem = howmany(NCPU, esbq_cpus_per_q);
4178     system_esbq.eq_len = 0;
4179     system_esbq.eq_head = system_esbq.eq_tail = NULL;
4180     system_esbq.eq_flags = 0;
4181 }

4183 /*
4184 * Set the QBACK or QB BACK flag in the given queue for
4185 * the given priority band.
4186 */
4187 void
4188 setqback(queue_t *q, unsigned char pri)
4189 {
4190     int i;
4191     qband_t *qbp;
4192     qband_t **qbpp;

4194     ASSERT(MUTEX_HELD(QLOCK(q)));
4195     if (pri != 0) {
4196         if (pri > q->q_nband) {
4197             qbpp = &q->q_bandp;
4198             while (*qbpp)

```

```

4199         qbpp = &(*qbpp)->qb_next;
4200         while (pri > q->q_nband) {
4201             if ((*qbpp = allocband()) == NULL) {
4202                 cmn_err(CE_WARN,
4203                     "setqback: can't allocate qband\n");
4204                 return;
4205             }
4206             (*qbpp)->qb_hiwat = q->q_hiwat;
4207             (*qbpp)->qb_lowat = q->q_lowat;
4208             q->q_nband++;
4209             qbpp = &(*qbpp)->qb_next;
4210         }
4211     }
4212     qbp = q->q_bandp;
4213     i = pri;
4214     while (--i)
4215         qbp = qbp->qb_next;
4216     qbp->qb_flag |= QB_BACK;
4217 } else {
4218     q->q_flag |= QBACK;
4219 }
4220 }

4222 int
4223 strcpyin(void *from, void *to, size_t len, int copyflag)
4224 {
4225     if (copyflag & U_TO_K) {
4226         ASSERT((copyflag & K_TO_K) == 0);
4227         if (copyin(from, to, len))
4228             return (EFAULT);
4229     } else {
4230         ASSERT(copyflag & K_TO_K);
4231         bcopy(from, to, len);
4232     }
4233     return (0);
4234 }

4236 int
4237 strcpyout(void *from, void *to, size_t len, int copyflag)
4238 {
4239     if (copyflag & U_TO_K) {
4240         if (copyout(from, to, len))
4241             return (EFAULT);
4242     } else {
4243         ASSERT(copyflag & K_TO_K);
4244         bcopy(from, to, len);
4245     }
4246     return (0);
4247 }

4249 /*
4250  * strsignal_nolock() posts a signal to the process(es) at the stream head.
4251  * It assumes that the stream head lock is already held, whereas strsignal()
4252  * acquires the lock first. This routine was created because a few callers
4253  * release the stream head lock before calling only to re-acquire it after
4254  * it returns.
4255  */
4256 void
4257 strsignal_nolock(stdata_t *stp, int sig, uchar_t band)
4258 {
4259     ASSERT(MUTEX_HELD(&stp->sd_lock));
4260     switch (sig) {
4261     case SIGPOLL:
4262         if (stp->sd_sigflags & S_MSG)
4263             strsendsig(stp->sd_siglist, S_MSG, band, 0);
4264         break;

```

```

4265     default:
4266         if (stp->sd_pgidp)
4267             pgsignal(stp->sd_pgidp, sig);
4268         break;
4269     }
4270 }

4272 void
4273 strsignal(stdata_t *stp, int sig, int32_t band)
4274 {
4275     TRACE_3(TR_FAC_STREAMS_FR, TR_SENDSIG,
4276         "strsignal:%p, %X, %X", stp, sig, band);
4277
4278     mutex_enter(&stp->sd_lock);
4279     switch (sig) {
4280     case SIGPOLL:
4281         if (stp->sd_sigflags & S_MSG)
4282             strsendsig(stp->sd_siglist, S_MSG, (uchar_t)band, 0);
4283         break;
4284
4285     default:
4286         if (stp->sd_pgidp) {
4287             pgsignal(stp->sd_pgidp, sig);
4288         }
4289         break;
4290     }
4291     mutex_exit(&stp->sd_lock);
4292 }

4294 void
4295 strhup(stdata_t *stp)
4296 {
4297     ASSERT(mutex_owned(&stp->sd_lock));
4298     pollwakep(&stp->sd_pollist, POLLHUP);
4299     if (stp->sd_sigflags & S_HANGUP)
4300         strsendsig(stp->sd_siglist, S_HANGUP, 0, 0);
4301 }

4303 /*
4304  * Backenable the first queue upstream from 'q' with a service procedure.
4305  */
4306 void
4307 backenable(queue_t *q, uchar_t pri)
4308 {
4309     queue_t *nq;
4310
4311     /*
4312      * Our presence might not prevent other modules in our own
4313      * stream from popping/pushing since the caller of getq might not
4314      * have a claim on the queue (some drivers do a getq on somebody
4315      * else's queue - they know that the queue itself is not going away
4316      * but the framework has to guarantee q_next in that stream).
4317      */
4318     claimstr(q);
4319
4320     /* Find nearest back queue with service proc */
4321     for (nq = backq(q); nq && !nq->q_qinfo->q_i_srvp; nq = backq(nq)) {
4322         ASSERT(STRMATED(q->q_stream) || STREAM(q) == STREAM(nq));
4323     }
4324
4325     if (nq) {
4326         kthread_t *freezer;
4327         /*
4328          * backenable can be called either with no locks held
4329          * or with the stream frozen (the latter occurs when a module
4330          * calls rmvq with the stream frozen). If the stream is frozen

```

```

4331     * by the caller the caller will hold all qlocks in the stream.
4332     * Note that a frozen stream doesn't freeze a mated stream,
4333     * so we explicitly check for that.
4334     */
4335     freezer = STREAM(q)->sd_freezer;
4336     if (freezer != curthread || STREAM(q) != STREAM(nq)) {
4337         mutex_enter(QLOCK(nq));
4338     }
4339 #ifdef DEBUG
4340     else {
4341         ASSERT(frozenstr(q));
4342         ASSERT(MUTEX_HELD(QLOCK(q)));
4343         ASSERT(MUTEX_HELD(QLOCK(nq)));
4344     }
4345 #endif
4346     setqback(nq, pri);
4347     qenable_locked(nq);
4348     if (freezer != curthread || STREAM(q) != STREAM(nq))
4349         mutex_exit(QLOCK(nq));
4350 }
4351     releasestr(q);
4352 }

4354 /*
4355  * Return the appropriate errno when one of flags_to_check is set
4356  * in sd_flags. Uses the exported error routines if they are set.
4357  * Will return 0 if non error is set (or if the exported error routines
4358  * do not return an error).
4359  *
4360  * If there is both a read and write error to check, we prefer the read error.
4361  * Also, give preference to recorded errno's over the error functions.
4362  * The flags that are handled are:
4363  *     STPLEX      return EINVAL
4364  *     STRDERR     return sd_rerror (and clear if STRDERRNONPERSIST)
4365  *     STWRERR    return sd_werror (and clear if STWRERRNONPERSIST)
4366  *     STRHUP     return sd_werror
4367  *
4368  * If the caller indicates that the operation is a peek, a nonpersistent error
4369  * is not cleared.
4370  */
4371 int
4372 strgeterr(stdata_t *stp, int32_t flags_to_check, int ispeek)
4373 {
4374     int32_t sd_flag = stp->sd_flag & flags_to_check;
4375     int error = 0;

4377     ASSERT(MUTEX_HELD(&stp->sd_lock));
4378     ASSERT((flags_to_check & ~(STRDERR|STWRERR|STRHUP|STPLEX)) == 0);
4379     if (sd_flag & STPLEX)
4380         error = EINVAL;
4381     else if (sd_flag & STRDERR) {
4382         error = stp->sd_rerror;
4383         if ((stp->sd_flag & STRDERRNONPERSIST) && !ispeek) {
4384             /*
4385              * Read errors are non-persistent i.e. discarded once
4386              * returned to a non-peeking caller,
4387              */
4388             stp->sd_rerror = 0;
4389             stp->sd_flag &= ~STRDERR;
4390         }
4391     }
4392     if (error == 0 && stp->sd_rderrfunc != NULL) {
4393         int clearerr = 0;

4394         error = (*stp->sd_rderrfunc)(stp->sd_vnode, ispeek,
4395             &clearerr);
4396         if (clearerr) {

```

```

4397         stp->sd_flag &= ~STRDERR;
4398         stp->sd_rderrfunc = NULL;
4399     }
4400 }
4401 } else if (sd_flag & STWRERR) {
4402     error = stp->sd_werror;
4403     if ((stp->sd_flag & STWRERRNONPERSIST) && !ispeek) {
4404         /*
4405          * Write errors are non-persistent i.e. discarded once
4406          * returned to a non-peeking caller,
4407          */
4408         stp->sd_werror = 0;
4409         stp->sd_flag &= ~STWRERR;
4410     }
4411     if (error == 0 && stp->sd_wrerrfunc != NULL) {
4412         int clearerr = 0;

4414         error = (*stp->sd_wrerrfunc)(stp->sd_vnode, ispeek,
4415             &clearerr);
4416         if (clearerr) {
4417             stp->sd_flag &= ~STWRERR;
4418             stp->sd_wrerrfunc = NULL;
4419         }
4420     }
4421 } else if (sd_flag & STRHUP) {
4422     /* sd_werror set when STRHUP */
4423     error = stp->sd_werror;
4424 }
4425 return (error);
4426 }

4429 /*
4430  * Single-thread open/close/push/pop
4431  * for twisted streams also
4432  */
4433 int
4434 strstartplumb(stdata_t *stp, int flag, int cmd)
4435 {
4436     int waited = 1;
4437     int error = 0;

4439     if (STRMATED(stp)) {
4440         struct stdata *stmatep = stp->sd_mate;

4442         STRLOCKMATES(stp);
4443         while (waited) {
4444             waited = 0;
4445             while (stmatep->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4446                 if ((cmd == I_POP) &&
4447                     (flag & (FNDELAY|FNONBLOCK))) {
4448                     STRUNLOCKMATES(stp);
4449                     return (EAGAIN);
4450                 }
4451                 waited = 1;
4452                 mutex_exit(&stp->sd_lock);
4453                 if (!cv_wait_sig(&stmatep->sd_monitor,
4454                     &stmatep->sd_lock)) {
4455                     mutex_exit(&stmatep->sd_lock);
4456                     return (EINTR);
4457                 }
4458                 mutex_exit(&stmatep->sd_lock);
4459                 STRLOCKMATES(stp);
4460             }
4461             while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4462                 if ((cmd == I_POP) &&

```

```

4463         (flag & (FNDELAY|FNONBLOCK))) {
4464             STRUNLOCKMATES(stp);
4465             return (EAGAIN);
4466         }
4467         waited = 1;
4468         mutex_exit(&stmatep->sd_lock);
4469         if (!cv_wait_sig(&stp->sd_monitor,
4470             &stp->sd_lock)) {
4471             mutex_exit(&stp->sd_lock);
4472             return (EINTR);
4473         }
4474         mutex_exit(&stp->sd_lock);
4475         STRUNLOCKMATES(stp);
4476     }
4477     if (stp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
4478         error = strgeterr(stp,
4479             STRDERR|STWRERR|STRHUP|STPLEX, 0);
4480         if (error != 0) {
4481             STRUNLOCKMATES(stp);
4482             return (error);
4483         }
4484     }
4485     }
4486     stp->sd_flag |= STRPLUMB;
4487     STRUNLOCKMATES(stp);
4488 } else {
4489     mutex_enter(&stp->sd_lock);
4490     while (stp->sd_flag & (STWOPEN|STRCLOSE|STRPLUMB)) {
4491         if (((cmd == I_POP) || (cmd == _I_REMOVE)) &&
4492             (flag & (FNDELAY|FNONBLOCK))) {
4493             mutex_exit(&stp->sd_lock);
4494             return (EAGAIN);
4495         }
4496         if (!cv_wait_sig(&stp->sd_monitor, &stp->sd_lock)) {
4497             mutex_exit(&stp->sd_lock);
4498             return (EINTR);
4499         }
4500         if (stp->sd_flag & (STRDERR|STWRERR|STRHUP|STPLEX)) {
4501             error = strgeterr(stp,
4502                 STRDERR|STWRERR|STRHUP|STPLEX, 0);
4503             if (error != 0) {
4504                 mutex_exit(&stp->sd_lock);
4505                 return (error);
4506             }
4507         }
4508     }
4509     stp->sd_flag |= STRPLUMB;
4510     mutex_exit(&stp->sd_lock);
4511 }
4512 return (0);
4513 }

4515 /*
4516  * Complete the plumbing operation associated with stream 'stp'.
4517  */
4518 void
4519 strendplumb(stdata_t *stp)
4520 {
4521     ASSERT(MUTEX_HELD(&stp->sd_lock));
4522     ASSERT(stp->sd_flag & STRPLUMB);
4523     stp->sd_flag &= ~STRPLUMB;
4524     cv_broadcast(&stp->sd_monitor);
4525 }

4527 /*
4528  * This describes how the STREAMS framework handles synchronization

```

```

4529  * during open/push and close/pop.
4530  * The key interfaces for open and close are qprocson and qprocsoff,
4531  * respectively. While the close case in general is harder both open
4532  * have close have significant similarities.
4533  *
4534  * During close the STREAMS framework has to both ensure that there
4535  * are no stale references to the queue pair (and syncq) that
4536  * are being closed and also provide the guarantees that are documented
4537  * in qprocsoff(9F).
4538  * If there are stale references to the queue that is closing it can
4539  * result in kernel memory corruption or kernel panics.
4540  *
4541  * Note that is it up to the module/driver to ensure that it itself
4542  * does not have any stale references to the closing queues once its close
4543  * routine returns. This includes:
4544  * - Cancelling any timeout/bufcall/qtimeout/qbufcall callback routines
4545  *   associated with the queues. For timeout and bufcall callbacks the
4546  *   module/driver also has to ensure (or wait for) any callbacks that
4547  *   are in progress.
4548  * - If the module/driver is using esballoc it has to ensure that any
4549  *   esballoc free functions do not refer to a queue that has closed.
4550  *   (Note that in general the close routine can not wait for the esballoc'ed
4551  *   messages to be freed since that can cause a deadlock.)
4552  * - Cancelling any interrupts that refer to the closing queues and
4553  *   also ensuring that there are no interrupts in progress that will
4554  *   refer to the closing queues once the close routine returns.
4555  * - For multiplexors removing any driver global state that refers to
4556  *   the closing queue and also ensuring that there are no threads in
4557  *   the multiplexor that has picked up a queue pointer but not yet
4558  *   finished using it.
4559  *
4560  * In addition, a driver/module can only reference the q_next pointer
4561  * in its open, close, put, or service procedures or in a
4562  * qtimeout/qbufcall callback procedure executing "on" the correct
4563  * stream. Thus it can not reference the q_next pointer in an interrupt
4564  * routine or a timeout, bufcall or esballoc callback routine. Likewise
4565  * it can not reference q_next of a different queue e.g. in a mux that
4566  * passes messages from one queues put/service procedure to another queue.
4567  * In all the cases when the driver/module can not access the q_next
4568  * field it must use the *next* versions e.g. canputnext instead of
4569  * canput(q->q_next) and putnextctl instead of putctl(q->q_next, ...).
4570  *
4571  *
4572  * Assuming that the driver/module conforms to the above constraints
4573  * the STREAMS framework has to avoid stale references to q_next for all
4574  * the framework internal cases which include (but are not limited to):
4575  * - Threads in canput/canputnext/backenable and elsewhere that are
4576  *   walking q_next.
4577  * - Messages on a syncq that have a reference to the queue through b_queue.
4578  * - Messages on an outer perimeter (syncq) that have a reference to the
4579  *   queue through b_queue.
4580  * - Threads that use q_nfsrv (e.g. canput) to find a queue.
4581  *   Note that only canput and bcanput use q_nfsrv without any locking.
4582  *
4583  * The STREAMS framework providing the qprocsoff(9F) guarantees means that
4584  * after qprocsoff returns, the framework has to ensure that no threads can
4585  * enter the put or service routines for the closing read or write-side queue.
4586  * In addition to preventing "direct" entry into the put procedures
4587  * the framework also has to prevent messages being drained from
4588  * the syncq or the outer perimeter.
4589  * XXX Note that currently qdetach does relies on D_MTOCEXCL as the only
4590  * mechanism to prevent qwriter(PERIM_OUTER) from running after
4591  * qprocsoff has returned.
4592  * Note that if a module/driver uses put(9F) on one of its own queues
4593  * it is up to the module/driver to ensure that the put() doesn't
4594  * get called when the queue is closing.

```

```

4595 *
4596 *
4597 * The framework aspects of the above "contract" is implemented by
4598 * qprocsoff, removeq, and strlock:
4599 * - qprocsoff (disable_svc) sets QWCLOSE to prevent runservice from
4600 * entering the service procedures.
4601 * - strlock acquires the sd_lock and sd_reflock to prevent putnext,
4602 * canputnext, backenable etc from dereferencing the q_next that will
4603 * soon change.
4604 * - strlock waits for sd_refcnt to be zero to wait for e.g. any canputnext
4605 * or other q_next walker that uses claimstr/releasestr to finish.
4606 * - optionally for every syncq in the stream strlock acquires all the
4607 * sq_lock's and waits for all sq_counts to drop to a value that indicates
4608 * that no thread executes in the put or service procedures and that no
4609 * thread is draining into the module/driver. This ensures that no
4610 * open, close, put, service, or qtimeout/qbufcall callback procedure is
4611 * currently executing hence no such thread can end up with the old stale
4612 * q_next value and no canput/backenable can have the old stale
4613 * q_nfsrv/q_next.
4614 * - qdetach (wait_svc) makes sure that any scheduled or running threads
4615 * have either finished or observed the QWCLOSE flag and gone away.
4616 */

4619 /*
4620 * Get all the locks necessary to change q_next.
4621 *
4622 * Wait for sd_refcnt to reach 0 and, if sqlist is present, wait for the
4623 * sq_count of each syncq in the list to drop to sq_rmcount, indicating that
4624 * the only threads inside the syncq are threads currently calling removeq().
4625 * Since threads calling removeq() are in the process of removing their queues
4626 * from the stream, we do not need to worry about them accessing a stale q_next
4627 * pointer and thus we do not need to wait for them to exit (in fact, waiting
4628 * for them can cause deadlock).
4629 *
4630 * This routine is subject to starvation since it does not set any flag to
4631 * prevent threads from entering a module in the stream (i.e. sq_count can
4632 * increase on some syncq while it is waiting on some other syncq).
4633 *
4634 * Assumes that only one thread attempts to call strlock for a given
4635 * stream. If this is not the case the two threads would deadlock.
4636 * This assumption is guaranteed since strlock is only called by insertq
4637 * and removeq and streams plumbing changes are single-threaded for
4638 * a given stream using the STWOPEN, STRCLOSE, and STRPLUMB flags.
4639 *
4640 * For pipes, it is not difficult to atomically designate a pair of streams
4641 * to be mated. Once mated atomically by the framework the twisted pair remain
4642 * configured that way until dismantled atomically by the framework.
4643 * When plumbing takes place on a twisted stream it is necessary to ensure that
4644 * this operation is done exclusively on the twisted stream since two such
4645 * operations, each initiated on different ends of the pipe will deadlock
4646 * waiting for each other to complete.
4647 *
4648 * On entry, no locks should be held.
4649 * The locks acquired and held by strlock depends on a few factors.
4650 * - If sqlist is non-NULL all the syncq locks in the sqlist will be acquired
4651 * and held on exit and all sq_count are at an acceptable level.
4652 * - In all cases, sd_lock and sd_reflock are acquired and held on exit with
4653 * sd_refcnt being zero.
4654 */

4656 static void
4657 strlock(struct stdata *stp, sqlist_t *sqlist)
4658 {
4659     syncq_t *sql, *sql2;
4660     retry:

```

```

4661 /*
4662 * Wait for any claimstr to go away.
4663 */
4664 if (STRMATED(stp)) {
4665     struct stdata *stp1, *stp2;

4667     STRLOCKMATES(stp);
4668     /*
4669     * Note that the selection of locking order is not
4670     * important, just that they are always acquired in
4671     * the same order. To assure this, we choose this
4672     * order based on the value of the pointer, and since
4673     * the pointer will not change for the life of this
4674     * pair, we will always grab the locks in the same
4675     * order (and hence, prevent deadlocks).
4676     */
4677     if (&(stp->sd_lock) > &(stp->sd_mate->sd_lock)) {
4678         stp1 = stp;
4679         stp2 = stp->sd_mate;
4680     } else {
4681         stp2 = stp;
4682         stp1 = stp->sd_mate;
4683     }
4684     mutex_enter(&stp1->sd_reflock);
4685     if (stp1->sd_refcnt > 0) {
4686         STRUNLOCKMATES(stp);
4687         cv_wait(&stp1->sd_refmonitor, &stp1->sd_reflock);
4688         mutex_exit(&stp1->sd_reflock);
4689         goto retry;
4690     }
4691     mutex_enter(&stp2->sd_reflock);
4692     if (stp2->sd_refcnt > 0) {
4693         STRUNLOCKMATES(stp);
4694         mutex_exit(&stp1->sd_reflock);
4695         cv_wait(&stp2->sd_refmonitor, &stp2->sd_reflock);
4696         mutex_exit(&stp2->sd_reflock);
4697         goto retry;
4698     }
4699     STREAM_PUTLOCKS_ENTER(stp1);
4700     STREAM_PUTLOCKS_ENTER(stp2);
4701 } else {
4702     mutex_enter(&stp->sd_lock);
4703     mutex_enter(&stp->sd_reflock);
4704     while (stp->sd_refcnt > 0) {
4705         mutex_exit(&stp->sd_lock);
4706         cv_wait(&stp->sd_refmonitor, &stp->sd_reflock);
4707         if (mutex_tryenter(&stp->sd_lock) == 0) {
4708             mutex_exit(&stp->sd_reflock);
4709             mutex_enter(&stp->sd_lock);
4710             mutex_enter(&stp->sd_reflock);
4711         }
4712     }
4713     STREAM_PUTLOCKS_ENTER(stp);
4714 }

4716 if (sqlist == NULL)
4717     return;

4719 for (sql = sqlist->sqlist_head; sql; sql = sql->sql_next) {
4720     syncq_t *sq = sql->sql_sq;
4721     uint16_t count;

4723     mutex_enter(SQLOCK(sq));
4724     count = sq->sq_count;
4725     ASSERT(sq->sq_rmcount <= count);
4726     SQ_PUTLOCKS_ENTER(sq);

```

```

4727     SUM_SQ_PUTCOUNTS(sq, count);
4728     if (count == sq->sq_rmccount)
4729         continue;

4731     /* Failed - drop all locks that we have acquired so far */
4732     if (STRMATED(stp)) {
4733         STREAM_PUTLOCKS_EXIT(stp);
4734         STREAM_PUTLOCKS_EXIT(stp->sd_mate);
4735         STRUNLOCKMATES(stp);
4736         mutex_exit(&stp->sd_reflock);
4737         mutex_exit(&stp->sd_mate->sd_reflock);
4738     } else {
4739         STREAM_PUTLOCKS_EXIT(stp);
4740         mutex_exit(&stp->sd_lock);
4741         mutex_exit(&stp->sd_reflock);
4742     }
4743     for (sql2 = sqlist->sqlist_head; sql2 != sql;
4744         sql2 = sql2->sql_next) {
4745         SQ_PUTLOCKS_EXIT(sql2->sql_sq);
4746         mutex_exit(SQLOCK(sql2->sql_sq));
4747     }

4749     /*
4750     * The wait loop below may starve when there are many threads
4751     * claiming the syncq. This is especially a problem with permmod
4752     * syncqs (IP). To lessen the impact of the problem we increment
4753     * sq_needexcl and clear fastbits so that putnexts will slow
4754     * down and call sgenable instead of draining right away.
4755     */
4756     sq->sq_needexcl++;
4757     SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
4758     while (count > sq->sq_rmccount) {
4759         sq->sq_flags |= SQ_WANTWAKEUP;
4760         SQ_PUTLOCKS_EXIT(sq);
4761         cv_wait(&sq->sq_wait, SQLOCK(sq));
4762         count = sq->sq_count;
4763         SQ_PUTLOCKS_ENTER(sq);
4764         SUM_SQ_PUTCOUNTS(sq, count);
4765     }
4766     sq->sq_needexcl--;
4767     if (sq->sq_needexcl == 0)
4768         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
4769     SQ_PUTLOCKS_EXIT(sq);
4770     ASSERT(count == sq->sq_rmccount);
4771     mutex_exit(SQLOCK(sq));
4772     goto retry;
4773 }
4774 }

4776 /*
4777 * Drop all the locks that strlock acquired.
4778 */
4779 static void
4780 strunlock(struct stdata *stp, sqlist_t *sqlist)
4781 {
4782     syncql_t *sql;

4784     if (STRMATED(stp)) {
4785         STREAM_PUTLOCKS_EXIT(stp);
4786         STREAM_PUTLOCKS_EXIT(stp->sd_mate);
4787         STRUNLOCKMATES(stp);
4788         mutex_exit(&stp->sd_reflock);
4789         mutex_exit(&stp->sd_mate->sd_reflock);
4790     } else {
4791         STREAM_PUTLOCKS_EXIT(stp);
4792         mutex_exit(&stp->sd_lock);

```

```

4793         mutex_exit(&stp->sd_reflock);
4794     }

4796     if (sqlist == NULL)
4797         return;

4799     for (sql = sqlist->sqlist_head; sql; sql = sql->sql_next) {
4800         SQ_PUTLOCKS_EXIT(sql->sql_sq);
4801         mutex_exit(SQLOCK(sql->sql_sq));
4802     }
4803 }

4805 /*
4806 * When the module has service procedure, we need check if the next
4807 * module which has service procedure is in flow control to trigger
4808 * the backenable.
4809 */
4810 static void
4811 backenable_insertedq(queue_t *q)
4812 {
4813     qband_t *qbp;

4815     claimstr(q);
4816     if (q->q_qinfo->q_srvp != NULL && q->q_next != NULL) {
4817         if (q->q_next->q_nfsrv->q_flag & QWANTW)
4818             backenable(q, 0);

4820         qbp = q->q_next->q_nfsrv->q_bandp;
4821         for (; qbp != NULL; qbp = qbp->qb_next)
4822             if ((qbp->qb_flag & QB_WANTW) && qbp->qb_first != NULL)
4823                 backenable(q, qbp->qb_first->b_band);
4824     }
4825     releasestr(q);
4826 }

4828 /*
4829 * Given two read queues, insert a new single one after another.
4830 *
4831 * This routine acquires all the necessary locks in order to change
4832 * q_next and related pointer using strlock().
4833 * It depends on the stream head ensuring that there are no concurrent
4834 * insertq or removeq on the same stream. The stream head ensures this
4835 * using the flags STWOPEN, STRCLOSE, and STRPLUMB.
4836 *
4837 * Note that no syncq locks are held during the q_next change. This is
4838 * applied to all streams since, unlike removeq, there is no problem of stale
4839 * pointers when adding a module to the stream. Thus drivers/modules that do a
4840 * canput(rq->q_next) would never get a closed/freed queue pointer even if we
4841 * applied this optimization to all streams.
4842 */
4843 void
4844 insertq(struct stdata *stp, queue_t *new)
4845 {
4846     queue_t *after;
4847     queue_t *wafter;
4848     queue_t *wnew = _WR(new);
4849     boolean_t have_fifo = B_FALSE;

4851     if (new->q_flag & _INSERTING) {
4852         ASSERT(stp->sd_vnode->v_type != VFIFO);
4853         after = new->q_next;
4854         wafter = _WR(new->q_next);
4855     } else {
4856         after = _RD(stp->sd_wrq);
4857         wafter = stp->sd_wrq;
4858     }

```

```

4860 TRACE_2(TR_FAC_STREAMS_FR, TR_INSERTQ,
4861         "insertq:%p, %p", after, new);
4862 ASSERT(after->q_flag & QREADR);
4863 ASSERT(new->q_flag & QREADR);

4865 strlock(stp, NULL);

4867 /* Do we have a FIFO? */
4868 if (wafter->q_next == after) {
4869     have_fifo = B_TRUE;
4870     wnew->q_next = new;
4871 } else {
4872     wnew->q_next = wafter->q_next;
4873 }
4874 new->q_next = after;

4876 set_nfsrv_ptr(new, wnew, after, wafter);
4877 /*
4878  * set_nfsrv_ptr() needs to know if this is an insertion or not,
4879  * so only reset this flag after calling it.
4880  */
4881 new->q_flag &= ~QINSERTING;

4883 if (have_fifo) {
4884     wafter->q_next = wnew;
4885 } else {
4886     if (wafter->q_next)
4887         _OTHERQ(wafter->q_next)->q_next = new;
4888     wafter->q_next = wnew;
4889 }

4891 set_qend(new);
4892 /* The QEND flag might have to be updated for the upstream guy */
4893 set_qend(after);

4895 ASSERT(_SAMESTR(new) == O_SAMESTR(new));
4896 ASSERT(_SAMESTR(wnew) == O_SAMESTR(wnew));
4897 ASSERT(_SAMESTR(after) == O_SAMESTR(after));
4898 ASSERT(_SAMESTR(wafter) == O_SAMESTR(wafter));
4899 strsetuio(stp);

4901 /*
4902  * If this was a module insertion, bump the push count.
4903  */
4904 if (!(new->q_flag & QISDRV))
4905     stp->sd_pushcnt++;

4907 strunlock(stp, NULL);

4909 /* check if the write Q needs backenable */
4910 backenable_insertedq(wnew);

4912 /* check if the read Q needs backenable */
4913 backenable_insertedq(new);
4914 }

4916 /*
4917  * Given a read queue, unlink it from any neighbors.
4918  */
4919 * This routine acquires all the necessary locks in order to
4920 * change q_next and related pointers and also guard against
4921 * stale references (e.g. through q_next) to the queue that
4922 * is being removed. It also plays part of the role in ensuring
4923 * that the module's/driver's put procedure doesn't get called
4924 * after qprocsoff returns.

```

```

4925 *
4926 * Removeq depends on the stream head ensuring that there are
4927 * no concurrent insertq or removeq on the same stream. The
4928 * stream head ensures this using the flags STWOPEN, STRCLOSE and
4929 * STRPLUMB.
4930 *
4931 * The set of locks needed to remove the queue is different in
4932 * different cases:
4933 *
4934 * Acquire sd_lock, sd_reflock, and all the syncq locks in the stream after
4935 * waiting for the syncq reference count to drop to 0 indicating that no
4936 * non-close threads are present anywhere in the stream. This ensures that any
4937 * module/driver can reference q_next in its open, close, put, or service
4938 * procedures.
4939 *
4940 * The sq_rmcount counter tracks the number of threads inside removeq().
4941 * strlock() ensures that there is either no threads executing inside perimeter
4942 * or there is only a thread calling qprocsoff().
4943 *
4944 * strlock() compares the value of sq_count with the number of threads inside
4945 * removeq() and waits until sq_count is equal to sq_rmcount. We need to wakeup
4946 * any threads waiting in strlock() when the sq_rmcount increases.
4947 */

4949 void
4950 removeq(queue_t *qp)
4951 {
4952     queue_t *wqp = _WR(qp);
4953     struct stdata *stp = STREAM(qp);
4954     sqliist_t *sqliist = NULL;
4955     boolean_t isdriver;
4956     int moved;
4957     syncq_t *sq = qp->q_syncq;
4958     syncq_t *wsq = wqp->q_syncq;

4960     ASSERT(stp);

4962     TRACE_2(TR_FAC_STREAMS_FR, TR_REMOVEQ,
4963            "removeq:%p %p", qp, wqp);
4964     ASSERT(qp->q_flag & QREADR);

4966     /*
4967      * For queues using Synchronous streams, we must wait for all threads in
4968      * rwnext() to drain out before proceeding.
4969      */
4970     if (qp->q_flag & QSYNCSTR) {
4971         /* First, we need wakeup any threads blocked in rwnext() */
4972         mutex_enter(SQLOCK(sq));
4973         if (sq->sq_flags & SQ_WANTWAKEUP) {
4974             sq->sq_flags &= ~SQ_WANTWAKEUP;
4975             cv_broadcast(&sq->sq_wait);
4976         }
4977         mutex_exit(SQLOCK(sq));

4979         if (wsq != sq) {
4980             mutex_enter(SQLOCK(wsq));
4981             if (wsq->sq_flags & SQ_WANTWAKEUP) {
4982                 wsq->sq_flags &= ~SQ_WANTWAKEUP;
4983                 cv_broadcast(&wsq->sq_wait);
4984             }
4985             mutex_exit(SQLOCK(wsq));
4986         }

4988         mutex_enter(QLOCK(qp));
4989         while (qp->q_rwcnt > 0) {
4990             qp->q_flag |= QWANTRMQSYNC;

```

```

4991         cv_wait(&qp->q_wait, QLOCK(qp));
4992     }
4993     mutex_exit(QLOCK(qp));

4995     mutex_enter(QLOCK(wqp));
4996     while (wqp->q_rvcnt > 0) {
4997         wqp->q_flag |= QWANTRMQSYNC;
4998         cv_wait(&wqp->q_wait, QLOCK(wqp));
4999     }
5000     mutex_exit(QLOCK(wqp));
5001 }

5003 mutex_enter(SQLOCK(sq));
5004 sq->sq_rmcount++;
5005 if (sq->sq_flags & SQ_WANTWAKEUP) {
5006     sq->sq_flags &= ~SQ_WANTWAKEUP;
5007     cv_broadcast(&sq->sq_wait);
5008 }
5009 mutex_exit(SQLOCK(sq));

5011 isdriver = (qp->q_flag & QISDRV);

5013 sqlist = sqlist_build(qp, stp, STRMATED(stp));
5014 strlock(stp, sqlist);

5016 reset_nfsrv_ptr(qp, wqp);

5018 ASSERT(wqp->q_next == NULL || backq(qp)->q_next == qp);
5019 ASSERT(qp->q_next == NULL || backq(wqp)->q_next == wqp);
5020 /* Do we have a FIFO? */
5021 if (wqp->q_next == qp) {
5022     stp->sd_wrq->q_next = _RD(stp->sd_wrq);
5023 } else {
5024     if (wqp->q_next)
5025         backq(qp)->q_next = qp->q_next;
5026     if (qp->q_next)
5027         backq(wqp)->q_next = wqp->q_next;
5028 }

5030 /* The QEND flag might have to be updated for the upstream guy */
5031 if (qp->q_next)
5032     set_qend(qp->q_next);

5034 ASSERT(_SAMESTR(stp->sd_wrq) == 0_SAMESTR(stp->sd_wrq));
5035 ASSERT(_SAMESTR(_RD(stp->sd_wrq)) == 0_SAMESTR(_RD(stp->sd_wrq)));

5037 /*
5038  * Move any messages destined for the put procedures to the next
5039  * syncq in line. Otherwise free them.
5040  */
5041 moved = 0;
5042 /*
5043  * Quick check to see whether there are any messages or events.
5044  */
5045 if (qp->q_syncqmsgs != 0 || (qp->q_syncq->sq_flags & SQ_EVENTS))
5046     moved += propagate_syncq(qp);
5047 if (wqp->q_syncqmsgs != 0 ||
5048     (wqp->q_syncq->sq_flags & SQ_EVENTS))
5049     moved += propagate_syncq(wqp);

5051 strsetuio(stp);

5053 /*
5054  * If this was a module removal, decrement the push count.
5055  */
5056 if (!isdriver)

```

```

5057         stp->sd_pushcnt--;

5059     strunlock(stp, sqlist);
5060     sqlist_free(sqlist);

5062     /*
5063     * Make sure any messages that were propagated are drained.
5064     * Also clear any QFULL bit caused by messages that were propagated.
5065     */

5067     if (qp->q_next != NULL) {
5068         clr_qfull(qp);
5069         /*
5070          * For the driver calling qprocsoff, propagate_syncq
5071          * frees all the messages instead of putting it in
5072          * the stream head
5073          */
5074         if (!isdriver && (moved > 0))
5075             emptysq(qp->q_next->q_syncq);
5076     }
5077     if (wqp->q_next != NULL) {
5078         clr_qfull(wqp);
5079         /*
5080          * We come here for any pop of a module except for the
5081          * case of driver being removed. We don't call emptysq
5082          * if we did not move any messages. This will avoid holding
5083          * PERMOD syncq locks in emptysq
5084          */
5085         if (moved > 0)
5086             emptysq(wqp->q_next->q_syncq);
5087     }

5089     mutex_enter(SQLOCK(sq));
5090     sq->sq_rmcount--;
5091     mutex_exit(SQLOCK(sq));
5092 }

5094 /*
5095  * Prevent further entry by setting a flag (like SQ_FROZEN, SQ_BLOCKED or
5096  * SQ_WRITER) on a syncq.
5097  * If maxcnt is not -1 it assumes that caller has "maxcnt" claim(s) on the
5098  * sync queue and waits until sq_count reaches maxcnt.
5099  *
5100  * If maxcnt is -1 there's no need to grab sq_putlocks since the caller
5101  * does not care about putnext threads that are in the middle of calling put
5102  * entry points.
5103  *
5104  * This routine is used for both inner and outer syncqs.
5105  */
5106 static void
5107 blocksq(syncq_t *sq, ushort_t flag, int maxcnt)
5108 {
5109     uint16_t count = 0;

5111     mutex_enter(SQLOCK(sq));
5112     /*
5113     * Wait for SQ_FROZEN/SQ_BLOCKED to be reset.
5114     * SQ_FROZEN will be set if there is a frozen stream that has a
5115     * queue which also refers to this "shared" syncq.
5116     * SQ_BLOCKED will be set if there is "off" queue which also
5117     * refers to this "shared" syncq.
5118     */
5119     if (maxcnt != -1) {
5120         count = sq->sq_count;
5121         SQ_PUTLOCKS_ENTER(sq);
5122         SQ_PUTCOUNT_CLRFAST_LOCKED(sq);

```

```

5123     SUM_SQ_PUTCOUNTS(sq, count);
5124 }
5125 sq->sq_needexcl++;
5126 ASSERT(sq->sq_needexcl != 0); /* wraparound */

5128 while ((sq->sq_flags & flag) ||
5129 (maxcnt != -1 && count > (unsigned)maxcnt)) {
5130     sq->sq_flags |= SQ_WANTWAKEUP;
5131     if (maxcnt != -1) {
5132         SQ_PUTLOCKS_EXIT(sq);
5133     }
5134     cv_wait(&sq->sq_wait, SLOCK(sq));
5135     if (maxcnt != -1) {
5136         count = sq->sq_count;
5137         SQ_PUTLOCKS_ENTER(sq);
5138         SUM_SQ_PUTCOUNTS(sq, count);
5139     }
5140 }
5141 sq->sq_needexcl--;
5142 sq->sq_flags |= flag;
5143 ASSERT(maxcnt == -1 || count == maxcnt);
5144 if (maxcnt != -1) {
5145     if (sq->sq_needexcl == 0) {
5146         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
5147     }
5148     SQ_PUTLOCKS_EXIT(sq);
5149 } else if (sq->sq_needexcl == 0) {
5150     SQ_PUTCOUNT_SETFAST(sq);
5151 }

5153 mutex_exit(SLOCK(sq));
5154 }

5156 /*
5157 * Reset a flag that was set with blocksq.
5158 *
5159 * Can not use this routine to reset SQ_WRITER.
5160 *
5161 * If "isouter" is set then the syncq is assumed to be an outer perimeter
5162 * and drain_syncq is not called. Instead we rely on the qwriter_outer thread
5163 * to handle the queued qwriter operations.
5164 *
5165 * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5166 * sq_putlocks are used.
5167 */
5168 static void
5169 unblocksq(syncq_t *sq, uint16_t resetflag, int isouter)
5170 {
5171     uint16_t flags;

5173     mutex_enter(SLOCK(sq));
5174     ASSERT(resetflag != SQ_WRITER);
5175     ASSERT(sq->sq_flags & resetflag);
5176     flags = sq->sq_flags & ~resetflag;
5177     sq->sq_flags = flags;
5178     if (flags & (SQ_QUEUED | SQ_WANTWAKEUP)) {
5179         if (flags & SQ_WANTWAKEUP) {
5180             flags &= ~SQ_WANTWAKEUP;
5181             cv_broadcast(&sq->sq_wait);
5182         }
5183         sq->sq_flags = flags;
5184         if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5185             if (!isouter) {
5186                 /* drain_syncq drops SLOCK */
5187                 drain_syncq(sq);
5188                 return;

```

```

5189     }
5190 }
5191 }
5192 mutex_exit(SLOCK(sq));
5193 }

5195 /*
5196 * Reset a flag that was set with blocksq.
5197 * Does not drain the syncq. Use emptysq() for that.
5198 * Returns 1 if SQ_QUEUED is set. Otherwise 0.
5199 *
5200 * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5201 * sq_putlocks are used.
5202 */
5203 static int
5204 dropsq(syncq_t *sq, uint16_t resetflag)
5205 {
5206     uint16_t flags;

5208     mutex_enter(SLOCK(sq));
5209     ASSERT(sq->sq_flags & resetflag);
5210     flags = sq->sq_flags & ~resetflag;
5211     if (flags & SQ_WANTWAKEUP) {
5212         flags &= ~SQ_WANTWAKEUP;
5213         cv_broadcast(&sq->sq_wait);
5214     }
5215     sq->sq_flags = flags;
5216     mutex_exit(SLOCK(sq));
5217     if (flags & SQ_QUEUED)
5218         return (1);
5219     return (0);
5220 }

5222 /*
5223 * Empty all the messages on a syncq.
5224 *
5225 * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5226 * sq_putlocks are used.
5227 */
5228 static void
5229 emptysq(syncq_t *sq)
5230 {
5231     uint16_t flags;

5233     mutex_enter(SLOCK(sq));
5234     flags = sq->sq_flags;
5235     if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5236         /*
5237          * To prevent potential recursive invocation of drain_syncq we
5238          * do not call drain_syncq if count is non-zero.
5239          */
5240         if (sq->sq_count == 0) {
5241             /* drain_syncq() drops SLOCK */
5242             drain_syncq(sq);
5243             return;
5244         } else
5245             sqenable(sq);
5246     }
5247     mutex_exit(SLOCK(sq));
5248 }

5250 /*
5251 * Ordered insert while removing duplicates.
5252 */
5253 static void
5254 sqliist_insert(sqliist_t *sqliist, syncq_t *sqp)

```

```

5255 {
5256     syncql_t *sqlp, **prev_sqlpp, *new_sqlp;

5258     prev_sqlpp = &sqlist->sqlist_head;
5259     while ((sqlp = *prev_sqlpp) != NULL) {
5260         if (sqlp->sql_sq >= sqp) {
5261             if (sqlp->sql_sq == sqp) /* duplicate */
5262                 return;
5263             break;
5264         }
5265         prev_sqlpp = &sqlp->sql_next;
5266     }
5267     new_sqlp = &sqlist->sqlist_array[sqlist->sqlist_index++];
5268     ASSERT((char *)new_sqlp < (char *)sqlist + sqlist->sqlist_size);
5269     new_sqlp->sql_next = sqlp;
5270     new_sqlp->sql_sq = sqp;
5271     *prev_sqlpp = new_sqlp;
5272 }

5274 /*
5275  * Walk the write side queues until we hit either the driver
5276  * or a twist in the stream (_SAMESTR will return false in both
5277  * these cases) then turn around and walk the read side queues
5278  * back up to the stream head.
5279  */
5280 static void
5281 sqlist_insertall(sqlist_t *sqlist, queue_t *q)
5282 {
5283     while (q != NULL) {
5284         sqlist_insert(sqlist, q->q_syncq);

5286         if (_SAMESTR(q))
5287             q = q->q_next;
5288         else if (!(q->q_flag & QREADR))
5289             q = _RD(q);
5290         else
5291             q = NULL;
5292     }
5293 }

5295 /*
5296  * Allocate and build a list of all syncqs in a stream and the syncq(s)
5297  * associated with the "q" parameter. The resulting list is sorted in a
5298  * canonical order and is free of duplicates.
5299  * Assumes the passed queue is a _RD(q).
5300  */
5301 static sqlist_t *
5302 sqlist_build(queue_t *q, struct stdata *stp, boolean_t do_twist)
5303 {
5304     sqlist_t *sqlist = sqlist_alloc(stp, KM_SLEEP);

