

```

*****
4270 Wed Jun 17 00:15:40 2009
new/usr/src/cmd/fs.d/nfs/etc/nfs.dfl
4953763 Need way to configure NFS window sizes without changing system wide defa
6216670 NFS server needs a bigger transmit buffer
*****
1 # ident "%Z%M% %I% %E% SMI"
1 #
2 # Copyright 2009 Sun Microsystems, Inc. All rights reserved.
3 # Copyright 2005 Sun Microsystems, Inc. All rights reserved.
3 # Use is subject to license terms.
4 #
5 # CDDL HEADER START
6 #
7 # The contents of this file are subject to the terms of the
8 # Common Development and Distribution License (the "License").
9 # You may not use this file except in compliance with the License.
9 # Common Development and Distribution License, Version 1.0 only
10 # (the "License"). You may not use this file except in compliance
11 # with the License.
10 #
11 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
12 # or http://www.opensolaris.org/os/licensing.
13 # See the License for the specific language governing permissions
14 # and limitations under the License.
15 #
16 # When distributing Covered Code, include this CDDL HEADER in each
17 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
18 # If applicable, add the following below this CDDL HEADER, with the
19 # fields enclosed by brackets "[]" replaced with your own identifying
20 # information: Portions Copyright [yyyy] [name of copyright owner]
21 #
22 # CDDL HEADER END
23 #

25 # Sets the maximum number of concurrent connection oriented connections.
26 # Default is unlimited and is obtained by not setting NFSD_MAX_CONNECTIONS.
27 # Equivalent to -c.
28 #NFSD_MAX_CONNECTIONS=

30 # Set connection queue length for the NFS over a connection-oriented
31 # transport. The default value is 32 entries.
32 # Equivalent to -l.
33 NFSD_LISTEN_BACKLOG=32

35 # Start NFS daemon over the specified protocol only.
36 # Equivalent to -p, ALL is equivalent to -a on the nfsd command line.
37 # Mutually exclusive with NFSD_DEVICE.
38 NFSD_PROTOCOL=ALL

40 # Start NFS daemon for the transport specified by the given device only.
41 # Equivalent to -t.
42 # Mutually exclusive with setting NFSD_PROTOCOL.
43 #NFSD_DEVICE=

45 # Maximum number of concurrent NFS requests.
46 # Equivalent to last numeric argument on nfsd command line.
47 NFSD_SERVERS=16

49 # Set connection queue length for lockd over a connection-oriented transport.
50 # Default and minimum value is 32.
51 LOCKD_LISTEN_BACKLOG=32

53 # Maximum number of concurrent lockd requests.
54 # Default is 20.
55 LOCKD_SERVERS=20

```

```

57 # Retransmit Timeout before lockd tries again.
58 # Default is 5.
59 LOCKD_RETRANSMIT_TIMEOUT=5

61 # Grace period in seconds that all clients (both NLM & NFSv4) have to
62 # reclaim locks after a server reboot. Also controls the NFSv4 lease
63 # interval.
64 # Overrides the deprecated setting LOCKD_GRACE_PERIOD.
65 # Default is 90 seconds.
66 GRACE_PERIOD=90

68 # Deprecated.
69 # As for GRACE_PERIOD, above.
70 # Default is 90 seconds.
71 #LOCKD_GRACE_PERIOD=90

73 # Sets the minimum version of the NFS protocol that will be registered
74 # and offered by the server. The default is 2.
75 #NFS_SERVER_VERSMIN=2

77 # Sets the maximum version of the NFS protocol that will be registered
78 # and offered by the server. The default is 4.
79 #NFS_SERVER_VERSMAX=4

81 # Sets the minimum version of the NFS protocol that will be used by
82 # the NFS client. Can be overridden by the "vers=" NFS mount option.
83 # The default is 2.
84 #NFS_CLIENT_VERSMIN=2

86 # Sets the maximum version of the NFS protocol that will be used by
87 # the NFS client. Can be overridden by the "vers=" NFS mount option.
88 # If "vers=" is not specified for an NFS mount, this is the version
89 # that will be attempted first. The default is 4.
90 #NFS_CLIENT_VERSMAX=4

92 # Determines if the NFS version 4 delegation feature will be enabled
93 # for the server. If it is enabled, the server will attempt to
94 # provide delegations to the NFS version 4 client. The default is on.
95 #NFS_SERVER_DELEGATION=on

97 # Specifies to nfsmapid daemon that it is to override its default
98 # behavior of using the DNS domain, and that it is to use 'domain' as
99 # the domain to append to outbound attribute strings, and that it is to
100 # use 'domain' to compare against inbound attribute strings.
101 #NFSMAPID_DOMAIN=domain

103 # Specifies TCP send and receive buffer size of NFS server connections.
104 #
105 # To override the default values and force NFS connections to use system-wide
106 # default TCP send and receive buffer size, set the corresponding option to 0.
107 #
108 # Default is 1048576 bytes, which is the current maximum allowable buffer size
109 # limited by system-wide 'tcp_max_buf' configuration variable. To set the buffer
110 # size beyond the current maximum allowable value, increase 'tcp_max_buf' to
111 # a value greater than, or equal to, the desired value for NFS_SERVER_SNDBUFSZ
112 # and NFS_SERVER_RCVBUFSZ.
113 #
114 #NFS_SERVER_SNDBUFSZ=1048576
115 #NFS_SERVER_RCVBUFSZ=1048576