5306     /*
5307      * start with the current queue/qpair
5308      */
5309     ASSERT(q->q_flag & QREADR);

5311     sqlist_insert(sqlist, q->q_syncq);
5312     sqlist_insert(sqlist, _WR(q)->q_syncq);

5314     sqlist_insertall(sqlist, stp->sd_wrq);
5315     if (do_twist)
5316         sqlist_insertall(sqlist, stp->sd_mate->sd_wrq);

5318     return (sqlist);
5319 }

```

```

5321 static sqlist_t *
5322 sqlist_alloc(struct stdata *stp, int kmflag)
5323 {
5324     size_t sqlist_size;
5325     sqlist_t *sqlist;

5327     /*
5328      * Allocate 2 syncql_t's for each pushed module. Note that
5329      * the sqlist_t structure already has 4 syncql_t's built in:
5330      * 2 for the stream head, and 2 for the driver/other stream head.
5331      */
5332     sqlist_size = 2 * sizeof (syncql_t) * stp->sd_pushcnt +
5333                 sizeof (sqlist_t);
5334     if (STRMATED(stp))
5335         sqlist_size += 2 * sizeof (syncql_t) * stp->sd_mate->sd_pushcnt;
5336     sqlist = kmem_alloc(sqlist_size, kmflag);

5338     sqlist->sqlist_head = NULL;
5339     sqlist->sqlist_size = sqlist_size;
5340     sqlist->sqlist_index = 0;

5342     return (sqlist);
5343 }

5345 /*
5346  * Free the list created by sqlist_alloc()
5347  */
5348 static void
5349 sqlist_free(sqlist_t *sqlist)
5350 {
5351     kmem_free(sqlist, sqlist->sqlist_size);
5352 }

5354 /*
5355  * Prevent any new entries into any syncq in this stream.
5356  * Used by freezestr.
5357  */
5358 void
5359 strblock(queue_t *q)
5360 {
5361     struct stdata *stp;
5362     syncql_t *sql;
5363     sqlist_t *sqlist;

5365     q = _RD(q);

5367     stp = STREAM(q);
5368     ASSERT(stp != NULL);

5370     /*
5371      * Get a sorted list with all the duplicates removed containing
5372      * all the syncqs referenced by this stream.
5373      */
5374     sqlist = sqlist_build(q, stp, B_FALSE);
5375     for (sql = sqlist->sqlist_head; sql != NULL; sql = sql->sql_next)
5376         blocksq(sql->sql_sq, SQ_FROZEN, -1);
5377     sqlist_free(sqlist);
5378 }

5380 /*
5381  * Release the block on new entries into this stream
5382  */
5383 void
5384 strunblock(queue_t *q)
5385 {
5386     struct stdata *stp;

```

```

5387     syncql_t      *sql;
5388     sqlist_t      *sqlist;
5389     int           drain_needed;

5391     q = _RD(q);

5393     /*
5394     * Get a sorted list with all the duplicates removed containing
5395     * all the syncqs referenced by this stream.
5396     * Have to drop the SQ_FROZEN flag on all the syncqs before
5397     * starting to drain them; otherwise the draining might
5398     * cause a freeze in some module on the stream (which
5399     * would deadlock).
5400     */
5401     stp = STREAM(q);
5402     ASSERT(stp != NULL);
5403     sqlist = sqlist_build(q, stp, B_FALSE);
5404     drain_needed = 0;
5405     for (sql = sqlist->sqlist_head; sql != NULL; sql = sql->sql_next)
5406         drain_needed += dropsq(sql->sql_sq, SQ_FROZEN);
5407     if (drain_needed) {
5408         for (sql = sqlist->sqlist_head; sql != NULL;
5409              sql = sql->sql_next)
5410             emptysq(sql->sql_sq);
5411     }
5412     sqlist_free(sqlist);
5413 }

5415 #ifdef DEBUG
5416 static int
5417 qprocsareon(queue_t *rq)
5418 {
5419     if (rq->q_next == NULL)
5420         return (0);
5421     return (_WR(rq->q_next)->q_next == _WR(rq));
5422 }

5424 int
5425 qclaimed(queue_t *q)
5426 {
5427     uint_t count;

5429     count = q->q_syncq->sq_count;
5430     SUM_SQ_PUTCOUNTS(q->q_syncq, count);
5431     return (count != 0);
5432 }

5434 /*
5435 * Check if anyone has frozen this stream with freeze in
5436 */
5437 int
5438 frozenstr(queue_t *q)
5439 {
5440     return ((q->q_syncq->sq_flags & SQ_FROZEN) != 0);
5441 }
5442 #endif /* DEBUG */

5444 /*
5445 * Enter a queue.
5446 * Obsolete interface. Should not be used.
5447 */
5448 void
5449 enterq(queue_t *q)
5450 {
5451     entersq(q->q_syncq, SQ_CALLBACK);
5452 }

```

```

5454 void
5455 leaveq(queue_t *q)
5456 {
5457     leavesq(q->q_syncq, SQ_CALLBACK);
5458 }

5460 /*
5461 * Enter a perimeter. c_inner and c_outer specifies which concurrency bits
5462 * to check.
5463 * Wait if SQ_QUEUED is set to preserve ordering between messages and qwriter
5464 * calls and the running of open, close and service procedures.
5465 *
5466 * If c_inner bit is set no need to grab sq_putlocks since we don't care
5467 * if other threads have entered or are entering put entry point.
5468 *
5469 * If c_inner bit is set it might have been possible to use
5470 * sq_putlocks/sq_putcounts instead of SLOCK/sq_count (e.g. to optimize
5471 * open/close path for IP) but since the count may need to be decremented in
5472 * qwait() we wouldn't know which counter to decrement. Currently counter is
5473 * selected by current cpu_segid and current CPU can change at any moment. XXX
5474 * in the future we might use curthread id bits to select the counter and this
5475 * would stay constant across routine calls.
5476 */
5477 void
5478 entersq(syncq_t *sq, int entrypoint)
5479 {
5480     uint16_t count = 0;
5481     uint16_t flags;
5482     uint16_t waitflags = SQ_STAYAWAY | SQ_EVENTS | SQ_EXCL;
5483     type_t type;
5484     uint_t c_inner = entrypoint & SQ_CI;
5485     uint_t c_outer = entrypoint & SQ_CO;

5487     /*
5488     * Increment ref count to keep closes out of this queue.
5489     */
5490     ASSERT(sq);
5491     ASSERT(c_inner && c_outer);
5492     mutex_enter(SLOCK(sq));
5493     flags = sq->sq_flags;
5494     type = sq->sq_type;
5495     if (!(type & c_inner)) {
5496         /* Make sure all putcounts now use slowlock. */
5497         count = sq->sq_count;
5498         SQ_PUTLOCKS_ENTER(sq);
5499         SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
5500         SUM_SQ_PUTCOUNTS(sq, count);
5501         sq->sq_needexcl++;
5502         ASSERT(sq->sq_needexcl != 0); /* wraparound */
5503         waitflags |= SQ_MESSAGES;
5504     }
5505     /*
5506     * Wait until we can enter the inner perimeter.
5507     * If we want exclusive access we wait until sq_count is 0.
5508     * We have to do this before entering the outer perimeter in order
5509     * to preserve put/close message ordering.
5510     */
5511     while ((flags & waitflags) || (!(type & c_inner) && count != 0)) {
5512         sq->sq_flags = flags | SQ_WANTWAKEUP;
5513         if (!(type & c_inner)) {
5514             SQ_PUTLOCKS_EXIT(sq);
5515         }
5516         cv_wait(&sq->sq_wait, SLOCK(sq));
5517         if (!(type & c_inner)) {
5518             count = sq->sq_count;

```

```

5519         SQ_PUTLOCKS_ENTER(sq);
5520         SUM_SQ_PUTCOUNTS(sq, count);
5521     }
5522     flags = sq->sq_flags;
5523 }

5525 if (!(type & c_inner)) {
5526     ASSERT(sq->sq_needexcl > 0);
5527     sq->sq_needexcl--;
5528     if (sq->sq_needexcl == 0) {
5529         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
5530     }
5531 }

5533 /* Check if we need to enter the outer perimeter */
5534 if (!(type & c_outer)) {
5535     /*
5536      * We have to enter the outer perimeter exclusively before
5537      * we can increment sq_count to avoid deadlock. This implies
5538      * that we have to re-check sq_flags and sq_count.
5539      *
5540      * is it possible to have c_inner set when c_outer is not set?
5541      */
5542     if (!(type & c_inner)) {
5543         SQ_PUTLOCKS_EXIT(sq);
5544     }
5545     mutex_exit(SQLOCK(sq));
5546     outer_enter(sq->sq_outer, SQ_GOAWAY);
5547     mutex_enter(SQLOCK(sq));
5548     flags = sq->sq_flags;
5549     /*
5550      * there should be no need to recheck sq_putcounts
5551      * because outer_enter() has already waited for them to clear
5552      * after setting SQ_WRITER.
5553      */
5554     count = sq->sq_count;
5555 #ifndef DEBUG
5556     /*
5557      * SUMCHECK_SQ_PUTCOUNTS should return the sum instead
5558      * of doing an ASSERT internally. Others should do
5559      * something like
5560      *     ASSERT(SUMCHECK_SQ_PUTCOUNTS(sq) == 0);
5561      * without the need to #ifdef DEBUG it.
5562      */
5563     SUMCHECK_SQ_PUTCOUNTS(sq, 0);
5564 #endif
5565     while ((flags & (SQ_EXCL|SQ_BLOCKED|SQ_FROZEN)) ||
5566           (!(type & c_inner) && count != 0)) {
5567         sq->sq_flags = flags | SQ_WANTWAKEUP;
5568         cv_wait(&sq->sq_wait, SQLOCK(sq));
5569         count = sq->sq_count;
5570         flags = sq->sq_flags;
5571     }
5572 }

5574 sq->sq_count++;
5575 ASSERT(sq->sq_count != 0); /* Wraparound */
5576 if (!(type & c_inner)) {
5577     /* Exclusive entry */
5578     ASSERT(sq->sq_count == 1);
5579     sq->sq_flags |= SQ_EXCL;
5580     if (type & c_outer) {
5581         SQ_PUTLOCKS_EXIT(sq);
5582     }
5583 }
5584 mutex_exit(SQLOCK(sq));

```

```

5585 }

5587 /*
5588  * Leave a syncq. Announce to framework that closes may proceed.
5589  * c_inner and c_outer specify which concurrency bits to check.
5590  *
5591  * Must never be called from driver or module put entry point.
5592  *
5593  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5594  * sq_putlocks are used.
5595  */
5596 void
5597 leavesq(syncq_t *sq, int entrypoint)
5598 {
5599     uint16_t    flags;
5600     uint16_t    type;
5601     uint_t      c_outer = entrypoint & SQ_CO;
5602 #ifdef DEBUG
5603     uint_t      c_inner = entrypoint & SQ_CI;
5604 #endif

5606     /*
5607      * Decrement ref count, drain the syncq if possible, and wake up
5608      * any waiting close.
5609      */
5610     ASSERT(sq);
5611     ASSERT(c_inner && c_outer);
5612     mutex_enter(SQLOCK(sq));
5613     flags = sq->sq_flags;
5614     type = sq->sq_type;
5615     if (flags & (SQ_QUEUED|SQ_WANTWAKEUP|SQ_WANTEXWAKEUP)) {
5617         if (flags & SQ_WANTWAKEUP) {
5618             flags &= ~SQ_WANTWAKEUP;
5619             cv_broadcast(&sq->sq_wait);
5620         }
5621         if (flags & SQ_WANTEXWAKEUP) {
5622             flags &= ~SQ_WANTEXWAKEUP;
5623             cv_broadcast(&sq->sq_exitwait);
5624         }
5626         if ((flags & SQ_QUEUED) && !(flags & SQ_STAYAWAY)) {
5627             /*
5628              * The syncq needs to be drained. "Exit" the syncq
5629              * before calling drain_syncq.
5630              */
5631             ASSERT(sq->sq_count != 0);
5632             sq->sq_count--;
5633             ASSERT((flags & SQ_EXCL) || (type & c_inner));
5634             sq->sq_flags = flags & ~SQ_EXCL;
5635             drain_syncq(sq);
5636             ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
5637             /* Check if we need to exit the outer perimeter */
5638             /* XXX will this ever be true? */
5639             if (!(type & c_outer))
5640                 outer_exit(sq->sq_outer);
5641             return;
5642         }
5643     }
5644     ASSERT(sq->sq_count != 0);
5645     sq->sq_count--;
5646     ASSERT((flags & SQ_EXCL) || (type & c_inner));
5647     sq->sq_flags = flags & ~SQ_EXCL;
5648     mutex_exit(SQLOCK(sq));

5650     /* Check if we need to exit the outer perimeter */

```

```

5651     if (!(sq->sq_type & c_outer))
5652         outer_exit(sq->sq_outer);
5653 }

5655 /*
5656  * Prevent q_next from changing in this stream by incrementing sq_count.
5657  */
5658 * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5659 * sq_putlocks are used.
5660 */
5661 void
5662 claimq(queue_t *qp)
5663 {
5664     syncq_t *sq = qp->q_syncq;

5666     mutex_enter(SQLOCK(sq));
5667     sq->sq_count++;
5668     ASSERT(sq->sq_count != 0);      /* Wraparound */
5669     mutex_exit(SQLOCK(sq));
5670 }

5672 /*
5673  * Undo claimq.
5674  */
5675 * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
5676 * sq_putlocks are used.
5677 */
5678 void
5679 releaseq(queue_t *qp)
5680 {
5681     syncq_t *sq = qp->q_syncq;
5682     uint16_t flags;

5684     mutex_enter(SQLOCK(sq));
5685     ASSERT(sq->sq_count > 0);
5686     sq->sq_count--;

5688     flags = sq->sq_flags;
5689     if (flags & (SQ_WANTWAKEUP|SQ_QUEUED)) {
5690         if (flags & SQ_WANTWAKEUP) {
5691             flags &= ~SQ_WANTWAKEUP;
5692             cv_broadcast(&sq->sq_wait);
5693         }
5694         sq->sq_flags = flags;
5695         if ((flags & SQ_QUEUED) && !(flags & (SQ_STAYAWAY|SQ_EXCL))) {
5696             /*
5697              * To prevent potential recursive invocation of
5698              * drain_syncq we do not call drain_syncq if count is
5699              * non-zero.
5700              */
5701             if (sq->sq_count == 0) {
5702                 drain_syncq(sq);
5703                 return;
5704             } else
5705                 sqenable(sq);
5706         }
5707     }
5708     mutex_exit(SQLOCK(sq));
5709 }

5711 /*
5712  * Prevent q_next from changing in this stream by incrementing sd_refcnt.
5713  */
5714 void
5715 claimstr(queue_t *qp)
5716 {

```

```

5717     struct stdata *stp = STREAM(qp);

5719     mutex_enter(&stp->sd_reflock);
5720     stp->sd_refcnt++;
5721     ASSERT(stp->sd_refcnt != 0);      /* Wraparound */
5722     mutex_exit(&stp->sd_reflock);
5723 }

5725 /*
5726  * Undo claimstr.
5727  */
5728 void
5729 releasestr(queue_t *qp)
5730 {
5731     struct stdata *stp = STREAM(qp);

5733     mutex_enter(&stp->sd_reflock);
5734     ASSERT(stp->sd_refcnt != 0);
5735     if (--stp->sd_refcnt == 0)
5736         cv_broadcast(&stp->sd_refmonitor);
5737     mutex_exit(&stp->sd_reflock);
5738 }

5740 static syncq_t *
5741 new_syncq(void)
5742 {
5743     return (kmem_cache_alloc(syncq_cache, KM_SLEEP));
5744 }

5746 static void
5747 free_syncq(syncq_t *sq)
5748 {
5749     ASSERT(sq->sq_head == NULL);
5750     ASSERT(sq->sq_outer == NULL);
5751     ASSERT(sq->sq_callbpend == NULL);
5752     ASSERT((sq->sq_onext == NULL && sq->sq_oprev == NULL) ||
5753            (sq->sq_onext == sq && sq->sq_oprev == sq));

5755     if (sq->sq_ciputctrl != NULL) {
5756         ASSERT(sq->sq_nciputctrl == n_ciputctrl - 1);
5757         SUMCHECK_CIPUTCTRL_COUNTS(sq->sq_ciputctrl,
5758                                 sq->sq_nciputctrl, 0);
5759         ASSERT(ciputctrl_cache != NULL);
5760         kmem_cache_free(ciputctrl_cache, sq->sq_ciputctrl);
5761     }

5763     sq->sq_tail = NULL;
5764     sq->sq_evhead = NULL;
5765     sq->sq_evtail = NULL;
5766     sq->sq_ciputctrl = NULL;
5767     sq->sq_nciputctrl = 0;
5768     sq->sq_count = 0;
5769     sq->sq_rmqcount = 0;
5770     sq->sq_callbflags = 0;
5771     sq->sq_cancelid = 0;
5772     sq->sq_next = NULL;
5773     sq->sq_needexcl = 0;
5774     sq->sq_svcflags = 0;
5775     sq->sq_nqueues = 0;
5776     sq->sq_pri = 0;
5777     sq->sq_onext = NULL;
5778     sq->sq_oprev = NULL;
5779     sq->sq_flags = 0;
5780     sq->sq_type = 0;
5781     sq->sq_servcount = 0;

```

```

5783     kmem_cache_free(syncq_cache, sq);
5784 }

5786 /* Outer perimeter code */

5788 /*
5789 * The outer syncq uses the fields and flags in the syncq slightly
5790 * differently from the inner syncqs.
5791 * sq_count      Incremented when there are pending or running
5792 *               writers at the outer perimeter to prevent the set of
5793 *               inner syncqs that belong to the outer perimeter from
5794 *               changing.
5795 * sq_head/tail  List of deferred qwriter(OUTER) operations.
5796 *
5797 * SQ_BLOCKED   Set to prevent traversing of sq_next,sq_prev while
5798 *               inner syncqs are added to or removed from the
5799 *               outer perimeter.
5800 * SQ_QUEUED    sq_head/tail has messages or events queued.
5801 *
5802 * SQ_WRITER    A thread is currently traversing all the inner syncqs
5803 *               setting the SQ_WRITER flag.
5804 */

5806 /*
5807 * Get write access at the outer perimeter.
5808 * Note that read access is done by entersq, putnext, and put by simply
5809 * incrementing sq_count in the inner syncq.
5810 *
5811 * Waits until "flags" is no longer set in the outer to prevent multiple
5812 * threads from having write access at the same time. SQ_WRITER has to be part
5813 * of "flags".
5814 *
5815 * Increases sq_count on the outer syncq to keep away outer_insert/remove
5816 * until the outer_exit is finished.
5817 *
5818 * outer_enter is vulnerable to starvation since it does not prevent new
5819 * threads from entering the inner syncqs while it is waiting for sq_count to
5820 * go to zero.
5821 */
5822 void
5823 outer_enter(syncq_t *outer, uint16_t flags)
5824 {
5825     syncq_t *sq;
5826     int wait_needed;
5827     uint16_t count;

5829     ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
5830           outer->sq_oprev != NULL);
5831     ASSERT(flags & SQ_WRITER);

5833 retry:
5834     mutex_enter(SQLOCK(outer));
5835     while (outer->sq_flags & flags) {
5836         outer->sq_flags |= SQ_WANTWAKEUP;
5837         cv_wait(&outer->sq_wait, SQLOCK(outer));
5838     }

5840     ASSERT(!(outer->sq_flags & SQ_WRITER));
5841     outer->sq_flags |= SQ_WRITER;
5842     outer->sq_count++;
5843     ASSERT(outer->sq_count != 0); /* wraparound */
5844     wait_needed = 0;
5845     /*
5846     * Set SQ_WRITER on all the inner syncqs while holding
5847     * the SQLOCK on the outer syncq. This ensures that the changing
5848     * of SQ_WRITER is atomic under the outer SQLOCK.

```

```

5849     */
5850     for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
5851         mutex_enter(SQLOCK(sq));
5852         count = sq->sq_count;
5853         SQ_PUTLOCKS_ENTER(sq);
5854         sq->sq_flags |= SQ_WRITER;
5855         SUM_SQ_PUTCOUNTS(sq, count);
5856         if (count != 0)
5857             wait_needed = 1;
5858         SQ_PUTLOCKS_EXIT(sq);
5859         mutex_exit(SQLOCK(sq));
5860     }
5861     mutex_exit(SQLOCK(outer));

5863     /*
5864     * Get everybody out of the syncqs sequentially.
5865     * Note that we don't actually need to acquire the PUTLOCKS, since
5866     * we have already cleared the fastbit, and set QWRITER. By
5867     * definition, the count can not increase since putnext will
5868     * take the slowlock path (and the purpose of acquiring the
5869     * putlocks was to make sure it didn't increase while we were
5870     * waiting).
5871     *
5872     * Note that we still acquire the PUTLOCKS to be safe.
5873     */
5874     if (wait_needed) {
5875         for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
5876             mutex_enter(SQLOCK(sq));
5877             count = sq->sq_count;
5878             SQ_PUTLOCKS_ENTER(sq);
5879             SUM_SQ_PUTCOUNTS(sq, count);
5880             while (count != 0) {
5881                 sq->sq_flags |= SQ_WANTWAKEUP;
5882                 SQ_PUTLOCKS_EXIT(sq);
5883                 cv_wait(&sq->sq_wait, SQLOCK(sq));
5884                 count = sq->sq_count;
5885                 SQ_PUTLOCKS_ENTER(sq);
5886                 SUM_SQ_PUTCOUNTS(sq, count);
5887             }
5888             SQ_PUTLOCKS_EXIT(sq);
5889             mutex_exit(SQLOCK(sq));
5890         }
5891     }
5892     /*
5893     * Verify that none of the flags got set while we
5894     * were waiting for the sq_counts to drop.
5895     * If this happens we exit and retry entering the
5896     * outer perimeter.
5897     */
5898     mutex_enter(SQLOCK(outer));
5899     if (outer->sq_flags & (flags & ~SQ_WRITER)) {
5900         mutex_exit(SQLOCK(outer));
5901         outer_exit(outer);
5902         goto retry;
5903     }
5904     mutex_exit(SQLOCK(outer));
5905 }

5907 /*
5908 * Drop the write access at the outer perimeter.
5909 * Read access is dropped implicitly (by putnext, put, and leavesq) by
5910 * decrementing sq_count.
5911 */
5912 void
5913 outer_exit(syncq_t *outer)
5914 {

```

```

5915     syncq_t *sq;
5916     int     drain_needed;
5917     uint16_t flags;

5919     ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
5920            outer->sq_oprev != NULL);
5921     ASSERT(MUTEX_NOT_HELD(SQLLOCK(outer)));

5923     /*
5924      * Atomically (from the perspective of threads calling become_writer)
5925      * drop the write access at the outer perimeter by holding
5926      * SQLLOCK(outer) across all the dropsq calls and the resetting of
5927      * SQ_WRITER.
5928      * This defines a locking order between the outer perimeter
5929      * SQLLOCK and the inner perimeter SQLLOCKS.
5930      */
5931     mutex_enter(SQLLOCK(outer));
5932     flags = outer->sq_flags;
5933     ASSERT(outer->sq_flags & SQ_WRITER);
5934     if (flags & SQ_QUEUED) {
5935         write_now(outer);
5936         flags = outer->sq_flags;
5937     }

5939     /*
5940      * sq_onext is stable since sq_count has not yet been decreased.
5941      * Reset the SQ_WRITER flags in all syncqs.
5942      * After dropping SQ_WRITER on the outer syncq we empty all the
5943      * inner syncqs.
5944      */
5945     drain_needed = 0;
5946     for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext)
5947         drain_needed += dropsq(sq, SQ_WRITER);
5948     ASSERT(!(outer->sq_flags & SQ_QUEUED));
5949     flags &= ~SQ_WRITER;
5950     if (drain_needed) {
5951         outer->sq_flags = flags;
5952         mutex_exit(SQLLOCK(outer));
5953         for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext)
5954             emptysq(sq);
5955         mutex_enter(SQLLOCK(outer));
5956         flags = outer->sq_flags;
5957     }
5958     if (flags & SQ_WANTWAKEUP) {
5959         flags &= ~SQ_WANTWAKEUP;
5960         cv_broadcast(&outer->sq_wait);
5961     }
5962     outer->sq_flags = flags;
5963     ASSERT(outer->sq_count > 0);
5964     outer->sq_count--;
5965     mutex_exit(SQLLOCK(outer));
5966 }

5968 /*
5969  * Add another syncq to an outer perimeter.
5970  * Block out all other access to the outer perimeter while it is being
5971  * changed using blocksq.
5972  * Assumes that the caller has *not* done an outer_enter.
5973  * Vulnerable to starvation in blocksq.
5974  */
5975 static void
5976 outer_insert(syncq_t *outer, syncq_t *sq)
5977 {
5978     ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
5979            outer->sq_oprev != NULL);

```

```

5981     ASSERT(sq->sq_outer == NULL && sq->sq_onext == NULL &&
5982            sq->sq_oprev == NULL); /* Can't be in an outer perimeter */

5984     /* Get exclusive access to the outer perimeter list */
5985     blocksq(outer, SQ_BLOCKED, 0);
5986     ASSERT(outer->sq_flags & SQ_BLOCKED);
5987     ASSERT(!(outer->sq_flags & SQ_WRITER));

5989     mutex_enter(SQLLOCK(sq));
5990     sq->sq_outer = outer;
5991     outer->sq_onext->sq_oprev = sq;
5992     sq->sq_onext = outer->sq_onext;
5993     outer->sq_onext = sq;
5994     sq->sq_oprev = outer;
5995     mutex_exit(SQLLOCK(sq));
5996     unblocksq(outer, SQ_BLOCKED, 1);
5997 }

5999 /*
6000  * Remove a syncq from an outer perimeter.
6001  * Block out all other access to the outer perimeter while it is being
6002  * changed using blocksq.
6003  * Assumes that the caller has *not* done an outer_enter.
6004  * Vulnerable to starvation in blocksq.
6005  */
6006 static void
6007 outer_remove(syncq_t *outer, syncq_t *sq)
6008 {
6009     ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
6010            outer->sq_oprev != NULL);
6011     ASSERT(sq->sq_outer == outer);

6014     /* Get exclusive access to the outer perimeter list */
6015     blocksq(outer, SQ_BLOCKED, 0);
6016     ASSERT(outer->sq_flags & SQ_BLOCKED);
6017     ASSERT(!(outer->sq_flags & SQ_WRITER));

6019     mutex_enter(SQLLOCK(sq));
6020     sq->sq_outer = NULL;
6021     sq->sq_onext->sq_oprev = sq->sq_oprev;
6022     sq->sq_oprev->sq_onext = sq->sq_onext;
6023     sq->sq_oprev = sq->sq_onext = NULL;
6024     mutex_exit(SQLLOCK(sq));
6025     unblocksq(outer, SQ_BLOCKED, 1);
6026 }

6028 /*
6029  * Queue a deferred qwriter(OUTER) callback for this outer perimeter.
6030  * If this is the first callback for this outer perimeter then add
6031  * this outer perimeter to the list of outer perimeters that
6032  * the qwriter_outer_thread will process.
6033  * Increments sq_count in the outer syncq to prevent the membership
6034  * of the outer perimeter (in terms of inner syncqs) to change while
6035  * the callback is pending.
6036  */
6037 static void
6038 queue_writer(syncq_t *outer, void (*func)(), queue_t *q, mblk_t *mp)
6039 {
6040     ASSERT(MUTEX_HELD(SQLLOCK(outer)));

6043     mp->b_prev = (mblk_t *)func;
6044     mp->b_queue = q;
6045     mp->b_next = NULL;
6046     outer->sq_count++; /* Decrement when dequeued */

```

```

6047     ASSERT(outer->sq_count != 0); /* Wraparound */
6048     if (outer->sq_evhead == NULL) {
6049         /* First message. */
6050         outer->sq_evhead = outer->sq_evtail = mp;
6051         outer->sq_flags |= SQ_EVENTS;
6052         mutex_exit(SQLOCK(outer));
6053         STRSTAT(qwr_outer);
6054         (void) taskq_dispatch(streams_taskq,
6055             (task_func_t *)qwriter_outer_service, outer, TQ_SLEEP);
6056     } else {
6057         ASSERT(outer->sq_flags & SQ_EVENTS);
6058         outer->sq_evtail->b_next = mp;
6059         outer->sq_evtail = mp;
6060         mutex_exit(SQLOCK(outer));
6061     }
6062 }

6064 /*
6065  * Try and upgrade to write access at the outer perimeter. If this can
6066  * not be done without blocking then queue the callback to be done
6067  * by the qwriter_outer_thread.
6068  *
6069  * This routine can only be called from put or service procedures plus
6070  * asynchronous callback routines that have properly entered the queue (with
6071  * entersq). Thus qwriter(OUTER) assumes the caller has one claim on the syncq
6072  * associated with q.
6073  */
6074 void
6075 qwriter_outer(queue_t *q, mblk_t *mp, void (*func)())
6076 {
6077     syncq_t *osq, *sq, *outer;
6078     int failed;
6079     uint16_t flags;

6081     osq = q->q_syncq;
6082     outer = osq->sq_outer;
6083     if (outer == NULL)
6084         panic("qwriter(PERIM_OUTER): no outer perimeter");
6085     ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
6086         outer->sq_oprev != NULL);

6088     mutex_enter(SQLOCK(outer));
6089     flags = outer->sq_flags;
6090     /*
6091      * If some thread is traversing sq_next, or if we are blocked by
6092      * outer_insert or outer_remove, or if the we already have queued
6093      * callbacks, then queue this callback for later processing.
6094      *
6095      * Also queue the qwriter for an interrupt thread in order
6096      * to reduce the time spent running at high IPL.
6097      * to identify there are events.
6098      */
6099     if ((flags & SQ_GOAWAY) || (curthread->t_pri >= kpreemptpri)) {
6100         /*
6101          * Queue the become_writer request.
6102          * The queueing is atomic under SQLOCK(outer) in order
6103          * to synchronize with outer_exit.
6104          * queue_writer will drop the outer SQLOCK
6105          */
6106         if (flags & SQ_BLOCKED) {
6107             /* Must set SQ_WRITER on inner perimeter */
6108             mutex_enter(SQLOCK(osq));
6109             osq->sq_flags |= SQ_WRITER;
6110             mutex_exit(SQLOCK(osq));
6111         } else {
6112             if (!(flags & SQ_WRITER)) {

```

```

6113         /*
6114          * The outer could have been SQ_BLOCKED thus
6115          * SQ_WRITER might not be set on the inner.
6116          */
6117         mutex_enter(SQLOCK(osq));
6118         osq->sq_flags |= SQ_WRITER;
6119         mutex_exit(SQLOCK(osq));
6120     }
6121     ASSERT(osq->sq_flags & SQ_WRITER);
6122 }
6123 queue_writer(outer, func, q, mp);
6124 return;
6125 }
6126 /*
6127  * We are half-way to exclusive access to the outer perimeter.
6128  * Prevent any outer_enter, qwriter(OUTER), or outer_insert/remove
6129  * while the inner syncqs are traversed.
6130  */
6131     outer->sq_count++;
6132     ASSERT(outer->sq_count != 0); /* wraparound */
6133     flags |= SQ_WRITER;
6134     /*
6135      * Check if we can run the function immediately. Mark all
6136      * syncqs with the writer flag to prevent new entries into
6137      * put and service procedures.
6138      *
6139      * Set SQ_WRITER on all the inner syncqs while holding
6140      * the SQLOCK on the outer syncq. This ensures that the changing
6141      * of SQ_WRITER is atomic under the outer SQLOCK.
6142      */
6143     failed = 0;
6144     for (sq = outer->sq_onext; sq != outer; sq = sq->sq_onext) {
6145         uint16_t count;
6146         uint_t maxcnt = (sq == osq) ? 1 : 0;

6148         mutex_enter(SQLOCK(sq));
6149         count = sq->sq_count;
6150         SQ_PUTLOCKS_ENTER(sq);
6151         SUM_SQ_PUTCOUNTS(sq, count);
6152         if (sq->sq_count > maxcnt)
6153             failed = 1;
6154         sq->sq_flags |= SQ_WRITER;
6155         SQ_PUTLOCKS_EXIT(sq);
6156         mutex_exit(SQLOCK(sq));
6157     }
6158     if (failed) {
6159         /*
6160          * Some other thread has a read claim on the outer perimeter.
6161          * Queue the callback for deferred processing.
6162          *
6163          * queue_writer will set SQ_QUEUED before we drop SQ_WRITER
6164          * so that other qwriter(OUTER) calls will queue their
6165          * callbacks as well. queue_writer increments sq_count so we
6166          * decrement to compensate for the our increment.
6167          *
6168          * Dropping SQ_WRITER enables the writer thread to work
6169          * on this outer perimeter.
6170          */
6171         outer->sq_flags = flags;
6172         queue_writer(outer, func, q, mp);
6173         /* queue_writer dropper the lock */
6174         mutex_enter(SQLOCK(outer));
6175         ASSERT(outer->sq_count > 0);
6176         outer->sq_count--;
6177         ASSERT(outer->sq_flags & SQ_WRITER);
6178         flags = outer->sq_flags;

```

```

6179         flags &= ~SQ_WRITER;
6180         if (flags & SQ_WANTWAKEUP) {
6181             flags &= ~SQ_WANTWAKEUP;
6182             cv_broadcast(&outer->sq_wait);
6183         }
6184         outer->sq_flags = flags;
6185         mutex_exit(SQLOCK(outer));
6186         return;
6187     } else {
6188         outer->sq_flags = flags;
6189         mutex_exit(SQLOCK(outer));
6190     }

6192     /* Can run it immediately */
6193     (*func)(q, mp);

6195     outer_exit(outer);
6196 }

6198 /*
6199  * Dequeue all writer callbacks from the outer perimeter and run them.
6200  */
6201 static void
6202 write_now(syncq_t *outer)
6203 {
6204     mblk_t      *mp;
6205     queue_t     *q;
6206     void        (*func)();

6208     ASSERT(MUTEX_HELD(SQLOCK(outer)));
6209     ASSERT(outer->sq_outer == NULL && outer->sq_onext != NULL &&
6210           outer->sq_oprev != NULL);
6211     while ((mp = outer->sq_evhead) != NULL) {
6212         /*
6213          * queues cannot be placed on the queuelist on the outer
6214          * perimeter.
6215          */
6216         ASSERT(!(outer->sq_flags & SQ_MESSAGES));
6217         ASSERT((outer->sq_flags & SQ_EVENTS));

6219         outer->sq_evhead = mp->b_next;
6220         if (outer->sq_evhead == NULL) {
6221             outer->sq_evtail = NULL;
6222             outer->sq_flags &= ~SQ_EVENTS;
6223         }
6224         ASSERT(outer->sq_count != 0);
6225         outer->sq_count--; /* Incremented when enqueued. */
6226         mutex_exit(SQLOCK(outer));
6227         /*
6228          * Drop the message if the queue is closing.
6229          * Make sure that the queue is "claimed" when the callback
6230          * is run in order to satisfy various ASSERTs.
6231          */
6232         q = mp->b_queue;
6233         func = (void (*)())mp->b_prev;
6234         ASSERT(func != NULL);
6235         mp->b_next = mp->b_prev = NULL;
6236         if (q->q_flag & QWCLOSE) {
6237             freemsg(mp);
6238         } else {
6239             claimq(q);
6240             (*func)(q, mp);
6241             releaseq(q);
6242         }
6243         mutex_enter(SQLOCK(outer));
6244     }

```

```

6245         ASSERT(MUTEX_HELD(SQLOCK(outer)));
6246     }

6248 /*
6249  * The list of messages on the inner syncq is effectively hashed
6250  * by destination queue. These destination queues are doubly
6251  * linked lists (hopefully) in priority order. Messages are then
6252  * put on the queue referenced by the q_sqhead/q_sqtail elements.
6253  * Additional messages are linked together by the b_next/b_prev
6254  * elements in the mblk, with (similar to putq()) the first message
6255  * having a NULL b_prev and the last message having a NULL b_next.
6256  *
6257  * Events, such as qwriter callbacks, are put onto a list in FIFO
6258  * order referenced by sq_evhead, and sq_evtail. This is a singly
6259  * linked list, and messages here MUST be processed in the order queued.
6260  */

6262 /*
6263  * Run the events on the syncq event list (sq_evhead).
6264  * Assumes there is only one claim on the syncq, it is
6265  * already exclusive (SQ_EXCL set), and the SQLOCK held.
6266  * Messages here are processed in order, with the SQ_EXCL bit
6267  * held all the way through till the last message is processed.
6268  */
6269 void
6270 sq_run_events(syncq_t *sq)
6271 {
6272     mblk_t      *bp;
6273     queue_t     *qp;
6274     uint16_t    flags = sq->sq_flags;
6275     void        (*func)();

6277     ASSERT(MUTEX_HELD(SQLOCK(sq)));
6278     ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6279           sq->sq_oprev == NULL) ||
6280           (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6281           sq->sq_oprev != NULL));

6283     ASSERT(flags & SQ_EXCL);
6284     ASSERT(sq->sq_count == 1);

6286     /*
6287      * We need to process all of the events on this list. It
6288      * is possible that new events will be added while we are
6289      * away processing a callback, so on every loop, we start
6290      * back at the beginning of the list.
6291      */
6292     /*
6293      * We have to reaccess sq_evhead since there is a
6294      * possibility of a new entry while we were running
6295      * the callback.
6296      */
6297     for (bp = sq->sq_evhead; bp != NULL; bp = sq->sq_evhead) {
6298         ASSERT(bp->b_queue->q_syncq == sq);
6299         ASSERT(sq->sq_flags & SQ_EVENTS);

6301         qp = bp->b_queue;
6302         func = (void (*)())bp->b_prev;
6303         ASSERT(func != NULL);

6305         /*
6306          * Messages from the event queue must be taken off in
6307          * FIFO order.
6308          */
6309         ASSERT(sq->sq_evhead == bp);
6310         sq->sq_evhead = bp->b_next;

```

```

6312         if (bp->b_next == NULL) {
6313             /* Deleting last */
6314             ASSERT(sq->sq_evthead == bp);
6315             sq->sq_evthead = NULL;
6316             sq->sq_flags &= ~SQ_EVENTS;
6317         }
6318         bp->b_prev = bp->b_next = NULL;
6319         ASSERT(bp->b_datap->db_ref != 0);
6321
6322         mutex_exit(SQLOCK(sq));
6323
6324         (*func)(qp, bp);
6325
6326         mutex_enter(SQLOCK(sq));
6327         /*
6328          * re-read the flags, since they could have changed.
6329          */
6330         flags = sq->sq_flags;
6331         ASSERT(flags & SQ_EXCL);
6332     }
6333     ASSERT(sq->sq_evthead == NULL && sq->sq_evthead == NULL);
6334     ASSERT(!(sq->sq_flags & SQ_EVENTS));
6335
6336     if (flags & SQ_WANTWAKEUP) {
6337         flags &= ~SQ_WANTWAKEUP;
6338         cv_broadcast(&sq->sq_wait);
6339     }
6340     if (flags & SQ_WANTEXWAKEUP) {
6341         flags &= ~SQ_WANTEXWAKEUP;
6342         cv_broadcast(&sq->sq_exitwait);
6343     }
6344     sq->sq_flags = flags;
6345 }
6346
6347 /*
6348  * Put messages on the event list.
6349  * If we can go exclusive now, do so and process the event list, otherwise
6350  * let the last claim service this list (or wake the sqthread).
6351  * This procedure assumes SQLOCK is held. To run the event list, it
6352  * must be called with no claims.
6353  */
6354 static void
6355 sqfill_events(syncq_t *sq, queue_t *q, mblk_t *mp, void (*func)())
6356 {
6357     uint16_t count;
6358
6359     ASSERT(MUTEX_HELD(SQLOCK(sq)));
6360     ASSERT(func != NULL);
6361
6362     /*
6363      * This is a callback. Add it to the list of callbacks
6364      * and see about upgrading.
6365      */
6366     mp->b_prev = (mblk_t *)func;
6367     mp->b_queue = q;
6368     mp->b_next = NULL;
6369     if (sq->sq_evthead == NULL) {
6370         sq->sq_evthead = sq->sq_evthead = mp;
6371         sq->sq_flags |= SQ_EVENTS;
6372     } else {
6373         ASSERT(sq->sq_evthead != NULL);
6374         ASSERT(sq->sq_evthead->b_next == NULL);
6375         ASSERT(sq->sq_flags & SQ_EVENTS);
6376         sq->sq_evthead->b_next = mp;
6377         sq->sq_evthead = mp;

```

```

6377     }
6378     /*
6379      * We have set SQ_EVENTS, so threads will have to
6380      * unwind out of the perimeter, and new entries will
6381      * not grab a putlock. But we still need to know
6382      * how many threads have already made a claim to the
6383      * syncq, so grab the putlocks, and sum the counts.
6384      * If there are no claims on the syncq, we can upgrade
6385      * to exclusive, and run the event list.
6386      * NOTE: We hold the SQLOCK, so we can just grab the
6387      * putlocks.
6388      */
6389     count = sq->sq_count;
6390     SQ_PUTLOCKS_ENTER(sq);
6391     SUM_SQ_PUTCOUNTS(sq, count);
6392     /*
6393      * We have no claim, so we need to check if there
6394      * are no others, then we can upgrade.
6395      */
6396     /*
6397      * There are currently no claims on
6398      * the syncq by this thread (at least on this entry). The thread who has
6399      * the claim should drain syncq.
6400      */
6401     if (count > 0) {
6402         /*
6403          * Can't upgrade - other threads inside.
6404          */
6405         SQ_PUTLOCKS_EXIT(sq);
6406         mutex_exit(SQLOCK(sq));
6407         return;
6408     }
6409     /*
6410      * Need to set SQ_EXCL and make a claim on the syncq.
6411      */
6412     ASSERT((sq->sq_flags & SQ_EXCL) == 0);
6413     sq->sq_flags |= SQ_EXCL;
6414     ASSERT(sq->sq_count == 0);
6415     sq->sq_count++;
6416     SQ_PUTLOCKS_EXIT(sq);
6417
6418     /* Process the events list */
6419     sq_run_events(sq);
6420
6421     /*
6422      * Release our claim...
6423      */
6424     sq->sq_count--;
6425
6426     /*
6427      * And release SQ_EXCL.
6428      * We don't need to acquire the putlocks to release
6429      * SQ_EXCL, since we are exclusive, and hold the SQLOCK.
6430      */
6431     sq->sq_flags &= ~SQ_EXCL;
6432
6433     /*
6434      * sq_run_events should have released SQ_EXCL
6435      */
6436     ASSERT(!(sq->sq_flags & SQ_EXCL));
6437
6438     /*
6439      * If anything happened while we were running the
6440      * events (or was there before), we need to process
6441      * them now. We shouldn't be exclusive sine we
6442      * released the perimeter above (plus, we asserted

```

```

6443     * for it).
6444     */
6445     if (!(sq->sq_flags & SQ_STAYAWAY) && (sq->sq_flags & SQ_QUEUED))
6446         drain_syncq(sq);
6447     else
6448         mutex_exit(SQLOCK(sq));
6449 }

6451 /*
6452  * Perform delayed processing. The caller has to make sure that it is safe
6453  * to enter the syncq (e.g. by checking that none of the SQ_STAYAWAY bits are
6454  * set).
6455  *
6456  * Assume that the caller has NO claims on the syncq. However, a claim
6457  * on the syncq does not indicate that a thread is draining the syncq.
6458  * There may be more claims on the syncq than there are threads draining
6459  * (i.e. #_threads_draining <= sq_count)
6460  *
6461  * drain_syncq has to terminate when one of the SQ_STAYAWAY bits gets set
6462  * in order to preserve qwriter(OUTER) ordering constraints.
6463  *
6464  * sq_putcount only needs to be checked when dispatching the queued
6465  * writer call for CIPUT sync queue, but this is handled in sq_run_events.
6466  */
6467 void
6468 drain_syncq(syncq_t *sq)
6469 {
6470     queue_t     *qp;
6471     uint16_t    count;
6472     uint16_t    type = sq->sq_type;
6473     uint16_t    flags = sq->sq_flags;
6474     boolean_t    bg_service = sq->sq_svcflags & SQ_SERVICE;

6476     TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_START,
6477            "drain_syncq start:%p", sq);
6478     ASSERT(MUTEX_HELD(SQLOCK(sq)));
6479     ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6480            sq->sq_oprev == NULL) ||
6481            (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6482            sq->sq_oprev != NULL));

6484     /*
6485      * Drop SQ_SERVICE flag.
6486      */
6487     if (bg_service)
6488         sq->sq_svcflags &= ~SQ_SERVICE;

6490     /*
6491      * If SQ_EXCL is set, someone else is processing this syncq - let him
6492      * finish the job.
6493      */
6494     if (flags & SQ_EXCL) {
6495         if (bg_service) {
6496             ASSERT(sq->sq_servcount != 0);
6497             sq->sq_servcount--;
6498         }
6499         mutex_exit(SQLOCK(sq));
6500         return;
6501     }

6503     /*
6504      * This routine can be called by a background thread if
6505      * it was scheduled by a hi-priority thread. SO, if there are
6506      * NOT messages queued, return (remember, we have the SQLOCK,
6507      * and it cannot change until we release it). Wakeup any waiters also.
6508      */

```

```

6509     if (!(flags & SQ_QUEUED)) {
6510         if (flags & SQ_WANTWAKEUP) {
6511             flags &= ~SQ_WANTWAKEUP;
6512             cv_broadcast(&sq->sq_wait);
6513         }
6514         if (flags & SQ_WANTEXWAKEUP) {
6515             flags &= ~SQ_WANTEXWAKEUP;
6516             cv_broadcast(&sq->sq_exitwait);
6517         }
6518         sq->sq_flags = flags;
6519         if (bg_service) {
6520             ASSERT(sq->sq_servcount != 0);
6521             sq->sq_servcount--;
6522         }
6523         mutex_exit(SQLOCK(sq));
6524         return;
6525     }

6527     /*
6528      * If this is not a concurrent put perimeter, we need to
6529      * become exclusive to drain. Also, if not CIPUT, we would
6530      * not have acquired a putlock, so we don't need to check
6531      * the putcounts. If not entering with a claim, we test
6532      * for sq_count == 0.
6533      */
6534     type = sq->sq_type;
6535     if (!(type & SQ_CIPUT)) {
6536         if (sq->sq_count > 1) {
6537             if (bg_service) {
6538                 ASSERT(sq->sq_servcount != 0);
6539                 sq->sq_servcount--;
6540             }
6541             mutex_exit(SQLOCK(sq));
6542             return;
6543         }
6544         sq->sq_flags |= SQ_EXCL;
6545     }

6547     /*
6548      * This is where we make a claim to the syncq.
6549      * This can either be done by incrementing a putlock, or
6550      * the sq_count. But since we already have the SQLOCK
6551      * here, we just bump the sq_count.
6552      *
6553      * Note that after we make a claim, we need to let the code
6554      * fall through to the end of this routine to clean itself
6555      * up. A return in the while loop will put the syncq in a
6556      * very bad state.
6557      */
6558     sq->sq_count++;
6559     ASSERT(sq->sq_count != 0); /* wraparound */

6561     while ((flags = sq->sq_flags) & SQ_QUEUED) {
6562         /*
6563          * If we are told to stayaway or went exclusive,
6564          * we are done.
6565          */
6566         if (flags & (SQ_STAYAWAY)) {
6567             break;
6568         }

6570         /*
6571          * If there are events to run, do so.
6572          * We have one claim to the syncq, so if there are
6573          * more than one, other threads are running.
6574          */

```

```

6575     if (sq->sq_evhead != NULL) {
6576         ASSERT(sq->sq_flags & SQ_EVENTS);

6578         count = sq->sq_count;
6579         SQ_PUTLOCKS_ENTER(sq);
6580         SUM_SQ_PUTCOUNTS(sq, count);
6581         if (count > 1) {
6582             SQ_PUTLOCKS_EXIT(sq);
6583             /* Can't upgrade - other threads inside */
6584             break;
6585         }
6586         ASSERT((flags & SQ_EXCL) == 0);
6587         sq->sq_flags = flags | SQ_EXCL;
6588         SQ_PUTLOCKS_EXIT(sq);
6589         /*
6590          * we have the only claim, run the events,
6591          * sq_run_events will clear the SQ_EXCL flag.
6592          */
6593         sq_run_events(sq);

6595         /*
6596          * If this is a CIPUT perimeter, we need
6597          * to drop the SQ_EXCL flag so we can properly
6598          * continue draining the syncq.
6599          */
6600         if (type & SQ_CIPUT) {
6601             ASSERT(sq->sq_flags & SQ_EXCL);
6602             sq->sq_flags &= ~SQ_EXCL;
6603         }

6605         /*
6606          * And go back to the beginning just in case
6607          * anything changed while we were away.
6608          */
6609         ASSERT((sq->sq_flags & SQ_EXCL) || (type & SQ_CIPUT));
6610         continue;
6611     }

6613     ASSERT(sq->sq_evhead == NULL);
6614     ASSERT(!(sq->sq_flags & SQ_EVENTS));

6616     /*
6617     * Find the queue that is not draining.
6618     *
6619     * q_draining is protected by QLOCK which we do not hold.
6620     * But if it was set, then a thread was draining, and if it gets
6621     * cleared, then it was because the thread has successfully
6622     * drained the syncq, or a GOAWAY state occurred. For the GOAWAY
6623     * state to happen, a thread needs the SLOCK which we hold, and
6624     * if there was such a flag, we would have already seen it.
6625     */

6627     for (qp = sq->sq_head;
6628          qp != NULL && (qp->q_draining ||
6629                       (qp->q_sqflags & Q_SQDRAINING));
6630          qp = qp->q_sqnext)
6631         ;

6633     if (qp == NULL)
6634         break;

6636     /*
6637     * We have a queue to work on, and we hold the
6638     * SLOCK and one claim, call qdrain syncq.
6639     * This means we need to release the SLOCK and
6640     * acquire the QLOCK (OK since we have a claim).

```

```

6641         * Note that qdrain_syncq will actually dequeue
6642         * this queue from the sq_head list when it is
6643         * convinced all the work is done and release
6644         * the QLOCK before returning.
6645         */
6646         qp->q_sqflags |= Q_SQDRAINING;
6647         mutex_exit(SLOCK(sq));
6648         mutex_enter(QLOCK(qp));
6649         qdrain_syncq(sq, qp);
6650         mutex_enter(SLOCK(sq));

6652         /* The queue is drained */
6653         ASSERT(qp->q_sqflags & Q_SQDRAINING);
6654         qp->q_sqflags &= ~Q_SQDRAINING;
6655         /*
6656          * NOTE: After this point qp should not be used since it may be
6657          * closed.
6658          */
6659     }

6661     ASSERT(MUTEX_HELD(SLOCK(sq)));
6662     flags = sq->sq_flags;

6664     /*
6665     * sq->sq_head cannot change because we hold the
6666     * slock. However, a thread CAN decide that it is no longer
6667     * going to drain that queue. However, this should be due to
6668     * a GOAWAY state, and we should see that here.
6669     *
6670     * This loop is not very efficient. One solution may be adding a second
6671     * pointer to the "draining" queue, but it is difficult to do when
6672     * queues are inserted in the middle due to priority ordering. Another
6673     * possibility is to yank the queue out of the sq list and put it onto
6674     * the "draining list" and then put it back if it can't be drained.
6675     */

6677     ASSERT((sq->sq_head == NULL) || (flags & SQ_GOAWAY) ||
6678           (type & SQ_CI) || sq->sq_head->q_draining);

6680     /* Drop SQ_EXCL for non-CIPUT perimeters */
6681     if (!(type & SQ_CIPUT))
6682         flags &= ~SQ_EXCL;
6683     ASSERT((flags & SQ_EXCL) == 0);

6685     /* Wake up any waiters. */
6686     if (flags & SQ_WANTWAKEUP) {
6687         flags &= ~SQ_WANTWAKEUP;
6688         cv_broadcast(&sq->sq_wait);
6689     }
6690     if (flags & SQ_WANTEXWAKEUP) {
6691         flags &= ~SQ_WANTEXWAKEUP;
6692         cv_broadcast(&sq->sq_exitwait);
6693     }
6694     sq->sq_flags = flags;

6696     ASSERT(sq->sq_count != 0);
6697     /* Release our claim. */
6698     sq->sq_count--;

6700     if (bg_service) {
6701         ASSERT(sq->sq_servcount != 0);
6702         sq->sq_servcount--;
6703     }

6705     mutex_exit(SLOCK(sq));

```

```

6707     TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_END,
6708             "drain_syncq end:%p", sq);
6709 }

6712 /*
6713 *
6714 * qdrain_syncq can be called (currently) from only one of two places:
6715 *   drain_syncq
6716 *   putnext (or some variation of it).
6717 * and eventually
6718 *   qwait(_sig)
6719 *
6720 * If called from drain_syncq, we found it in the list of queues needing
6721 * service, so there is work to be done (or it wouldn't be in the list).
6722 *
6723 * If called from some putnext variation, it was because the
6724 * perimeter is open, but messages are blocking a putnext and
6725 * there is not a thread working on it. Now a thread could start
6726 * working on it while we are getting ready to do so ourself, but
6727 * the thread would set the q_draining flag, and we can spin out.
6728 *
6729 * As for qwait(_sig), I think I shall let it continue to call
6730 * drain_syncq directly (after all, it will get here eventually).
6731 *
6732 * qdrain_syncq has to terminate when:
6733 * - one of the SQ_STAYAWAY bits gets set to preserve qwriter(OUTER) ordering
6734 * - SQ_EVENTS gets set to preserve qwriter(INNER) ordering
6735 *
6736 * ASSUMES:
6737 *   One claim
6738 *   QLOCK held
6739 *   SLOCK not held
6740 *   Will release QLOCK before returning
6741 */
6742 void
6743 qdrain_syncq(syncq_t *sq, queue_t *q)
6744 {
6745     mblk_t      *bp;
6746 #ifdef DEBUG
6747     uint16_t    count;
6748 #endif

6750     TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_START,
6751             "drain_syncq start:%p", sq);
6752     ASSERT(q->q_syncq == sq);
6753     ASSERT(MUTEX_HELD(QLOCK(q)));
6754     ASSERT(MUTEX_NOT_HELD(SLOCK(sq)));
6755     /*
6756      * For non-CIPUT perimeters, we should be called with the exclusive bit
6757      * set already. For CIPUT perimeters, we will be doing a concurrent
6758      * drain, so it better not be set.
6759      */
6760     ASSERT((sq->sq_flags & (SQ_EXCL|SQ_CIPUT)));
6761     ASSERT(!((sq->sq_type & SQ_CIPUT) && (sq->sq_flags & SQ_EXCL)));
6762     ASSERT((sq->sq_type & SQ_CIPUT) || (sq->sq_flags & SQ_EXCL));
6763     /*
6764      * All outer pointers are set, or none of them are
6765      */
6766     ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6767            sq->sq_oprev == NULL) ||
6768            (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6769            sq->sq_oprev != NULL));
6770 #ifdef DEBUG
6771     count = sq->sq_count;
6772     /*

```

```

6773     * This is OK without the putlocks, because we have one
6774     * claim either from the sq_count, or a putcount. We could
6775     * get an erroneous value from other counts, but ours won't
6776     * change, so one way or another, we will have at least a
6777     * value of one.
6778     */
6779     SUM_SQ_PUTCOUNTS(sq, count);
6780     ASSERT(count >= 1);
6781 #endif /* DEBUG */

6783     /*
6784     * The first thing to do is find out if a thread is already draining
6785     * this queue. If so, we are done, just return.
6786     */
6787     if (q->q_draining) {
6788         mutex_exit(QLOCK(q));
6789         return;
6790     }

6792     /*
6793     * If the perimeter is exclusive, there is nothing we can do right now,
6794     * go away. Note that there is nothing to prevent this case from
6795     * changing right after this check, but the spin-out will catch it.
6796     */

6798     /* Tell other threads that we are draining this queue */
6799     q->q_draining = 1; /* Protected by QLOCK */

6801     /*
6802     * If there is nothing to do, clear QFULL as necessary. This caters for
6803     * the case where an empty queue was enqueued onto the syncq.
6804     */
6805     if (q->q_sqhead == NULL) {
6806         ASSERT(q->q_syncqmsgs == 0);
6807         mutex_exit(QLOCK(q));
6808         clr_qfull(q);
6809         mutex_enter(QLOCK(q));
6810     }

6812     /*
6813     * Note that q_sqhead must be re-checked here in case another message
6814     * was enqueued whilst QLOCK was dropped during the call to clr_qfull.
6815     */
6816     for (bp = q->q_sqhead; bp != NULL; bp = q->q_sqhead) {
6817         /*
6818          * Because we can enter this routine just because a putnext is
6819          * blocked, we need to spin out if the perimeter wants to go
6820          * exclusive as well as just blocked. We need to spin out also
6821          * if events are queued on the syncq.
6822          * Don't check for SQ_EXCL, because non-CIPUT perimeters would
6823          * set it, and it can't become exclusive while we hold a claim.
6824          */
6825         if (sq->sq_flags & (SQ_STAYAWAY | SQ_EVENTS)) {
6826             break;
6827         }

6829 #ifdef DEBUG
6830     /*
6831     * Since we are in qdrain_syncq, we already know the queue,
6832     * but for sanity, we want to check this against the qp that
6833     * was passed in by bp->b_queue.
6834     */

6836     ASSERT(bp->b_queue == q);
6837     ASSERT(bp->b_queue->q_syncq == sq);
6838     bp->b_queue = NULL;

```

```

6840      /*
6841      * We would have the following check in the DEBUG code:
6842      *
6843      * if (bp->b_prev != NULL) {
6844      *     ASSERT(bp->b_prev == (void (*)())q->q_info->q_i_putp);
6845      * }
6846      *
6847      * This can't be done, however, since IP modifies qinfo
6848      * structure at run-time (switching between IPv4 qinfo and IPv6
6849      * qinfo), invalidating the check.
6850      * So the assignment to func is left here, but the ASSERT itself
6851      * is removed until the whole issue is resolved.
6852      */
6853 #endif
6854     ASSERT(q->q_sqhead == bp);
6855     q->q_sqhead = bp->b_next;
6856     bp->b_prev = bp->b_next = NULL;
6857     ASSERT(q->q_syncqmsgs > 0);
6858     mutex_exit(QLOCK(q));

6860     ASSERT(bp->b_datap->db_ref != 0);

6862     (void) (*q->q_info->q_i_putp)(q, bp);

6864     mutex_enter(QLOCK(q));

6866     /*
6867     * q_syncqmsgs should only be decremented after executing the
6868     * put procedure to avoid message re-ordering. This is due to an
6869     * optimisation in putnext() which can call the put procedure
6870     * directly if it sees q_syncqmsgs == 0 (despite Q_SQQUEUED
6871     * being set).
6872     *
6873     * We also need to clear QFULL in the next service procedure
6874     * queue if this is the last message destined for that queue.
6875     *
6876     * It would make better sense to have some sort of tunable for
6877     * the low water mark, but these semantics are not yet defined.
6878     * So, alas, we use a constant.
6879     */
6880     if (--q->q_syncqmsgs == 0) {
6881         mutex_exit(QLOCK(q));
6882         clr_qfull(q);
6883         mutex_enter(QLOCK(q));
6884     }

6886     /*
6887     * Always clear SQ_EXCL when CIPUT in order to handle
6888     * qwriter(INNER). The putp() can call qwriter and get exclusive
6889     * access IFF this is the only claim. So, we need to test for
6890     * this possibility, acquire the mutex and clear the bit.
6891     */
6892     if ((sq->sq_type & SQ_CIPUT) && (sq->sq_flags & SQ_EXCL)) {
6893         mutex_enter(SQLOCK(sq));
6894         sq->sq_flags &= ~SQ_EXCL;
6895         mutex_exit(SQLOCK(sq));
6896     }
6897 }

6899 /*
6900 * We should either have no messages on this queue, or we were told to
6901 * goaway by a waiter (which we will wake up at the end of this
6902 * function).
6903 */
6904 ASSERT((q->q_sqhead == NULL) ||

```

```

6905     (sq->sq_flags & (SQ_STAYAWAY | SQ_EVENTS)));

6907     ASSERT(MUTEX_HELD(QLOCK(q)));
6908     ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));

6910     /* Remove the q from the syncq list if all the messages are drained. */
6911     if (q->q_sqhead == NULL) {
6912         ASSERT(q->q_syncqmsgs == 0);
6913         mutex_enter(SQLOCK(sq));
6914         if (q->q_sqflags & Q_SQQUEUED)
6915             SQRM_Q(sq, q);
6916         mutex_exit(SQLOCK(sq));
6917         /*
6918          * Since the queue is removed from the list, reset its priority.
6919          */
6920         q->q_spri = 0;
6921     }

6923     /*
6924     * Remember, the q_draining flag is used to let another thread know
6925     * that there is a thread currently draining the messages for a queue.
6926     * Since we are now done with this queue (even if there may be messages
6927     * still there), we need to clear this flag so some thread will work on
6928     * it if needed.
6929     */
6930     ASSERT(q->q_draining);
6931     q->q_draining = 0;

6933     /* Called with a claim, so OK to drop all locks. */
6934     mutex_exit(QLOCK(q));

6936     TRACE_1(TR_FAC_STREAMS_FR, TR_DRAIN_SYNCQ_END,
6937            "drain_syncq end:%p", sq);
6938 }
6939 /* END OF QDRAIN_SYNCQ */

6942 /*
6943 * This is the mate to qdrain_syncq, except that it is putting the message onto
6944 * the queue instead of draining. Since the message is destined for the queue
6945 * that is selected, there is no need to identify the function because the
6946 * message is intended for the put routine for the queue. For debug kernels,
6947 * this routine will do it anyway just in case.
6948 *
6949 * After the message is enqueued on the syncq, it calls putnext_tail()
6950 * which will schedule a background thread to actually process the message.
6951 *
6952 * Assumes that there is a claim on the syncq (sq->sq_count > 0) and
6953 * SQLOCK(sq) and QLOCK(q) are not held.
6954 */
6955 void
6956 qfill_syncq(syncq_t *sq, queue_t *q, mblk_t *mp)
6957 {
6958     ASSERT(MUTEX_NOT_HELD(SQLOCK(sq)));
6959     ASSERT(MUTEX_NOT_HELD(QLOCK(q)));
6960     ASSERT(sq->sq_count > 0);
6961     ASSERT(q->q_syncq == sq);
6962     ASSERT((sq->sq_outer == NULL && sq->sq_onext == NULL &&
6963            sq->sq_oprev == NULL) ||
6964            (sq->sq_outer != NULL && sq->sq_onext != NULL &&
6965            sq->sq_oprev != NULL));

6967     mutex_enter(QLOCK(q));

6969 #ifdef DEBUG
6970     /*

```

```

6971  * This is used for debug in the qfill_syncq/qdrain_syncq case
6972  * to trace the queue that the message is intended for. Note
6973  * that the original use was to identify the queue and function
6974  * to call on the drain. In the new syncq, we have the context
6975  * of the queue that we are draining, so call it's putproc and
6976  * don't rely on the saved values. But for debug this is still
6977  * useful information.
6978  */
6979  mp->b_prev = (mblk_t *)q->q_info->q_i_putp;
6980  mp->b_queue = q;
6981  mp->b_next = NULL;
6982 #endif
6983  ASSERT(q->q_syncq == sq);
6984  /*
6985   * Enqueue the message on the list.
6986   * SQPUT_MP() accesses q_syncqmsgs. We are already holding QLOCK to
6987   * protect it. So it's ok to acquire SLOCK after SQPUT_MP().
6988   */
6989  SQPUT_MP(q, mp);
6990  mutex_enter(SLOCK(sq));

6992  /*
6993   * And queue on syncq for scheduling, if not already queued.
6994   * Note that we need the SLOCK for this, and for testing flags
6995   * at the end to see if we will drain. So grab it now, and
6996   * release it before we call qdrain_syncq or return.
6997   */
6998  if (!(q->q_sqflags & Q_SQUEUEUED)) {
6999      q->q_spri = curthread->t_pri;
7000      SQPUT_Q(sq, q);
7001  }
7002 #ifndef DEBUG
7003  else {
7004      /*
7005       * All of these conditions MUST be true!
7006       */
7007      ASSERT(sq->sq_tail != NULL);
7008      if (sq->sq_tail == sq->sq_head) {
7009          ASSERT((q->q_sqprev == NULL) &&
7010              (q->q_sqnext == NULL));
7011      } else {
7012          ASSERT((q->q_sqprev != NULL) ||
7013              (q->q_sqnext != NULL));
7014      }
7015      ASSERT(sq->sq_flags & SQ_QUEUEUED);
7016      ASSERT(q->q_syncqmsgs != 0);
7017      ASSERT(q->q_sqflags & Q_SQUEUEUED);
7018  }
7019 #endif
7020  mutex_exit(QLOCK(q));
7021  /*
7022   * SLOCK is still held, so sq_count can be safely decremented.
7023   */
7024  sq->sq_count--;

7026  putnext_tail(sq, q, 0);
7027  /* Should not reference sq or q after this point. */
7028 }

7030 /* End of qfill_syncq */

7032 /*
7033  * Remove all messages from a syncq (if qp is NULL) or remove all messages
7034  * that would be put into qp by drain_syncq.
7035  * Used when deleting the syncq (qp == NULL) or when detaching
7036  * a queue (qp != NULL).

```

```

7037  * Return non-zero if one or more messages were freed.
7038  */
7039  * No need to grab sq_putlocks here. See comment in strsubr.h that explains when
7040  * sq_putlocks are used.
7041  */
7042  * NOTE: This function assumes that it is called from the close() context and
7043  * that all the queues in the syncq are going away. For this reason it doesn't
7044  * acquire QLOCK for modifying q_sqhead/q_sqtail fields. This assumption is
7045  * currently valid, but it is useful to rethink this function to behave properly
7046  * in other cases.
7047  */
7048  int
7049  flush_syncq(syncq_t *sq, queue_t *qp)
7050  {
7051      mblk_t      *bp, *mp_head, *mp_next, *mp_prev;
7052      queue_t     *q;
7053      int         ret = 0;

7055      mutex_enter(SLOCK(sq));

7057      /*
7058       * Before we leave, we need to make sure there are no
7059       * events listed for this queue. All events for this queue
7060       * will just be freed.
7061       */
7062      if (qp != NULL && sq->sq_evhead != NULL) {
7063          ASSERT(sq->sq_flags & SQ_EVENTS);

7065          mp_prev = NULL;
7066          for (bp = sq->sq_evhead; bp != NULL; bp = mp_next) {
7067              mp_next = bp->b_next;
7068              if (bp->b_queue == qp) {
7069                  /* Delete this message */
7070                  if (mp_prev != NULL) {
7071                      mp_prev->b_next = mp_next;
7072                  }
7073                  /* Update sq_evtail if the last element
7074                   * is removed.
7075                   */
7076                  if (bp == sq->sq_evtail) {
7077                      ASSERT(mp_next == NULL);
7078                      sq->sq_evtail = mp_prev;
7079                  }
7080              } else
7081                  sq->sq_evhead = mp_next;
7082              if (sq->sq_evhead == NULL)
7083                  sq->sq_flags &= ~SQ_EVENTS;
7084              bp->b_prev = bp->b_next = NULL;
7085              freemsg(bp);
7086              ret++;
7087          } else {
7088              mp_prev = bp;
7089          }
7090      }
7091  }

7093  /*
7094   * Walk sq_head and:
7095   * - match qp if qp is set, remove it's messages
7096   * - all if qp is not set
7097   */
7098  q = sq->sq_head;
7099  while (q != NULL) {
7100      ASSERT(q->q_syncq == sq);
7101      if ((qp == NULL) || (qp == q)) {
7102          /*

```

```

7103     * Yank the messages as a list off the queue
7104     */
7105     mp_head = q->q_sqhead;
7106     /*
7107     * We do not have QLOCK(q) here (which is safe due to
7108     * assumptions mentioned above). To obtain the lock we
7109     * need to release SLOCK which may allow lots of things
7110     * to change upon us. This place requires more analysis.
7111     */
7112     q->q_sqhead = q->q_sqtail = NULL;
7113     ASSERT(mp_head->b_queue &&
7114            mp_head->b_queue->q_syncq == sq);

7116     /*
7117     * Free each of the messages.
7118     */
7119     for (bp = mp_head; bp != NULL; bp = mp_next) {
7120         mp_next = bp->b_next;
7121         bp->b_prev = bp->b_next = NULL;
7122         freemsg(bp);
7123         ret++;
7124     }
7125     /*
7126     * Now remove the queue from the syncq.
7127     */
7128     ASSERT(q->q_sqflags & Q_SQUEUEUED);
7129     SQRM_Q(sq, q);
7130     q->q_spri = 0;
7131     q->q_syncqmsgs = 0;

7133     /*
7134     * If qp was specified, we are done with it and are
7135     * going to drop SLOCK(sq) and return. We wakeup syncq
7136     * waiters while we still have the SLOCK.
7137     */
7138     if ((qp != NULL) && (sq->sq_flags & SQ_WANTWAKEUP)) {
7139         sq->sq_flags &= ~SQ_WANTWAKEUP;
7140         cv_broadcast(&sq->sq_wait);
7141     }
7142     /* Drop SLOCK across clr_qfull */
7143     mutex_exit(SLOCK(sq));

7145     /*
7146     * We avoid doing the test that drain_syncq does and
7147     * unconditionally clear qfull for every flushed
7148     * message. Since flush_syncq is only called during
7149     * close this should not be a problem.
7150     */
7151     clr_qfull(q);
7152     if (qp != NULL) {
7153         return (ret);
7154     } else {
7155         mutex_enter(SLOCK(sq));
7156         /*
7157         * The head was removed by SQRM_Q above.
7158         * reread the new head and flush it.
7159         */
7160         q = sq->sq_head;
7161     }
7162     } else {
7163         q = q->q_sqnext;
7164     }
7165     ASSERT(MUTEX_HELD(SLOCK(sq)));
7166 }

7168 if (sq->sq_flags & SQ_WANTWAKEUP) {

```

```

7169         sq->sq_flags &= ~SQ_WANTWAKEUP;
7170         cv_broadcast(&sq->sq_wait);
7171     }

7173     mutex_exit(SLOCK(sq));
7174     return (ret);
7175 }

7177 /*
7178 * Propagate all messages from a syncq to the next syncq that are associated
7179 * with the specified queue. If the queue is attached to a driver or if the
7180 * messages have been added due to a qwriter(PERIM_INNER), free the messages.
7181 *
7182 * Assumes that the stream is strlock()'ed. We don't come here if there
7183 * are no messages to propagate.
7184 *
7185 * NOTE : If the queue is attached to a driver, all the messages are freed
7186 * as there is no point in propagating the messages from the driver syncq
7187 * to the closing stream head which will in turn get freed later.
7188 */
7189 static int
7190 propagate_syncq(queue_t *qp)
7191 {
7192     mblk_t      *bp, *head, *tail, *prev, *next;
7193     syncq_t     *sq;
7194     queue_t     *nqp;
7195     syncq_t     *nsq;
7196     boolean_t   isdriver;
7197     int         moved = 0;
7198     uint16_t    flags;
7199     pri_t       priority = curthread->t_pri;
7200 #ifdef DEBUG
7201     void        (*func)();
7202 #endif

7204     sq = qp->q_syncq;
7205     ASSERT(MUTEX_HELD(SLOCK(sq)));
7206     /* debug macro */
7207     SQ_PUTLOCKS_HELD(sq);
7208     /*
7209     * As entersq() does not increment the sq_count for
7210     * the write side, check sq_count for non-QPERQ
7211     * perimeters alone.
7212     */
7213     ASSERT((qp->q_flag & QPERQ) || (sq->sq_count >= 1));

7215     /*
7216     * propagate_syncq() can be called because of either messages on the
7217     * queue syncq or because on events on the queue syncq. Do actual
7218     * message propagations if there are any messages.
7219     */
7220     if (qp->q_syncqmsgs) {
7221         isdriver = (qp->q_flag & QISDRV);

7223         if (!isdriver) {
7224             nqp = qp->q_next;
7225             nsq = nqp->q_syncq;
7226             ASSERT(MUTEX_HELD(SLOCK(nsq)));
7227             /* debug macro */
7228             SQ_PUTLOCKS_HELD(nsq);
7229 #ifdef DEBUG
7230             func = (void (*)())nqp->q_info->qi_putp;
7231 #endif
7232         }

7234         SQRM_Q(sq, qp);

```

```

7235     priority = MAX(qp->q_spri, priority);
7236     qp->q_spri = 0;
7237     head = qp->q_sqhead;
7238     tail = qp->q_sqtail;
7239     qp->q_sqhead = qp->q_sqtail = NULL;
7240     qp->q_syncqmsgs = 0;

7242     /*
7243     * Walk the list of messages, and free them if this is a driver,
7244     * otherwise reset the b_prev and b_queue value to the new putp.
7245     * Afterward, we will just add the head to the end of the next
7246     * syncq, and point the tail to the end of this one.
7247     */

7249     for (bp = head; bp != NULL; bp = next) {
7250         next = bp->b_next;
7251         if (isdriver) {
7252             bp->b_prev = bp->b_next = NULL;
7253             freemsg(bp);
7254             continue;
7255         }
7256         /* Change the q values for this message */
7257         bp->b_queue = nqp;
7258 #ifdef DEBUG
7259         bp->b_prev = (mblk_t *)func;
7260 #endif
7261         moved++;
7262     }
7263     /*
7264     * Attach list of messages to the end of the new queue (if there
7265     * is a list of messages).
7266     */

7268     if (!isdriver && head != NULL) {
7269         ASSERT(tail != NULL);
7270         if (nqp->q_sqhead == NULL) {
7271             nqp->q_sqhead = head;
7272         } else {
7273             ASSERT(nqp->q_sqtail != NULL);
7274             nqp->q_sqtail->b_next = head;
7275         }
7276         nqp->q_sqtail = tail;
7277     }
7278     /*
7279     * When messages are moved from high priority queue to
7280     * another queue, the destination queue priority is
7281     * upgraded.
7282     */

7283     if (priority > nqp->q_spri)
7284         nqp->q_spri = priority;

7286     SQPUT_Q(nsq, nqp);

7288     nqp->q_syncqmsgs += moved;
7289     ASSERT(nqp->q_syncqmsgs != 0);
7290 }

7293 /*
7294 * Before we leave, we need to make sure there are no
7295 * events listed for this queue. All events for this queue
7296 * will just be freed.
7297 */
7298 if (sq->sq_evhead != NULL) {
7299     ASSERT(sq->sq_flags & SQ_EVENTS);
7300     prev = NULL;

```

```

7301     for (bp = sq->sq_evhead; bp != NULL; bp = next) {
7302         next = bp->b_next;
7303         if (bp->b_queue == qp) {
7304             /* Delete this message */
7305             if (prev != NULL) {
7306                 prev->b_next = next;
7307             }
7308             /*
7309             * Update sq_evtail if the last element
7310             * is removed.
7311             */
7312             if (bp == sq->sq_evtail) {
7313                 ASSERT(next == NULL);
7314                 sq->sq_evtail = prev;
7315             }
7316             } else
7317                 sq->sq_evhead = next;
7318             if (sq->sq_evhead == NULL)
7319                 sq->sq_flags &= ~SQ_EVENTS;
7320             bp->b_prev = bp->b_next = NULL;
7321             freemsg(bp);
7322         } else {
7323             prev = bp;
7324         }
7325     }

7327     flags = sq->sq_flags;

7329     /* Wake up any waiter before leaving. */
7330     if (flags & SQ_WANTWAKEUP) {
7331         flags &= ~SQ_WANTWAKEUP;
7332         cv_broadcast(&sq->sq_wait);
7333     }
7334     sq->sq_flags = flags;

7336     return (moved);
7337 }

7339 /*
7340 * Try and upgrade to exclusive access at the inner perimeter. If this can
7341 * not be done without blocking then request will be queued on the syncq
7342 * and drain_syncq will run it later.
7343 */
7344 * This routine can only be called from put or service procedures plus
7345 * asynchronous callback routines that have properly entered the queue (with
7346 * entersq). Thus qwriter_inner assumes the caller has one claim on the syncq
7347 * associated with q.
7348 */
7349 void
7350 qwriter_inner(queue_t *q, mblk_t *mp, void (*func)())
7351 {
7352     syncq_t *sq = q->q_syncq;
7353     uint16_t count;

7355     mutex_enter(SQLOCK(sq));
7356     count = sq->sq_count;
7357     SQ_PUTLOCKS_ENTER(sq);
7358     SUM_SQ_PUTCOUNTS(sq, count);
7359     ASSERT(count >= 1);
7360     ASSERT(sq->sq_type & (SQ_CIPUT|SQ_CISVC));

7362     if (count == 1) {
7363         /*
7364         * Can upgrade. This case also handles nested qwriter calls
7365         * (when the qwriter callback function calls qwriter). In that
7366         * case SQ_EXCL is already set.

```

```

7367     */
7368     sq->sq_flags |= SQ_EXCL;
7369     SQ_PUTLOCKS_EXIT(sq);
7370     mutex_exit(SQLOCK(sq));
7371     (*func)(q, mp);
7372     /*
7373     * Assumes that leavesq, putnext, and drain_syncq will reset
7374     * SQ_EXCL for SQ_CIPUT/SQ_CISVC queues. We leave SQ_EXCL on
7375     * until putnext, leavesq, or drain_syncq drops it.
7376     * That way we handle nested qwriter(INNER) without dropping
7377     * SQ_EXCL until the outermost qwriter callback routine is
7378     * done.
7379     */
7380     return;
7381 }
7382 SQ_PUTLOCKS_EXIT(sq);
7383 sqfill_events(sq, q, mp, func);
7384 }

7386 /*
7387  * Synchronous callback support functions
7388  */

7390 /*
7391  * Allocate a callback parameter structure.
7392  * Assumes that caller initializes the flags and the id.
7393  * Acquires SQLOCK(sq) if non-NULL is returned.
7394  */
7395 callbparams_t *
7396 callbparams_alloc(syncq_t *sq, void (*func)(void *), void *arg, int kmflags)
7397 {
7398     callbparams_t *cbp;
7399     size_t size = sizeof (callbparams_t);

7401     cbp = kmem_alloc(size, kmflags & ~KM_PANIC);

7403     /*
7404     * Only try tryhard allocation if the caller is ready to panic.
7405     * Otherwise just fail.
7406     */
7407     if (cbp == NULL) {
7408         if (kmflags & KM_PANIC)
7409             cbp = kmem_alloc_tryhard(sizeof (callbparams_t),
7410                                     &size, kmflags);
7411         else
7412             return (NULL);
7413     }

7415     ASSERT(size >= sizeof (callbparams_t));
7416     cbp->cbp_size = size;
7417     cbp->cbp_sq = sq;
7418     cbp->cbp_func = func;
7419     cbp->cbp_arg = arg;
7420     mutex_enter(SQLOCK(sq));
7421     cbp->cbp_next = sq->sq_callbpend;
7422     sq->sq_callbpend = cbp;
7423     return (cbp);
7424 }

7426 void
7427 callbparams_free(syncq_t *sq, callbparams_t *cbp)
7428 {
7429     callbparams_t **pp, *p;

7431     ASSERT(MUTEX_HELD(SQLOCK(sq)));

```

```

7433     for (pp = &sq->sq_callbpend; (p = *pp) != NULL; pp = &p->cbp_next) {
7434         if (p == cbp) {
7435             *pp = p->cbp_next;
7436             kmem_free(p, p->cbp_size);
7437             return;
7438         }
7439     }
7440     (void) (STRLOG(0, 0, 0, SL_CONSOLE,
7441                 "callbparams_free: not found\n"));
7442 }

7444 void
7445 callbparams_free_id(syncq_t *sq, callbparams_id_t id, int32_t flag)
7446 {
7447     callbparams_t **pp, *p;

7449     ASSERT(MUTEX_HELD(SQLOCK(sq)));

7451     for (pp = &sq->sq_callbpend; (p = *pp) != NULL; pp = &p->cbp_next) {
7452         if (p->cbp_id == id && p->cbp_flags == flag) {
7453             *pp = p->cbp_next;
7454             kmem_free(p, p->cbp_size);
7455             return;
7456         }
7457     }
7458     (void) (STRLOG(0, 0, 0, SL_CONSOLE,
7459                 "callbparams_free_id: not found\n"));
7460 }

7462 /*
7463  * Callback wrapper function used by once-only callbacks that can be
7464  * cancelled (qtimeout and qbufcall)
7465  * Contains inline version of entersq(sq, SQ_CALLBACK) that can be
7466  * cancelled by the qun* functions.
7467  */
7468 void
7469 qcallbackwrapper(void *arg)
7470 {
7471     callbparams_t *cbp = arg;
7472     syncq_t *sq;
7473     uint16_t count = 0;
7474     uint16_t waitflags = SQ_STAYAWAY | SQ_EVENTS | SQ_EXCL;
7475     uint16_t type;

7477     sq = cbp->cbp_sq;
7478     mutex_enter(SQLOCK(sq));
7479     type = sq->sq_type;
7480     if (!(type & SQ_CICB)) {
7481         count = sq->sq_count;
7482         SQ_PUTLOCKS_ENTER(sq);
7483         SQ_PUTCOUNT_CLRFAST_LOCKED(sq);
7484         SUM_SQ_PUTCOUNTS(sq, count);
7485         sq->sq_needexcl++;
7486         ASSERT(sq->sq_needexcl != 0); /* wraparound */
7487         waitflags |= SQ_MESSAGES;
7488     }
7489     /* Can not handle exclusive entry at outer perimeter */
7490     ASSERT(type & SQ_COCB);

7492     while ((sq->sq_flags & waitflags) || (!(type & SQ_CICB) && count != 0)) {
7493         if ((sq->sq_callbflags & cbp->cbp_flags) &&
7494             (sq->sq_cancelid == cbp->cbp_id)) {
7495             /* timeout has been cancelled */
7496             sq->sq_callbflags |= SQ_CALLB_BYPASSED;
7497             callbparams_free(sq, cbp);
7498             if (!(type & SQ_CICB)) {

```

```

7499     ASSERT(sq->sq_needexcl > 0);
7500     sq->sq_needexcl--;
7501     if (sq->sq_needexcl == 0) {
7502         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
7503     }
7504     SQ_PUTLOCKS_EXIT(sq);
7505     }
7506     mutex_exit(SQLOCK(sq));
7507     return;
7508 }
7509 sq->sq_flags |= SQ_WANTWAKEUP;
7510 if (!(type & SQ_CICB)) {
7511     SQ_PUTLOCKS_EXIT(sq);
7512 }
7513 cv_wait(&sq->sq_wait, SQLOCK(sq));
7514 if (!(type & SQ_CICB)) {
7515     count = sq->sq_count;
7516     SQ_PUTLOCKS_ENTER(sq);
7517     SUM_SQ_PUTCOUNTS(sq, count);
7518 }
7519 }

7521 sq->sq_count++;
7522 ASSERT(sq->sq_count != 0);      /* Wraparound */
7523 if (!(type & SQ_CICB)) {
7524     ASSERT(count == 0);
7525     sq->sq_flags |= SQ_EXCL;
7526     ASSERT(sq->sq_needexcl > 0);
7527     sq->sq_needexcl--;
7528     if (sq->sq_needexcl == 0) {
7529         SQ_PUTCOUNT_SETFAST_LOCKED(sq);
7530     }
7531     SQ_PUTLOCKS_EXIT(sq);
7532 }

7534 mutex_exit(SQLOCK(sq));

7536 cbp->cbp_func(cbp->cbp_arg);

7538 /*
7539  * We drop the lock only for leavesq to re-acquire it.
7540  * Possible optimization is inline of leavesq.
7541  */
7542 mutex_enter(SQLOCK(sq));
7543 callbparams_free(sq, cbp);
7544 mutex_exit(SQLOCK(sq));
7545 leavesq(sq, SQ_CALLBACK);
7546 }

7548 /*
7549  * No need to grab sq_putlocks here. See comment in strsubr.h that
7550  * explains when sq_putlocks are used.
7551  *
7552  * sq_count (or one of the sq_putcounts) has already been
7553  * decremented by the caller, and if SQ_QUEUED, we need to call
7554  * drain_syncq (the global syncq drain).
7555  * If putnext_tail is called with the SQ_EXCL bit set, we are in
7556  * one of two states, non-CIPUT perimeter, and we need to clear
7557  * it, or we went exclusive in the put procedure. In any case,
7558  * we want to clear the bit now, and it is probably easier to do
7559  * this at the beginning of this function (remember, we hold
7560  * the SQLOCK). Lastly, if there are other messages queued
7561  * on the syncq (and not for our destination), enable the syncq
7562  * for background work.
7563  */

```

```

7565 /* ARGSUSED */
7566 void
7567 putnext_tail(syncq_t *sq, queue_t *qp, uint32_t passflags)
7568 {
7569     uint16_t     flags = sq->sq_flags;

7571     ASSERT(MUTEX_HELD(SQLOCK(sq)));
7572     ASSERT(MUTEX_NOT_HELD(QLOCK(qp)));

7574     /* Clear SQ_EXCL if set in passflags */
7575     if (passflags & SQ_EXCL) {
7576         flags &= ~SQ_EXCL;
7577     }
7578     if (flags & SQ_WANTWAKEUP) {
7579         flags &= ~SQ_WANTWAKEUP;
7580         cv_broadcast(&sq->sq_wait);
7581     }
7582     if (flags & SQ_WANTEXWAKEUP) {
7583         flags &= ~SQ_WANTEXWAKEUP;
7584         cv_broadcast(&sq->sq_exitwait);
7585     }
7586     sq->sq_flags = flags;

7588     /*
7589     * We have cleared SQ_EXCL if we were asked to, and started
7590     * the wakeup process for waiters. If there are no writers
7591     * then we need to drain the syncq if we were told to, or
7592     * enable the background thread to do it.
7593     */
7594     if (!(flags & (SQ_STAYAWAY|SQ_EXCL))) {
7595         if ((passflags & SQ_QUEUED) ||
7596             (sq->sq_svcflags & SQ_DISABLED)) {
7597             /* drain_syncq will take care of events in the list */
7598             drain_syncq(sq);
7599             return;
7600         } else if (flags & SQ_QUEUED) {
7601             sqenable(sq);
7602         }
7603     }

7604     /* Drop the SQLOCK on exit */
7605     mutex_exit(SQLOCK(sq));
7606     TRACE_3(TR_FAC_STREAMS_FR, TR_PUTNEXT_END,
7607            "putnext_end:(%p, %p, %p) done", NULL, qp, sq);
7608 }

7610 void
7611 set_qend(queue_t *q)
7612 {
7613     mutex_enter(QLOCK(q));
7614     if (!O_SAMESTR(q))
7615         q->q_flag |= QEND;
7616     else
7617         q->q_flag &= ~QEND;
7618     mutex_exit(QLOCK(q));
7619     q = _OTHERQ(q);
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 }

7628 /*
7629  * Set QFULL in next service procedure queue (that cares) if not already
7630  * set and if there are already more messages on the syncq than

```

```

7631 * sq_max_size. If sq_max_size is 0, no flow control will be asserted on
7632 * any syncq.
7633 *
7634 * The fq here is the next queue with a service procedure. This is where
7635 * we would fail canputnext, so this is where we need to set QFULL.
7636 * In the case when fq != q we need to take QLOCK(fq) to set QFULL flag.
7637 *
7638 * We already have QLOCK at this point. To avoid cross-locks with
7639 * freezestr() which grabs all QLOCKS and with strlock() which grabs both
7640 * SLOCK and sd_reflock, we need to drop respective locks first.
7641 */
7642 void
7643 set_qfull(queue_t *q)
7644 {
7645     queue_t      *fq = NULL;
7646
7647     ASSERT(MUTEX_HELD(QLOCK(q)));
7648     if ((sq_max_size != 0) && (!(q->q_nfsrv->q_flag & QFULL)) &&
7649         (q->q_syncmsgs > sq_max_size)) {
7650         if ((fq = q->q_nfsrv) == q) {
7651             fq->q_flag |= QFULL;
7652         } else {
7653             mutex_exit(QLOCK(q));
7654             mutex_enter(QLOCK(fq));
7655             fq->q_flag |= QFULL;
7656             mutex_exit(QLOCK(fq));
7657             mutex_enter(QLOCK(q));
7658         }
7659     }
7660 }
7661
7662 void
7663 clr_qfull(queue_t *q)
7664 {
7665     queue_t *oq = q;
7666
7667     q = q->q_nfsrv;
7668     /* Fast check if there is any work to do before getting the lock. */
7669     if ((q->q_flag & (QFULL|QWANTW)) == 0) {
7670         return;
7671     }
7672
7673     /*
7674      * Do not reset QFULL (and backenable) if the q_count is the reason
7675      * for QFULL being set.
7676      */
7677     mutex_enter(QLOCK(q));
7678     /*
7679      * If queue is empty i.e q_mblkcnt is zero, queue can not be full.
7680      * Hence clear the QFULL.
7681      * If both q_count and q_mblkcnt are less than the hiwat mark,
7682      * clear the QFULL.
7683      */
7684     if (q->q_mblkcnt == 0 || ((q->q_count < q->q_hiwat) &&
7685         (q->q_mblkcnt < q->q_hiwat))) {
7686         q->q_flag &= ~QFULL;
7687         /*
7688          * A little more confusing, how about this way:
7689          * if someone wants to write,
7690          * AND
7691          * both counts are less than the lowat mark
7692          * OR
7693          * the lowat mark is zero
7694          * THEN
7695          * backenable
7696          */

```