```

new/usr/src/cmd/fs.d/nfs/lib/nfs_tbind.h

1

```
*****
2855 Wed Jun 17 00:15:41 2009
new/usr/src/cmd/fs.d/nfs/lib/nfs_tbind.h
4953763 Need way to configure NFS window sizes without changing system wide defa
6216670 NFS server needs a bigger transmit buffer
*****
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 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */
26 /*
27 * nfs_tbind.h, common code for nfsd and lockd
28 */
29
30 #ifndef _NFS_TBIND_H
31 #define _NFS_TBIND_H
32
33 #pragma ident "%Z%M% %I% %E% SMI"
34
35 #include <netconfig.h>
36 #include <netdir.h>
37
38 #ifdef __cplusplus
39 extern "C" {
40 #endif
41
42 /*
43  * Globals which should be initialised by daemon main().
44  */
45 extern size_t end_listen_fds;
46 extern size_t num_fds;
47 extern int listen_backlog;
48 extern int (*Mysvc)(int, struct netbuf, struct netconfig *);
49 extern int (*Mysvc4)(int, struct netbuf *, struct netconfig *,
50 int, struct netbuf *);
51 extern int max_conns_allowed;
52
53 /*
54  * RPC protocol block. Useful for passing registration information.
55  */
56 struct protob {
57     char *serv; /* ASCII service name, e.g. "NFS" */
58     int versmin; /* minimum version no. to be registered */
59     int versmax; /* maximum version no. to be registered */
60 };
61
62 #endif
63
64 #endif
65
66 #endif
```

new/usr/src/cmd/fs.d/nfs/lib/nfs_tbind.h

2

```
58     int program; /* program no. to be registered */
59     struct protob *next; /* next entry on list */
60 };
61
62 /*
63  * Declarations for protocol types and comparison.
64  */
65 #define NETSELDECL(x) char *x
66 #define NETSELPDECL(x) char **x
67 #define NETSELEQ(x, y) (strcmp((x), (y)) == 0)
68
69 /*
70  * nfs library routines
71  */
72 extern int nfslib_transport_open(struct netconfig *);
73 extern int nfslib_bindit(struct netconfig *, struct netbuf **,
74 struct nd_hostserv *, int, int sndbufsz, int rcvbufsz);
75 extern void nfslib_log_tli_error(char *, int, struct netconfig *);
76 extern int do_all_setbuf(struct protob *,
77 int (*)(int, struct netbuf, struct netconfig *),
78 int use_pmap, int sndbufsz, int rcvbufsz);
79 extern int do_all(struct protob *,
80 int (*)(int, struct netbuf, struct netconfig *),
81 int use_pmap);
82 extern void do_one_setbuf(char *, char *, struct protob *,
83 int (*)(int, struct netbuf, struct netconfig *),
84 int use_pmap, int sndbufsz, int rcvbufsz);
85 extern void do_one(char *, char *, struct protob *,
86 int (*)(int, struct netbuf, struct netconfig *),
87 int use_pmap);
88 extern void poll_for_action(void);
89
90 #ifdef __cplusplus
91 }
92
93 unchanged_portion_omitted
94
95 #endif
```

```

*****
44823 Wed Jun 17 00:15:42 2009
new/usr/src/cmd/fs.d/nfs/lib/nfs_tbind.c
4953763 Need way to configure NFS window sizes without changing system wide defa
6216670 NFS server needs a bigger transmit buffer
*****
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  * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25 */

26 /*
27  * nfs_tbind.c, common part for nfsd and lockd.
28 */

30 #pragma ident "%Z%%M% %I% %E% SMI"

30 #define PORTMAP

32 #include <tiuser.h>
33 #include <fcntl.h>
34 #include <netconfig.h>
35 #include <stropts.h>
36 #include <errno.h>
37 #include <syslog.h>
38 #include <rpc/rpc.h>
39 #include <rpc/pmap_prot.h>
40 #include <sys/time.h>
41 #include <sys/resource.h>
42 #include <signal.h>
43 #include <netdir.h>
44 #include <unistd.h>
45 #include <string.h>
46 #include <netinet/tcp.h>
47 #include <malloc.h>
48 #include <stdlib.h>
49 #include "nfs_tbind.h"
50 #include <nfs/nfs.h>
51 #include <nfs/nfs_acl.h>
52 #include <nfs/nfssys.h>
53 #include <nfs/nfs4.h>
54 #include <zone.h>
55 #include <sys/socket.h>
56 #include <tsol/label.h>