```

7697         if ((q->q_flag & QWANTW) &&
7698             ((q->q_count < q->q_lowat) &&
7699              (q->q_mblkcnt < q->q_lowat)) || q->q_lowat == 0) {
7700             q->q_flag &= ~QWANTW;
7701             mutex_exit(QLOCK(q));
7702             backenable(oq, 0);
7703         } else
7704             mutex_exit(QLOCK(q));
7705     } else
7706         mutex_exit(QLOCK(q));
7707 }
7708
7709 /*
7710  * Set the forward service procedure pointer.
7711  *
7712  * Called at insert-time to cache a queue's next forward service procedure in
7713  * q_nfsrv; used by canput() and canputnext(). If the queue to be inserted
7714  * has a service procedure then q_nfsrv points to itself. If the queue to be
7715  * inserted does not have a service procedure, then q_nfsrv points to the next
7716  * queue forward that has a service procedure. If the queue is at the logical
7717  * end of the stream (driver for write side, stream head for the read side)
7718  * and does not have a service procedure, then q_nfsrv also points to itself.
7719  */
7720 void
7721 set_nfsrv_ptr(
7722     queue_t *rnew,      /* read queue pointer to new module */
7723     queue_t *wnew,      /* write queue pointer to new module */
7724     queue_t *prev_rq,   /* read queue pointer to the module above */
7725     queue_t *prev_wq)  /* write queue pointer to the module above */
7726 {
7727     queue_t *qp;
7728
7729     if (prev_wq->q_next == NULL) {
7730         /*
7731          * Insert the driver, initialize the driver and stream head.
7732          * In this case, prev_rq/prev_wq should be the stream head.
7733          * _I_INSERT does not allow inserting a driver. Make sure
7734          * that it is not an insertion.
7735          */
7736         ASSERT(!(rnew->q_flag & _QINSERTING));
7737         wnew->q_nfsrv = wnew;
7738         if (rnew->q_qinfo->q_i_srvp)
7739             rnew->q_nfsrv = rnew;
7740         else
7741             rnew->q_nfsrv = prev_rq;
7742         prev_rq->q_nfsrv = prev_rq;
7743         prev_wq->q_nfsrv = prev_wq;
7744     } else {
7745         /*
7746          * set up read side q_nfsrv pointer. This MUST be done
7747          * before setting the write side, because the setting of
7748          * the write side for a fifo may depend on it.
7749          *
7750          * Suppose we have a fifo that only has pipemod pushed.
7751          * pipemod has no read or write service procedures, so
7752          * nfsrv for both pipemod queues points to prev_rq (the
7753          * stream read head). Now push bufmod (which has only a
7754          * read service procedure). Doing the write side first,
7755          * wnew->q_nfsrv is set to pipemod's writeq nfsrv, which
7756          * is WRONG; the next queue forward from wnew with a
7757          * service procedure will be rnew, not the stream read head.
7758          * Since the downstream queue (which in the case of a fifo
7759          * is the read queue rnew) can affect upstream queues, it
7760          * needs to be done first. Setting up the read side first
7761          * sets nfsrv for both pipemod queues to rnew and then
7762          * when the write side is set up, wnew-q_nfsrv will also

```

```

7763     * point to rnew.
7764     */
7765     if (rnew->q_qinfo->q_i_srvp) {
7766         /*
7767          * use _OTHERQ() because, if this is a pipe, next
7768          * module may have been pushed from other end and
7769          * q_next could be a read queue.
7770          */
7771         qp = _OTHERQ(prev_wq->q_next);
7772         while (qp && qp->q_nfsrv != qp) {
7773             qp->q_nfsrv = rnew;
7774             qp = backq(qp);
7775         }
7776         rnew->q_nfsrv = rnew;
7777     } else
7778         rnew->q_nfsrv = prev_rq->q_nfsrv;

7780     /* set up write side q_nfsrv pointer */
7781     if (wnew->q_qinfo->q_i_srvp) {
7782         wnew->q_nfsrv = wnew;

7784         /*
7785          * For insertion, need to update nfsrv of the modules
7786          * above which do not have a service routine.
7787          */
7788         if (rnew->q_flag & _QINSERTING) {
7789             for (qp = prev_wq;
7790                  qp != NULL && qp->q_nfsrv != qp;
7791                  qp = backq(qp)) {
7792                 qp->q_nfsrv = wnew->q_nfsrv;
7793             }
7794         }
7795     } else {
7796         if (prev_wq->q_next == prev_rq)
7797             /*
7798              * Since prev_wq/prev_rq are the middle of a
7799              * fifo, wnew/rnew will also be the middle of
7800              * a fifo and wnew's nfsrv is same as rnew's.
7801              */
7802             wnew->q_nfsrv = rnew->q_nfsrv;
7803         else
7804             wnew->q_nfsrv = prev_wq->q_next->q_nfsrv;
7805     }
7806 }
7807 }

7809 /*
7810  * Reset the forward service procedure pointer; called at remove-time.
7811  */
7812 void
7813 reset_nfsrv_ptr(queue_t *rq, queue_t *wqp)
7814 {
7815     queue_t *tmp_qp;

7817     /* Reset the write side q_nfsrv pointer for _I_REMOVE */
7818     if ((rq->q_flag & _QREMOVING) && (wqp->q_qinfo->q_i_srvp != NULL)) {
7819         for (tmp_qp = backq(wqp);
7820              tmp_qp != NULL && tmp_qp->q_nfsrv == wqp;
7821              tmp_qp = backq(tmp_qp)) {
7822             tmp_qp->q_nfsrv = wqp->q_nfsrv;
7823         }
7824     }

7826     /* reset the read side q_nfsrv pointer */
7827     if (rq->q_qinfo->q_i_srvp) {
7828         if (wqp->q_next) { /* non-driver case */

```

```

7829         tmp_qp = _OTHERQ(wqp->q_next);
7830         while (tmp_qp && tmp_qp->q_nfsrv == rq) {
7831             /* Note that rq->q_next cannot be NULL */
7832             ASSERT(rq->q_next != NULL);
7833             tmp_qp->q_nfsrv = rq->q_next->q_nfsrv;
7834             tmp_qp = backq(tmp_qp);
7835         }
7836     }
7837 }
7838 }

7840 /*
7841  * This routine should be called after all stream geometry changes to update
7842  * the stream head cached struio() rd/wr queue pointers. Note must be called
7843  * with the streamlock(jed).
7844  *
7845  * Note: only enables Synchronous STREAMS for a side of a Stream which has
7846  * an explicit synchronous barrier module queue. That is, a queue that
7847  * has specified a struio() type.
7848  */
7849 static void
7850 strsetuio(stdata_t *stp)
7851 {
7852     queue_t *wrq;

7854     if (stp->sd_flag & STPLEX) {
7855         /*
7856          * Not streamhead, but a mux, so no Synchronous STREAMS.
7857          */
7858         stp->sd_struiowrq = NULL;
7859         stp->sd_struiordq = NULL;
7860         return;
7861     }
7862     /*
7863      * Scan the write queue(s) while synchronous
7864      * until we find a qinfo uio type specified.
7865      */
7866     wrq = stp->sd_wrq->q_next;
7867     while (wrq) {
7868         if (wrq->q_struiot == STRUIOT_NONE) {
7869             wrq = 0;
7870             break;
7871         }
7872         if (wrq->q_struiot != STRUIOT_DONTCARE)
7873             break;
7874         if (!_SAMESTR(wrq)) {
7875             wrq = 0;
7876             break;
7877         }
7878         wrq = wrq->q_next;
7879     }
7880     stp->sd_struiowrq = wrq;
7881     /*
7882      * Scan the read queue(s) while synchronous
7883      * until we find a qinfo uio type specified.
7884      */
7885     wrq = stp->sd_wrq->q_next;
7886     while (wrq) {
7887         if (_RD(wrq)->q_struiot == STRUIOT_NONE) {
7888             wrq = 0;
7889             break;
7890         }
7891         if (_RD(wrq)->q_struiot != STRUIOT_DONTCARE)
7892             break;
7893         if (!_SAMESTR(wrq)) {
7894             wrq = 0;

```

```

7895         break;
7896     }
7897     wrq = wrq->q_next;
7898 }
7899 stp->sd_struicordq = wrq ? _RD(wrq) : 0;
7900 }

7902 /*
7903  * pass_wput, unblocks the passthru queues, so that
7904  * messages can arrive at muxs lower read queue, before
7905  * I_LINK/I_UNLINK is acked/nacked.
7906  */
7907 static void
7908 pass_wput(queue_t *q, mblk_t *mp)
7909 {
7910     syncq_t *sq;

7912     sq = _RD(q)->q_syncq;
7913     if (sq->sq_flags & SQ_BLOCKED)
7914         unblocksq(sq, SQ_BLOCKED, 0);
7915     putnext(q, mp);
7916 }

7918 /*
7919  * Set up queues for the link/unlink.
7920  * Create a new queue and block it and then insert it
7921  * below the stream head on the lower stream.
7922  * This prevents any messages from arriving during the setq
7923  * as well as while the mux is processing the LINK/I_UNLINK.
7924  * The blocked passq is unblocked once the LINK/I_UNLINK has
7925  * been acked or nacked or if a message is generated and sent
7926  * down muxs write put procedure.
7927  * See pass_wput().
7928  *
7929  * After the new queue is inserted, all messages coming from below are
7930  * blocked. The call to strlock will ensure that all activity in the stream head
7931  * read queue syncq is stopped (sq_count drops to zero).
7932  */
7933 static queue_t *
7934 link_addpassthru(stdata_t *stpdwn)
7935 {
7936     queue_t *passq;
7937     sqliist_t sqliist;

7939     passq = allocq();
7940     STREAM(passq) = STREAM(_WR(passq)) = stpdwn;
7941     /* setq might sleep in allocator - avoid holding locks. */
7942     setq(passq, &passthru_rinit, &passthru_winit, NULL, QPERQ,
7943           SQ_CI|SQ_CO, B_FALSE);
7944     claimq(passq);
7945     blocksq(passq->q_syncq, SQ_BLOCKED, 1);
7946     insertq(STREAM(passq), passq);

7948     /*
7949     * Use strlock() to wait for the stream head sq_count to drop to zero
7950     * since we are going to change q_ptr in the stream head. Note that
7951     * insertq() doesn't wait for any syncq counts to drop to zero.
7952     */
7953     sqliist.sqliist_head = NULL;
7954     sqliist.sqliist_index = 0;
7955     sqliist.sqliist_size = sizeof (sqliist_t);
7956     sqliist.insert(&sqliist, _RD(stpdwn->sd_wrq)->q_syncq);
7957     strlock(stpdwn, &sqliist);
7958     strunlock(stpdwn, &sqliist);

7960     releaseq(passq);

```

```

7961         return (passq);
7962     }

7964 /*
7965  * Let messages flow up into the mux by removing
7966  * the passq.
7967  */
7968 static void
7969 link_rempassthru(queue_t *passq)
7970 {
7971     claimq(passq);
7972     removeq(passq);
7973     releaseq(passq);
7974     freeq(passq);
7975 }

7977 /*
7978  * Wait for the condition variable pointed to by 'cvp' to be signaled,
7979  * or for 'tim' milliseconds to elapse, whichever comes first. If 'tim'
7980  * is negative, then there is no time limit. If 'nosigs' is non-zero,
7981  * then the wait will be non-interruptible.
7982  *
7983  * Returns >0 if signaled, 0 if interrupted, or -1 upon timeout.
7984  */
7985 clock_t
7986 str_cv_wait(kcondvar_t *cvp, kmutex_t *mp, clock_t tim, int nosigs)
7987 {
7988     clock_t ret;

7990     if (tim < 0) {
7991         if (nosigs) {
7992             cv_wait(cvp, mp);
7993             ret = 1;
7994         } else {
7995             ret = cv_wait_sig(cvp, mp);
7996         }
7997     } else if (tim > 0) {
7998         /*
7999         * convert milliseconds to clock ticks
8000         */
8001         if (nosigs) {
8002             ret = cv_reltimedwait(cvp, mp,
8003                                   MSEC_TO_TICK_ROUNDUP(tim), TR_CLOCK_TICK);
8004         } else {
8005             ret = cv_reltimedwait_sig(cvp, mp,
8006                                       MSEC_TO_TICK_ROUNDUP(tim), TR_CLOCK_TICK);
8007         }
8008     } else {
8009         ret = -1;
8010     }
8011     return (ret);
8012 }

8014 /*
8015  * Wait until the stream head can determine if it is at the mark but
8016  * don't wait forever to prevent a race condition between the "mark" state
8017  * in the stream head and any mark state in the caller/user of this routine.
8018  *
8019  * This is used by sockets and for a socket it would be incorrect
8020  * to return a failure for SIOCATMARK when there is no data in the receive
8021  * queue and the marked urgent data is traveling up the stream.
8022  *
8023  * This routine waits until the mark is known by waiting for one of these
8024  * three events:
8025  *   The stream head read queue becoming non-empty (including an EOF).
8026  *   The STRATMARK flag being set (due to a MSGMARKNEXT message).

```

```

8027 *   The STRNOTATMARK flag being set (which indicates that the transport
8028 *   has sent a MSGNOTMARKNEXT message to indicate that it is not at
8029 *   the mark).
8030 *
8031 * The routine returns 1 if the stream is at the mark; 0 if it can
8032 * be determined that the stream is not at the mark.
8033 * If the wait times out and it can't determine
8034 * whether or not the stream might be at the mark the routine will return -1.
8035 *
8036 * Note: This routine should only be used when a mark is pending i.e.,
8037 * in the socket case the SIGURG has been posted.
8038 * Note2: This can not wakeup just because synchronous streams indicate
8039 * that data is available since it is not possible to use the synchronous
8040 * streams interfaces to determine the b_flag value for the data queued below
8041 * the stream head.
8042 */
8043 int
8044 strwaitmark(vnode_t *vp)
8045 {
8046     struct stdata *stp = vp->v_stream;
8047     queue_t *rq = _RD(stp->sd_wrq);
8048     int mark;

8050     mutex_enter(&stp->sd_lock);
8051     while (rq->q_first == NULL &&
8052           !(stp->sd_flag & (STRATMARK|STRNOTATMARK|STREOF))) {
8053         stp->sd_flag |= RSLEEP;

8055         /* Wait for 100 milliseconds for any state change. */
8056         if (str_cv_wait(&rq->q_wait, &stp->sd_lock, 100, 1) == -1) {
8057             mutex_exit(&stp->sd_lock);
8058             return (-1);
8059         }
8060     }
8061     if (stp->sd_flag & STRATMARK)
8062         mark = 1;
8063     else if (rq->q_first != NULL && (rq->q_first->b_flag & MSGMARK))
8064         mark = 1;
8065     else
8066         mark = 0;

8068     mutex_exit(&stp->sd_lock);
8069     return (mark);
8070 }

8072 /*
8073 * Set a read side error. If persist is set change the socket error
8074 * to persistent. If errfunc is set install the function as the exported
8075 * error handler.
8076 */
8077 void
8078 strsetrerror(vnode_t *vp, int error, int persist, errfunc_t errfunc)
8079 {
8080     struct stdata *stp = vp->v_stream;

8082     mutex_enter(&stp->sd_lock);
8083     stp->sd_rerror = error;
8084     if (error == 0 && errfunc == NULL)
8085         stp->sd_flag &= ~STRDERR;
8086     else
8087         stp->sd_flag |= STRDERR;
8088     if (persist) {
8089         stp->sd_flag &= ~STRDERRNONPERSIST;
8090     } else {
8091         stp->sd_flag |= STRDERRNONPERSIST;
8092     }

```

```

8093     stp->sd_rerrfunc = errfunc;
8094     if (error != 0 || errfunc != NULL) {
8095         cv_broadcast(&_RD(stp->sd_wrq)->q_wait); /* readers */
8096         cv_broadcast(&stp->sd_wrq->q_wait); /* writers */
8097         cv_broadcast(&stp->sd_monitor); /* ioctllers */

8099         mutex_exit(&stp->sd_lock);
8100         pollwakeup(&stp->sd_pollist, POLLERR);
8101         mutex_enter(&stp->sd_lock);

8103         if (stp->sd_sigflags & S_ERROR)
8104             strsendsig(stp->sd_siglist, S_ERROR, 0, error);
8105     }
8106     mutex_exit(&stp->sd_lock);
8107 }

8109 /*
8110 * Set a write side error. If persist is set change the socket error
8111 * to persistent.
8112 */
8113 void
8114 strsetwerror(vnode_t *vp, int error, int persist, errfunc_t errfunc)
8115 {
8116     struct stdata *stp = vp->v_stream;

8118     mutex_enter(&stp->sd_lock);
8119     stp->sd_werror = error;
8120     if (error == 0 && errfunc == NULL)
8121         stp->sd_flag &= ~STWRERR;
8122     else
8123         stp->sd_flag |= STWRERR;
8124     if (persist) {
8125         stp->sd_flag &= ~STWRERRNONPERSIST;
8126     } else {
8127         stp->sd_flag |= STWRERRNONPERSIST;
8128     }
8129     stp->sd_werrfunc = errfunc;
8130     if (error != 0 || errfunc != NULL) {
8131         cv_broadcast(&_RD(stp->sd_wrq)->q_wait); /* readers */
8132         cv_broadcast(&stp->sd_wrq->q_wait); /* writers */
8133         cv_broadcast(&stp->sd_monitor); /* ioctllers */

8135         mutex_exit(&stp->sd_lock);
8136         pollwakeup(&stp->sd_pollist, POLLERR);
8137         mutex_enter(&stp->sd_lock);

8139         if (stp->sd_sigflags & S_ERROR)
8140             strsendsig(stp->sd_siglist, S_ERROR, 0, error);
8141     }
8142     mutex_exit(&stp->sd_lock);
8143 }

8145 /*
8146 * Make the stream return 0 (EOF) when all data has been read.
8147 * No effect on write side.
8148 */
8149 void
8150 strseteof(vnode_t *vp, int eof)
8151 {
8152     struct stdata *stp = vp->v_stream;

8154     mutex_enter(&stp->sd_lock);
8155     if (!eof) {
8156         stp->sd_flag &= ~STREOF;
8157         mutex_exit(&stp->sd_lock);
8158     }

```

```
8159     }
8160     stp->sd_flag |= STREOF;
8161     if (stp->sd_flag & RSLEEP) {
8162         stp->sd_flag &= ~RSLEEP;
8163         cv_broadcast(&_RD(stp->sd_wrq)->q_wait);
8164     }
8165
8166     mutex_exit(&stp->sd_lock);
8167     pollwakep(&stp->sd_pollist, POLLIN|POLLRDNORM);
8168     mutex_enter(&stp->sd_lock);
8169
8170     if (stp->sd_sigflags & (S_INPUT|S_RDNORM))
8171         strsendsig(stp->sd_siglist, S_INPUT|S_RDNORM, 0, 0);
8172     mutex_exit(&stp->sd_lock);
8173 }
8174
8175 void
8176 strflushrq(vnode_t *vp, int flag)
8177 {
8178     struct stdata *stp = vp->v_stream;
8179
8180     mutex_enter(&stp->sd_lock);
8181     flushq(_RD(stp->sd_wrq), flag);
8182     mutex_exit(&stp->sd_lock);
8183 }
8184
8185 void
8186 strsetrpathooks(vnode_t *vp, uint_t flags, msgfunc_t protofunc,
8187                msgfunc_t miscfunc)
8188 {
8189     struct stdata *stp = vp->v_stream;
8190
8191     mutex_enter(&stp->sd_lock);
8192
8193     if (protofunc == NULL)
8194         stp->sd_rprotofunc = strrput_proto;
8195     else
8196         stp->sd_rprotofunc = protofunc;
8197
8198     if (miscfunc == NULL)
8199         stp->sd_rmiscfunc = strrput_misc;
8200     else
8201         stp->sd_rmiscfunc = miscfunc;
8202
8203     if (flags & SH_CONSOL_DATA)
8204         stp->sd_rput_opt |= SR_CONSOL_DATA;
8205     else
8206         stp->sd_rput_opt &= ~SR_CONSOL_DATA;
8207
8208     if (flags & SH_SIGALldata)
8209         stp->sd_rput_opt |= SR_SIGALldata;
8210     else
8211         stp->sd_rput_opt &= ~SR_SIGALldata;
8212
8213     if (flags & SH_IGN_ZEROLEN)
8214         stp->sd_rput_opt |= SR_IGN_ZEROLEN;
8215     else
8216         stp->sd_rput_opt &= ~SR_IGN_ZEROLEN;
8217
8218     mutex_exit(&stp->sd_lock);
8219 }
8220
8221 unchanged_portion_omitted
```

```

*****
22222 Wed Jul 13 01:32:41 2016
new/usr/src/uts/common/sys/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 #
23 # Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright 2014, Joyent, Inc. All rights reserved.
25 # Copyright 2013 Garrett D'Amore <garrett@damore.org>
26 # Copyright 2013 Saso Kiselkov. All rights reserved.
27 # Copyright 2015 Igor Kozhukhov <ikozhukhov@gmail.com>
28 # Copyright 2016 Nexenta Systems, Inc.
29 #
30 #
31 include $(SRC)/uts/Makefile.uts
32 #
33 FILEMODE=644
34 #
35 #
36 # Note that the following headers are present in the kernel but
37 # neither installed or shipped as part of the product:
38 # cpuid_drv.h: Private interface for cpuid consumers
39 # unix_bb_info.h: Private interface to kcov
40 #
41 #
42 i386_HDRS= \
43 agp/agpamd64gart_io.h \
44 agp/agpdefs.h \
45 agp/agpgart_impl.h \
46 agp/agpmaster_io.h \
47 agp/agptarget_io.h \
48 agpgart.h \
49 asy.h \
50 fd_debug.h \
51 fd_c.h \
52 fdmedia.h \
53 mouse.h \
54 ucode.h
55 #
56 sparc_HDRS= \
57 mouse.h \
58 scsi/targets/ssddef.h \
59 $(MDESCHDRS)
60 #
61 # Generated headers

```

```

62 GENHDRS= \
63 priv_const.h \
64 priv_names.h \
65 usb/usbdevs.h
66 #
67 CHKHDRS= \
68 acpi_drv.h \
69 acct.h \
70 acctctl.h \
71 acl.h \
72 acl_impl.h \
73 aggr.h \
74 aggr_impl.h \
75 aio.h \
76 aio_impl.h \
77 aio_req.h \
78 aiocb.h \
79 ascii.h \
80 asynch.h \
81 atomic.h \
82 attr.h \
83 audio.h \
84 audioio.h \
85 autoconf.h \
86 auxv.h \
87 auxv_386.h \
88 auxv_SPARC.h \
89 avl.h \
90 avl_impl.h \
91 bitmap.h \
92 bitset.h \
93 bl.h \
94 blkdev.h \
95 bofi.h \
96 bofi_impl.h \
97 bpp_io.h \
98 bootstat.h \
99 brand.h \
100 buf.h \
101 bufmod.h \
102 bustypes.h \
103 byteorder.h \
104 callb.h \
105 callo.h \
106 cap_util.h \
107 cpucaps.h \
108 cpucaps_impl.h \
109 ccompile.h \
110 cdio.h \
111 cladm.h \
112 class.h \
113 clconf.h \
114 clock_impl.h \
115 cmlb.h \
116 cmn_err.h \
117 compress.h \
118 condvar.h \
119 condvar_impl.h \
120 conf.h \
121 consdev.h \
122 console.h \
123 consplat.h \
124 vt.h \
125 vtdaemon.h \
126 kd.h \
127 contract.h

```

```

128     contract_impl.h  \
129     copyops.h        \
130     core.h           \
131     corectl.h       \
132     cpc_impl.h      \
133     cpc_pcbe.h     \
134     cpr.h           \
135     cpupart.h      \
136     cpuvar.h       \
137     crc32.h        \
138     cred.h         \
139     cred_impl.h    \
140     pidnode.h      \
141 #endif /* ! codereview */
142     crtctl.h        \
143     cryptmod.h     \
144     csioctl.h      \
145     ctf.h          \
146     ctfs.h         \
147     ctfs_impl.h   \
148     ctf_api.h     \
149     ctype.h        \
150     cyclic.h       \
151     cyclic_impl.h \
152     dacf.h         \
153     dacf_impl.h   \
154     damap.h        \
155     damap_impl.h  \
156     dc_ki.h        \
157     ddi.h          \
158     ddifm.h        \
159     ddifm_impl.h  \
160     ddi_hp.h       \
161     ddi_hp_impl.h \
162     ddi_intr.h     \
163     ddi_intr_impl.h \
164     ddi_impldefs.h \
165     ddi_implfuncs.h \
166     ddi_obsolete.h \
167     ddi_periodic.h \
168     ddidevmap.h   \
169     ddi_dmareq.h  \
170     ddi_mapreq.h  \
171     ddi_propdefs.h \
172     ddi_types.h   \
173     debug.h       \
174     des.h         \
175     devctl.h      \
176     devcache.h    \
177     devcache_impl.h \
178     devfm.h       \
179     devid_cache.h \
180     devinfo_impl.h \
181     devops.h      \
182     devpolicy.h   \
183     devpoll.h     \
184     dirent.h      \
185     disp.h        \
186     dkbad.h       \
187     dkio.h        \
188     dklabel.h     \
189     dl.h          \
190     dlpi.h        \
191     dld.h         \
192     dld_impl.h   \
193     dld_ioc.h    \

```

```

194     dls.h          \
195     dls_mgmt.h    \
196     dls_impl.h    \
197     dma_i8237A.h  \
198     dnlc.h        \
199     door.h        \
200     door_data.h   \
201     door_impl.h   \
202     dtrace.h      \
203     dtrace_impl.h \
204     dumpadm.h     \
205     dumphdr.h     \
206     ecppsys.h     \
207     ecppio.h      \
208     ecppreg.h     \
209     ecppvar.h     \
210     edonr.h       \
211     efi_partition.h \
212     elf.h         \
213     elf_386.h     \
214     elf_SPARC.h   \
215     elf_notes.h   \
216     elf_amd64.h   \
217     elftypes.h    \
218     emul64.h      \
219     emul64cmd.h   \
220     emul64var.h   \
221     epm.h         \
222     epoll.h       \
223     errno.h       \
224     errorq.h      \
225     errorq_impl.h \
226     esunddi.h     \
227     ethernet.h    \
228     euc.h         \
229     eucioctl.h    \
230     eventfd.h     \
231     exacct.h      \
232     exacct_catalog.h \
233     exacct_impl.h \
234     exec.h        \
235     exechdr.h     \
236     extdirent.h   \
237     fault.h       \
238     fasttrap.h    \
239     fasttrap_impl.h \
240     fbio.h        \
241     fbuf.h        \
242     fcntl.h       \
243     fct.h         \
244     fct_defines.h \
245     fctio.h       \
246     fdbuffer.h    \
247     fdio.h        \
248     feature_tests.h \
249     fem.h         \
250     file.h        \
251     filio.h       \
252     flock.h       \
253     flock_impl.h  \
254     fork.h        \
255     fss.h         \
256     fssprioctl.h  \
257     fsid.h        \
258     fssnap.h      \
259     fssnap_if.h   \

```

```

260     fstyp.h          \
261     ftrace.h        \
262     fx.h            \
263     fxpriocntl.h   \
264     gfs.h          \
265     gld.h          \
266     gldpriv.h      \
267     group.h        \
268     hdio.h         \
269     hook.h         \
270     hook_event.h   \
271     hook_impl.h    \
272     hwconf.h       \
273     ia.h           \
274     iapriocntl.h   \
275     ibpart.h       \
276     id32.h         \
277     idmap.h        \
278     ieeefp.h       \
279     id_space.h     \
280     instance.h     \
281     int_const.h    \
282     int_fmtio.h    \
283     int_limits.h   \
284     int_types.h    \
285     inttypes.h     \
286     ioccom.h       \
287     ioctl.h        \
288     ipc.h          \
289     ipc_impl.h     \
290     ipc_rctl.h     \
291     ipd.h          \
292     ipmi.h         \
293     isa_defs.h     \
294     iscsi_authclient.h \
295     iscsi_authclientglue.h \
296     iscsi_protocol.h \
297     jioctl.h       \
298     kbd.h          \
299     kbdrereg.h     \
300     kbio.h         \
301     kcpc.h         \
302     kdi.h          \
303     kdi_impl.h    \
304     kiconv.h       \
305     kiconv_big5_utf8.h \
306     kiconv_ck_common.h \
307     kiconv_cp950hkscs_utf8.h \
308     kiconv_emea1.h \
309     kiconv_emea2.h \
310     kiconv_euckr_utf8.h \
311     kiconv_euctw_utf8.h \
312     kiconv_gb18030_utf8.h \
313     kiconv_gb2312_utf8.h \
314     kiconv_hkscs_utf8.h \
315     kiconv_ja.h    \
316     kiconv_ja_jis_to_unicode.h \
317     kiconv_ja_unicode_to_jis.h \
318     kiconv_ko.h    \
319     kiconv_latin1.h \
320     kiconv_sc.h    \
321     kiconv_tc.h    \
322     kiconv_uhc_utf8.h \
323     kiconv_utf8_big5.h \
324     kiconv_utf8_cp950hkscs.h \
325     kiconv_utf8_euckr.h \

```

```

326     kiconv_utf8_euctw.h \
327     kiconv_utf8_gb18030.h \
328     kiconv_utf8_gb2312.h \
329     kiconv_utf8_hkscs.h \
330     kiconv_utf8_uhc.h \
331     kidmap.h         \
332     klpd.h          \
333     klwp.h          \
334     kmdb.h          \
335     kmem.h          \
336     kmem_impl.h    \
337     kobj.h          \
338     kobj_impl.h    \
339     ksocket.h       \
340     kstat.h         \
341     kstr.h          \
342     ksyms.h         \
343     ksynch.h        \
344     ldterm.h        \
345     lgrp.h          \
346     lgrp_user.h    \
347     libc_kernel.h  \
348     link.h          \
349     list.h          \
350     list_impl.h    \
351     llc1.h          \
352     loadavg.h       \
353     lock.h          \
354     lockfs.h        \
355     lockstat.h     \
356     lofi.h          \
357     log.h           \
358     logindmux.h    \
359     logindmux_impl.h \
360     lwp.h           \
361     lwp_timer_impl.h \
362     lwp_upimutex_impl.h \
363     lpif.h          \
364     mac.h           \
365     mac_client.h   \
366     mac_client_impl.h \
367     mac_ether.h    \
368     mac_flow.h     \
369     mac_flow_impl.h \
370     mac_impl.h     \
371     mac_provider.h \
372     mac_soft_ring.h \
373     mac_stat.h     \
374     machelf.h      \
375     map.h          \
376     md4.h          \
377     md5.h          \
378     md5_consts.h   \
379     mdi_impldefs.h \
380     mem.h           \
381     mem_config.h   \
382     memlist.h      \
383     mkdev.h        \
384     mhd.h          \
385     mii.h          \
386     miiregs.h      \
387     mixer.h        \
388     mman.h         \
389     mmapobj.h      \
390     mntent.h       \
391     mntio.h        \

```

```

392 mnttab.h \
393 modctl.h \
394 mode.h \
395 model.h \
396 modhash.h \
397 modhash_impl.h \
398 mount.h \
399 mouse.h \
400 msacct.h \
401 msg.h \
402 msg_impl.h \
403 msio.h \
404 msreg.h \
405 mtio.h \
406 multidata.h \
407 multidata_impl.h \
408 mutex.h \
409 nbmlock.h \
410 ndifm.h \
411 ndi_impldefs.h \
412 net80211.h \
413 net80211_crypto.h \
414 net80211_ht.h \
415 net80211_proto.h \
416 netconfig.h \
417 neti.h \
418 netstack.h \
419 nexusdefs.h \
420 note.h \
421 null.h \
422 nvpair.h \
423 nvpair_impl.h \
424 objfs.h \
425 objfs_impl.h \
426 ontrap.h \
427 open.h \
428 openpromio.h \
429 panic.h \
430 param.h \
431 pathconf.h \
432 pathname.h \
433 pattn.h \
434 queue.h \
435 serializer.h \
436 pbio.h \
437 pccard.h \
438 pci.h \
439 pcie.h \
440 pci_impl.h \
441 pci_tools.h \
442 pcmcia.h \
443 ptypes.h \
444 pfmod.h \
445 pg.h \
446 pghw.h \
447 physmem.h \
448 pkp_hash.h \
449 pm.h \
450 policy.h \
451 poll.h \
452 poll_impl.h \
453 pool.h \
454 pool_impl.h \
455 pool_pset.h \
456 port.h \
457 port_impl.h \

```

```

458 port_kernel.h \
459 portif.h \
460 ppmio.h \
461 pppt_ic_if.h \
462 pppt_ioctl.h \
463 priocntl.h \
464 priv.h \
465 priv_impl.h \
466 prnio.h \
467 proc.h \
468 processor.h \
469 procfs.h \
470 procset.h \
471 project.h \
472 protosw.h \
473 prsystem.h \
474 pset.h \
475 pshot.h \
476 ptem.h \
477 ptms.h \
478 ptyvar.h \
479 raidioctl.h \
480 ramdisk.h \
481 random.h \
482 rctl.h \
483 rctl_impl.h \
484 rds.h \
485 reboot.h \
486 refstr.h \
487 refstr_impl.h \
488 resource.h \
489 rliocntl.h \
490 rt.h \
491 rtpriocntl.h \
492 rwlock.h \
493 rwlock_impl.h \
494 rwstlock.h \
495 sad.h \
496 schedctl.h \
497 sdt.h \
498 select.h \
499 sem.h \
500 sem_impl.h \
501 sema_impl.h \
502 semaphore.h \
503 sendfile.h \
504 ser_sync.h \
505 session.h \
506 shal.h \
507 shal_consts.h \
508 sha2.h \
509 sha2_consts.h \
510 share.h \
511 shm.h \
512 shm_impl.h \
513 sid.h \
514 siginfo.h \
515 signal.h \
516 signalfd.h \
517 skein.h \
518 sleepq.h \
519 sbios.h \
520 sbios_impl.h \
521 subject.h \
522 socket.h \
523 socket_impl.h \

```

```

524 socket_proto.h \
525 socketvar.h \
526 sockfilter.h \
527 sockio.h \
528 soundcard.h \
529 squeue.h \
530 squeue_impl.h \
531 srn.h \
532 sservice.h \
533 stat.h \
534 statfs.h \
535 statvfs.h \
536 stdbool.h \
537 stdint.h \
538 stermio.h \
539 stmf.h \
540 stmf_defines.h \
541 stmf_ioctl.h \
542 stmf_sbd_ioctl.h \
543 stream.h \
544 strft.h \
545 strlog.h \
546 strmdp.h \
547 stropts.h \
548 strredir.h \
549 strstat.h \
550 strsubr.h \
551 strsun.h \
552 strtty.h \
553 sunddi.h \
554 sunldi.h \
555 sunldi_impl.h \
556 sunmdi.h \
557 sunndi.h \
558 sunos_dhcp_class.h \
559 sunpm.h \
560 suntpl.h \
561 suntty.h \
562 swap.h \
563 synch.h \
564 sysdc.h \
565 sysdc_impl.h \
566 syscall.h \
567 sysconf.h \
568 sysconfig.h \
569 sysevent.h \
570 sysevent_impl.h \
571 sysinfo.h \
572 syslog.h \
573 sysmacros.h \
574 sysmsg_impl.h \
575 systeminfo.h \
576 system.h \
577 task.h \
578 taskq.h \
579 taskq_impl.h \
580 t_kuser.h \
581 t_lock.h \
582 telioctl.h \
583 termio.h \
584 termios.h \
585 termiox.h \
586 thread.h \
587 ticlts.h \
588 ticots.h \
589 ticotsord.h \

```

```

590 tihdr.h \
591 time.h \
592 time_impl.h \
593 time_std_impl.h \
594 timeb.h \
595 timer.h \
596 timerfd.h \
597 times.h \
598 timex.h \
599 timod.h \
600 tirdwr.h \
601 tiuser.h \
602 tl.h \
603 tnf.h \
604 tnf_com.h \
605 tnf_probe.h \
606 tnf_writer.h \
607 todio.h \
608 tpicommon.h \
609 ts.h \
610 tspriocntl.h \
611 ttcompat.h \
612 ttold.h \
613 tty.h \
614 ttychars.h \
615 ttydev.h \
616 tuneable.h \
617 turnstile.h \
618 types.h \
619 types32.h \
620 tzfile.h \
621 u8_textprep.h \
622 u8_textprep_data.h \
623 uadmin.h \
624 ucred.h \
625 uio.h \
626 ulimit.h \
627 un.h \
628 unistd.h \
629 user.h \
630 ustat.h \
631 utime.h \
632 utsname.h \
633 utssys.h \
634 uuid.h \
635 va_impl.h \
636 va_list.h \
637 var.h \
638 varargs.h \
639 vfs.h \
640 vfs_opreg.h \
641 vfstab.h \
642 vgareg.h \
643 videodev2.h \
644 visual_io.h \
645 vlan.h \
646 vm.h \
647 vm_usage.h \
648 vmem.h \
649 vmem_impl.h \
650 vmsystem.h \
651 vnic.h \
652 vnic_impl.h \
653 vnode.h \
654 vscan.h \
655 vtoc.h \

```

```

656      vtrace.h          \
657      vvoid_event.h     \
658      vvoid_wheel.h     \
659      vvoid_queue.h     \
660      vvoid_state.h     \
661      vvoid_store.h     \
662      wait.h            \
663      waitq.h           \
664      wanboot_impl.h    \
665      watchpoint.h     \
666      winlockio.h      \
667      zcons.h           \
668      zone.h            \
669      xti_inet.h        \
670      xti_osi.h         \
671      xti_xtiopt.h     \
672      zmod.h            \

674 HDRS=                  \
675      $(GENHDRS)         \
676      $(CHKHDRS)        \

678 AUDIOHDRS=            \
679      ac97.h             \
680      audio_common.h    \
681      audio_driver.h    \
682      audio_oss.h       \
683      g711.h            \

685 AVHDRS=               \
686      iec61883.h        \

688 BSCHDRS=              \
689      bscbus.h          \
690      bscv_impl.h       \
691      lom_ebuscodes.h   \
692      lom_io.h          \
693      lom_priv.h        \
694      lombus.h          \

696 MDESCHDRS=            \
697      mdesc.h           \
698      mdesc_impl.h     \

700 CPUDRVHDRS=           \
701      cpudrv.h         \

703 CRYPTOHDRS=          \
704      elfsign.h         \
705      ioctl.h           \
706      ioctladmin.h     \
707      common.h          \
708      impl.h            \
709      spi.h             \
710      api.h             \
711      ops_impl.h       \
712      sched_impl.h     \

714 DCAMHDRS=            \
715      dcam1394_io.h    \

717 IBHDRS=               \
718      ib_types.h       \
719      ib_pkt_hdrs.h    \

721 IBTLHDRS=             \

```

```

722      ibtl_types.h     \
723      ibtl_status.h   \
724      ibti.h           \
725      ibti_cm.h       \
726      ibci.h          \
727      ibti_common.h   \
728      ibvti.h         \
729      ibtl_ci_types.h \

731 IBTMLIMPLHDRS=       \
732      ibtl_util.h     \

734 IBNEXHDRS=           \
735      ibnex_devctl.h  \

737 IBMFHDRS=            \
738      ibmf.h          \
739      ibmf_msg.h      \
740      ibmf_saa.h      \
741      ibmf_utils.h    \

743 IBMGTHDRS=           \
744      ib_dm_attr.h    \
745      ib_mad.h        \
746      sm_attr.h       \
747      sa_recs.h       \

749 IBDHDRS=             \
750      ibd.h           \

752 OFHDRS=              \
753      ofa_solaris.h   \
754      ofed_kernel.h   \

756 RDMAHDRS=           \
757      ib_addr.h       \
758      ib_user_mad.h   \
759      ib_user_sa.h    \
760      ib_user_verbs.h \
761      ib_verbs.h      \
762      rdma_cm.h       \
763      rdma_user_cm.h \

765 SOL_UVERBSHDRS=     \
766      sol_uverbs.h    \
767      sol_uverbs2ucma.h \
768      sol_uverbs_comp.h \
769      sol_uverbs_hca.h \
770      sol_uverbs_qp.h \
771      sol_uverbs_event.h \

773 SOL_UMADHDRS=       \
774      sol_umad.h     \

776 SOL_UCMAHDRS=       \
777      sol_ucma.h     \
778      sol_rdma_user_cm.h \

780 SOL_OFSHDRS=        \
781      sol_cma.h      \
782      sol_ib_cma.h   \
783      sol_ofs_common.h \
784      sol_kverb_impl.h \

786 TAVORHDRS=         \
787      tavor_ioctl.h \

```

```

789 HERMONHDRS= \
790     hermon_ioctl.h

792 MLNXHDRS= \
793     mlnx_umap.h

795 IDMHDRS= \
796     idm.h \
797     idm_impl.h \
798     idm_so.h \
799     idm_text.h \
800     idm_transport.h \
801     idm_conn_sm.h

803 ISCSITHDRS= \
804     radius_packet.h \
805     radius_protocol.h \
806     chap.h \
807     isns_protocol.h \
808     iscsi_if.h \
809     iscsit_common.h

811 ISOHDRS= \
812     signal_iso.h

814 DERIVED_LVMHDRS= \
815     md_mdiox.h \
816     md_basic.h \
817     mdmed.h \
818     md_mhdx.h \
819     mdmn_commd.h

821 LVMHDRS= \
822     md_convert.h \
823     md_crc.h \
824     md_hotspares.h \
825     md_mddb.h \
826     md_mirror.h \
827     md_mirror_shared.h \
828     md_names.h \
829     md_notify.h \
830     md_raid.h \
831     md_rename.h \
832     md_sp.h \
833     md_stripe.h \
834     md_trans.h \
835     mdio.h \
836     mdvar.h

838 ALL_LVMHDRS= \
839     $(LVMHDRS) \
840     $(DERIVED_LVMHDRS)

842 FMHDRS= \
843     protocol.h \
844     util.h

846 FMFSHDRS= \
847     zfs.h

849 FMIOHDRS= \
850     ddi.h \
851     disk.h \
852     pci.h \
853     scsi.h

```

```

854     sun4upci.h \
855     opl_mc_fm.h

857 FSHDRS= \
858     autofs.h \
859     decomp.h \
860     dv_node.h \
861     sdev_impl.h \
862     fifonode.h \
863     hsfs_isospec.h \
864     hsfs_node.h \
865     hsfs_rrip.h \
866     hsfs_spec.h \
867     hsfs_susp.h \
868     lofs_info.h \
869     lofs_node.h \
870     mntdata.h \
871     namenode.h \
872     pc_dir.h \
873     pc_fs.h \
874     pc_label.h \
875     pc_node.h \
876     pxfh_ki.h \
877     snode.h \
878     swapnode.h \
879     tmp.h \
880     tmpnode.h \
881     udf_inode.h \
882     udf_volume.h \
883     ufs_acl.h \
884     ufs_bio.h \
885     ufs_filio.h \
886     ufs_fs.h \
887     ufs_fsdire.h \
888     ufs_inode.h \
889     ufs_lockfs.h \
890     ufs_log.h \
891     ufs_mount.h \
892     ufs_panic.h \
893     ufs_prot.h \
894     ufs_quota.h \
895     ufs_snap.h \
896     ufs_trans.h \
897     zfs.h \
898     zut.h

900 SCSIHDRS= \
901     scsi.h \
902     scsi_address.h \
903     scsi_ctl.h \
904     scsi_fm.h \
905     scsi_names.h \
906     scsi_params.h \
907     scsi_pkt.h \
908     scsi_resource.h \
909     scsi_types.h \
910     scsi_watch.h

912 SCSSICONFHDRS= \
913     autoconf.h \
914     device.h

916 SCSSIGENHDRS= \
917     commands.h \
918     dad_mode.h \
919     inquiry.h

```

```

920     message.h      \
921     mode.h         \
922     persist.h     \
923     sense.h       \
924     sff_frames.h  \
925     smp_frames.h  \
926     status.h      \

928 SCSIIMPLHDRS=    \
929     commands.h    \
930     inquiry.h     \
931     mode.h        \
932     scsi_reset_notify.h \
933     scsi_sas.h    \
934     sense.h       \
935     services.h    \
936     smp_transport.h \
937     spc3_types.h  \
938     status.h      \
939     transport.h   \
940     types.h       \
941     uscsi.h       \
942     usmp.h        \

944 SCSTITARGETSHDRS= \
945     ses.h         \
946     sesio.h      \
947     sgendef.h    \
948     stdef.h      \
949     sdef.h       \
950     smp.h        \

952 SCSIADHDRS=

954 SCASICADHDRS=

956 SCSIISCSIHDRS=  \
957     iscsi_door.h \
958     iscsi_if.h   \

960 SCIVHCIHDRS=    \
961     scsi_vhci.h  \
962     mpapi_impl.h \
963     mpapi_scsi_vhci.h \

965 SDCARDHDRS=     \
966     sda.h        \
967     sda_impl.h   \
968     sda_ioctl.h  \

970 FC4HDRS=        \
971     fc_transport.h \
972     linkapp.h     \
973     fc.h          \
974     fcp.h        \
975     fcal_transport.h \
976     fcal.h       \
977     fcal_linkapp.h \
978     fcio.h       \

980 FCHDRS=         \
981     fc.h         \
982     fcio.h      \
983     fc_types.h  \
984     fc_appif.h  \

```

```

986 FCIMPLHDRS=     \
987     fc_error.h  \
988     fcph.h      \

990 FCULPHDRS=      \
991     fcp_util.h  \
992     fcsm.h      \

994 SATAGENHDRS=    \
995     sata_hba.h  \
996     sata_defs.h \
997     sata_cfgadm.h \

999 SYSEVENTHDRS=   \
1000     ap_driver.h \
1001     dev.h       \
1002     domain.h   \
1003     dr.h        \
1004     env.h       \
1005     eventdefs.h \
1006     ipmp.h     \
1007     pwrctl.h   \
1008     svm.h      \
1009     vrrp.h     \

1011 CONTRACTHDRS=   \
1012     process.h   \
1013     process_impl.h \
1014     device.h    \
1015     device_impl.h \

1017 USBHDRS=        \
1018     usba.h     \
1019     usbai.h    \

1021 USBAUDHDRS=     \
1022     usb_audio.h \

1024 USBHUBDHDRS=    \
1025     hub.h       \
1026     hubd_impl.h \

1028 USBHIDHDRS=     \
1029     hid.h       \

1031 USBMSHDRS=      \
1032     usb_bulkonly.h \
1033     usb_cbi.h    \

1035 USBPRNHDRS=     \
1036     usb_printer.h \

1038 USBDCDCHDRS=    \
1039     usb_cdc.h   \

1041 USBVIDHDRS=     \
1042     usbvc.h     \

1044 USBWCMHDRS=     \
1045     usbwcm.h   \

1047 UGENHDRS=       \
1048     usb_ugen.h \

1050 HOTPLUGHDRS=    \
1051     hpcsvc.h    \

```

```

1052      hpctrl.h
1054 HOTPLUGPCIHDRS= \
1055      pcicfg.h      \
1056      pcihp.h
1058 RSMHDRS= \
1059      rsm.h      \
1060      rsm_common.h \
1061      rsmapi_common.h \
1062      rsmapi.h \
1063      rsmapi_driver.h \
1064      rsmka_path_int.h
1066 TSOLHDRS= \
1067      label.h      \
1068      label_macro.h \
1069      priv.h      \
1070      tndb.h      \
1071      tsyscall.h
1073 I1394HDRS= \
1074      cmd1394.h \
1075      id1394.h \
1076      ieee1212.h \
1077      ieee1394.h \
1078      ixl1394.h \
1079      sl394_impl.h \
1080      t1394.h
1082 # "cmdk" headers used on sparc
1083 SDKTPHDRS= \
1084      dadkio.h \
1085      fdisk.h
1087 # "cmdk" headers used on i386
1088 DKTPHDRS= \
1089      altsctr.h \
1090      bbh.h \
1091      cm.h \
1092      cmdev.h \
1093      cmdk.h \
1094      cmpkt.h \
1095      controller.h \
1096      dadev.h \
1097      dadk.h \
1098      dadkio.h \
1099      fctypes.h \
1100      fdisk.h \
1101      flowctrl.h \
1102      gda.h \
1103      quetypes.h \
1104      queue.h \
1105      tgcom.h \
1106      tgdk.h
1108 # "pc" header files used on i386
1109 PCHDRS= \
1110      avintr.h \
1111      dma_engine.h \
1112      i8272A.h \
1113      pcic_reg.h \
1114      pcic_var.h \
1115      pic.h \
1116      pit.h \
1117      rtc.h

```

```

1119 NXGEHDRS= \
1120      nxge.h \
1121      nxge_common.h \
1122      nxge_common_impl.h \
1123      nxge_defs.h \
1124      nxge_hw.h \
1125      nxge_impl.h \
1126      nxge_ipp.h \
1127      nxge_ipp_hw.h \
1128      nxge_mac.h \
1129      nxge_mac_hw.h \
1130      nxge_fflp.h \
1131      nxge_fflp_hw.h \
1132      nxge_mii.h \
1133      nxge_rxdma.h \
1134      nxge_rxdma_hw.h \
1135      nxge_txc.h \
1136      nxge_txc_hw.h \
1137      nxge_txdma.h \
1138      nxge_txdma_hw.h \
1139      nxge_virtual.h \
1140      nxge_espc.h
1142 include Makefile.sysshdrs
1144 dcam/%.check: dcam/%.h
1145      $(DOT_H_CHECK)
1147 CHECKHDRS= \
1148      $(MACH)_HDRS:%.h=%.check \
1149      $(AUDIOHDRS:%.h=audio/%.check) \
1150      $(AVHDRS:%.h=av/%.check) \
1151      $(BSCHDRS:%.h=%.check) \
1152      $(CHKHDRS:%.h=%.check) \
1153      $(CPUDRVHDRS:%.h=%.check) \
1154      $(CRYPTOHDRS:%.h=crypto/%.check) \
1155      $(DCAMHDRS:%.h=dcam/%.check) \
1156      $(FC4HDRS:%.h=fc4/%.check) \
1157      $(FCHDRS:%.h=fibre-channel/%.check) \
1158      $(FCIMPLHDRS:%.h=fibre-channel/impl/%.check) \
1159      $(FCULPHDRS:%.h=fibre-channel/ulp/%.check) \
1160      $(IBHDRS:%.h=ib/%.check) \
1161      $(IBDHDRS:%.h=ib/clients/ibd/%.check) \
1162      $(IBTLHDRS:%.h=ib/ibtl/%.check) \
1163      $(IBTLIMPLHDRS:%.h=ib/ibtl/impl/%.check) \
1164      $(IBNEXHDRS:%.h=ib/ibnex/%.check) \
1165      $(IBMGTHDRS:%.h=ib/mgt/%.check) \
1166      $(IBMFHDRS:%.h=ib/mgt/ibmf/%.check) \
1167      $(OFHDRS:%.h=ib/clients/of/%.check) \
1168      $(RDMAHDRS:%.h=ib/clients/of/rdma/%.check) \
1169      $(SOL_UVERBSHDRS:%.h=ib/clients/of/sol_uverbs/%.check) \
1170      $(SOL_UCMAHDRS:%.h=ib/clients/of/sol_ucma/%.check) \
1171      $(SOL_OFSHDRS:%.h=ib/clients/of/sol_ofs/%.check) \
1172      $(TAVORHDRS:%.h=ib/adapters/tavor/%.check) \
1173      $(HERMONHDRS:%.h=ib/adapters/hermon/%.check) \
1174      $(MLNXHDRS:%.h=ib/adapters/%.check) \
1175      $(IDMHDRS:%.h=idm/%.check) \
1176      $(ISCSIHDRS:%.h=iscsi/%.check) \
1177      $(ISCSITHDRS:%.h=iscsit/%.check) \
1178      $(ISOHDRS:%.h=iso/%.check) \
1179      $(FMHDRS:%.h=fm/%.check) \
1180      $(FMFSHDRS:%.h=fm/fs/%.check) \
1181      $(FMIOHDRS:%.h=fm/io/%.check) \
1182      $(FSHDRS:%.h=fs/%.check) \
1183      $(LVMHDRS:%.h=lvm/%.check)

```

```

1184 $(SCSIHDRS:%.h=scsi/.check) \
1185 $(SCSIADHDRS:%.h=scsi/adapters/.check) \
1186 $(SCSICONFHDRS:%.h=scsi/conf/.check) \
1187 $(SCSIIMPLHDRS:%.h=scsi/impl/.check) \
1188 $(SCSIISCSIHDRS:%.h=scsi/adapters/.check) \
1189 $(SCSIGHDRS:%.h=scsi/generic/.check) \
1190 $(SCSITARGETSHDRS:%.h=scsi/targets/.check) \
1191 $(SCSIVHCIHDRS:%.h=scsi/adapters/.check) \
1192 $(SATAGENHDRS:%.h=sata/.check) \
1193 $(SDCARDHDRS:%.h=sdcard/.check) \
1194 $(SYSEVENTHDRS:%.h=sysevent/.check) \
1195 $(CONTRACTHDRS:%.h=contract/.check) \
1196 $(USBAUDHDRS:%.h=usb/clients/audio/.check) \
1197 $(USBHUBDHDRS:%.h=usb/hubd/.check) \
1198 $(USBHIDHDRS:%.h=usb/clients/hid/.check) \
1199 $(USBMSHDRS:%.h=usb/clients/mass_storage/.check) \
1200 $(USBPRNHDRS:%.h=usb/clients/printer/.check) \
1201 $(USBCDCHDRS:%.h=usb/clients/usbcdc/.check) \
1202 $(USBVIDHDRS:%.h=usb/clients/video/usbvc/.check) \
1203 $(USBWCMHDRS:%.h=usb/clients/usbinput/usbwcm/.check) \
1204 $(UGENHDRS:%.h=usb/clients/ugen/.check) \
1205 $(USBHDRS:%.h=usb/.check) \
1206 $(I1394HDRS:%.h=1394/.check) \
1207 $(RSMHDRS:%.h=rsm/.check) \
1208 $(TSOLHDRS:%.h=tsol/.check) \
1209 $(NXGEHDRS:%.h=nxge/.check)

```

```
1212 .KEEP_STATE:
```

```

1214 .PARALLEL: \
1215 $(CHECKHDRS) \
1216 $(ROOTHDRS) \
1217 $(ROOTAUDHDRS) \
1218 $(ROOTAVHDRS) \
1219 $(ROOTCRYPTOHDRS) \
1220 $(ROOTDCAMHDRS) \
1221 $(ROOTISOHDRS) \
1222 $(ROOTIDMHDRS) \
1223 $(ROOTISCSIHDRS) \
1224 $(ROOTISCSITHDRS) \
1225 $(ROOTFC4HDRS) \
1226 $(ROOTFCHDRS) \
1227 $(ROOTFCIMPLHDRS) \
1228 $(ROOTFCULPHDRS) \
1229 $(ROOTFMHDRS) \
1230 $(ROOTFMIOHDRS) \
1231 $(ROOTFMFSDHDRS) \
1232 $(ROOTFSDHDRS) \
1233 $(ROOTIBDHDRS) \
1234 $(ROOTIBHDRS) \
1235 $(ROOTIBTLHDRS) \
1236 $(ROOTIBTLIMPLHDRS) \
1237 $(ROOTIBNEXHDRS) \
1238 $(ROOTIBMGTHDRS) \
1239 $(ROOTIBMFHDRS) \
1240 $(ROOTOFHDRS) \
1241 $(ROOTRDMAHDRS) \
1242 $(ROOTSOL_OFSDHDRS) \
1243 $(ROOTSOL_UMADHDRS) \
1244 $(ROOTSOL_UVERBSHDRS) \
1245 $(ROOTSOL_UCMAHDRS) \
1246 $(ROOTTAVORHDRS) \
1247 $(ROOTTHERMONHDRS) \
1248 $(ROOTMLNXHDRS) \
1249 $(ROOTLVMHDRS) \

```

```

1250 $(ROOTSCSIHDRS) \
1251 $(ROOTSCSIADHDRS) \
1252 $(ROOTSCSICONFHDRS) \
1253 $(ROOTSCSIISCSIHDRS) \
1254 $(ROOTSCSIGHDRS) \
1255 $(ROOTSCSIIMPLHDRS) \
1256 $(ROOTSCSIVHCIHDRS) \
1257 $(ROOTSDCARDHDRS) \
1258 $(ROOTSYSEVENTHDRS) \
1259 $(ROOTCONTRACTHDRS) \
1260 $(ROOTUSBHDRS) \
1261 $(ROOTUWBHDRS) \
1262 $(ROOTUWBAHDRS) \
1263 $(ROOTUSBAUDHDRS) \
1264 $(ROOTUSBHUBDHDRS) \
1265 $(ROOTUSBHIDHDRS) \
1266 $(ROOTUSBHRCCHDRS) \
1267 $(ROOTUSBMSHDRS) \
1268 $(ROOTUSBPRNHDRS) \
1269 $(ROOTUSBCDCHDRS) \
1270 $(ROOTUSBVIDHDRS) \
1271 $(ROOTUSBWCMHDRS) \
1272 $(ROOTUGENHDRS) \
1273 $(ROOT1394HDRS) \
1274 $(ROOTHOTPLUGHDRS) \
1275 $(ROOTHOTPLUGPCIHDRS) \
1276 $(ROOTRSMHDRS) \
1277 $(ROOTTSOLHDRS) \
1278 $( $(MACH)_ROOTHDRS)

```

```

1281 install_h: \
1282 $(ROOTDIRS) \
1283 LVMDERIVED_H \
1284 .WAIT \
1285 $(ROOTHDRS) \
1286 $(ROOTAUDHDRS) \
1287 $(ROOTAVHDRS) \
1288 $(ROOTCRYPTOHDRS) \
1289 $(ROOTDCAMHDRS) \
1290 $(ROOTISOHDRS) \
1291 $(ROOTIDMHDRS) \
1292 $(ROOTISCSIHDRS) \
1293 $(ROOTISCSITHDRS) \
1294 $(ROOTFC4HDRS) \
1295 $(ROOTFCHDRS) \
1296 $(ROOTFCIMPLHDRS) \
1297 $(ROOTFCULPHDRS) \
1298 $(ROOTFMHDRS) \
1299 $(ROOTFMFSDHDRS) \
1300 $(ROOTFMIOHDRS) \
1301 $(ROOTFSDHDRS) \
1302 $(ROOTIBDHDRS) \
1303 $(ROOTIBHDRS) \
1304 $(ROOTIBTLHDRS) \
1305 $(ROOTIBTLIMPLHDRS) \
1306 $(ROOTIBNEXHDRS) \
1307 $(ROOTIBMGTHDRS) \
1308 $(ROOTIBMFHDRS) \
1309 $(ROOTOFHDRS) \
1310 $(ROOTRDMAHDRS) \
1311 $(ROOTSOL_OFSDHDRS) \
1312 $(ROOTSOL_UMADHDRS) \
1313 $(ROOTSOL_UVERBSHDRS) \
1314 $(ROOTSOL_UCMAHDRS) \
1315 $(ROOTTAVORHDRS) \

```

```
1316 $(ROOTHERMONHDRS) \
1317 $(ROOTMLNXHDRS) \
1318 $(ROOTLVMHDRS) \
1319 $(ROOTSCSIHDRS) \
1320 $(ROOTSCSIADHDRS) \
1321 $(ROOTSCSIISCSIHDRS) \
1322 $(ROOTSCSICONFHDRS) \
1323 $(ROOTSCSIGENHDRS) \
1324 $(ROOTSCSIIMPLHDRS) \
1325 $(ROOTSCSIVHCIHDRS) \
1326 $(ROOTSDCARDHDRS) \
1327 $(ROOTSYSEVENTHDRS) \
1328 $(ROOTCONTRACTHDRS) \
1329 $(ROOTUWBHDRS) \
1330 $(ROOTUWBAHDRS) \
1331 $(ROOTUSBHDRS) \
1332 $(ROOTUSBAUDHDRS) \
1333 $(ROOTUSBHUBDHDRS) \
1334 $(ROOTUSBHIDHDRS) \
1335 $(ROOTUSBHCHDRS) \
1336 $(ROOTUSBMSHDRS) \
1337 $(ROOTUSBPRNHDRS) \
1338 $(ROOTUSBCDCHDRS) \
1339 $(ROOTUSBVIDHDRS) \
1340 $(ROOTUSBWCMHDRS) \
1341 $(ROOTUGENHDRS) \
1342 $(ROOT1394HDRS) \
1343 $(ROOTHOTPLUGHDRS) \
1344 $(ROOTHOTPLUGPCIHDRS) \
1345 $(ROOTRSMHDRS) \
1346 $(ROOTTSOLHDRS) \
1347 $(MACH)_ROOTHDRS)

1349 all_h: $(GENHDRS)

1351 priv_const.h: $(PRIVS_AWK) $(PRIVS_DEF)
1352 $(AWK) -f $(PRIVS_AWK) < $(PRIVS_DEF) -v privhfile=$@

1354 priv_names.h: $(PRIVS_AWK) $(PRIVS_DEF)
1355 $(AWK) -f $(PRIVS_AWK) < $(PRIVS_DEF) -v pubhfile=$@

1357 usb/usbdevs.h: $(USBDEVS_AWK) $(USBDEVS_DATA)
1358 $(AWK) -f $(USBDEVS_AWK) $(USBDEVS_DATA) -H > $@

1360 LVMDERIVED_H:
1361 cd $(SRC)/uts/common/sys/lvm; pwd; $(MAKE) all_h

1363 clean:
1364 $(RM) $(GENHDRS)

1366 clobber: clean
1367 cd $(SRC)/uts/common/sys/lvm; pwd; $(MAKE) clobber

1369 check: $(CHECKHDRS)

1371 FRC:
```

new/usr/src/uts/common/sys/fcntl.h

1

```
*****
13060 Wed Jul 13 01:32:41 2016
new/usr/src/uts/common/sys/fcntl.h
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
22 /*
23  * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
24 */
26 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
27 /*      All Rights Reserved */
29 /*
30  * University Copyright- Copyright (c) 1982, 1986, 1988
31  * The Regents of the University of California
32  * All Rights Reserved
33  *
34  * University Acknowledgment- Portions of this document are derived from
35  * software developed by the University of California, Berkeley, and its
36  * contributors.
37  */
39 /* Copyright (c) 2013, OmniTI Computer Consulting, Inc. All rights reserved. */
40 /* Copyright 2015, Joyent, Inc. */
42 #ifndef _SYS_FCNTL_H
43 #define _SYS_FCNTL_H
45 #include <sys/feature_tests.h>
47 #include <sys/types.h>
49 #ifdef __cplusplus
50 extern "C" {
51 #endif
53 /*
54  * Flag values accessible to open(2) and fcntl(2)
55  * The first five can only be set (exclusively) by open(2).
56  */
57 #define O_RDONLY      0
58 #define O_WRONLY      1
59 #define O_RDWR        2
60 #define O_SEARCH      0x200000
61 #define O_EXEC         0x400000
```

new/usr/src/uts/common/sys/fcntl.h

2

```
62 #if defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE)
63 #define O_NDELAY      0x04 /* non-blocking I/O */
64 #endif /* defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE) */
65 #define O_APPEND      0x08 /* append (writes guaranteed at the end) */
66 #if defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE) || \
67     (_POSIX_C_SOURCE > 2) || defined(_XOPEN_SOURCE)
68 #define O_SYNC        0x10 /* synchronized file update option */
69 #define O_DSYNC       0x40 /* synchronized data update option */
70 #define O_RSYNC       0x8000 /* synchronized file update option */
71 /* defines read/write file integrity */
72 #endif /* defined(__EXTENSIONS__) || !defined(_POSIX_C_SOURCE) ... */
73 #define O_NONBLOCK    0x80 /* non-blocking I/O (POSIX) */
74 #ifdef _LARGEFILE_SOURCE
75 #define O_LARGEFILE   0x2000
76 #endif
78 /*
79  * Flag values accessible only to open(2).
80  */
81 #define O_CREAT        0x100 /* open with file create (uses third arg) */
82 #define O_TRUNC        0x200 /* open with truncation */
83 #define O_EXCL         0x400 /* exclusive open */
84 #define O_NOCTTY       0x800 /* don't allocate controlling tty (POSIX) */
85 #define O_XATTR        0x4000 /* extended attribute */
86 #define O_NOFOLLOW     0x20000 /* don't follow symlinks */
87 #define O_NOLINKS      0x40000 /* don't allow multiple hard links */
88 #define O_CLOEXEC      0x800000 /* set the close-on-exec flag */
90 /*
91  * fcntl(2) requests
92  *
93  * N.B.: values are not necessarily assigned sequentially below.
94  */
95 #define F_DUPFD        0 /* Duplicate fildes */
96 #define F_GETFD        1 /* Get fildes flags */
97 #define F_SETFD        2 /* Set fildes flags */
98 #define F_GETFL        3 /* Get file flags */
99 #define F_GETXFL        45 /* Get file flags including open-only flags */
100 #define F_SETFL        4 /* Set file flags */
102 /*
103  * Applications that read /dev/mem must be built like the kernel. A
104  * new symbol "KMEMUSER" is defined for this purpose.
105  */
106 #if defined(_KERNEL) || defined(_KMEMUSER)
107 #define F_O_GETTLK     5 /* SVR3 Get file lock (need for rfs, across */
108 /* the wire compatibility */
109 /* clustering: lock id contains both per-node sysid and node id */
110 #define SYSIDMASK      0x0000ffff
111 #define GETSYSID(id)   (id & SYSIDMASK)
112 #define NODEIDMASK     0xffff0000
113 #define BITS_IN_SYSID  16
114 #define GETNLNID(sysid) ((int)((((uint_t)(sysid) & NODEIDMASK) >> \
115     BITS_IN_SYSID))
117 /* Clustering: Macro used for PXFS locks */
118 #define GETPXFSID(sysid) ((int)((((uint_t)(sysid) & NODEIDMASK) >> \
119     BITS_IN_SYSID))
120 #endif /* defined(_KERNEL) */
122 #define F_CHKFL        8 /* Unused */
123 #define F_DUP2FD        9 /* Duplicate fildes at third arg */
124 #define F_DUP2FD_CLOEXEC 36 /* Like F_DUP2FD with O_CLOEXEC set */
125 /* EINVAL is fildes matches arg1 */
126 #define F_DUPFD_CLOEXEC 37 /* Like F_DUPFD with O_CLOEXEC set */
```

```

128 #define F_ISSTREAM      13    /* Is the file desc. a stream ? */
129 #define F_PRIV         15    /* Turn on private access to file */
130 #define F_NPRIV        16    /* Turn off private access to file */
131 #define F_QUOTACTL     17    /* UFS quota call */
132 #define F_BLOCKS       18    /* Get number of BLKSIZE blocks allocated */
133 #define F_BLKSIZE      19    /* Get optimal I/O block size */
134 /*
135  * Numbers 20-22 have been removed and should not be reused.
136  */
137 #define F_GETOWN        23    /* Get owner (socket emulation) */
138 #define F_SETOWN        24    /* Set owner (socket emulation) */
139 #define F_REVOKE        25    /* Object reuse revoke access to file desc. */

141 #define F_HASREMOLECKS 26    /* Does vp have NFS locks; private to lock */
142                               /* manager */

144 /*
145  * Commands that refer to flock structures. The argument types differ between
146  * the large and small file environments; therefore, the #defined values must
147  * as well.
148  * The NBMAND forms are private and should not be used.
149  * The FLOCK forms are also private and should not be used.
150  */

152 #if defined(_LP64) || _FILE_OFFSET_BITS == 32
153 /* "Native" application compilation environment */
154 #define F_SETLK         6     /* Set file lock */
155 #define F_SETLKW        7     /* Set file lock and wait */
156 #define F_ALLOCSP      10    /* Allocate file space */
157 #define F_FREESP        11    /* Free file space */
158 #define F_GETLK         14    /* Get file lock */
159 #define F_SETLK_NBMAND  42    /* private */
160 #if !defined(_STRICT_SYMBOLS)
161 #define F_OFD_GETLK     47    /* Get file lock owned by file */
162 #define F_OFD_SETLK     48    /* Set file lock owned by file */
163 #define F_OFD_SETLKW   49    /* Set file lock owned by file and wait */
164 #define F_FLOCK         53    /* private - set flock owned by file */
165 #define F_FLOCKW        54    /* private - set flock owned by file and wait */
166 #endif /* _STRICT_SYMBOLS */
167 #else
168 /* ILP32 large file application compilation environment version */
169 #define F_SETLK         34    /* Set file lock */
170 #define F_SETLKW        35    /* Set file lock and wait */
171 #define F_ALLOCSP      28    /* Allocate file space */
172 #define F_FREESP        27    /* Free file space */
173 #define F_GETLK         33    /* Get file lock */
174 #define F_SETLK_NBMAND  44    /* private */
175 #if !defined(_STRICT_SYMBOLS)
176 #define F_OFD_GETLK     50    /* Get file lock owned by file */
177 #define F_OFD_SETLK     51    /* Set file lock owned by file */
178 #define F_OFD_SETLKW   52    /* Set file lock owned by file and wait */
179 #define F_FLOCK         55    /* private - set flock owned by file */
180 #define F_FLOCKW        56    /* private - set flock owned by file and wait */
181 #endif /* _STRICT_SYMBOLS */
182 #endif /* !_LP64 || _FILE_OFFSET_BITS == 32 */

184 #define F_ASSOCI_PID    (USHRT_MAX)
185 #define F_DASSOC_PID    (USHRT_MAX - 1)

187 #endif /* !codereview */
188 #if defined(_LARGEFILE64_SOURCE)

190 #if !defined(_LP64) || defined(_KERNEL)
191 /*
192  * transitional large file interface version
193  * These are only valid in a 32 bit application compiled with large files

```

```

194  * option, for source compatibility, the 64-bit versions are mapped back
195  * to the native versions.
196  */
197 #define F_SETLK64       34    /* Set file lock */
198 #define F_SETLKW64      35    /* Set file lock and wait */
199 #define F_ALLOCSP64     28    /* Allocate file space */
200 #define F_FREESP64      27    /* Free file space */
201 #define F_GETLK64       33    /* Get file lock */
202 #define F_SETLK64_NBMAND 44    /* private */
203 #if !defined(_STRICT_SYMBOLS)
204 #define F_OFD_GETLK64   50    /* Get file lock owned by file */
205 #define F_OFD_SETLK64   51    /* Set file lock owned by file */
206 #define F_OFD_SETLKW64 52    /* Set file lock owned by file and wait */
207 #define F_FLOCK64       55    /* private - set flock owned by file */
208 #define F_FLOCKW64      56    /* private - set flock owned by file and wait */
209 #endif /* !_STRICT_SYMBOLS */
210 #else
211 #define F_SETLK64       6     /* Set file lock */
212 #define F_SETLKW64      7     /* Set file lock and wait */
213 #define F_ALLOCSP64     10    /* Allocate file space */
214 #define F_FREESP64      11    /* Free file space */
215 #define F_GETLK64       14    /* Get file lock */
216 #define F_SETLK64_NBMAND 42    /* private */
217 #if !defined(_STRICT_SYMBOLS)
218 #define F_OFD_GETLK64   47    /* Get file lock owned by file */
219 #define F_OFD_SETLK64   48    /* Set file lock owned by file */
220 #define F_OFD_SETLKW64 49    /* Set file lock owned by file and wait */
221 #define F_FLOCK64       53    /* private - set flock owned by file */
222 #define F_FLOCKW64      54    /* private - set flock owned by file and wait */
223 #endif /* !_STRICT_SYMBOLS */
224 #endif /* !_LP64 || _KERNEL */

226 #endif /* _LARGEFILE64_SOURCE */

228 #define F_SHARE         40    /* Set a file share reservation */
229 #define F_UNSHARE       41    /* Remove a file share reservation */
230 #define F_SHARE_NBMAND  43    /* private */

232 #define F_BADFD         46    /* Create Poison FD */

234 /*
235  * File segment locking set data type - information passed to system by user.
236  */

238 /* regular version, for both small and large file compilation environment */
239 typedef struct flock {
240     short  l_type;
241     short  l_whence;
242     off_t  l_start;
243     off_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 } flock_t;

249 #if defined(_SYSCTL32)

251 /* Kernel's view of ILP32 flock structure */

253 typedef struct flock32 {
254     int16_t l_type;
255     int16_t l_whence;
256     off32_t l_start;
257     off32_t l_len;         /* len == 0 means until end of file */
258     int32_t l_sysid;
259     pid32_t l_pid;

```

```

260     int32_t l_pad[4];          /* reserve area */
261 } flock32_t;

263 #endif /* _SYSCALL32 */

265 /* transitional large file interface version */

267 #if defined(_LARGEFILE64_SOURCE)

269 typedef struct flock64 {
270     short l_type;
271     short l_whence;
272     off64_t l_start;
273     off64_t l_len;          /* len == 0 means until end of file */
274     int l_sysid;
275     pid_t l_pid;
276     long l_pad[4];        /* reserve area */
277 } flock64_t;

279 #if defined(_SYSCALL32)

281 /* Kernel's view of ILP32 flock64 */

283 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
284 #pragma pack(4)
285 #endif

287 typedef struct flock64_32 {
288     int16_t l_type;
289     int16_t l_whence;
290     off64_t l_start;
291     off64_t l_len;          /* len == 0 means until end of file */
292     int32_t l_sysid;
293     pid32_t l_pid;
294     int32_t l_pad[4];      /* reserve area */
295 } flock64_32_t;

297 #if _LONG_LONG_ALIGNMENT == 8 && _LONG_LONG_ALIGNMENT_32 == 4
298 #pragma pack()
299 #endif

301 /* Kernel's view of LP64 flock64 */

303 typedef struct flock64_64 {
304     int16_t l_type;
305     int16_t l_whence;
306     off64_t l_start;
307     off64_t l_len;          /* len == 0 means until end of file */
308     int32_t l_sysid;
309     pid32_t l_pid;
310     int64_t l_pad[4];      /* reserve area */
311 } flock64_64_t;

313 #endif /* _SYSCALL32 */

315 #endif /* _LARGEFILE64_SOURCE */

317 #if defined(_KERNEL) || defined(_KMEMUSER)
318 /* SVr3 flock type; needed for rfs across the wire compatibility */
319 typedef struct o_flock {
320     int16_t l_type;
321     int16_t l_whence;
322     int32_t l_start;
323     int32_t l_len;          /* len == 0 means until end of file */
324     int16_t l_sysid;
325     int16_t l_pid;

```

```

326 } o_flock_t;
327 #endif /* defined(_KERNEL) */

329 /*
330 * File segment locking types.
331 */
332 #define F_RDLCK      01    /* Read lock */
333 #define F_WRLCK     02    /* Write lock */
334 #define F_UNLCK     03    /* Remove lock(s) */
335 #define F_UNLKSYS   04    /* remove remote locks for a given system */

337 /*
338 * POSIX constants
339 */

341 /* Mask for file access modes */
342 #define O_ACCMODE    (O_SEARCH | O_EXEC | O_WRONLY)
343 #define FD_CLOEXEC   1    /* close on exec flag */

345 /*
346 * DIRECTIO
347 */
348 #if defined(__EXTENSIONS__) || !defined(__XOPEN_OR_POSIX)
349 #define DIRECTIO_OFF (0)
350 #define DIRECTIO_ON  (1)
351 #endif

352 /*
353 * File share reservation type
354 */
355 typedef struct fshare {
356     short f_access;
357     short f_deny;
358     int f_id;
359 } fshare_t;

361 /*
362 * f_access values
363 */
364 #define F_RDACC      0x1    /* Read-only share access */
365 #define F_WRACC      0x2    /* Write-only share access */
366 #define F_RWACC      0x3    /* Read-Write share access */
367 #define F_RMACC      0x4    /* private flag: Delete share access */
368 #define F_MDACC      0x20   /* private flag: Metadata share access */

370 /*
371 * f_deny values
372 */
373 #define F_NODNY      0x0    /* Don't deny others access */
374 #define F_RDDNY      0x1    /* Deny others read share access */
375 #define F_WRDNY      0x2    /* Deny others write share access */
376 #define F_RWDNY      0x3    /* Deny others read or write share access */
377 #define F_RMDNY      0x4    /* private flag: Deny delete share access */
378 #define F_COMPAT      0x8    /* Set share to old DOS compatibility mode */
379 #define F_MANDDNY     0x10   /* private flag: mandatory enforcement */
380 #endif /* defined(__EXTENSIONS__) || !defined(__XOPEN_OR_POSIX) */

382 /*
383 * Special flags for functions such as openat(), fstatat()...
384 */
385 #if !defined(__XOPEN_OR_POSIX) || defined(_ATFILE_SOURCE) || \
386     defined(__EXTENSIONS__) \
387     /* || defined(_XPG7) */
388 #define AT_FDCWD      0xffd19553
389 #define AT_SYMLINK_NOFOLLOW 0x1000
390 #define AT_SYMLINK_FOLLOW 0x2000 /* only for linkat() */
391 #define AT_REMOVEDIR 0x1

```

```
392 #define _AT_TRIGGER          0x2
393 #define AT_EACCESS           0x4    /* use EUID/EGID for access */
394 #endif

396 #if !defined(__XOPEN_OR_POSIX) || defined(_XPG6) || defined(__EXTENSIONS__)
397 /* advice for posix_fadvise */
398 #define POSIX_FADV_NORMAL    0
399 #define POSIX_FADV_RANDOM    1
400 #define POSIX_FADV_SEQUENTIAL 2
401 #define POSIX_FADV_WILLNEED 3
402 #define POSIX_FADV_DONTNEED 4
403 #define POSIX_FADV_NOREUSE   5
404 #endif

406 #ifdef __cplusplus
407 }
408 #endif

410 #endif /* _SYS_FCNTL_H */
```

```

*****
7516 Wed Jul 13 01:32:42 2016
new/usr/src/uts/common/sys/file.h
XXXX adding PID information to netstat output
*****
_____unchanged_portion_omitted_____

81 /* f_flag */

83 #define FOPEN          0xffffffff
84 #define FREAD          0x01 /* <sys/aioch.h> LIO_READ must be identical */
85 #define FWRITE         0x02 /* <sys/aioch.h> LIO_WRITE must be identical */
86 #define FNDELAY        0x04
87 #define FAPPEND        0x08
88 #define FSYNC          0x10 /* file (data+inode) integrity while writing */
89 #define FREVOKED       0x20 /* Object reuse Revoked file */
90 #define FDSYNC          0x40 /* file data only integrity while writing */
91 #define FNONBLOCK      0x80

93 #define FMASK          0xa0ff /* all flags that can be changed by F_SETFL */

95 /* open-only modes */

97 #define FCREAT         0x0100
98 #define FTRUNC         0x0200
99 #define FEXCL          0x0400
100 #define FASYNC         0x1000 /* asyncio in progress pseudo flag */
101 #define FOFFMAX        0x2000 /* large file */
102 #define FXATTR         0x4000 /* open as extended attribute */
103 #define FNOCTTY        0x0800
104 #define FRSYNC         0x8000 /* sync read operations at same level of */
105 /* integrity as specified for writes by */
106 /* FSYNC and FDSYNC flags */

108 #define FNODSYNC       0x10000 /* fsync pseudo flag */

110 #define FNOFOLLOW      0x20000 /* don't follow symlinks */
111 #define FNOLINKS       0x40000 /* don't allow multiple hard links */
112 #define FIGNORECASE    0x80000 /* request case-insensitive lookups */
113 #define FXATTRDIROPEN  0x100000 /* only opening hidden attribute directory */

115 /* f_flag2 (open-only) */

117 #define FSEARCH        0x200000 /* O_SEARCH = 0x200000 */
118 #define FEXEC          0x400000 /* O_EXEC = 0x400000 */

120 #define FCLOEXEC       0x800000 /* O_CLOEXEC = 0x800000 */

122 #ifndef _KERNEL

124 /*
125 * This is a flag that is set on f_flag2, but is never user-visible
126 */
127 #define FEPOLLED       0x8000

129 /*
130 * Fake flags for driver ioctl calls to inform them of the originating
131 * process' model. See <sys/model.h>
132 *
133 * Part of the Solaris 2.6+ DDI/DKI
134 */
135 #define FMODELS DATAMODEL_MASK /* Note: 0x0fff00000 */
136 #define FILP32 DATAMODEL_ILP32
137 #define FLP64 DATAMODEL_LP64
138 #define FNATIVE DATAMODEL_NATIVE

```

```

140 /*
141 * Large Files: The macro gets the offset maximum (refer to LFS API doc)
142 * corresponding to a file descriptor. We had the choice of storing
143 * this value in file descriptor. Right now we only have two
144 * offset maximums one if MAXOFF_T and other is MAXOFFSET_T. It is
145 * inefficient to store these two values in a separate member in
146 * file descriptor. To avoid wasting spaces we define this macro.
147 * The day there are more than two offset maximum we may want to
148 * rewrite this macro.
149 */

151 #define OFFSET_MAX(fd) ((fd->f_flag & FOFFMAX) ? MAXOFFSET_T : MAXOFF32_T)

153 /*
154 * Fake flag => internal ioctl call for layered drivers.
155 * Note that this flag deliberately *won't* fit into
156 * the f_flag field of a file_t.
157 *
158 * Part of the Solaris 2.x DDI/DKI.
159 */
160 #define FKIOCTL          0x80000000 /* ioctl addresses are from kernel */

162 /*
163 * Fake flag => this time to specify that the open(9E)
164 * comes from another part of the kernel, not userland.
165 *
166 * Part of the Solaris 2.x DDI/DKI.
167 */
168 #define FKLVR           0x40000000 /* layered driver call */

170 #endif /* _KERNEL */

172 /* miscellaneous defines */

174 #ifndef L_SET
175 #define L_SET 0 /* for lseek */
176 #endif /* L_SET */

178 /*
179 * For flock(3C). These really don't belong here but for historical reasons
180 * the interface defines them to be here.
181 */
182 #define LOCK_SH 1
183 #define LOCK_EX 2
184 #define LOCK_NB 4
185 #define LOCK_UN 8

187 #if !defined(_STRICT_SYMBOLS)
188 extern int flock(int, int);
189 #endif

191 #if defined(_KERNEL)

193 /*
194 * Routines dealing with user per-open file flags and
195 * user open files.
196 */
197 struct proc; /* forward reference for function prototype */
198 struct vnodeops;
199 struct vattr;

201 extern file_t *getf(int);
202 extern void releasef(int);
203 extern void areleasef(int, uf_info_t *);
204 #ifndef _BOOT
205 extern void closeall(uf_info_t *);

```

```
206 #endif
207 extern void flist_fork(proc_t *, proc_t *);
207 extern void flist_fork(uf_info_t *, uf_info_t *);
208 extern int closef(file_t *);
209 extern int closeandsetf(int, file_t *);
210 extern int ufalloc_file(int, file_t *);
211 extern int ufalloc(int);
212 extern int ufcalloc(struct proc *, uint_t);
213 extern int falloc(struct vnode *, int, file_t **, int *);
214 extern void finit(void);
215 extern void unfalloc(file_t *);
216 extern void setf(int, file_t *);
217 extern int f_getfd_error(int, int *);
218 extern char f_getfd(int);
219 extern int f_setfd_error(int, int);
220 extern void f_setfd(int, char);
221 extern int f_getfl(int, int *);
222 extern int f_badfd(int, int *, int);
223 extern int fassign(struct vnode **, int, int *);
224 extern void fcnt_add(uf_info_t *, int);
225 extern void close_exec(uf_info_t *);
226 extern void clear_stale_fd(void);
227 extern void clear_active_fd(int);
228 extern void free_afd(afd_t *afd);
229 extern int fgetstartvp(int, char *, struct vnode **);
230 extern int fsetattr(int, char *, int, struct vattr *);
231 extern int fisopen(struct vnode *);
232 extern void delfpollinfo(int);
233 extern void addfpollinfo(int);
234 extern int sock_getfasync(struct vnode *);
235 extern int files_can_change_zones(void);
236 #ifdef DEBUG
237 /* The following functions are only used in ASSERT()s */
238 extern void checkwfdlist(struct vnode *, fpollinfo_t *);
239 extern void checkfpollinfo(void);
240 extern int infpollinfo(int);
241 #endif /* DEBUG */

243 #endif /* defined(_KERNEL) */

245 #ifdef __cplusplus
246 }
  unchanged_portion_omitted

```

```
*****
1539 Wed Jul 13 01:32:42 2016
new/usr/src/uts/common/sys/pidnode.h
XXXX adding PID information to netstat output
*****
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2015 Mohamed A. Khalfella <khalfella@gmail.com>
14  */

16 #ifndef _SYS_PIDNODE_H
17 #define _SYS_PIDNODE_H

19 #include <sys/avl.h>

21 #ifdef __cplusplus
22 extern "C" {
23 #endif

25 /*
26  * Network connections initiated/accepted by user processes are either
27  * socket based connections or stream-based connections. conn_pid_info_t
28  * expose the type of the connection in the cpi_contents field.
29  */

31 #define CONN_PID_INFO_NON      0      /* terminated process/kernel sockets */
32 #define CONN_PID_INFO_SOCK    1      /* socket network connection */
33 #define CONN_PID_INFO_STREAM  2      /* stream network connection */

35 typedef struct conn_pid_info_s {
36     uint16_t      cpi_contents; /* CONN_PID_INFO_* */
37     uint32_t      cpi_pids_cnt; /* # of elements in cpi_pids */
38     uint32_t      cpi_tot_size; /* total size of hdr + pids */
39     pid_t         cpi_pids[1]; /* variable length array of pids */
40 } conn_pid_info_t;

42 #if defined(_KERNEL)

44 typedef struct pid_node_s {
45     avl_node_t     pn_ref_link;
46     uint32_t       pn_count;
47     pid_t          pn_pid;
48 } pid_node_t;

50 extern int pid_node_comparator(const void *, const void *);

52 #endif /* defined(_KERNEL) */

54 #ifdef __cplusplus
55 }
56 #endif

58 #endif /* _SYS_PIDNODE_H */
59 #endif /* !codereview */
```

```

*****
      8332 Wed Jul 13 01:32:42 2016
new/usr/src/uts/common/sys/socket_proto.h
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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
23 */

25 #ifndef _SYS_SOCKET_PROTO_H_
26 #define _SYS_SOCKET_PROTO_H_

28 #ifdef __cplusplus
29 extern "C" {
30 #endif

32 #include <sys/socket.h>
33 #include <sys/pidnode.h>
34 #endif /* ! codereview */

36 /*
37  * Generation count
38  */
39 typedef uint64_t sock_connid_t;

41 #define SOCK_CONNID_INIT(id) { \
42     (id) = 0; \
43 }
44 #define SOCK_CONNID_BUMP(id)      (++(id))
45 #define SOCK_CONNID_LT(id1, id2) ((int64_t)((id1)-(id2)) < 0)

47 /* Socket protocol properties */
48 struct sock_proto_props {
49     uint_t  sopp_flags;           /* options to set */
50     ushort_t sopp_wroff;         /* write offset */
51     ssize_t  sopp_txhiwat;       /* tx hi water mark */
52     ssize_t  sopp_txlowat;       /* tx lo water mark */
53     ssize_t  sopp_rxhiwat;       /* rcv high water mark */
54     ssize_t  sopp_rxlowat;       /* rcv low water mark */
55     ssize_t  sopp_maxblk;        /* maximum message block size */
56     ssize_t  sopp_maxpsz;        /* maximum packet size */
57     ssize_t  sopp_minpsz;        /* minimum packet size */
58     ushort_t sopp_tail;         /* space available at the end */
59     uint_t   sopp_zcopyflag;     /* zero copy flag */
60     boolean_t sopp_oobinline;    /* OOB inline */
61     uint_t   sopp_rcvtimer;     /* delayed rcv notification (time) */

```

```

62     uint32_t sopp_rcvthresh;     /* delayed rcv notification (bytes) */
63     socklen_t sopp_maxaddrlen;   /* maximum size of protocol address */
64     boolean_t sopp_loopback;     /* loopback connection */
65 };

67 /* flags to determine which socket options are set */
68 #define SOCKOPT_WROFF      0x0001 /* set write offset */
69 #define SOCKOPT_RCVHIWAT  0x0002 /* set read side high water */
70 #define SOCKOPT_RCVLOWAT  0x0004 /* set read side low water */
71 #define SOCKOPT_MAXBLK    0x0008 /* set maximum message block size */
72 #define SOCKOPT_TAIL      0x0010 /* set the extra allocated space */
73 #define SOCKOPT_ZCOPY     0x0020 /* set/unset zero copy for sendfile */
74 #define SOCKOPT_MAXPSZ    0x0040 /* set maxpsz for protocols */
75 #define SOCKOPT_OOBINLINE 0x0080 /* set oob inline processing */
76 #define SOCKOPT_RCVTIMER  0x0100
77 #define SOCKOPT_RCVTHRESH 0x0200
78 #define SOCKOPT_MAXADDRLEN 0x0400 /* set max address length */
79 #define SOCKOPT_MINPSZ    0x0800 /* set minpsz for protocols */
80 #define SOCKOPT_LOOPBACK  0x1000 /* set loopback */

82 #define IS_SO_OOB_INLINE(so)  ((so)->so_proto_props.sopp_oobinline)

84 #ifdef _KERNEL

86 struct T_capability_ack;

88 typedef struct sock_upcalls_s sock_upcalls_t;
89 typedef struct sock_downcalls_s sock_downcalls_t;

91 /*
92  * Upcall and downcall handle for sockfns and transport layer.
93  */
94 typedef struct __sock_upper_handle *sock_upper_handle_t;
95 typedef struct __sock_lower_handle *sock_lower_handle_t;

97 struct sock_downcalls_s {
98     void (*sd_activate)(sock_lower_handle_t, sock_upper_handle_t,
99         sock_upcalls_t *, int, cred_t *);
100     int (*sd_accept)(sock_lower_handle_t, sock_lower_handle_t,
101         sock_upper_handle_t, cred_t *);
102     int (*sd_bind)(sock_lower_handle_t, struct sockaddr *, socklen_t,
103         cred_t *);
104     int (*sd_listen)(sock_lower_handle_t, int, cred_t *);
105     int (*sd_connect)(sock_lower_handle_t, const struct sockaddr *,
106         socklen_t, sock_connid_t *, cred_t *);
107     int (*sd_getpeername)(sock_lower_handle_t, struct sockaddr *,
108         socklen_t *, cred_t *);
109     int (*sd_getsockname)(sock_lower_handle_t, struct sockaddr *,
110         socklen_t *, cred_t *);
111     int (*sd_getsockopt)(sock_lower_handle_t, int, int, void *,
112         socklen_t *, cred_t *);
113     int (*sd_setsockopt)(sock_lower_handle_t, int, int, const void *,
114         socklen_t, cred_t *);
115     int (*sd_send)(sock_lower_handle_t, mblk_t *, struct nmsg_hdr *,
116         cred_t *);
117     int (*sd_send_uio)(sock_lower_handle_t, uio_t *, struct nmsg_hdr *,
118         cred_t *);
119     int (*sd_rcv_uio)(sock_lower_handle_t, uio_t *, struct nmsg_hdr *,
120         cred_t *);
121     short (*sd_poll)(sock_lower_handle_t, short, int, cred_t *);
122     int (*sd_shutdown)(sock_lower_handle_t, int, cred_t *);
123     void (*sd_clr_flowctrl)(sock_lower_handle_t);
124     int (*sd_ioctl)(sock_lower_handle_t, int, intptr_t, int,
125         int32_t *, cred_t *);
126     int (*sd_close)(sock_lower_handle_t, int, cred_t *);
127 };

```

```

129 typedef sock_lower_handle_t (*so_proto_create_func_t)(int, int, int,
130 sock_downcalls_t **, uint_t *, int *, int, cred_t *);

132 typedef struct sock_quiesce_arg {
133     mblk_t *soqa_exdata_mp;
134     mblk_t *soqa_urgmark_mp;
135 } sock_quiesce_arg_t;
136 typedef mblk_t *(*so_proto_quiesced_cb_t)(sock_upper_handle_t,
137 sock_quiesce_arg_t *, struct T_capability_ack *, struct sockaddr *,
138 socklen_t, struct sockaddr *, socklen_t, short);
139 typedef int (*so_proto_fallback_func_t)(sock_lower_handle_t, queue_t *,
140 boolean_t, so_proto_quiesced_cb_t, sock_quiesce_arg_t *);

142 /*
143  * These functions return EOPNOTSUPP and are intended for the sockfs
144  * developer that doesn't wish to supply stubs for every function themselves.
145  */
146 extern int sock_accept_notsupp(sock_lower_handle_t, sock_lower_handle_t,
147 sock_upper_handle_t, cred_t *);
148 extern int sock_bind_notsupp(sock_lower_handle_t, struct sockaddr *,
149 socklen_t, cred_t *);
150 extern int sock_listen_notsupp(sock_lower_handle_t, int, cred_t *);
151 extern int sock_connect_notsupp(sock_lower_handle_t,
152 const struct sockaddr *, socklen_t, sock_connid_t *, cred_t *);
153 extern int sock_getpeername_notsupp(sock_lower_handle_t, struct sockaddr *,
154 socklen_t *, cred_t *);
155 extern int sock_getsockname_notsupp(sock_lower_handle_t, struct sockaddr *,
156 socklen_t *, cred_t *);
157 extern int sock_getsockopt_notsupp(sock_lower_handle_t, int, int, void *,
158 socklen_t *, cred_t *);
159 extern int sock_setsockopt_notsupp(sock_lower_handle_t, int, int,
160 const void *, socklen_t, cred_t *);
161 extern int sock_send_notsupp(sock_lower_handle_t, mblk_t *,
162 struct nmsgHdr *, cred_t *);
163 extern int sock_send_uio_notsupp(sock_lower_handle_t, uio_t *,
164 struct nmsgHdr *, cred_t *);
165 extern int sock_recv_uio_notsupp(sock_lower_handle_t, uio_t *,
166 struct nmsgHdr *, cred_t *);
167 extern short sock_poll_notsupp(sock_lower_handle_t, short, int, cred_t *);
168 extern int sock_shutdown_notsupp(sock_lower_handle_t, int, cred_t *);
169 extern void sock_clr_flowctrl_notsupp(sock_lower_handle_t);
170 extern int sock_ioctl_notsupp(sock_lower_handle_t, int, intptr_t, int,
171 int32_t *, cred_t *);
172 extern int sock_close_notsupp(sock_lower_handle_t, int, cred_t *);

174 /*
175  * Upcalls and related information
176  */

178 /*
179  * su_opctl() actions
180  */
181 typedef enum sock_opctl_action {
182     SOCK_OPCTL_ENAB_ACCEPT = 0,
183     SOCK_OPCTL_SHUT_SEND,
184     SOCK_OPCTL_SHUT_RECV
185 } sock_opctl_action_t;

187 struct sock_upcalls_s {
188     sock_upper_handle_t (*su_newconn)(sock_upper_handle_t,
189 sock_lower_handle_t, sock_downcalls_t *, cred_t *, pid_t,
190 sock_upcalls_t **);
191     void (*su_connected)(sock_upper_handle_t, sock_connid_t, cred_t *,
192 pid_t);
193     int (*su_disconnected)(sock_upper_handle_t, sock_connid_t, int);

```

```

194     void (*su_opctl)(sock_upper_handle_t, sock_opctl_action_t,
195 uintptr_t);
196     ssize_t (*su_recv)(sock_upper_handle_t, mblk_t *, size_t, int,
197 int *, boolean_t *);
198     void (*su_set_proto_props)(sock_upper_handle_t,
199 struct sock_proto_props *);
200     void (*su_txq_full)(sock_upper_handle_t, boolean_t);
201     void (*su_signal_oob)(sock_upper_handle_t, ssize_t);
202     void (*su_zcopy_notify)(sock_upper_handle_t);
203     void (*su_set_error)(sock_upper_handle_t, int);
204     void (*su_closed)(sock_upper_handle_t);
205     mblk_t *(*su_get_sock_pid_mblk)(sock_upper_handle_t);
206 #endif /* ! codereview */
207 };

209 #define SOCK_UC_VERSION      sizeof (sock_upcalls_t)
210 #define SOCK_DC_VERSION      sizeof (sock_downcalls_t)

212 #define SOCKET_RECVHIWATER  (48 * 1024)
213 #define SOCKET_RECVLOWATER  1024

215 #define SOCKET_NO_RCVTIMER   0
216 #define SOCKET_TIMER_INTERVAL 50

218 #endif /* _KERNEL */

220 #ifdef __cplusplus
221 }
222 #endif

224 #endif /* _SYS_SOCKET_PROTO_H */

```

```

*****
35592 Wed Jul 13 01:32:42 2016
new/usr/src/uts/common/sys/socketvar.h
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 1996, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T      */
27 /*      All Rights Reserved      */

29 /*
30  * University Copyright- Copyright (c) 1982, 1986, 1988
31  * The Regents of the University of California
32  * All Rights Reserved
33  *
34  * University Acknowledgment- Portions of this document are derived from
35  * software developed by the University of California, Berkeley, and its
36  * contributors.
37  */
38 /*
39  * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
40  */

42 #ifndef _SYS_SOCKETVAR_H
43 #define _SYS_SOCKETVAR_H

45 #include <sys/types.h>
46 #include <sys/stream.h>
47 #include <sys/t_lock.h>
48 #include <sys/cred.h>
49 #include <sys/pidnode.h>
50 #endif /* ! codereview */
51 #include <sys/vnode.h>
52 #include <sys/file.h>
53 #include <sys/param.h>
54 #include <sys/zone.h>
55 #include <sys/sdt.h>
56 #include <sys/modctl.h>
57 #include <sys/atomic.h>
58 #include <sys/socket.h>
59 #include <sys/ksocket.h>
60 #include <sys/kstat.h>

```

```

62 #ifndef _KERNEL
63 #include <sys/vfs_opreg.h>
64 #endif

66 #ifndef __cplusplus
67 extern "C" {
68 #endif

70 /*
71  * Internal representation of the address used to represent addresses
72  * in the loopback transport for AF_UNIX. While the sockaddr_un is used
73  * as the sockfs layer address for AF_UNIX the pathnames contained in
74  * these addresses are not unique (due to relative pathnames) thus can not
75  * be used in the transport.
76  *
77  * The transport level address consists of a magic number (used to separate the
78  * name space for specific and implicit binds). For a specific bind
79  * this is followed by a "vnode *" which ensures that all specific binds
80  * have a unique transport level address. For implicit binds the latter
81  * part of the address is a byte string (of the same length as a pointer)
82  * that is assigned by the loopback transport.
83  *
84  * The uniqueness assumes that the loopback transport has a separate namespace
85  * for sockets in order to avoid name conflicts with e.g. TLI use of the
86  * same transport.
87  */
88 struct so_ux_addr {
89     void *soua_vp;          /* vnode pointer or assigned by tl */
90     uint_t soua_magic;     /* See below */
91 };

93 #define SOU_MAGIC_EXPLICIT 0x75787670 /* "uxvp" */
94 #define SOU_MAGIC_IMPLICIT 0x616e666e /* "anon" */

96 struct sockaddr_ux {
97     sa_family_t sou_family; /* AF_UNIX */
98     struct so_ux_addr sou_addr;
99 };

101 #if defined(_KERNEL) || defined(_KMEMUSER)

103 #include <sys/socket_proto.h>

105 typedef struct sonodeops sonodeops_t;
106 typedef struct sonode sonode_t;

108 struct sodirect_s;

110 /*
111  * The sonode represents a socket. A sonode never exist in the file system
112  * name space and can not be opened using open() - only the socket, socketpair
113  * and accept calls create sonodes.
114  *
115  * The locking of sockfs uses the so_lock mutex plus the SOLOCKED and
116  * SOREADLOCKED flags in so_flag. The mutex protects all the state in the
117  * sonode. It is expected that the underlying transport protocol serializes
118  * socket operations, so sockfs will not normally not single-thread
119  * operations. However, certain sockets, including TPI based ones, can only
120  * handle one control operation at a time. The SOLOCKED flag is used to
121  * single-thread operations from sockfs users to prevent e.g. multiple bind()
122  * calls to operate on the same sonode concurrently. The SOREADLOCKED flag is
123  * used to ensure that only one thread sleeps in kstrgetmsg for a given
124  * sonode. This is needed to ensure atomic operation for things like
125  * MSG_WAITALL.
126  *
127  * The so_fallback_rwlock is used to ensure that for sockets that can

```

```

128 * fall back to TPI, the fallback is not initiated until all pending
129 * operations have completed.
130 *
131 * Note that so_lock is sometimes held across calls that might go to sleep
132 * (kmem_alloc and soallocproto*). This implies that no other lock in
133 * the system should be held when calling into sockfs; from the system call
134 * side or from strput (in case of TPI based sockets). If locks are held
135 * while calling into sockfs the system might hang when running low on memory.
136 */
137 struct sonode {
138     struct vnode *so_vnode; /* vnode associated with this sonode */
139
140     sonodeops_t *so_ops; /* operations vector for this sonode */
141     void *so_priv; /* sonode private data */
142
143     krwlock_t so_fallback_rwlock;
144     kmutex_t so_lock; /* protects sonode fields */
145
146     kcondvar_t so_state_cv; /* synchronize state changes */
147     kcondvar_t so_single_cv; /* wait due to SOLOCKED */
148     kcondvar_t so_read_cv; /* wait due to SOREADLOCKED */
149
150     /* These fields are protected by so_lock */
151
152     uint_t so_state; /* internal state flags SS_*, below */
153     uint_t so_mode; /* characteristics on socket. SM_* */
154     ushort_t so_flag; /* flags, see below */
155     int so_count; /* count of opened references */
156
157     sock_connid_t so_proto_connid; /* protocol generation number */
158
159     ushort_t so_error; /* error affecting connection */
160
161     struct sockparams *so_sockparams; /* vnode or socket module */
162     /* Needed to recreate the same socket for accept */
163     short so_family;
164     short so_type;
165     short so_protocol;
166     short so_version; /* From so_socket call */
167
168     /* Accept queue */
169     kmutex_t so_acceptq_lock; /* protects accept queue */
170     list_t so_acceptq_list; /* pending conns */
171     list_t so_acceptq_defer; /* deferred conns */
172     list_node_t so_acceptq_node; /* acceptq list node */
173     unsigned int so_acceptq_len; /* # of conns (both lists) */
174     unsigned int so_backlog; /* Listen backlog */
175     kcondvar_t so_acceptq_cv; /* wait for new conn. */
176     struct sonode *so_listener; /* parent socket */
177
178     /* Options */
179     short so_options; /* From socket call, see socket.h */
180     struct linger so_linger; /* SO_LINGER value */
181 #define so_sndbuf so_proto_props.sopp_txhiwat /* SO_SNDBUF value */
182 #define so_sndlowat so_proto_props.sopp_txlowat /* tx low water mark */
183 #define so_rcvbuf so_proto_props.sopp_rxhiwat /* SO_RCVBUF value */
184 #define so_rcvlowat so_proto_props.sopp_rxlowat /* rx low water mark */
185 #define so_max_addr_len so_proto_props.sopp_maxaddrlen
186 #define so_minpsz so_proto_props.sopp_minpsz
187 #define so_maxpsz so_proto_props.sopp_maxpsz
188
189     int so_xpg_rcvbuf; /* SO_RCVBUF value for XPG4 socket */
190     clock_t so_sndtimeo; /* send timeout */
191     clock_t so_rcvtimeo; /* recv timeout */
192
193     mblk_t *so_oobmsg; /* outofline oob data */

```

```

194     ssize_t so_oobmark; /* offset of the oob data */
195
196     pid_t so_pgrp; /* pgrp for signals */
197
198     cred_t *so_peercred; /* connected socket peer cred */
199     pid_t so_cpuid; /* connected socket peer cached pid */
200     zoneid_t so_zoneid; /* opener's zoneid */
201
202     struct pollhead so_poll_list; /* common pollhead */
203     short so_pollev; /* events that should be generated */
204
205     /* Receive */
206     unsigned int so_rcv_queued; /* # bytes on both rcv lists */
207     mblk_t *so_rcv_q_head; /* processing/copyout rcv queue */
208     mblk_t *so_rcv_q_last_head;
209     mblk_t *so_rcv_head; /* protocol prequeue */
210     mblk_t *so_rcv_last_head; /* last mblk in b_next chain */
211     kcondvar_t so_rcv_cv; /* wait for data */
212     uint_t so_rcv_wanted; /* # of bytes wanted by app */
213     timeout_id_t so_rcv_timer_tid;
214
215 #define so_rcv_thresh so_proto_props.sopp_rcvthresh
216 #define so_rcv_timer_interval so_proto_props.sopp_rcvtimer
217
218     kcondvar_t so_snd_cv; /* wait for snd buffers */
219     uint32_t
220         so_snd_qfull: 1, /* Transmit full */
221         so_rcv_wakeup: 1,
222         so_snd_wakeup: 1,
223         so_not_str: 1, /* B_TRUE if not streams based socket */
224         so_pad_to_bit_31: 28;
225
226     /* Communication channel with protocol */
227     sock_lower_handle_t so_proto_handle;
228     sock_downcalls_t *so_downcalls;
229
230     struct sock_proto_props so_proto_props; /* protocol settings */
231     boolean_t so_flowctrlrd; /* Flow controlled */
232     uint_t so_copyflag; /* Copy related flag */
233     kcondvar_t so_copy_cv; /* Copy cond variable */
234
235     /* kernel sockets */
236     ksocket_callbacks_t so_ksock_callbacks;
237     void *so_ksock_cb_arg; /* callback argument */
238     kcondvar_t so_closing_cv;
239
240     /* != NULL for sodirect enabled socket */
241     struct sodirect_s *so_direct;
242
243     /* socket filters */
244     uint_t so_filter_active; /* # of active fil */
245     uint_t so_filter_tx; /* pending tx ops */
246     struct sof_instance *so_filter_top; /* top of stack */
247     struct sof_instance *so_filter_bottom; /* bottom of stack */
248     clock_t so_filter_defertime; /* time when deferred */
249
250     /* pid tree */
251     avl_tree_t so_pid_tree;
252     kmutex_t so_pid_tree_lock;
253 #endif /* ! codereview */
254 };
255
256 #define SO_HAVE_DATA(so) \
257     /* \
258     * For the (tid == 0) case we must check so_rcv_{q,}head \
259     * 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|POLLRDNRN|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_SAVED_EOR        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.

```

```

392 */
393 typedef struct sonode *(*so_create_func_t)(struct sockparams *,
394     int, int, int, int, int, int *, cred_t *);
395 typedef void (*so_destroy_func_t)(struct sonode *);

397 /* STREAM device information */
398 typedef struct sdev_info {
399     char    *sd_devpath;
400     int     sd_devpathlen; /* Is 0 if sp_devpath is a static string */
401     vnode_t *sd_vnode;
402 } sdev_info_t;

404 #define SOCKMOD_VERSION_1    1
405 #define SOCKMOD_VERSION    2

407 /* name of the TPI pseudo socket module */
408 #define SOTPI_SMOD_NAME    "socktpi"

410 typedef struct __smod_priv_s {
411     so_create_func_t    smodp_sock_create_func;
412     so_destroy_func_t   smodp_sock_destroy_func;
413     so_proto_fallback_func_t smodp_proto_fallback_func;
414     const char          *smodp_fallback_devpath_v4;
415     const char          *smodp_fallback_devpath_v6;
416 } __smod_priv_t;

418 /*
419  * Socket module register information
420  */
421 typedef struct smod_reg_s {
422     int         smod_version;
423     char        *smod_name;
424     size_t      smod_uc_version;
425     size_t      smod_dc_version;
426     so_proto_create_func_t  smod_proto_create_func;

428     /* __smod_priv_data must be NULL */
429     __smod_priv_t    *__smod_priv;
430 } smod_reg_t;

432 /*
433  * Socket module information
434  */
435 typedef struct smod_info {
436     int         smod_version;
437     char        *smod_name;
438     uint_t      smod_refcnt; /* # of entries */
439     size_t      smod_uc_version; /* upcall version */
440     size_t      smod_dc_version; /* down call version */
441     so_proto_create_func_t  smod_proto_create_func;
442     so_proto_fallback_func_t smod_proto_fallback_func;
443     const char  *smod_fallback_devpath_v4;
444     const char  *smod_fallback_devpath_v6;
445     so_create_func_t    smod_sock_create_func;
446     so_destroy_func_t   smod_sock_destroy_func;
447     list_node_t    smod_node;
448 } smod_info_t;

450 typedef struct sockparams_stats {
451     kstat_named_t    sps_nfallback; /* # of fallbacks to TPI */
452     kstat_named_t    sps_nactive; /* # of active sockets */
453     kstat_named_t    sps_ncreate; /* total # of created sockets */
454 } sockparams_stats_t;

456 /*
457  * sockparams

```

```

458 *
459 * Used for mapping family/type/protocol to a socket module or STREAMS device
460 */
461 struct sockparams {
462     /*
463      * The family, type, protocol, sdev_info and smod_name are
464      * set when the entry is created, and they will never change
465      * thereafter.
466      */
467     int         sp_family;
468     int         sp_type;
469     int         sp_protocol;

471     sdev_info_t    sp_sdev_info; /* STREAM device */
472     char          *sp_smod_name; /* socket module name */

474     kmutex_t      sp_lock; /* lock for refcnt and smod_info */
475     uint64_t      sp_refcnt; /* entry reference count */
476     smod_info_t    *sp_smod_info; /* socket module */

478     sockparams_stats_t sp_stats;
479     kstat_t        *sp_kstat;

481     /*
482      * The entries below are only modified while holding
483      * sockconf_lock as a writer.
484      */
485     int         sp_flags; /* see below */
486     list_node_t sp_node;

488     list_t       sp_auto_filters; /* list of automatic filters */
489     list_t       sp_prog_filters; /* list of programmatic filters */
490 };

492 struct sof_entry;

494 typedef struct sp_filter {
495     struct sof_entry *spf_filter;
496     list_node_t      spf_node;
497 } sp_filter_t;

500 /*
501  * sockparams flags
502  */
503 #define SOCKPARAMS_EPHEMERAL    0x1 /* temp. entry, not on global list */

505 extern void sockparams_init(void);
506 extern struct sockparams *sockparams_hold_ephemeral_bydev(int, int, int,
507     const char *, int, int *);
508 extern struct sockparams *sockparams_hold_ephemeral_bymod(int, int, int,
509     const char *, int, int *);
510 extern void sockparams_ephemeral_drop_last_ref(struct sockparams *);

512 extern struct sockparams *sockparams_create(int, int, int, char *, char *, int,
513     int, int, int *);
514 extern void sockparams_destroy(struct sockparams *);
515 extern int sockparams_add(struct sockparams *);
516 extern int sockparams_delete(int, int, int);
517 extern int sockparams_new_filter(struct sof_entry *);
518 extern void sockparams_filter_cleanup(struct sof_entry *);
519 extern int sockparams_copyout_socktable(uintptr_t);

521 extern void smod_init(void);
522 extern void smod_add(smod_info_t *);
523 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  * Decrease 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(lm).
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 };

```

```

657 #define READ_OP          1
658 #define SNFQ_TIMEOUT      (60 * 5 * hz) /* 5 minutes */

660 /* Socket network operations switch */
661 struct sonodeops {
662     int      (*sop_init)(struct sonode *, struct sonode *, cred_t *,
663                        int);
664     int      (*sop_accept)(struct sonode *, int, cred_t *, struct sonode **);
665     int      (*sop_bind)(struct sonode *, struct sockaddr *, socklen_t,
666                        int, cred_t *);
667     int      (*sop_listen)(struct sonode *, int, cred_t *);
668     int      (*sop_connect)(struct sonode *, struct sockaddr *,
669                        socklen_t, int, int, cred_t *);
670     int      (*sop_recvmmsg)(struct sonode *, struct msghdr *,
671                        struct uio *, cred_t *);
672     int      (*sop_sendmsg)(struct sonode *, struct msghdr *,
673                        struct uio *, cred_t *);
674     int      (*sop_sendmblk)(struct sonode *, struct msghdr *, int,
675                        cred_t *, mblk_t **);
676     int      (*sop_getpeername)(struct sonode *, struct sockaddr *,
677                        socklen_t *, boolean_t, cred_t *);
678     int      (*sop_getsockname)(struct sonode *, struct sockaddr *,
679                        socklen_t *, cred_t *);
680     int      (*sop_shutdown)(struct sonode *, int, cred_t *);
681     int      (*sop_getsockopt)(struct sonode *, int, int, void *,
682                        socklen_t *, int, cred_t *);
683     int      (*sop_setsockopt)(struct sonode *, int, int, const void *,
684                        socklen_t, cred_t *);
685     int      (*sop_ioctl)(struct sonode *, int, intptr_t, int,
686                        cred_t *, int32_t *);
687     int      (*sop_poll)(struct sonode *, short, int, short *,
688                        struct pollhead **);
689     int      (*sop_close)(struct sonode *, int, cred_t *);
690 };

692 #define SOP_INIT(so, flag, cr, flags) \
693     ((so)->so_ops->sop_init((so), (flag), (cr), (flags)))
694 #define SOP_ACCEPT(so, fflag, cr, nsop) \
695     ((so)->so_ops->sop_accept((so), (fflag), (cr), (nsop)))
696 #define SOP_BIND(so, name, namelen, flags, cr) \
697     ((so)->so_ops->sop_bind((so), (name), (namelen), (flags), (cr)))
698 #define SOP_LISTEN(so, backlog, cr) \
699     ((so)->so_ops->sop_listen((so), (backlog), (cr)))
700 #define SOP_CONNECT(so, name, namelen, fflag, flags, cr) \
701     ((so)->so_ops->sop_connect((so), (name), (namelen), (fflag), (flags), \
702     (cr)))
703 #define SOP_RECVMSG(so, msg, uiop, cr) \
704     ((so)->so_ops->sop_recvmmsg((so), (msg), (uiop), (cr)))
705 #define SOP_SENDMSG(so, msg, uiop, cr) \
706     ((so)->so_ops->sop_sendmsg((so), (msg), (uiop), (cr)))
707 #define SOP_SENDMBLK(so, msg, size, cr, mpp) \
708     ((so)->so_ops->sop_sendmblk((so), (msg), (size), (cr), (mpp)))
709 #define SOP_GETPEERNAME(so, addr, addrlen, accept, cr) \
710     ((so)->so_ops->sop_getpeername((so), (addr), (addrlen), (accept), (cr)))
711 #define SOP_GETSOCKNAME(so, addr, addrlen, cr) \
712     ((so)->so_ops->sop_getsockname((so), (addr), (addrlen), (cr)))
713 #define SOP_SHUTDOWN(so, how, cr) \
714     ((so)->so_ops->sop_shutdown((so), (how), (cr)))
715 #define SOP_GETSOCKOPT(so, level, optionname, optval, optlenp, flags, cr) \
716     ((so)->so_ops->sop_getsockopt((so), (level), (optionname), \
717     (optval), (optlenp), (flags), (cr)))
718 #define SOP_SETSOCKOPT(so, level, optionname, optval, optlen, cr) \
719     ((so)->so_ops->sop_setsockopt((so), (level), (optionname), \
720     (optval), (optlen), (cr)))
721 #define SOP_IOCTL(so, cmd, arg, mode, cr, rvalp) \

```

```

722     ((so)->so_ops->sop_ioctl((so), (cmd), (arg), (mode), (cr), (rvalp)))
723 #define SOP_POLL(so, events, anyyet, reventsp, phpp) \
724     ((so)->so_ops->sop_poll((so), (events), (anyyet), (reventsp), (phpp)))
725 #define SOP_CLOSE(so, flag, cr) \
726     ((so)->so_ops->sop_close((so), (flag), (cr)))

728 #endif /* defined(_KERNEL) || defined(_KMEMUSER) */

730 #ifndef _KERNEL

732 #define ISALIGNED_cmsgHDR(addr) \
733     (((uintptr_t)(addr) & (_CMSG_HDR_ALIGNMENT - 1)) == 0)

735 #define ROUNDUP_cmsgLEN(len) \
736     (((len) + _CMSG_HDR_ALIGNMENT - 1) & ~(_CMSG_HDR_ALIGNMENT - 1))

738 #define IS_NON_STREAM SOCK(vp) \
739     ((vp)->v_type == VSOCK && (vp)->v_stream == NULL)
740 /*
741  * Macros that operate on struct cmsghdr.
742  * Used in parsing msg_control.
743  * The MSG_VALID macro does not assume that the last option buffer is padded.
744  */
745 #define CMSG_NEXT(cmsg) \
746     (struct cmsghdr *)((uintptr_t)(cmsg) + \
747     ROUNDUP_cmsgLEN((cmsg)->cmsg_len))
748 #define CMSG_CONTENT(cmsg) (&((cmsg)[1]))
749 #define CMSG_CONTENTLEN(cmsg) ((cmsg)->cmsg_len - sizeof(struct cmsghdr))
750 #define CMSG_VALID(cmsg, start, end) \
751     (ISALIGNED_cmsgHDR(cmsg) && \
752     ((uintptr_t)(cmsg) >= (uintptr_t)(start)) && \
753     ((uintptr_t)(cmsg) < (uintptr_t)(end)) && \
754     ((ssize_t)(cmsg)->cmsg_len >= sizeof(struct cmsghdr)) && \
755     ((uintptr_t)(cmsg) + (cmsg)->cmsg_len <= (uintptr_t)(end)))

757 /*
758  * Maximum size of any argument that is copied in (addresses, options,
759  * access rights). MUST be at least MAXPATHLEN + 3.
760  * BSD and SunOS 4.X limited this to MLEN or MCLBYTES.
761  */
762 #define SO_MAXARGSIZE 8192

764 /*
765  * Convert between vnode and sonode
766  */
767 #define VTOSO(vp) ((struct sonode *)((vp)->v_data))
768 #define SOTOV(sp) ((sp)->so_vnode)

770 /*
771  * Internal flags for sobind()
772  */
773 #define _SOBIND_REBIND 0x01 /* Bind to existing local address */
774 #define _SOBIND_UNSPEC 0x02 /* Bind to unspecified address */
775 #define _SOBIND_LOCK_HELD 0x04 /* so_excl_lock held by caller */
776 #define _SOBIND_NOXLATE 0x08 /* No addr translation for AF_UNIX */
777 #define _SOBIND_XPG4_2 0x10 /* xpg4.2 semantics */
778 #define _SOBIND_SOCKBSD 0x20 /* BSD semantics */
779 #define _SOBIND_LISTEN 0x40 /* Make into SS_ACCEPTCONN */
780 #define _SOBIND_SOCKETPAIR 0x80 /* Internal flag for so_socketpair() */
781 /* to enable listen with backlog = 1 */

783 /*
784  * Internal flags for sounbind()
785  */
786 #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_obstate(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 eprintln(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 eprintln(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 *);

```

```

920 extern void      soisconnected(struct sonode *);
921 extern void      soisdisconnected(struct sonode *, int);
922 extern void      socantsendmore(struct sonode *);
923 extern void      socantrcvmore(struct sonode *);
924 extern void      sosetError(struct sonode *, int);
925 extern int       sogeterr(struct sonode *, boolean_t);
926 extern int       sowaitconnected(struct sonode *, int, int);

928 extern ssize_t   soreadfile(file_t *, uchar_t *, u_offset_t, int *, size_t);
929 extern void      *sock_kstat_init(zoneid_t);
930 extern void      sock_kstat_fini(zoneid_t, void *);
931 extern struct sonode *getsonode(int, int *, file_t **);
932 /*
933  * Function wrappers (mostly around the sonode switch) for
934  * backward compatibility.
935  */
936 extern int       soaccept(struct sonode *, int, struct sonode **);
937 extern int       sobind(struct sonode *, struct sockaddr *, socklen_t,
938                        int, int);
939 extern int       solisten(struct sonode *, int);
940 extern int       soconnect(struct sonode *, struct sockaddr *, socklen_t,
941                          int, int);
942 extern int       sorecvmsg(struct sonode *, struct nmsgHDR *, struct uio *);
943 extern int       sosendmsg(struct sonode *, struct nmsgHDR *, struct uio *);
944 extern int       soshutdown(struct sonode *, int);
945 extern int       sogetsockopt(struct sonode *, int, int, void *, socklen_t *,
946                             int);
947 extern int       sosetsockopt(struct sonode *, int, int, const void *,
948                             t_uscalar_t);

950 extern struct sonode *screate(struct sockparams *, int, int, int, int,
951                             int *);

953 extern int       so_copyin(const void *, void *, size_t, int);
954 extern int       so_copyout(const void *, void *, size_t, int);

956 #endif

958 /*
959  * Internal structure for obtaining sonode information from the socklist.
960  * These types match those corresponding in the sonode structure.
961  * This is not a published interface, and may change at any time. It is
962  * used for passing information back up to the kstat consumers. By converting
963  * kernel addresses to strings, we should be able to pass information from
964  * the kernel to userland regardless of n-bit kernel we are using.
965  * This is not a published interface, and may change at any time.
966  */

967 #define ADRSTRLEN (2 * sizeof (uint64_t) + 1)

969 #endif /* ! codereview */
970 struct sockinfo {
971     uint_t      si_size;           /* real length of this struct */
972     short       si_family;
973     short       si_type;
974     ushort_t    si_flag;
975     uint_t      si_state;
976     uint_t      si_ux_laddr_sou_magic;
977     uint_t      si_ux_faddr_sou_magic;
978     t_scalar_t  si_serv_type;
979     t_uscalar_t si_laddr_soa_len;
980     t_uscalar_t si_faddr_soa_len;
981     uint16_t    si_laddr_family;
982     uint16_t    si_faddr_family;
983     char        si_laddr_sun_path[MAXPATHLEN + 1]; /* NULL terminated */
984     char        si_faddr_sun_path[MAXPATHLEN + 1];

```

```

985     boolean_t   si_faddr_noxlate;
986     zoneid_t    si_szoneid;
987     char        si_son_straddr[ADRSTRLEN];
988     char        si_lvn_straddr[ADRSTRLEN];
989     char        si_fvn_straddr[ADRSTRLEN];
990     uint_t      si_pn_cnt;
991     pid_t       si_pids[1];
992 #endif /* ! codereview */
993 };

995 /*
996  * Subcodes for sockconf() system call
997  */
998 #define SOCKCONFIG_ADD_SOCKET      0
999 #define SOCKCONFIG_REMOVE_SOCKET  1
1000 #define SOCKCONFIG_ADD_FILTER     2
1001 #define SOCKCONFIG_REMOVE_FILTER   3
1002 #define SOCKCONFIG_GET_SOCKETABLE  4

1004 /*
1005  * Data structures for configuring socket filters.
1006  */

1008 /*
1009  * Placement hint for automatic filters
1010  */
1011 typedef enum {
1012     SOF_HINT_NONE,
1013     SOF_HINT_TOP,
1014     SOF_HINT_BOTTOM,
1015     SOF_HINT_BEFORE,
1016     SOF_HINT_AFTER
1017 } sof_hint_t;

1019 /*
1020  * Socket tuple. Used by sockconfig_filter_props to list socket
1021  * types of interest.
1022  */
1023 typedef struct sof_socktuple {
1024     int     sofst_family;
1025     int     sofst_type;
1026     int     sofst_protocol;
1027 } sof_socktuple_t;

1029 /*
1030  * Socket filter properties used by sockconfig() system call.
1031  */
1032 struct sockconfig_filter_props {
1033     char        *sfp_modname;
1034     boolean_t   sfp_autoattach;
1035     sof_hint_t  sfp_hint;
1036     char        *sfp_hintarg;
1037     uint_t      sfp_socktuple_cnt;
1038     sof_socktuple_t *sfp_socktuple;
1039 };

1041 /*
1042  * Data structures for the in-kernel socket configuration table.
1043  */
1044 typedef struct sockconfig_socktable_entry {
1045     int     se_family;
1046     int     se_type;
1047     int     se_protocol;
1048     int     se_refcnt;
1049     int     se_flags;
1050     char    se_modname[MODMAXNAMELEN];

```

```
1051     char          se_strdev[MAXPATHLEN];
1052 } sockconfig_socktable_entry_t;

1054 typedef struct sockconfig_socktable {
1055     uint_t          num_of_entries;
1056     sockconfig_socktable_entry_t *st_entries;
1057 } sockconfig_socktable_t;

1059 #ifdef _SYSCALL32

1061 typedef struct sof_socktuple32 {
1062     int32_t         sofst_family;
1063     int32_t         sofst_type;
1064     int32_t         sofst_protocol;
1065 } sof_socktuple32_t;

1067 struct sockconfig_filter_props32 {
1068     caddr32_t       sfp_modname;
1069     boolean_t       sfp_autoattach;
1070     sof_hint_t       sfp_hint;
1071     caddr32_t       sfp_hintarg;
1072     uint32_t        sfp_socktuple_cnt;
1073     caddr32_t       sfp_socktuple;
1074 };

1076 typedef struct sockconfig_socktable32 {
1077     uint_t          num_of_entries;
1078     caddr32_t       st_entries;
1079 } sockconfig_socktable32_t;

1081 #endif /* _SYSCALL32 */

1083 #define SOCKMOD_PATH    "socketmod"    /* dir where sockmods are stored */

1085 #ifdef __cplusplus
1086 }
1087 #endif

1089 #endif /* _SYS_SOCKETVAR_H */
```

```

*****
48841 Wed Jul 13 01:32:42 2016
new/usr/src/uts/common/sys/strsubr.h
XXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
22 /*      All Rights Reserved      */

25 /*
26  * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
27  * Use is subject to license terms.
28  */

30 #ifndef _SYS_STRSUBR_H
31 #define _SYS_STRSUBR_H

33 /*
34  * WARNING:
35  * Everything in this file is private, belonging to the
36  * STREAMS subsystem.  The only guarantee made about the
37  * contents of this file is that if you include it, your
38  * code will not port to the next release.
39  */
40 #include <sys/stream.h>
41 #include <sys/stropts.h>
42 #include <sys/kstat.h>
43 #include <sys/uiio.h>
44 #include <sys/proc.h>
45 #include <sys/netstack.h>
46 #include <sys/modhash.h>
47 #include <sys/pidnode.h>
48 #endif /* ! codereview */

50 #ifdef __cplusplus
51 extern "C" {
52 #endif

54 /*
55  * In general, the STREAMS locks are disjoint; they are only held
56  * locally, and not simultaneously by a thread.  However, module
57  * code, including at the stream head, requires some locks to be
58  * acquired in order for its safety.
59  *
60  * 1. Stream level claim.  This prevents the value of q_next
61  *    from changing while module code is executing.
62  *
63  * 2. Queue level claim.  This prevents the value of q_ptr

```

```

62 *    from changing while put or service code is executing.
63 *    In addition, it provides for queue single-threading
64 *    for QPAIR and PERQ MT-safe modules.
65 * 3. Stream head lock.  May be held by the stream head module
66 *    to implement a read/write/open/close monitor.
67 *    Note: that the only types of twisted stream supported are
68 *    the pipe and transports which have read and write service
69 *    procedures on both sides of the twist.
70 * 4. Queue lock.  May be acquired by utility routines on
71 *    behalf of a module.
72 */

74 /*
75  * In general, sd_lock protects the consistency of the stdata
76  * structure.  Additionally, it is used with sd_monitor
77  * to implement an open/close monitor.  In particular, it protects
78  * the following fields:
79  *   sd_iocblk
80  *   sd_flag
81  *   sd_copyflag
82  *   sd_iocid
83  *   sd_iocwait
84  *   sd_sidp
85  *   sd_pgidp
86  *   sd_wroff
87  *   sd_tail
88  *   sd_rerror
89  *   sd_werror
90  *   sd_pushcnt
91  *   sd_sigflags
92  *   sd_siglist
93  *   sd_pollist
94  *   sd_mark
95  *   sd_closetime
96  *   sd_wakeq
97  *   sd_maxblk
98  *
99  * The following fields are modified only by the allocator, which
100 * has exclusive access to them at that time:
101 *   sd_wrq
102 *   sd_strtab
103 *
104 * The following field is protected by the overlying file system
105 * code, guaranteeing single-threading of opens:
106 *   sd_vnode
107 *
108 * Stream-level locks should be acquired before any queue-level locks
109 * are acquired.
110 *
111 * The stream head write queue lock(sd_wrq) is used to protect the
112 * fields qn_maxpsz and qn_minpsz because freezestr() which is
113 * necessary for strqset() only gets the queue lock.
114 */

116 /*
117  * Function types for the parameterized stream head.
118  * The msgfunc_t takes the parameters:
119  *   msgfunc(vnode_t *vp, mblk_t *mp, strwakeupt *wakeups,
120  *           strsigset_t *firstmsgsig, strsigset_t *allmsgsig,
121  *           strpollset_t *pollwakeups);
122  * It returns an optional message to be processed by the stream head.
123  *
124  * The parameters for errfunc_t are:
125  *   errfunc(vnode *vp, int ispeek, int *clearerr);
126  * It returns an errno and zero if there was no pending error.
127 */

```

```

128 typedef uint_t  strwakeupt;
129 typedef uint_t  strsigset_t;
130 typedef short   strpollset_t;
131 typedef uintptr_t callbparams_id_t;
132 typedef mblk_t  *(*msgfunc_t)(vnode_t *, mblk_t *, strwakeupt *,
133                               strsigset_t *, strsigset_t *, strpollset_t *);
134 typedef int     (*errfunc_t)(vnode_t *, int, int *);

136 /*
137  * Per stream sd_lock in putnext may be replaced by per cpu stream putlocks
138  * each living in a separate cache line. putnext/canputnext grabs only one of
139  * stream putlocks while strlock() (called on behalf of insertq()/removeq())
140  * acquires all stream putlocks. Normally stream putlocks are only employed
141  * for highly contended streams that have SQ_CIPUT queues in the critical path
142  * (e.g. NFS/UDP stream).
143  *
144  * stream putlocks are dynamically assigned to stdata structure through
145  * sd_ciputctrl pointer possibly when a stream is already in use. Since
146  * strlock() uses stream putlocks only under sd_lock acquiring sd_lock when
147  * assigning stream putlocks to the stream ensures synchronization with
148  * strlock().
149  *
150  * For lock ordering purposes stream putlocks are treated as the extension of
151  * sd_lock and are always grabbed right after grabbing sd_lock and released
152  * right before releasing sd_lock except putnext/canputnext where only one of
153  * stream putlocks locks is used and where it is the first lock to grab.
154  */

156 typedef struct ciputctrl_str {
157     union _ciput_un {
158         uchar_t pad[64];
159         struct _ciput_str {
160             kmutex_t      ciput_lck;
161             ushort_t     ciput_cnt;
162         } ciput_str;
163     } ciput_un;
164 } ciputctrl_t;

166 #define ciputctrl_lock  ciput_un.ciput_str.ciput_lck
167 #define ciputctrl_count  ciput_un.ciput_str.ciput_cnt

169 /*
170  * Header for a stream: interface to rest of system.
171  *
172  * NOTE: While this is a consolidation-private structure, some unbundled and
173  * third-party products inappropriately make use of some of the fields.
174  * As such, please take care to not gratuitously change any offsets of
175  * existing members.
176  */
177 typedef struct stdata {
178     struct queue *sd_wrq;          /* write queue */
179     struct msgb *sd_iocblk;       /* return block for ioctl */
180     struct vnode *sd_vnode;      /* pointer to associated vnode */
181     struct streamtab *sd_strtab; /* pointer to streamtab for stream */
182     uint_t sd_flag;              /* state/flags */
183     uint_t sd_iocid;             /* ioctl id */
184     struct pid *sd_sidp;         /* controlling session info */
185     struct pid *sd_pgidp;       /* controlling process group info */
186     ushort_t sd_tail;           /* reserved space in written mblks */
187     ushort_t sd_wroff;          /* write offset */
188     int sd_rerror;              /* error to return on read ops */
189     int sd_werror;              /* error to return on write ops */
190     int sd_pushcnt;             /* number of pushes done on stream */
191     int sd_sigflags;            /* logical OR of all siglist events */
192     struct strsig *sd_siglist;   /* pid linked list to rcv SIGPOLL sig */
193     struct pollhead sd_pollist; /* list of all pollers to wake up */

```

```

194     struct msgb *sd_mark;        /* "marked" message on read queue */
195     clock_t sd_closetime;       /* time to wait to drain q in close */
196     kmutex_t sd_lock;          /* protect head consistency */
197     kcondvar_t sd_monitor;     /* open/close/push/pop monitor */
198     kcondvar_t sd_iocmonitor;  /* ioctl single-threading */
199     kcondvar_t sd_refmonitor;  /* sd_refcnt monitor */
200     ssize_t sd_qn_minpsz;      /* These two fields are a performance */
201     ssize_t sd_qn_maxpsz;     /* enhancements, cache the values in */
202                                 /* the stream head so we don't have */
203                                 /* to ask the module below the stream */
204                                 /* head to get this information. */
205     struct stdata *sd_mate;    /* pointer to twisted stream mate */
206     kthread_id_t sd_freezer;   /* thread that froze stream */
207     kmutex_t sd_reflock;      /* Protects sd_refcnt */
208     int sd_refcnt;            /* number of claimstr */
209     uint_t sd_wakeq;           /* strwakeq()'s copy of sd_flag */
210     struct queue *sd_struiordq; /* sync barrier struio() read queue */
211     struct queue *sd_struiowrq; /* sync barrier struio() write queue */
212     char *sd_struiodnak;       /* defer NAK of M_IOCTL by rput() */
213     struct msgb *sd_struionak; /* pointer M_IOCTL mblk(s) to NAK */
214     caddr_t sd_t_audit_data;   /* For audit purposes only */
215     ssize_t sd_maxblk;        /* maximum message block size */
216     uint_t sd_rput_opt;       /* options/flags for strrput */
217     uint_t sd_wput_opt;       /* options/flags for write/putmsg */
218     uint_t sd_read_opt;       /* options/flags for stread */
219     msgfunc_t sd_rprotofunc;  /* rput M_PROTO routine */
220     msgfunc_t sd_rputdatafunc; /* read M_DATA routine */
221     msgfunc_t sd_rmiscfunc;   /* rput routine (non-data/proto) */
222     msgfunc_t sd_wputdatafunc; /* wput M_DATA routine */
223     errfunc_t sd_rderrfunc;   /* read side error callback */
224     errfunc_t sd_wrerrfunc;   /* write side error callback */
225     /*
226      * support for low contention concurrent putnext.
227      */
228     ciputctrl_t *sd_ciputctrl;
229     uint_t sd_nciputctrl;

231     int sd_anchor;            /* position of anchor in stream */
232     /*
233      * Service scheduling at the stream head.
234      */
235     kmutex_t sd_qlock;
236     struct queue *sd_qhead;    /* Head of queues to be serviced. */
237     struct queue *sd_qtail;    /* Tail of queues to be serviced. */
238     void *sd_servid;          /* Service ID for bckgrnd schedule */
239     ushort_t sd_svclags;      /* Servicing flags */
240     short sd_nqueues;         /* Number of queues in the list */
241     kcondvar_t sd_qcv;        /* Waiters for qhead to become empty */
242     kcondvar_t sd_zcopy_wait; /* copy-related flags */
243     uint_t sd_copyflag;       /* copy-related flags */
244     zoneid_t sd_anchorzone;   /* Allow removal from same zone only */
245     struct msgb *sd_cmdblk;   /* reply from _I_CMD */
246     /*
247      * pids associated with this stream head.
248      */
249     avl_tree_t sd_pid_tree;
250     kmutex_t sd_pid_tree_lock;
251 #endif /* ! codereview */
252 } stdata_t;

254 /*
255  * stdata servicing flags.
256  */
257 #define STRS_WILLSERVICE 0x01
258 #define STRS_SCHEDULED 0x02

```

```

260 #define STREAM_NEEDSERVICE(stp) ((stp)->sd_qhead != NULL)
262 /*
263  * stdata flag field defines
264  */
265 #define IOCWAIT      0x00000001    /* Someone is doing an ioctl */
266 #define RSLEEP       0x00000002    /* Someone wants to read/recv msg */
267 #define WSLEEP       0x00000004    /* Someone wants to write */
268 #define STRPRI       0x00000008    /* An M_PCPROTO is at stream head */
269 #define STRHUP       0x00000010    /* Device has vanished */
270 #define STWOPEN      0x00000020    /* waiting for 1st open */
271 #define STPLEX       0x00000040    /* stream is being multiplexed */
272 #define STRISTTY     0x00000080    /* stream is a terminal */
273 #define STRGETINPROG 0x00000100    /* (k)strgetmsg is running */
274 #define IOCWAITNE    0x00000200    /* STR_NOERROR ioctl running */
275 #define STRDERR      0x00000400    /* fatal read error from M_ERROR */
276 #define STRWRERR     0x00000800    /* fatal write error from M_ERROR */
277 #define STRDERRNONPERSIST 0x00001000 /* nonpersistent read errors */
278 #define STRWRERRNONPERSIST 0x00002000 /* nonpersistent write errors */
279 #define STRCLOSE     0x00004000    /* wait for a close to complete */
280 #define SNDMREAD     0x00008000    /* used for read notification */
281 #define OLDNDDELAY   0x00010000    /* use old TTY semantics for */
282                                     /* NDELAY reads and writes */
283                                     /* unused */
284                                     /* unused */
285 #define STRTOSTOP    0x00080000    /* block background writes */
286 #define STRCMDWAIT   0x00100000    /* someone is doing an _I_CMD */
287                                     /* unused */
288 #define STRMOUNT     0x00400000    /* stream is mounted */
289 #define STRNOTATMARK 0x00800000    /* Not at mark (when empty read q) */
290 #define STRDELIM     0x01000000    /* generate delimited messages */
291 #define STRATMARK    0x02000000    /* At mark (due to MSGMARKNEXT) */
292 #define STZCNOTIFY   0x04000000    /* wait for zerocopy mblk to be acked */
293 #define STRPLUMB     0x08000000    /* push/pop pending */
294 #define STREOF       0x10000000    /* End-of-file indication */
295 #define STREOPENFAIL 0x20000000    /* indicates if re-open has failed */
296 #define STRMATE      0x40000000    /* this stream is a mate */
297 #define STRHASLINKS 0x80000000    /* I_LINKs under this stream */
299 /*
300  * Copy-related flags (sd_copyflag), set by SO_COPYOPT.
301  */
302 #define STZCVMSAFE   0x00000001    /* safe to borrow file (segmapped) */
303                                     /* pages instead of bcopy */
304 #define STZCVMUNSAFE 0x00000002    /* unsafe to borrow file pages */
305 #define STRCOPYCACHED 0x00000004    /* copy should NOT bypass cache */
307 /*
308  * Options and flags for strrrput (sd_rput_opt)
309  */
310 #define SR_POLLIN    0x00000001    /* pollwakep needed for band0 data */
311 #define SR_SIGALLDATA 0x00000002    /* Send SIGPOLL for all M_DATA */
312 #define SR_CONSOL_DATA 0x00000004    /* Consolidate M_DATA onto q_last */
313 #define SR_IGN_ZEROLEN 0x00000008    /* Ignore zero-length M_DATA */
315 /*
316  * Options and flags for strwrite/strputmsg (sd_wput_opt)
317  */
318 #define SW_SIGPIPE   0x00000001    /* Send SIGPIPE for write error */
319 #define SW_RECHECK_ERR 0x00000002    /* Recheck errors in strwrite loop */
320 #define SW_SNDZERO   0x00000004    /* send 0-length msg down pipe/FIFO */
322 /*
323  * Options and flags for strread (sd_read_opt)
324  */
325 #define RD_MSGDIS    0x00000001    /* read msg discard */

```

```

326 #define RD_MSGNODIS 0x00000002    /* read msg no discard */
327 #define RD_PROTDAT  0x00000004    /* read M_[PC]PROTO contents as data */
328 #define RD_PROTDIS  0x00000008    /* discard M_[PC]PROTO blocks and */
329                                     /* retain data blocks */
330 /*
331  * Flags parameter for strsetrputhooks() and strsetwputhooks().
332  * These flags define the interface for setting the above internal
333  * flags in sd_rput_opt and sd_wput_opt.
334  */
335 #define SH_CONSOL_DATA 0x00000001    /* Consolidate M_DATA onto q_last */
336 #define SH_SIGALLDATA 0x00000002    /* Send SIGPOLL for all M_DATA */
337 #define SH_IGN_ZEROLEN 0x00000004    /* Drop zero-length M_DATA */
339 #define SH_SIGPIPE    0x00000100    /* Send SIGPIPE for write error */
340 #define SH_RECHECK_ERR 0x00000200    /* Recheck errors in strwrite loop */
342 /*
343  * Each queue points to a sync queue (the inner perimeter) which keeps
344  * track of the number of threads that are inside a given queue (sq_count)
345  * and also is used to implement the asynchronous putnext
346  * (by queuing messages if the queue can not be entered.)
347  */
348 * Messages are queued on sq_head/sq_tail including deferred qwriter(INNER)
349 * messages. The sq_head/sq_tail list is a singly-linked list with
350 * b_queue recording the queue and b_prev recording the function to
351 * be called (either the put procedure or a qwriter callback function.)
352 *
353 * The sq_count counter tracks the number of threads that are
354 * executing inside the perimeter or (in the case of outer perimeters)
355 * have some work queued for them relating to the perimeter. The sq_rmcount
356 * counter tracks the subset which are in removeq() (usually invoked from
357 * qprocsoff(9F)).
358 *
359 * In addition a module writer can declare that the module has an outer
360 * perimeter (by setting D_MTOUTPERIM) in which case all inner perimeter
361 * syncq's for the module point (through sq_outer) to an outer perimeter
362 * syncq. The outer perimeter consists of the doubly linked list (sq_onext and
363 * sq_orev) linking all the inner perimeter syncq's with out outer perimeter
364 * syncq. This is used to implement qwriter(OUTER) (an asynchronous way of
365 * getting exclusive access at the outer perimeter) and outer_enter/exit
366 * which are used by the framework to acquire exclusive access to the outer
367 * perimeter during open and close of modules that have set D_MTOUTPERIM.
368 *
369 * In the inner perimeter case sq_save is available for use by machine
370 * dependent code. sq_head/sq_tail are used to queue deferred messages on
371 * the inner perimeter syncqs and to queue become_writer requests on the
372 * outer perimeter syncqs.
373 *
374 * Note: machine dependent optimized versions of putnext may depend
375 * on the order of sq_flags and sq_count (so that they can e.g.
376 * read these two fields in a single load instruction.)
377 *
378 * Per perimeter SLOCK/sq_count in putnext/put may be replaced by per cpu
379 * sq_putlocks/sq_putcounts each living in a separate cache line. Obviously
380 * sq_putlock[x] protects sq_putcount[x]. putnext/put routine will grab only 1
381 * of sq_putlocks and update only 1 of sq_putcounts. strlock() and many
382 * other routines in strsubr.c and ddi.c will grab all sq_putlocks (as well as
383 * SLOCK) and figure out the count value as the sum of sq_count and all of
384 * sq_putcounts. The idea is to make critical fast path -- putnext -- much
385 * faster at the expense of much less often used slower path like
386 * strlock(). One known case where entersq/strlock is executed pretty often is
387 * SpecWeb but since IP is S_CIOC and socket TCP/IP stream is nextless
388 * there's no need to grab multiple sq_putlocks and look at sq_putcounts. See
389 * strsubr.c for more comments.
390 *
391 * Note regular SLOCK and sq_count are still used in many routines

```

```

392 * (e.g. entersq(), rwnext()) in the same way as before sq_putlocks were
393 * introduced.
394 *
395 * To understand when all sq_putlocks need to be held and all sq_putcounts
396 * need to be added up one needs to look closely at putnext code. Basically if
397 * a routine like e.g. wait_syncq() needs to be sure that perimeter is empty
398 * all sq_putlocks/sq_putcounts need to be held/added up. On the other hand
399 * there's no need to hold all sq_putlocks and count all sq_putcounts in
400 * routines like leavesq()/dropsq() and etc. since the are usually exit
401 * counterparts of entersq/outer_enter() and etc. which have already either
402 * prevented put entry points from executing or did not care about put
403 * entrypoints. entersq() doesn't need to care about sq_putlocks/sq_putcounts
404 * if the entry point has a shared access since put has the highest degree of
405 * concurrency and such entersq() does not intend to block out put
406 * entrypoints.
407 *
408 * Before sq_putcounts were introduced the standard way to wait for perimeter
409 * to become empty was:
410 *
411 *     mutex_enter(SQLOCK(sq));
412 *     while (sq->sq_count > 0) {
413 *         sq->sq_flags |= SQ_WANTWAKEUP;
414 *         cv_wait(&sq->sq_wait, SQLOCK(sq));
415 *     }
416 *     mutex_exit(SQLOCK(sq));
417 *
418 * The new way is:
419 *
420 *     mutex_enter(SQLOCK(sq));
421 *     count = sq->sq_count;
422 *     SQ_PUTLOCKS_ENTER(sq);
423 *     SUM_SQ_PUTCOUNTS(sq, count);
424 *     while (count != 0) {
425 *         sq->sq_flags |= SQ_WANTWAKEUP;
426 *         SQ_PUTLOCKS_EXIT(sq);
427 *         cv_wait(&sq->sq_wait, SQLOCK(sq));
428 *         count = sq->sq_count;
429 *         SQ_PUTLOCKS_ENTER(sq);
430 *         SUM_SQ_PUTCOUNTS(sq, count);
431 *     }
432 *     SQ_PUTLOCKS_EXIT(sq);
433 *     mutex_exit(SQLOCK(sq));
434 *
435 * Note that SQ_WANTWAKEUP is set before dropping SQ_PUTLOCKS. This makes sure
436 * putnext won't skip a wakeup.
437 *
438 * sq_putlocks are treated as the extension of SQLOCK for lock ordering
439 * purposes and are always grabbed right after grabbing SQLOCK and released
440 * right before releasing SQLOCK. This also allows dynamic creation of
441 * sq_putlocks while holding SQLOCK (by making sq_ciputctrl non null even when
442 * the stream is already in use). Only in putnext one of sq_putlocks
443 * is grabbed instead of SQLOCK. putnext return path remembers what counter it
444 * incremented and decrements the right counter on its way out.
445 */
446
447 struct syncq {
448     kmutex_t      sq_lock;          /* atomic access to syncq */
449     uint16_t      sq_count;         /* # threads inside */
450     uint16_t      sq_flags;        /* state and some type info */
451     /*
452     * Distributed syncq scheduling
453     * The list of queue's is handled by sq_head and
454     * sq_tail fields.
455     *
456     * The list of events is handled by the sq_evhead and sq_evtail
457     * fields.

```

```

458     /*
459     queue_t      *sq_head;         /* queue of deferred messages */
460     queue_t      *sq_tail;         /* queue of deferred messages */
461     mblk_t       *sq_evhead;       /* Event message on the syncq */
462     mblk_t       *sq_evtail;
463     uint_t       sq_nqueues;       /* # of queues on this sq */
464     /*
465     * Concurrency and condition variables
466     */
467     uint16_t      sq_type;          /* type (concurrency) of syncq */
468     uint16_t      sq_rmccount;      /* # threads inside removeq() */
469     kcondvar_t   sq_wait;          /* block on this sync queue */
470     kcondvar_t   sq_exitwait;      /* waiting for thread to leave the */
471     /* inner perimeter */
472     /*
473     * Handling synchronous callbacks such as qtimeout and qbufcall
474     */
475     ushort_t     sq_callbflags;     /* flags for callback synchronization */
476     callbparams_id_t sq_cancelid;   /* id of callback being cancelled */
477     struct callbparams *sq_callbpend; /* Pending callbacks */
478
479     /*
480     * Links forming an outer perimeter from one outer syncq and
481     * a set of inner sync queues.
482     */
483     struct syncq *sq_outer;         /* Pointer to outer perimeter */
484     struct syncq *sq_onext;         /* Linked list of syncq's making */
485     struct syncq *sq_oprev;         /* up the outer perimeter. */
486     /*
487     * support for low contention concurrent putnext.
488     */
489     ciputctrl_t  *sq_ciputctrl;
490     uint_t       sq_nciputctrl;
491     /*
492     * Counter for the number of threads wanting to become exclusive.
493     */
494     uint_t       sq_needexcl;
495     /*
496     * These two fields are used for scheduling a syncq for
497     * background processing. The sq_svcflag is protected by
498     * SQLOCK lock.
499     */
500     struct syncq *sq_next;          /* for syncq scheduling */
501     void *       sq_servid;
502     uint_t       sq_servcount;      /* # pending background threads */
503     uint_t       sq_svcflags;       /* Scheduling flags */
504     clock_t      sq_tstamp;         /* Time when was enabled */
505     /*
506     * Maximum priority of the queues on this syncq.
507     */
508     pri_t        sq_pri;
509 };
510 typedef struct syncq syncq_t;
511
512 /*
513 * sync queue scheduling flags (for sq_svcflags).
514 */
515 #define SQ_SERVICE      0x1        /* being serviced */
516 #define SQ_BGTHREAD    0x2        /* awaiting service by bg thread */
517 #define SQ_DISABLED    0x4        /* don't put syncq in service list */
518
519 /*
520 * FASTPUT bit in sd_count/putcount.
521 */
522 #define SQ_FASTPUT     0x8000
523 #define SQ_FASTMASK    0x7FFF

```

```

525 /*
526 * sync queue state flags
527 */
528 #define SQ_EXCL      0x0001      /* exclusive access to inner */
529                               /* perimeter */
530 #define SQ_BLOCKED  0x0002      /* qprocsoff */
531 #define SQ_FROZEN   0x0004      /* freezestr */
532 #define SQ_WRITER   0x0008      /* qwriter(OUTER) pending or running */
533 #define SQ_MESSAGES 0x0010      /* messages on syncq */
534 #define SQ_WANTWAKEUP 0x0020    /* do cv_broadcast on sq_wait */
535 #define SQ_WANTEXWAKEUP 0x0040  /* do cv_broadcast on sq_exitwait */
536 #define SQ_EVENTS   0x0080      /* Events pending */
537 #define SQ_QUEUED   (SQ_MESSAGES | SQ_EVENTS)
538 #define SQ_FLAGMASK 0x00FF

540 /*
541 * Test a queue to see if inner perimeter is exclusive.
542 */
543 #define PERIM_EXCL(q) ((q)->q_syncq->sq_flags & SQ_EXCL)

545 /*
546 * If any of these flags are set it is not possible for a thread to
547 * enter a put or service procedure. Instead it must either block
548 * or put the message on the syncq.
549 */
550 #define SQ_GOAWAY    (SQ_EXCL|SQ_BLOCKED|SQ_FROZEN|SQ_WRITER|\
551                    SQ_QUEUED)
552 /*
553 * If any of these flags are set it not possible to drain the syncq
554 */
555 #define SQ_STAYAWAY (SQ_BLOCKED|SQ_FROZEN|SQ_WRITER)

557 /*
558 * Flags to trigger syncq tail processing.
559 */
560 #define SQ_TAIL      (SQ_QUEUED|SQ_WANTWAKEUP|SQ_WANTEXWAKEUP)

562 /*
563 * Syncq types (stored in sq_type)
564 * The SQ_TYPES_IN_FLAGS (ciput) are also stored in sq_flags
565 * for performance reasons. Thus these type values have to be in the low
566 * 16 bits and not conflict with the sq_flags values above.
567 *
568 * Notes:
569 * - putnext() and put() assume that the put procedures have the highest
570 *   degree of concurrency. Thus if any of the SQ_CI* are set then SQ_CIPUT
571 *   has to be set. This restriction can be lifted by adding code to putnext
572 *   and put that check that sq_count == 0 like entersq does.
573 * - putnext() and put() does currently not handle !SQ_COPUT
574 * - In order to implement !SQ_COCB outer_enter has to be fixed so that
575 *   the callback can be cancelled while cv.waiting in outer_enter.
576 * - If SQ_CISVC needs to be implemented, qprocsoff() needs to wait
577 *   for the currently running services to stop (wait for QINSERVICE
578 *   to go off). disable_svc called from qprocsoff disables only
579 *   services that will be run in future.
580 *
581 * All the SQ_CO flags are set when there is no outer perimeter.
582 */
583 #define SQ_CIPUT      0x0100      /* Concurrent inner put proc */
584 #define SQ_CISVC     0x0200      /* Concurrent inner svc proc */
585 #define SQ_CIOC       0x0400      /* Concurrent inner open/close */
586 #define SQ_CICB       0x0800      /* Concurrent inner callback */
587 #define SQ_COPUT     0x1000      /* Concurrent outer put proc */
588 #define SQ_COSVC     0x2000      /* Concurrent outer svc proc */
589 #define SQ_COOC       0x4000      /* Concurrent outer open/close */

```

```

590 #define SQ_COCB      0x8000      /* Concurrent outer callback */

592 /* Types also kept in sq_flags for performance */
593 #define SQ_TYPES_IN_FLAGS (SQ_CIPUT)

595 #define SQ_CI         (SQ_CIPUT|SQ_CISVC|SQ_CIOC|SQ_CICB)
596 #define SQ_CO         (SQ_COPUT|SQ_COSVC|SQ_COOC|SQ_COCB)
597 #define SQ_TYPEMASK   (SQ_CI|SQ_CO)

599 /*
600 * Flag combinations passed to entersq and leavesq to specify the type
601 * of entry point.
602 */
603 #define SQ_PUT         (SQ_CIPUT|SQ_COPUT)
604 #define SQ_SVC         (SQ_CISVC|SQ_COSVC)
605 #define SQ_OPENCLOSE  (SQ_CIOC|SQ_COOC)
606 #define SQ_CALLBACK   (SQ_CICB|SQ_COCB)

608 /*
609 * Other syncq types which are not copied into flags.
610 */
611 #define SQ_PERMOD      0x01      /* Syncq is PERMOD */

613 /*
614 * Asynchronous callback qun*** flag.
615 * The mechanism these flags are used in is one where callbacks enter
616 * the perimeter thanks to framework support. To use this mechanism
617 * the q* and qun* flavors of the callback routines must be used.
618 * e.g. qtimeout and quntimeout. The synchronization provided by the flags
619 * avoids deadlocks between blocking qun* routines and the perimeter
620 * lock.
621 */
622 #define SQ_CALLB_BYPASSED 0x01    /* bypassed callback fn */

624 /*
625 * Cancel callback mask.
626 * The mask expands as the number of cancelable callback types grows
627 * Note - separate callback flag because different callbacks have
628 * overlapping id space.
629 */
630 #define SQ_CALLB_CANCEL_MASK (SQ_CANCEL_TOUT|SQ_CANCEL_BUFCALL)

632 #define SQ_CANCEL_TOUT 0x02      /* cancel timeout request */
633 #define SQ_CANCEL_BUFCALL 0x04   /* cancel bufcall request */

635 typedef struct callbparams {
636     syncq_t      *cbp_sq;
637     void          (*cbp_func)(void *);
638     void          *cbp_arg;
639     callbparams_id_t cbp_id;
640     uint_t       cbp_flags;
641     struct callbparams *cbp_next;
642     size_t       cbp_size;
643 } callbparams_t;

645 typedef struct strbufcall {
646     void          (*bc_func)(void *);
647     void          *bc_arg;
648     size_t       bc_size;
649     bufcall_id_t bc_id;
650     struct strbufcall *bc_next;
651     kthread_id_t bc_executor;
652 } strbufcall_t;

654 /*
655 * Structure of list of processes to be sent SIGPOLL/SIGURG signal

```

```

656 * on request. The valid S_* events are defined in stropts.h.
657 */
658 typedef struct strsig {
659     struct pid      *ss_pidp;      /* pid/pgrp pointer */
660     pid_t           ss_pid;        /* positive pid, negative pgrp */
661     int             ss_events;     /* S_* events */
662     struct strsig   *ss_next;
663 } strsig_t;

665 /*
666 * bufcall list
667 */
668 struct bclist {
669     strbufcall_t    *bc_head;
670     strbufcall_t    *bc_tail;
671 };

673 /*
674 * Structure used to track mux links and unlinks.
675 */
676 struct mux_node {
677     major_t         mn_imaj;      /* internal major device number */
678     uint16_t        mn_indegree;  /* number of incoming edges */
679     struct mux_node *mn_originp;  /* where we came from during search */
680     struct mux_edge *mn_startp;   /* where search left off in mn_outp */
681     struct mux_edge *mn_outp;    /* list of outgoing edges */
682     uint_t          mn_flags;     /* see below */
683 };

685 /*
686 * Flags for mux_nodes.
687 */
688 #define VISITED 1

690 /*
691 * Edge structure - a list of these is hung off the
692 * mux_node to represent the outgoing edges.
693 */
694 struct mux_edge {
695     struct mux_node *me_nodep;    /* edge leads to this node */
696     struct mux_edge *me_nextp;   /* next edge */
697     int             me_muxid;    /* id of link */
698     dev_t           me_dev;      /* dev_t - used for kernel PUNLINK */
699 };

701 /*
702 * Queue info
703 */
704 * The syncq is included here to reduce memory fragmentation
705 * for kernel memory allocators that only allocate in sizes that are
706 * powers of two. If the kernel memory allocator changes this should
707 * be revisited.
708 */
709 typedef struct queinfo {
710     struct queue    qu_rqueue;    /* read queue - must be first */
711     struct queue    qu_wqueue;    /* write queue - must be second */
712     struct syncq    qu_syncq;     /* syncq - must be third */
713 } queinfo_t;

715 /*
716 * Multiplexed streams info
717 */
718 typedef struct linkinfo {
719     struct linkblk  li_lblk;      /* must be first */
720     struct file     *li_fpdwn;    /* file pointer for lower stream */
721     struct linkinfo *li_next;    /* next in list */

```

```

722     struct linkinfo *li_prev;    /* previous in list */
723 } linkinfo_t;

725 /*
726 * List of syncq's used by freezestr/unfreezestr
727 */
728 typedef struct syncql {
729     struct syncql  *sql_next;
730     syncq_t        *sql_sq;
731 } syncql_t;

733 typedef struct sqliist {
734     syncql_t        *sqliist_head;
735     size_t          sqliist_size; /* structure size in bytes */
736     size_t          sqliist_index; /* next free entry in array */
737     syncql_t        sqliist_array[4]; /* 4 or more entries */
738 } sqliist_t;

740 typedef struct perdm {
741     struct perdm   *dm_next;
742     syncq_t        *dm_sq;
743     struct streamtab *dm_str;
744     uint_t         dm_ref;
745 } perdm_t;

747 #define NEED_DM(dmp, qflag) \
748     (dmp == NULL && (qflag & (QPERMOD | QMTOUTPERIM)))

750 /*
751 * fmodsw_impl_t is used within the kernel. fmodsw is used by
752 * the modules/drivers. The information is copied from fmodsw
753 * defined in the module/driver into the fmodsw_impl_t structure
754 * during the module/driver initialization.
755 */
756 typedef struct fmodsw_impl    fmodsw_impl_t;

758 struct fmodsw_impl {
759     fmodsw_impl_t    *f_next;
760     char              f_name[FMNAMESZ + 1];
761     struct streamtab *f_str;
762     uint32_t          f_qflag;
763     uint32_t          f_sqtype;
764     perdm_t           *f_dmp;
765     uint32_t          f_ref;
766     uint32_t          f_hits;
767 };

769 typedef enum {
770     FMODSW_HOLD = 0x00000001,
771     FMODSW_LOAD = 0x00000002
772 } fmodsw_flags_t;

774 typedef struct cdevsw_impl {
775     struct streamtab *d_str;
776     uint32_t          d_qflag;
777     uint32_t          d_sqtype;
778     perdm_t           *d_dmp;
779 } cdevsw_impl_t;

781 /*
782 * Enumeration of the types of access that can be requested for a
783 * controlling terminal under job control.
784 */
785 enum jaccess {
786     JCREAD, /* read data on a ctty */
787     JCWRITE, /* write data to a ctty */

```

```

788     JCSETP,          /* set ctty parameters */
789     JCGETP          /* get ctty parameters */
790 };

792 struct str_stack {
793     netstack_t      *ss_netstack; /* Common netstack */

795     kmutex_t        ss_sad_lock; /* autopush lock */
796     mod_hash_t      *ss_sad_hash;
797     size_t          ss_sad_hash_nchains;
798     struct saddev   *ss_saddev; /* sad device array */
799     int             ss_sadcnt; /* number of sad devices */

801     int             ss_devcnt; /* number of mux_nodes */
802     struct mux_node *ss_mux_nodes; /* mux info for cycle checking */
803 };
804 typedef struct str_stack str_stack_t;

806 /*
807  * Finding related queues
808  */
809 #define STREAM(q) ((q)->q_stream)
810 #define SQ(rq) ((syncq_t *)((rq) + 2))

812 /*
813  * Get the module/driver name for a queue. Since some queues don't have
814  * q_info structures (e.g., see log_makeq()), fall back to "?".
815  */
816 #define Q2NAME(q) \
817     (((q)->q_qinfo != NULL && (q)->q_qinfo->q_i_mininfo->mi_idname != NULL) ? \
818     (q)->q_qinfo->q_i_mininfo->mi_idname : "?")

820 /*
821  * Locking macros
822  */
823 #define QLOCK(q) (&(q)->q_lock)
824 #define SLOCK(sq) (&(sq)->sq_lock)

826 #define STREAM_PUTLOCKS_ENTER(stp) { \
827     ASSERT(MUTEX_HELD(&(stp)->sd_lock)); \
828     if ((stp)->sd_ciputctrl != NULL) { \
829         int i; \
830         int nlocks = (stp)->sd_nciputctrl; \
831         ciputctrl_t *cip = (stp)->sd_ciputctrl; \
832         for (i = 0; i <= nlocks; i++) { \
833             mutex_enter(&cip[i].ciputctrl_lock); \
834         } \
835     } \
836 }

838 #define STREAM_PUTLOCKS_EXIT(stp) { \
839     ASSERT(MUTEX_HELD(&(stp)->sd_lock)); \
840     if ((stp)->sd_ciputctrl != NULL) { \
841         int i; \
842         int nlocks = (stp)->sd_nciputctrl; \
843         ciputctrl_t *cip = (stp)->sd_ciputctrl; \
844         for (i = 0; i <= nlocks; i++) { \
845             mutex_exit(&cip[i].ciputctrl_lock); \
846         } \
847     } \
848 }

850 #define SQ_PUTLOCKS_ENTER(sq) { \
851     ASSERT(MUTEX_HELD(SLOCK(sq))); \
852     if ((sq)->sq_ciputctrl != NULL) { \
853         int i; \

```

```

854         int nlocks = (sq)->sq_nciputctrl; \
855         ciputctrl_t *cip = (sq)->sq_ciputctrl; \
856         ASSERT((sq)->sq_type & SQ_CIPUT); \
857         for (i = 0; i <= nlocks; i++) { \
858             mutex_enter(&cip[i].ciputctrl_lock); \
859         } \
860     } \
861 }

863 #define SQ_PUTLOCKS_EXIT(sq) { \
864     ASSERT(MUTEX_HELD(SLOCK(sq))); \
865     if ((sq)->sq_ciputctrl != NULL) { \
866         int i; \
867         int nlocks = (sq)->sq_nciputctrl; \
868         ciputctrl_t *cip = (sq)->sq_ciputctrl; \
869         ASSERT((sq)->sq_type & SQ_CIPUT); \
870         for (i = 0; i <= nlocks; i++) { \
871             mutex_exit(&cip[i].ciputctrl_lock); \
872         } \
873     } \
874 }

876 #define SQ_PUTCOUNT_SETFAST(sq) { \
877     ASSERT(MUTEX_HELD(SLOCK(sq))); \
878     if ((sq)->sq_ciputctrl != NULL) { \
879         int i; \
880         int nlocks = (sq)->sq_nciputctrl; \
881         ciputctrl_t *cip = (sq)->sq_ciputctrl; \
882         ASSERT((sq)->sq_type & SQ_CIPUT); \
883         for (i = 0; i <= nlocks; i++) { \
884             mutex_enter(&cip[i].ciputctrl_lock); \
885             cip[i].ciputctrl_count |= SQ_FASTPUT; \
886             mutex_exit(&cip[i].ciputctrl_lock); \
887         } \
888     } \
889 }

891 #define SQ_PUTCOUNT_CLRFAST(sq) { \
892     ASSERT(MUTEX_HELD(SLOCK(sq))); \
893     if ((sq)->sq_ciputctrl != NULL) { \
894         int i; \
895         int nlocks = (sq)->sq_nciputctrl; \
896         ciputctrl_t *cip = (sq)->sq_ciputctrl; \
897         ASSERT((sq)->sq_type & SQ_CIPUT); \
898         for (i = 0; i <= nlocks; i++) { \
899             mutex_enter(&cip[i].ciputctrl_lock); \
900             cip[i].ciputctrl_count &= ~SQ_FASTPUT; \
901             mutex_exit(&cip[i].ciputctrl_lock); \
902         } \
903     } \
904 }

907 #ifndef DEBUG

909 #define SQ_PUTLOCKS_HELD(sq) { \
910     ASSERT(MUTEX_HELD(SLOCK(sq))); \
911     if ((sq)->sq_ciputctrl != NULL) { \
912         int i; \
913         int nlocks = (sq)->sq_nciputctrl; \
914         ciputctrl_t *cip = (sq)->sq_ciputctrl; \
915         ASSERT((sq)->sq_type & SQ_CIPUT); \
916         for (i = 0; i <= nlocks; i++) { \
917             ASSERT(MUTEX_HELD(&cip[i].ciputctrl_lock)); \
918         } \
919     } \

```

```

920     }
922 #define SUMCHECK_SQ_PUTCOUNTS(sq, countcheck) { \
923     if ((sq)->sq_ciputctrl != NULL) { \
924         int i; \
925         uint_t count = 0; \
926         int ncounts = (sq)->sq_nciputctrl; \
927         ASSERT((sq)->sq_type & SQ_CIPUT); \
928         for (i = 0; i <= ncounts; i++) { \
929             count += \
930                 (((sq)->sq_ciputctrl[i].ciputctrl_count) & \
931                 SQ_FASTMASK); \
932             } \
933         ASSERT(count == (countcheck)); \
934     } \
935 }
937 #define SUMCHECK_CIPUTCTRL_COUNTS(ciput, nciput, countcheck) { \
938     int i; \
939     uint_t count = 0; \
940     ASSERT((ciput) != NULL); \
941     for (i = 0; i <= (nciput); i++) { \
942         count += (((ciput)[i].ciputctrl_count) & \
943                 SQ_FASTMASK); \
944     } \
945     ASSERT(count == (countcheck)); \
946 }
948 #else /* DEBUG */
950 #define SQ_PUTLOCKS_HELD(sq)
951 #define SUMCHECK_SQ_PUTCOUNTS(sq, countcheck)
952 #define SUMCHECK_CIPUTCTRL_COUNTS(sq, nciput, countcheck)
954 #endif /* DEBUG */
956 #define SUM_SQ_PUTCOUNTS(sq, count) { \
957     if ((sq)->sq_ciputctrl != NULL) { \
958         int i; \
959         int ncounts = (sq)->sq_nciputctrl; \
960         ciputctrl_t *cip = (sq)->sq_ciputctrl; \
961         ASSERT((sq)->sq_type & SQ_CIPUT); \
962         for (i = 0; i <= ncounts; i++) { \
963             (count) += ((cip[i].ciputctrl_count) & \
964                       SQ_FASTMASK); \
965         } \
966     } \
967 }
969 #define CLAIM_QNEXT_LOCK(stp) mutex_enter(&(stp)->sd_lock)
970 #define RELEASE_QNEXT_LOCK(stp) mutex_exit(&(stp)->sd_lock)
972 /*
973  * syncq message manipulation macros.
974  */
975 /*
976  * Put a message on the queue syncq.
977  * Assumes QLOCK held.
978  */
979 #define SQPUT_MP(qp, mp) \
980     { \
981         qp->q_syncqmsgs++; \
982         if (qp->q_sqhead == NULL) { \
983             qp->q_sqhead = qp->q_sqtail = mp; \
984         } else { \
985             qp->q_sqtail->b_next = mp; \

```

```

986         qp->q_sqtail = mp; \
987     } \
988     set_qfull(qp); \
989 }
991 /*
992  * Miscellaneous parameters and flags.
993  */
995 /*
996  * Default timeout in milliseconds for ioctls and close
997  */
998 #define STRTIMEOUT 15000
1000 /*
1001  * Flag values for stream io
1002  */
1003 #define WRITEWAIT    0x1    /* waiting for write event */
1004 #define READWAIT    0x2    /* waiting for read event */
1005 #define NOINTR       0x4    /* error is not to be set for signal */
1006 #define GETWAIT     0x8    /* waiting for getmsg event */
1008 /*
1009  * These flags need to be unique for stream io name space
1010  * and copy modes name space. These flags allow strwaitq
1011  * and strdoioctl to proceed as if signals or errors on the stream
1012  * head have not occurred; i.e. they will be detected by some other
1013  * means.
1014  * STR_NOSIG does not allow signals to interrupt the call
1015  * STR_NOERROR does not allow stream head read, write or hup errors to
1016  * affect the call. When used with strdoioctl(), if a previous ioctl
1017  * is pending and times out, STR_NOERROR will cause strdoioctl() to not
1018  * return ETIME. If, however, the requested ioctl times out, ETIME
1019  * will be returned (use ic_timeout instead)
1020  * STR_PEEK is used to inform strwaitq that the reader is peeking at data
1021  * and that a non-persistent error should not be cleared.
1022  * STR_DELAYERR is used to inform strwaitq that it should not check errors
1023  * after being awoken since, in addition to an error, there might also be
1024  * data queued on the stream head read queue.
1025  */
1026 #define STR_NOSIG    0x10    /* Ignore signals during strdoioctl/strwaitq */
1027 #define STR_NOERROR  0x20    /* Ignore errors during strdoioctl/strwaitq */
1028 #define STR_PEEK     0x40    /* Peeking behavior on non-persistent errors */
1029 #define STR_DELAYERR 0x80    /* Do not check errors on return */
1031 /*
1032  * Copy modes for tty and I_STR ioctls
1033  */
1034 #define U_TO_K  01    /* User to Kernel */
1035 #define K_TO_K  02    /* Kernel to Kernel */
1037 /*
1038  * Mux defines.
1039  */
1040 #define LINKNORMAL    0x01    /* normal mux link */
1041 #define LINKPERSIST   0x02    /* persistent mux link */
1042 #define LINKTYPEEMASK 0x03    /* bitmask of all link types */
1043 #define LINKCLOSE     0x04    /* unlink from strclose */
1045 /*
1046  * Definitions of Streams macros and function interfaces.
1047  */
1049 /*
1050  * Obsolete queue scheduling macros. They are not used anymore, but still kept
1051  * here for 3-d party modules and drivers who might still use them.

```

```

1052 */
1053 #define setqsched()
1054 #define qready()      1

1056 #ifdef _KERNEL
1057 #define runqueues()
1058 #define queuerun()
1059 #endif

1061 /* compatibility module for style 2 drivers with DR race condition */
1062 #define DRMODNAME      "drcompat"

1064 /*
1065  * Macros dealing with mux_nodes.
1066  */
1067 #define MUX_VISIT(X)    ((X)->mn_flags |= VISITED)
1068 #define MUX_CLEAR(X)   ((X)->mn_flags &= (~VISITED)); \
1069                       ((X)->mn_originp = NULL)
1070 #define MUX_DIDVISIT(X) ((X)->mn_flags & VISITED)

1073 /*
1074  * Twisted stream macros
1075  */
1076 #define STRMATED(X)    ((X)->sd_flag & STRMATE)
1077 #define STRLOCKMATES(X) if (&((X)->sd_lock) > &((X)->sd_mate->sd_lock)) { \
1078                       mutex_enter(&((X)->sd_lock)); \
1079                       mutex_enter(&((X)->sd_mate->sd_lock)); \
1080                       } else { \
1081                       mutex_enter(&((X)->sd_mate->sd_lock)); \
1082                       mutex_enter(&((X)->sd_lock)); \
1083                       }
1084 #define STRUNLOCKMATES(X) mutex_exit(&((X)->sd_lock)); \
1085                          mutex_exit(&((X)->sd_mate->sd_lock))

1087 #ifdef _KERNEL

1089 extern void strinit(void);
1090 extern int strdoioctl(struct stdata *, struct strioctl *, int, int,
1091                      cred_t *, int *);
1092 extern void strsendsig(struct strsig *, int, uchar_t, int);
1093 extern void str_sendsig(vnode_t *, int, uchar_t, int);
1094 extern void strhup(struct stdata *);
1095 extern int gattach(queue_t *, dev_t *, int, cred_t *, fmodsw_impl_t *,
1096                  boolean_t);
1097 extern int qreopen(queue_t *, dev_t *, int, cred_t *);
1098 extern void qdetach(queue_t *, int, int, cred_t *, boolean_t);
1099 extern void enterq(queue_t *);
1100 extern void leaveq(queue_t *);
1101 extern int putiocd(mblk_t *, caddr_t, int, cred_t *);
1102 extern int getiocd(mblk_t *, caddr_t, int);
1103 extern struct linkinfo *alloclink(queue_t *, queue_t *, struct file *);
1104 extern void lbfree(struct linkinfo *);
1105 extern int linkcycle(stdata_t *, stdata_t *, str_stack_t *);
1106 extern struct linkinfo *findlinks(stdata_t *, int, int, str_stack_t *);
1107 extern queue_t *getendq(queue_t *);
1108 extern int mlink(vnode_t *, int, int, cred_t *, int *, int);
1109 extern int mlink_file(vnode_t *, int, struct file *, cred_t *, int *, int);
1110 extern int munlink(struct stdata *, struct linkinfo *, int, cred_t *, int *,
1111                  str_stack_t *);
1112 extern int munlinkall(struct stdata *, int, cred_t *, int *, str_stack_t *);
1113 extern void mux_addedge(stdata_t *, stdata_t *, int, str_stack_t *);
1114 extern void mux_rmvedge(stdata_t *, int, str_stack_t *);
1115 extern int devflg_to_qflag(struct streamtab *, uint32_t, uint32_t *,
1116                          uint32_t *);
1117 extern void setq(queue_t *, struct qinit *, struct qinit *, perdm_t *,

```

```

1118     uint32_t, uint32_t, boolean_t);
1119 extern perdm_t *hold_dm(struct streamtab *, uint32_t, uint32_t);
1120 extern void rele_dm(perdm_t *);
1121 extern int strmakectl(struct strbuf *, int32_t, int32_t, mblk_t **);
1122 extern int strmakeedata(ssize_t *, struct uio *, stdata_t *, int32_t, mblk_t **);
1123 extern int strmakemsg(struct strbuf *, ssize_t *, struct uio *,
1124                      struct stdata *, int32_t, mblk_t **);
1125 extern int strgetmsg(vnode_t *, struct strbuf *, struct strbuf *, uchar_t *,
1126                    int *, int, rval_t *);
1127 extern int strputmsg(vnode_t *, struct strbuf *, struct strbuf *, uchar_t,
1128                    int flag, int fmode);
1129 extern int strstartplumb(struct stdata *, int, int);
1130 extern void strendplumb(struct stdata *);
1131 extern int stropen(struct vnode *, dev_t *, int, cred_t *);
1132 extern int strclose(struct vnode *, int, cred_t *);
1133 extern int strpoll(register struct stdata *, short, int, short *,
1134                  struct pollhead **);
1135 extern void strclean(struct vnode *);
1136 extern void str_cn_clean(); /* XXX hook for consoles signal cleanup */
1137 extern int strwrite(struct vnode *, struct uio *, cred_t *);
1138 extern int strwrite_common(struct vnode *, struct uio *, cred_t *, int);
1139 extern int stread(struct vnode *, struct uio *, cred_t *);
1140 extern int strioctl(struct vnode *, int, intptr_t, int, int, cred_t *, int *);
1141 extern int strrrput(queue_t *, mblk_t *);
1142 extern int strrrput_nondata(queue_t *, mblk_t *);
1143 extern mblk_t *strrrput_proto(vnode_t *, mblk_t *,
1144                              strwakeupt *, strsigset_t *, strsigset_t *, strpollset_t *);
1145 extern mblk_t *strrrput_misc(vnode_t *, mblk_t *,
1146                              strwakeupt *, strsigset_t *, strsigset_t *, strpollset_t *);
1147 extern int getiocseqno(void);
1148 extern int strwaitbuf(size_t, int);
1149 extern int strwaitq(stdata_t *, int, ssize_t, int, clock_t, int *);
1150 extern struct stdata *shalloc(queue_t *);
1151 extern void sh_insert_pid(struct stdata *, pid_t);
1152 extern void sh_remove_pid(struct stdata *, pid_t);
1153 extern mblk_t *sh_get_pid_mblk(struct stdata *);
1154 #endif /* ! codereview */
1155 extern void shfree(struct stdata *s);
1156 extern queue_t *allocq(void);
1157 extern void freeq(queue_t *);
1158 extern qband_t *allocband(void);
1159 extern void freeband(qband_t *);
1160 extern void freebs_enqueue(mblk_t *, dblk_t *);
1161 extern void setqback(queue_t *, unsigned char);
1162 extern int strcopyin(void *, void *, size_t, int);
1163 extern int strcopyout(void *, void *, size_t, int);
1164 extern void strsignal(struct stdata *, int, int32_t);
1165 extern clock_t str_cv_wait(kcondvar_t *, kmutex_t *, clock_t, int);
1166 extern void disable_svc(queue_t *);
1167 extern void enable_svc(queue_t *);
1168 extern void remove_runlist(queue_t *);
1169 extern void wait_svc(queue_t *);
1170 extern void backenable(queue_t *, uchar_t);
1171 extern void set_qend(queue_t *);
1172 extern int strgeterr(stdata_t *, int32_t, int);
1173 extern void qenable_locked(queue_t *);
1174 extern mblk_t *getq_noenab(queue_t *, ssize_t);
1175 extern void rmvq_noenab(queue_t *, mblk_t *);
1176 extern void qbackenable(queue_t *, uchar_t);
1177 extern void set_qfull(queue_t *);

1179 extern void strbblock(queue_t *);
1180 extern void strunblock(queue_t *);
1181 extern int qclaimed(queue_t *);
1182 extern int straccess(struct stdata *, enum jaccess);

```

```

1184 extern void entersq(syncq_t *, int);
1185 extern void leavesq(syncq_t *, int);
1186 extern void claimq(queue_t *);
1187 extern void releaseq(queue_t *);
1188 extern void claimstr(queue_t *);
1189 extern void releasestr(queue_t *);
1190 extern void removeq(queue_t *);
1191 extern void insertq(struct stdata *, queue_t *);
1192 extern void drain_syncq(syncq_t *);
1193 extern void qfill_syncq(syncq_t *, queue_t *, mblk_t *);
1194 extern void qdrain_syncq(syncq_t *, queue_t *);
1195 extern int flush_syncq(syncq_t *, queue_t *);
1196 extern void wait_sq_svc(syncq_t *);

1198 extern void outer_enter(syncq_t *, uint16_t);
1199 extern void outer_exit(syncq_t *);
1200 extern void qwriter_inner(queue_t *, mblk_t *, void (*)());
1201 extern void qwriter_outer(queue_t *, mblk_t *, void (*)());

1203 extern callbparams_t *callbparams_alloc(syncq_t *, void (*) (void *,
1204     void *, int));
1205 extern void callbparams_free(syncq_t *, callbparams_t *);
1206 extern void callbparams_free_id(syncq_t *, callbparams_id_t, int32_t);
1207 extern void qcallbwrapper(void *);

1209 extern mblk_t *esballoc_wait(unsigned char *, size_t, uint_t, frtn_t *);
1210 extern mblk_t *esballoca(unsigned char *, size_t, uint_t, frtn_t *);
1211 extern mblk_t *desballoca(unsigned char *, size_t, uint_t, frtn_t *);
1212 extern int do_sendfp(struct stdata *, struct file *, struct cred *);
1213 extern int frozenstr(queue_t *);
1214 extern size_t xmsgsize(mblk_t *);

1216 extern void putnext_tail(syncq_t *, queue_t *, uint32_t);
1217 extern void stream_willservice(stdata_t *);
1218 extern void stream_runservice(stdata_t *);

1220 extern void strmate(vnode_t *, vnode_t *);
1221 extern queue_t *strvp2wq(vnode_t *);
1222 extern vnode_t *strq2vp(queue_t *);
1223 extern mblk_t *allocb_wait(size_t, uint_t, uint_t, int *);
1224 extern mblk_t *allocb_cred(size_t, cred_t *, pid_t);
1225 extern mblk_t *allocb_cred_wait(size_t, uint_t, int *, cred_t *, pid_t);
1226 extern mblk_t *allocb_tmpl(size_t, const mblk_t *);
1227 extern mblk_t *allocb_tryhard(size_t);
1228 extern void mblk_copycred(mblk_t *, const mblk_t *);
1229 extern void mblk_setcred(mblk_t *, cred_t *, pid_t);
1230 extern cred_t *msg_getcred(const mblk_t *, pid_t *);
1231 extern struct ts_label_s *msg_getlabel(const mblk_t *);
1232 extern cred_t *msg_extractcred(mblk_t *, pid_t *);
1233 extern void strpollwakeuq(vnode_t *, short);
1234 extern int putnextctl_wait(queue_t *, int);

1236 extern int kstrputmsg(struct vnode *, mblk_t *, struct uio *, ssize_t,
1237     unsigned char, int, int);
1238 extern int kstrgetmsg(struct vnode *, mblk_t **, struct uio *,
1239     unsigned char *, int *, clock_t, rval_t *);

1241 extern void strseterror(vnode_t *, int, int, errfunc_t);
1242 extern void strsetwerror(vnode_t *, int, int, errfunc_t);
1243 extern void strseteof(vnode_t *, int);
1244 extern void strflushrq(vnode_t *, int);
1245 extern void strsetrputhooks(vnode_t *, uint_t, msgfunc_t, msgfunc_t);
1246 extern void strsetwputhooks(vnode_t *, uint_t, clock_t);
1247 extern void strsetrwputdatahooks(vnode_t *, msgfunc_t, msgfunc_t);
1248 extern int strwaitmark(vnode_t *);
1249 extern void strsignal_nolock(stdata_t *, int, uchar_t);

```

```

1251 struct multidata_s;
1252 struct pdesc_s;
1253 extern int hcksum_assoc(mblk_t *, struct multidata_s *, struct pdesc_s *,
1254     uint32_t, uint32_t, uint32_t, uint32_t, uint32_t, int);
1255 extern void hcksum_retrieve(mblk_t *, struct multidata_s *, struct pdesc_s *,
1256     uint32_t *, uint32_t *, uint32_t *, uint32_t *, uint32_t *);
1257 extern void lso_info_set(mblk_t *, uint32_t, uint32_t);
1258 extern void lso_info_cleanup(mblk_t *);
1259 extern unsigned int bcksum(uchar_t *, int, unsigned int);
1260 extern boolean_t is_vmloaned_mblk(mblk_t *, struct multidata_s *,
1261     struct pdesc_s *);

1263 extern int fmodsw_register(const char *, struct streamtab *, int);
1264 extern int fmodsw_unregister(const char *);
1265 extern fmodsw_impl_t *fmodsw_find(const char *, fmodsw_flags_t);
1266 extern void fmodsw_rele(fmodsw_impl_t *);

1268 extern void freemsgchain(mblk_t *);
1269 extern mblk_t *copymsgchain(mblk_t *);

1271 extern mblk_t *mcopyinuo(struct stdata *, uio_t *, ssize_t, ssize_t, int *);

1273 /*
1274  * shared or externally configured data structures
1275  */
1276 extern ssize_t strmsgsz; /* maximum stream message size */
1277 extern ssize_t strctlsz; /* maximum size of ctl message */
1278 extern int nstrpush; /* maximum number of pushes allowed */

1280 /*
1281  * Bufcalls related variables.
1282  */
1283 extern struct bclist strbcalls; /* List of bufcalls */
1284 extern kmutex_t strbcall_lock; /* Protects the list of bufcalls */
1285 extern kcondvar_t strbcall_cv; /* Signaling when a bufcall is added */
1286 extern kcondvar_t bcall_cv; /* wait of executing bufcall completes */

1288 extern frtn_t frnop;

1290 extern struct kmem_cache *ciputctrl_cache;
1291 extern int n_ciputctrl;
1292 extern int max_n_ciputctrl;
1293 extern int min_n_ciputctrl;

1295 extern cdevsw_impl_t *devimpl;

1297 /*
1298  * esballoc queue for throttling
1299  */
1300 typedef struct esb_queue {
1301     kmutex_t eq_lock;
1302     uint_t eq_len; /* number of queued messages */
1303     mblk_t *eq_head; /* head of queue */
1304     mblk_t *eq_tail; /* tail of queue */
1305     uint_t eq_flags; /* esballoc queue flags */
1306 } esb_queue_t;

1308 /*
1309  * esballoc flags for queue processing.
1310  */
1311 #define ESBQ_PROCESSING 0x01 /* queue is being processed */
1312 #define ESBQ_TIMER 0x02 /* timer is active */

1314 extern void esballoc_queue_init(void);

```

```
1316 #endif /* _KERNEL */

1318 /*
1319  * Note: Use of these macros are restricted to kernel/unix and
1320  * intended for the STREAMS framework.
1321  * All modules/drivers should include sys/ddi.h.
1322  *
1323  * Finding related queues
1324  */
1325 #define _OTHERQ(q) ((q)->q_flag&QREADR? (q)+1: (q)-1)
1326 #define _WR(q) ((q)->q_flag&QREADR? (q)+1: (q))
1327 #define _RD(q) ((q)->q_flag&QREADR? (q): (q)-1)
1328 #define _SAMESTR(q) (!(q)->q_flag & QEND))

1330 /*
1331  * These are also declared here for modules/drivers that erroneously
1332  * include strsubr.h after ddi.h or fail to include ddi.h at all.
1333  */
1334 extern struct queue *OTHERQ(queue_t *); /* stream.h */
1335 extern struct queue *RD(queue_t *);
1336 extern struct queue *WR(queue_t *);
1337 extern int SAMESTR(queue_t *);

1339 /*
1340  * The following hardware checksum related macros are private
1341  * interfaces that are subject to change without notice.
1342  */
1343 #ifdef _KERNEL
1344 #define DB_CKSUMSTART(mp) ((mp)->b_datap->db_cksumstart)
1345 #define DB_CKSUMEND(mp) ((mp)->b_datap->db_cksumend)
1346 #define DB_CKSUMSTUFF(mp) ((mp)->b_datap->db_cksumstuff)
1347 #define DB_CKSUMFLAGS(mp) ((mp)->b_datap->db_struioun.cksum.flags)
1348 #define DB_CKSUM16(mp) ((mp)->b_datap->db_cksum16)
1349 #define DB_CKSUM32(mp) ((mp)->b_datap->db_cksum32)
1350 #define DB_LSOFMFLAGS(mp) ((mp)->b_datap->db_struioun.cksum.flags)
1351 #define DB_LSOMSS(mp) ((mp)->b_datap->db_struioun.cksum.pad)
1352 #endif /* _KERNEL */

1354 #ifdef __cplusplus
1355 }
1356 #endif

1359 #endif /* _SYS_STRSUBR_H */
```

```

*****
24841 Wed Jul 13 01:32:43 2016
new/usr/src/uts/common/syscall/fcntl.c
XXXX adding PID information to netstat output
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
24  * Copyright (c) 2013, OmniTI Computer Consulting, Inc. All rights reserved.
25  * Copyright 2015, Joyent, Inc.
26 */

28 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T      */
29 /*      All Rights Reserved      */

31 /*
32  * Portions of this source code were derived from Berkeley 4.3 BSD
33  * under license from the Regents of the University of California.
34 */

37 #include <sys/param.h>
38 #include <sys/isa_defs.h>
39 #include <sys/types.h>
40 #include <sys/sysmacros.h>
41 #include <sys/system.h>
42 #include <sys/errno.h>
43 #include <sys/fcntl.h>
44 #include <sys/flock.h>
45 #include <sys/vnode.h>
46 #include <sys/file.h>
47 #include <sys/mode.h>
48 #include <sys/proc.h>
49 #include <sys/filio.h>
50 #include <sys/share.h>
51 #include <sys/debug.h>
52 #include <sys/rctl.h>
53 #include <sys/nbmlck.h>

55 #include <sys/cmn_err.h>

57 static int flock_check(vnode_t *, flock64_t *, offset_t, offset_t);
58 static int flock_get_start(vnode_t *, flock64_t *, offset_t, u_offset_t *);
59 static void fd_too_big(proc_t *);
61 */

```

```

62  * File control.
63  */
64 int
65 fcntl(int fdes, int cmd, intp_t arg)
66 {
67     int iarg;
68     int error = 0;
69     int retval;
70     proc_t *p;
71     file_t *fp;
72     vnode_t *vp;
73     u_offset_t offset;
74     u_offset_t start;
75     struct vattr vattr;
76     int in_crit;
77     int flag;
78     struct flock sbf;
79     struct flock64 bf;
80     struct o_flock obf;
81     struct flock64_32 bf64_32;
82     struct fshare fsh;
83     struct shrlock shr;
84     struct shr_locowner shr_own;
85     offset_t maxoffset;
86     model_t datamodel;
87
89 #if defined(_ILP32) && !defined(lint) && defined(_SYSCALL32)
90     ASSERT(sizeof (struct flock) == sizeof (struct flock32));
91     ASSERT(sizeof (struct flock64) == sizeof (struct flock64_32));
92 #endif
93 #if defined(_LP64) && !defined(lint) && defined(_SYSCALL32)
94     ASSERT(sizeof (struct flock) == sizeof (struct flock64_64));
95     ASSERT(sizeof (struct flock64) == sizeof (struct flock64_64));
96 #endif

98     /*
99      * First, for speed, deal with the subset of cases
100      * that do not require getf() / releasef().
101      */
102     switch (cmd) {
103     case F_GETFD:
104         if ((error = f_getfd_error(fdes, &flag)) == 0)
105             retval = flag;
106         goto out;

108     case F_SETFD:
109         error = f_setfd_error(fdes, (int)arg);
110         retval = 0;
111         goto out;

113     case F_GETFL:
114         if ((error = f_getfl(fdes, &flag)) == 0) {
115             retval = (flag & (FMASK | FASYNC));
116             if ((flag & (FSEARCH | FEXEC)) == 0)
117                 retval += FOPEN;
118             else
119                 retval |= (flag & (FSEARCH | FEXEC));
120         }
121         goto out;

123     case F_GETXFL:
124         if ((error = f_getfl(fdes, &flag)) == 0) {
125             retval = flag;
126             if ((flag & (FSEARCH | FEXEC)) == 0)
127                 retval += FOPEN;

```

```

128     }
129     goto out;

131 case F_BADFD:
132     if ((error = f_badfd(fdes, &fdres, (int)arg)) == 0)
133         retval = fdres;
134     goto out;
135 }

137 /*
138  * Second, for speed, deal with the subset of cases that
139  * require getf() / releasef() but do not require copyin.
140  */
141 if ((fp = getf(fdes)) == NULL) {
142     error = EBADF;
143     goto out;
144 }
145 iarg = (int)arg;

147 switch (cmd) {
148 case F_DUPFD:
149 case F_DUPFD_CLOEXEC:
150     p = curproc;
151     if ((uint_t)iarg >= p->p_fno_ctl) {
152         if (iarg >= 0)
153             fd_too_big(p);
154         error = EINVAL;
155         goto done;
156     }
157     /*
158     * We need to increment the f_count reference counter
159     * before allocating a new file descriptor.
160     * Doing it other way round opens a window for race condition
161     * with closeandsetf() on the target file descriptor which can
162     * close the file still referenced by the original
163     * file descriptor.
164     */
165     mutex_enter(&fp->f_tlock);
166     fp->f_count++;
167     mutex_exit(&fp->f_tlock);
168     if ((retval = ufalloc_file(iarg, fp)) == -1) {
169         /*
170         * New file descriptor can't be allocated.
171         * Revert the reference count.
172         */
173         mutex_enter(&fp->f_tlock);
174         fp->f_count--;
175         mutex_exit(&fp->f_tlock);
176         error = EMFILE;
177     } else {
178         if (cmd == F_DUPFD_CLOEXEC) {
179             f_setfd(retval, FD_CLOEXEC);
180         }
181     }

183     if (error == 0 && fp->f_vnode != NULL) {
184         (void) VOP_IOCTL(fp->f_vnode, F_ASSOCI_PID,
185             (intptr_t)p->p_pidp->pid_id, FKIOCTL, kcred,
186             NULL, NULL);
187     }

189 #endif /* ! codereview */
190     goto done;

192 case F_DUP2FD_CLOEXEC:
193     if (fdes == iarg) {

```

```

194         error = EINVAL;
195         goto done;
196     }

198     /*FALLTHROUGH*/

200 case F_DUP2FD:
201     p = curproc;
202     if (fdes == iarg) {
203         retval = iarg;
204     } else if ((uint_t)iarg >= p->p_fno_ctl) {
205         if (iarg >= 0)
206             fd_too_big(p);
207         error = EBADF;
208     } else {
209         /*
210         * We can't hold our getf(fdes) across the call to
211         * closeandsetf() because it creates a window for
212         * deadlock: if one thread is doing dup2(a, b) while
213         * another is doing dup2(b, a), each one will block
214         * waiting for the other to call releasef(). The
215         * solution is to increment the file reference count
216         * (which we have to do anyway), then releasef(fdes),
217         * then closeandsetf(). Incrementing f_count ensures
218         * that fp won't disappear after we call releasef().
219         * When closeandsetf() fails, we try avoid calling
220         * closef() because of all the side effects.
221         */
222         mutex_enter(&fp->f_tlock);
223         fp->f_count++;
224         mutex_exit(&fp->f_tlock);
225         releasef(fdes);

227         /*
228         * Assume we succeed to duplicate the file descriptor
229         * and associate the pid to the vnode.
230         */
231         if (fp->f_vnode != NULL) {
232             (void) VOP_IOCTL(fp->f_vnode, F_ASSOCI_PID,
233                 (intptr_t)p->p_pidp->pid_id, FKIOCTL,
234                 kcred, NULL, NULL);
235         }

237 #endif /* ! codereview */
238     if ((error = closeandsetf(iarg, fp)) == 0) {
239         if (cmd == F_DUP2FD_CLOEXEC) {
240             f_setfd(iarg, FD_CLOEXEC);
241         }
242         retval = iarg;
243     } else {
244         mutex_enter(&fp->f_tlock);
245         if (fp->f_count > 1) {
246             fp->f_count--;
247             mutex_exit(&fp->f_tlock);
248             /*
249             * Failed to duplicate fdes,
250             * disassociate the pid from the vnode.
251             */
252             if (fp->f_vnode != NULL) {
253                 (void) VOP_IOCTL(fp->f_vnode,
254                     F_DASSOC_PID,
255                     (intptr_t)p->p_pidp->pid_id,
256                     FKIOCTL, kcred, NULL, NULL);
257             }

259 #endif /* ! codereview */

```

```

260     } else {
261         mutex_exit(&fp->f_tlock);
262         (void) closef(fp);
263     }
264     }
265     goto out;
266 }
267 goto done;

269 case F_SETFL:
270     vp = fp->f_vnode;
271     flag = fp->f_flag;
272     if ((iarg & (FNONBLOCK|FNDELAY)) == (FNONBLOCK|FNDELAY))
273         iarg &= ~FNDELAY;
274     if ((error = VOP_SETFL(vp, flag, iarg, fp->f_cred, NULL)) ==
275         0) {
276         iarg &= FMASK;
277         mutex_enter(&fp->f_tlock);
278         fp->f_flag &= ~FMASK | (FREAD|FWRITE);
279         fp->f_flag |= (iarg - FOPEN) & ~(FREAD|FWRITE);
280         mutex_exit(&fp->f_tlock);
281     }
282     retval = 0;
283     goto done;
284 }

286 /*
287  * Finally, deal with the expensive cases.
288  */
289     retval = 0;
290     in_crit = 0;
291     maxoffset = MAXOFF_T;
292     datamodel = DATAMODEL_NATIVE;
293 #if defined(_SYSCALL32_IMPL)
294     if ((datamodel = get_udatamodel()) == DATAMODEL_ILP32)
295         maxoffset = MAXOFF32_T;
296 #endif

298     vp = fp->f_vnode;
299     flag = fp->f_flag;
300     offset = fp->f_offset;

302     switch (cmd) {
303     /*
304      * The file system and vnode layers understand and implement
305      * locking with flock64 structures. So here once we pass through
306      * the test for compatibility as defined by LFS API, (for F_SETLK,
307      * F_SETLKW, F_GETLK, F_GETLKW, F_OFD_GETLK, F_OFD_SETLK, F_OFD_SETLKW,
308      * F_FREESP) we transform the flock structure to a flock64 structure
309      * and send it to the lower layers. Similarly in case of GETLK and
310      * OFD_GETLK the returned flock64 structure is transformed to a flock
311      * structure if everything fits in nicely, otherwise we return
312      * EOVERFLOW.
313      */

315     case F_GETLK:
316     case F_O_GETLK:
317     case F_SETLK:
318     case F_SETLKW:
319     case F_SETLK_NBMAND:
320     case F_OFD_GETLK:
321     case F_OFD_SETLK:
322     case F_OFD_SETLKW:
323     case F_FLOCK:
324     case F_FLOCKW:

```

```

326     /*
327     * Copy in input fields only.
328     */

330     if (cmd == F_O_GETLK) {
331         if (datamodel != DATAMODEL_ILP32) {
332             error = EINVAL;
333             break;
334         }

336         if (copyin((void *)arg, &obf, sizeof (obf))) {
337             error = EFAULT;
338             break;
339         }
340         bf.l_type = obf.l_type;
341         bf.l_whence = obf.l_whence;
342         bf.l_start = (off64_t)obf.l_start;
343         bf.l_len = (off64_t)obf.l_len;
344         bf.l_sysid = (int)obf.l_sysid;
345         bf.l_pid = obf.l_pid;
346     } else if (datamodel == DATAMODEL_NATIVE) {
347         if (copyin((void *)arg, &sbf, sizeof (sbf))) {
348             error = EFAULT;
349             break;
350         }
351         /*
352          * XXX In an LP64 kernel with an LP64 application
353          * there's no need to do a structure copy here
354          * struct flock == struct flock64. However,
355          * we did it this way to avoid more conditional
356          * compilation.
357          */
358         bf.l_type = sbf.l_type;
359         bf.l_whence = sbf.l_whence;
360         bf.l_start = (off64_t)sbf.l_start;
361         bf.l_len = (off64_t)sbf.l_len;
362         bf.l_sysid = sbf.l_sysid;
363         bf.l_pid = sbf.l_pid;
364     }
365 #if defined(_SYSCALL32_IMPL)
366     else {
367         struct flock32 sbf32;
368         if (copyin((void *)arg, &sbf32, sizeof (sbf32))) {
369             error = EFAULT;
370             break;
371         }
372         bf.l_type = sbf32.l_type;
373         bf.l_whence = sbf32.l_whence;
374         bf.l_start = (off64_t)sbf32.l_start;
375         bf.l_len = (off64_t)sbf32.l_len;
376         bf.l_sysid = sbf32.l_sysid;
377         bf.l_pid = sbf32.l_pid;
378     }
379 #endif /* _SYSCALL32_IMPL */

381     /*
382     * 64-bit support: check for overflow for 32-bit lock ops
383     */
384     if ((error = flock_check(vp, &bf, offset, maxoffset)) != 0)
385         break;

387     if (cmd == F_FLOCK || cmd == F_FLOCKW) {
388         /* FLOCK* locking is always over the entire file. */
389         if (bf.l_whence != 0 || bf.l_start != 0 ||
390             bf.l_len != 0) {
391             error = EINVAL;

```

```

392         break;
393     }
394     if (bf.l_type < F_RDLCK || bf.l_type > F_UNLCK) {
395         error = EINVAL;
396         break;
397     }
398 }

400 if (cmd == F_OFD_SETLK || cmd == F_OFD_SETLKW) {
401     /*
402     * TBD OFD-style locking is currently limited to
403     * covering the entire file.
404     */
405     if (bf.l_whence != 0 || bf.l_start != 0 ||
406         bf.l_len != 0) {
407         error = EINVAL;
408         break;
409     }
410 }

412 /*
413 * Not all of the filesystems understand F_O_GETLK, and
414 * there's no need for them to know. Map it to F_GETLK.
415 *
416 * The *_flock functions in the various file systems basically
417 * do some validation and then funnel everything through the
418 * fs_flock function. For OFD-style locks fs_flock will do
419 * nothing so that once control returns here we can call the
420 * ofdlock function with the correct fp. For OFD-style locks
421 * the unsupported remote file systems, such as NFS, detect and
422 * reject the OFD-style cmd argument.
423 */
424 if ((error = VOP_FRLOCK(vp, (cmd == F_O_GETLK) ? F_GETLK : cmd,
425     &bf, flag, offset, NULL, fp->f_cred, NULL)) != 0)
426     break;

428 if (cmd == F_FLOCK || cmd == F_FLOCKW || cmd == F_OFD_GETLK ||
429     cmd == F_OFD_SETLK || cmd == F_OFD_SETLKW) {
430     /*
431     * This is an OFD-style lock so we need to handle it
432     * here. Because OFD-style locks are associated with
433     * the file_t we didn't have enough info down the
434     * VOP_FRLOCK path immediately above.
435     */
436     if ((error = ofdlock(fp, cmd, &bf, flag, offset)) != 0)
437         break;
438 }

440 /*
441 * If command is GETLK and no lock is found, only
442 * the type field is changed.
443 */
444 if ((cmd == F_O_GETLK || cmd == F_GETLK ||
445     cmd == F_OFD_GETLK) && bf.l_type == F_UNLCK) {
446     /* l_type always first entry, always a short */
447     if (copyout(&bf.l_type, &((struct flock *)arg)->l_type,
448         sizeof(bf.l_type)))
449         error = EFAULT;
450     break;
451 }

453 if (cmd == F_O_GETLK) {
454     /*
455     * Return an SVR3 flock structure to the user.
456     */
457     obf.l_type = (int16_t)bf.l_type;

```

```

458     obf.l_whence = (int16_t)bf.l_whence;
459     obf.l_start = (int32_t)bf.l_start;
460     obf.l_len = (int32_t)bf.l_len;
461     if (bf.l_sysid > SHRT_MAX || bf.l_pid > SHRT_MAX) {
462         /*
463         * One or both values for the above fields
464         * is too large to store in an SVR3 flock
465         * structure.
466         */
467         error = EOVERFLOW;
468         break;
469     }
470     obf.l_sysid = (int16_t)bf.l_sysid;
471     obf.l_pid = (int16_t)bf.l_pid;
472     if (copyout(&obf, (void *)arg, sizeof(obf)))
473         error = EFAULT;
474 } else if (cmd == F_GETLK || cmd == F_OFD_GETLK) {
475     /*
476     * Copy out SVR4 flock.
477     */
478     int i;

480     if (bf.l_start > maxoffset || bf.l_len > maxoffset) {
481         error = EOVERFLOW;
482         break;
483     }

485     if (datamodel == DATAMODEL_NATIVE) {
486         for (i = 0; i < 4; i++)
487             sbf.l_pad[i] = 0;
488     }
489     /*
490     * XXX In an LP64 kernel with an LP64
491     * application there's no need to do a
492     * structure copy here as currently
493     * struct flock == struct flock64.
494     * We did it this way to avoid more
495     * conditional compilation.
496     */
497     sbf.l_type = bf.l_type;
498     sbf.l_whence = bf.l_whence;
499     sbf.l_start = (off_t)bf.l_start;
500     sbf.l_len = (off_t)bf.l_len;
501     sbf.l_sysid = bf.l_sysid;
502     sbf.l_pid = bf.l_pid;
503     if (copyout(&sbf, (void *)arg, sizeof(sbf)))
504         error = EFAULT;
505 }
506 #if defined(_SYSCALL32_IMPL)
507     else {
508         struct flock32 sbf32;
509         if (bf.l_start > MAXOFF32_T ||
510             bf.l_len > MAXOFF32_T) {
511             error = EOVERFLOW;
512             break;
513         }
514         for (i = 0; i < 4; i++)
515             sbf32.l_pad[i] = 0;
516         sbf32.l_type = (int16_t)bf.l_type;
517         sbf32.l_whence = (int16_t)bf.l_whence;
518         sbf32.l_start = (off32_t)bf.l_start;
519         sbf32.l_len = (off32_t)bf.l_len;
520         sbf32.l_sysid = (int32_t)bf.l_sysid;
521         sbf32.l_pid = (pid32_t)bf.l_pid;
522         if (copyout(&sbf32,
523             (void *)arg, sizeof(sbf32)))
524             error = EFAULT;
525     }

```

```

524     }
525 #endif
526     }
527     break;

529     case F_CHKFL:
530     /*
531      * This is for internal use only, to allow the vnode layer
532      * to validate a flags setting before applying it. User
533      * programs can't issue it.
534      */
535     error = EINVAL;
536     break;

538     case F_ALLOCSP:
539     case F_FREESP:
540     case F_ALLOCSP64:
541     case F_FREESP64:
542     /*
543      * Test for not-a-regular-file (and returning EINVAL)
544      * before testing for open-for-writing (and returning EBADF).
545      * This is relied upon by posix_fallocate() in libc.
546      */
547     if (vp->v_type != VREG) {
548         error = EINVAL;
549         break;
550     }

552     if ((flag & FWRITE) == 0) {
553         error = EBADF;
554         break;
555     }

557     if (datamodel != DATAMODEL_ILP32 &&
558         (cmd == F_ALLOCSP64 || cmd == F_FREESP64)) {
559         error = EINVAL;
560         break;
561     }

563 #if defined(_ILP32) || defined(_SYSCALL32_IMPL)
564     if (datamodel == DATAMODEL_ILP32 &&
565         (cmd == F_ALLOCSP || cmd == F_FREESP)) {
566         struct flock32 sbf32;
567         /*
568          * For compatibility we overlay an SVR3 flock on an SVR4
569          * flock. This works because the input field offsets
570          * in "struct flock" were preserved.
571          */
572         if (copyin((void *)arg, &sbf32, sizeof (sbf32))) {
573             error = EFAULT;
574             break;
575         } else {
576             bf.l_type = sbf32.l_type;
577             bf.l_whence = sbf32.l_whence;
578             bf.l_start = (off64_t)sbf32.l_start;
579             bf.l_len = (off64_t)sbf32.l_len;
580             bf.l_sysid = sbf32.l_sysid;
581             bf.l_pid = sbf32.l_pid;
582         }
583     }
584 #endif /* _ILP32 || _SYSCALL32_IMPL */

586 #if defined(_LP64)
587     if (datamodel == DATAMODEL_LP64 &&
588         (cmd == F_ALLOCSP || cmd == F_FREESP)) {
589         if (copyin((void *)arg, &bf, sizeof (bf))) {

```

```

590             error = EFAULT;
591             break;
592         }
593     }
594 #endif /* defined(_LP64) */

596 #if !defined(_LP64) || defined(_SYSCALL32_IMPL)
597     if (datamodel == DATAMODEL_ILP32 &&
598         (cmd == F_ALLOCSP64 || cmd == F_FREESP64)) {
599         if (copyin((void *)arg, &bf64_32, sizeof (bf64_32))) {
600             error = EFAULT;
601             break;
602         } else {
603             /*
604              * Note that the size of flock64 is different in
605              * the ILP32 and LP64 models, due to the l_pad
606              * field. We do not want to assume that the
607              * flock64 structure is laid out the same in
608              * ILP32 and LP64 environments, so we will
609              * copy in the ILP32 version of flock64
610              * explicitly and copy it to the native
611              * flock64 structure.
612              */
613             bf.l_type = (short)bf64_32.l_type;
614             bf.l_whence = (short)bf64_32.l_whence;
615             bf.l_start = bf64_32.l_start;
616             bf.l_len = bf64_32.l_len;
617             bf.l_sysid = (int)bf64_32.l_sysid;
618             bf.l_pid = (pid_t)bf64_32.l_pid;
619         }
620     }
621 #endif /* !defined(_LP64) || defined(_SYSCALL32_IMPL) */

623     if (cmd == F_ALLOCSP || cmd == F_FREESP)
624         error = flock_check(vp, &bf, offset, maxoffset);
625     else if (cmd == F_ALLOCSP64 || cmd == F_FREESP64)
626         error = flock_check(vp, &bf, offset, MAXOFFSET_T);
627     if (error)
628         break;

630     if (vp->v_type == VREG && bf.l_len == 0 &&
631         bf.l_start > OFFSET_MAX(fp)) {
632         error = EFBIG;
633         break;
634     }

636     /*
637      * Make sure that there are no conflicting non-blocking
638      * mandatory locks in the region being manipulated. If
639      * there are such locks then return EACCES.
640      */
641     if ((error = flock_get_start(vp, &bf, offset, &start)) != 0)
642         break;

644     if (nbl_need_check(vp)) {
645         u_offset_t    begin;
646         ssize_t       length;

648         nbl_start_crit(vp, RW_READER);
649         in_crit = 1;
650         vattr.va_mask = AT_SIZE;
651         if ((error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
652             != 0)
653             break;
654         begin = start > vattr.va_size ? vattr.va_size : start;
655         length = vattr.va_size > start ? vattr.va_size - start :

```

```

656         start - vattr.va_size;
657         if (nbl_conflict(vp, NBL_WRITE, begin, length, 0,
658             NULL)) {
659             error = EACCES;
660             break;
661         }
662     }
663
664     if (cmd == F_ALLOCSP64)
665         cmd = F_ALLOCSP;
666     else if (cmd == F_FREESP64)
667         cmd = F_FREESP;
668
669     error = VOP_SPACE(vp, cmd, &bf, flag, offset, fp->f_cred, NULL);
670
671     break;
672
673 #if !defined(LP64) || defined(_SYSALL32_IMPL)
674     case F_GETLK64:
675     case F_SETLK64:
676     case F_SETLKW64:
677     case F_SETLK64_NBMAND:
678     case F_OFD_GETLK64:
679     case F_OFD_SETLK64:
680     case F_OFD_SETLKW64:
681     case F_FLOCK64:
682     case F_FLOCKW64:
683         /*
684          * Large Files: Here we set cmd as *LK and send it to
685          * lower layers. *LK64 is only for the user land.
686          * Most of the comments described above for F_SETLK
687          * applies here too.
688          * Large File support is only needed for ILP32 apps!
689          */
690         if (datamodel != DATAMODEL_ILP32) {
691             error = EINVAL;
692             break;
693         }
694
695         if (cmd == F_GETLK64)
696             cmd = F_GETLK;
697         else if (cmd == F_SETLK64)
698             cmd = F_SETLK;
699         else if (cmd == F_SETLKW64)
700             cmd = F_SETLKW;
701         else if (cmd == F_SETLK64_NBMAND)
702             cmd = F_SETLK_NBMAND;
703         else if (cmd == F_OFD_GETLK64)
704             cmd = F_OFD_GETLK;
705         else if (cmd == F_OFD_SETLK64)
706             cmd = F_OFD_SETLK;
707         else if (cmd == F_OFD_SETLKW64)
708             cmd = F_OFD_SETLKW;
709         else if (cmd == F_FLOCK64)
710             cmd = F_FLOCK;
711         else if (cmd == F_FLOCKW64)
712             cmd = F_FLOCKW;
713
714         /*
715          * Note that the size of flock64 is different in the ILP32
716          * and LP64 models, due to the sucking l_pad field.
717          * We do not want to assume that the flock64 structure is
718          * laid out in the same in ILP32 and LP64 environments, so
719          * we will copy in the ILP32 version of flock64 explicitly
720          * and copy it to the native flock64 structure.
721          */

```

```

723     if (copyin((void *)arg, &bf64_32, sizeof (bf64_32))) {
724         error = EFAULT;
725         break;
726     }
727
728     bf.l_type = (short)bf64_32.l_type;
729     bf.l_whence = (short)bf64_32.l_whence;
730     bf.l_start = bf64_32.l_start;
731     bf.l_len = bf64_32.l_len;
732     bf.l_sysid = (int)bf64_32.l_sysid;
733     bf.l_pid = (pid_t)bf64_32.l_pid;
734
735     if ((error = flock_check(vp, &bf, offset, MAXOFFSET_T)) != 0)
736         break;
737
738     if (cmd == F_FLOCK || cmd == F_FLOCKW) {
739         /* FLOCK* locking is always over the entire file. */
740         if (bf.l_whence != 0 || bf.l_start != 0 ||
741             bf.l_len != 0) {
742             error = EINVAL;
743             break;
744         }
745         if (bf.l_type < F_RDLCK || bf.l_type > F_UNLCK) {
746             error = EINVAL;
747             break;
748         }
749     }
750
751     if (cmd == F_OFD_SETLK || cmd == F_OFD_SETLKW) {
752         /*
753          * TBD OFD-style locking is currently limited to
754          * covering the entire file.
755          */
756         if (bf.l_whence != 0 || bf.l_start != 0 ||
757             bf.l_len != 0) {
758             error = EINVAL;
759             break;
760         }
761     }
762
763     /*
764      * The *_flock functions in the various file systems basically
765      * do some validation and then funnel everything through the
766      * fs_flock function. For OFD-style locks fs_flock will do
767      * nothing so that once control returns here we can call the
768      * ofdlock function with the correct fp. For OFD-style locks
769      * the unsupported remote file systems, such as NFS, detect and
770      * reject the OFD-style cmd argument.
771      */
772     if ((error = VOP_FRLOCK(vp, cmd, &bf, flag, offset,
773         NULL, fp->f_cred, NULL)) != 0)
774         break;
775
776     if (cmd == F_FLOCK || cmd == F_FLOCKW || cmd == F_OFD_GETLK ||
777         cmd == F_OFD_SETLK || cmd == F_OFD_SETLKW) {
778         /*
779          * This is an OFD-style lock so we need to handle it
780          * here. Because OFD-style locks are associated with
781          * the file_t we didn't have enough info down the
782          * VOP_FRLOCK path immediately above.
783          */
784         if ((error = ofdlock(fp, cmd, &bf, flag, offset)) != 0)
785             break;
786     }

```

```

788     if ((cmd == F_GETLK || cmd == F_OFD_GETLK) &&
789         bf.l_type == F_UNLCK) {
790         if (copyout(&bf.l_type, &((struct flock *)arg)->l_type,
791             sizeof (bf.l_type)))
792             error = EFAULT;
793         break;
794     }

796     if (cmd == F_GETLK || cmd == F_OFD_GETLK) {
797         int i;

799         /*
800          * We do not want to assume that the flock64 structure
801          * is laid out in the same in ILP32 and LP64
802          * environments, so we will copy out the ILP32 version
803          * of flock64 explicitly after copying the native
804          * flock64 structure to it.
805          */
806         for (i = 0; i < 4; i++)
807             bf64_32.l_pad[i] = 0;
808         bf64_32.l_type = (int16_t)bf.l_type;
809         bf64_32.l_whence = (int16_t)bf.l_whence;
810         bf64_32.l_start = bf.l_start;
811         bf64_32.l_len = bf.l_len;
812         bf64_32.l_sysid = (int32_t)bf.l_sysid;
813         bf64_32.l_pid = (pid32_t)bf.l_pid;
814         if (copyout(&bf64_32, (void *)arg, sizeof (bf64_32)))
815             error = EFAULT;
816     }
817     break;
818 #endif /* !defined(LP64) || defined(_SYSCALL32_IMPL) */

820     case F_SHARE:
821     case F_SHARE_NEMAND:
822     case F_UNSHARE:

824         /*
825          * Copy in input fields only.
826          */
827         if (copyin((void *)arg, &fsh, sizeof (fsh))) {
828             error = EFAULT;
829             break;
830         }

832         /*
833          * Local share reservations always have this simple form
834          */
835         shr.s_access = fsh.f_access;
836         shr.s_deny = fsh.f_deny;
837         shr.s_sysid = 0;
838         shr.s_pid = ttproc(curthread)->p_pid;
839         shr_own.sl_pid = shr.s_pid;
840         shr_own.sl_id = fsh.f_id;
841         shr.s_own_len = sizeof (shr_own);
842         shr.s_owner = (caddr_t)&shr_own;
843         error = VOP_SHRLOCK(vp, cmd, &shr, flag, fp->f_cred, NULL);
844         break;

846     default:
847         error = EINVAL;
848         break;
849 }

851 if (in_crit)
852     nbl_end_crit(vp);

```

```

854 done:
855     releasef(fdex);
856 out:
857     if (error)
858         return (set_errno(error));
859     return (retval);
860 }

862 int
863 flock_check(vnode_t *vp, flock64_t *flp, offset_t offset, offset_t max)
864 {
865     struct vattr    vattr;
866     int            error;
867     u_offset_t     start, end;

869     /*
870      * Determine the starting point of the request
871      */
872     switch (flp->l_whence) {
873     case 0:        /* SEEK_SET */
874         start = (u_offset_t)flp->l_start;
875         if (start > max)
876             return (EINVAL);
877         break;
878     case 1:        /* SEEK_CUR */
879         if (flp->l_start > (max - offset))
880             return (EOVERFLOW);
881         start = (u_offset_t)(flp->l_start + offset);
882         if (start > max)
883             return (EINVAL);
884         break;
885     case 2:        /* SEEK_END */
886         vattr.va_mask = AT_SIZE;
887         if (error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
888             return (error);
889         if (flp->l_start > (max - (offset_t)vattr.va_size))
890             return (EOVERFLOW);
891         start = (u_offset_t)(flp->l_start + (offset_t)vattr.va_size);
892         if (start > max)
893             return (EINVAL);
894         break;
895     default:
896         return (EINVAL);
897     }

899     /*
900      * Determine the range covered by the request.
901      */
902     if (flp->l_len == 0)
903         end = MAXEND;
904     else if ((offset_t)flp->l_len > 0) {
905         if (flp->l_len > (max - start + 1))
906             return (EOVERFLOW);
907         end = (u_offset_t)(start + (flp->l_len - 1));
908         ASSERT(end <= max);
909     } else {
910         /*
911          * Negative length; why do we even allow this ?
912          * Because this allows easy specification of
913          * the last n bytes of the file.
914          */
915         end = start;
916         start += (u_offset_t)flp->l_len;
917         (start)++;
918         if (start > max)
919             return (EINVAL);

```

```
920     ASSERT(end <= max);
921 }
922 ASSERT(start <= max);
923 if (flp->l_type == F_UNLCK && flp->l_len > 0 &&
924     end == (offset_t)max) {
925     flp->l_len = 0;
926 }
927 if (start > end)
928     return (EINVAL);
929 return (0);
930 }

932 static int
933 flock_get_start(vnode_t *vp, flock64_t *flp, offset_t offset, u_offset_t *start)
934 {
935     struct vattr    vattr;
936     int             error;

938     /*
939      * Determine the starting point of the request. Assume that it is
940      * a valid starting point.
941      */
942     switch (flp->l_whence) {
943     case 0: /* SEEK_SET */
944         *start = (u_offset_t)flp->l_start;
945         break;
946     case 1: /* SEEK_CUR */
947         *start = (u_offset_t)(flp->l_start + offset);
948         break;
949     case 2: /* SEEK_END */
950         vattr.va_mask = AT_SIZE;
951         if (error = VOP_GETATTR(vp, &vattr, 0, CRED(), NULL))
952             return (error);
953         *start = (u_offset_t)(flp->l_start + (offset_t)vattr.va_size);
954         break;
955     default:
956         return (EINVAL);
957     }

959     return (0);
960 }

962 /*
963  * Take rctl action when the requested file descriptor is too big.
964  */
965 static void
966 fd_too_big(proc_t *p)
967 {
968     mutex_enter(&p->p_lock);
969     (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
970         p->p_rctls, p, RCA_SAFE);
971     mutex_exit(&p->p_lock);
972 }
```