```

```

58 /*
59  * Determine valid semantics for most applications.
60 */
61 #define OK_TPI_TYPE(_nconf) \
62     (_nconf->nc_semantics == NC_TPI_CLTS || \
63      _nconf->nc_semantics == NC_TPI_COTS || \
64      _nconf->nc_semantics == NC_TPI_COTS_ORD)

66 #define BE32_TO_U32(a) \
67     (((ulong_t)((uchar_t *)a)[0] & 0xFF) << (ulong_t)24) | \
68     (((ulong_t)((uchar_t *)a)[1] & 0xFF) << (ulong_t)16) | \
69     (((ulong_t)((uchar_t *)a)[2] & 0xFF) << (ulong_t)8) | \
70     ((ulong_t)((uchar_t *)a)[3] & 0xFF)

72 /*
73  * Number of elements to add to the poll array on each allocation.
74 */
75 #define POLL_ARRAY_INC_SIZE    64

77 /*
78  * Number of file descriptors by which the process soft limit may be
79  * increased on each call to nofile_increas(0).
80 */
81 #define NOFILE_INC_SIZE 64

83 struct conn_ind {
84     struct conn_ind *conn_next;
85     struct conn_ind *conn_prev;
86     struct t_call    *conn_call;
87 };
88
89 unchanged_portion_omitted

94 /*
95  * this file contains transport routines common to nfsd and lockd
96 */
97 static int    nofile_increas(int);
98 static int    reuseaddr(int);
99 static int    recvucred(int);
100 static int    anonmlp(int);
101 static void   add_to_poll_list(int, struct netconfig *);
102 static char   *serv_name_to_port_name(char *);
103 static int    bind_to_proto(char *, char *, struct netbuf **,
104                             struct netconfig **, int, int);
105 static int    bind_to_provider(char *, char *, struct netbuf **,
106                                struct netconfig **, int, int);
107 static void   conn_close_oldest(void);
108 static boolean_t conn_get(int, struct netconfig *, struct conn_ind **);
109 static void   cots_listen_event(int, int);
110 static int    discon_get(int, struct netconfig *, struct conn_ind **);
111 static int    do_poll_clts_action(int, int);
112 static int    do_poll_cots_action(int, int);
113 static void   remove_from_poll_list(int);
114 static int    set_addrmask(int, struct netconfig *, struct netbuf *);
115 static int    is_listen_fd_index(int);

117 static struct pollfd *poll_array;
118 static struct conn_entry *conn_polled;
119 static int    num_conns; /* Current number of connections */
120 int          (*Mysvc4)(int, struct netbuf *, struct netconfig *, int,
121                       struct netbuf *);
122 static int    setopt(int fd, int level, int name, int value);

124 extern boolean_t __pmap_set(const rpcprog_t program, const rpcvers_t version,
125                             const struct netconfig *nconf, const struct netbuf *address);

```

```

127 /*
128 * Called to create and prepare a transport descriptor for in-kernel
129 * RPC service.
130 * Returns -1 on failure and a valid descriptor on success.
131 */
132 int
133 nfslib_transport_open(struct netconfig *nconf)
134 {
135     int fd;
136     struct striocctl strioc;

138     if ((nconf == (struct netconfig *)NULL) ||
139         (nconf->nc_device == (char *)NULL)) {
140         syslog(LOG_ERR, "no netconfig device");
141         return (-1);
142     }

144     /*
145      * Open the transport device.
146      */
147     fd = t_open(nconf->nc_device, O_RDWR, (struct t_info *)NULL);
148     if (fd == -1) {
149         if (t_errno == TSYSEERR && errno == EMFILE &&
150             (nofile_increase(0) == 0)) {
151             /* Try again with a higher NOFILE limit. */
152             fd = t_open(nconf->nc_device, O_RDWR,
153                 (struct t_info *)NULL);
154         }
155         if (fd == -1) {
156             syslog(LOG_ERR, "t_open %s failed: t_errno %d, %m",
157                 nconf->nc_device, t_errno);
158             return (-1);
159         }
160     }

162     /*
163      * Pop timod because the RPC module must be as close as possible
164      * to the transport.
165      */
166     if (ioctl(fd, I_POP, 0) < 0) {
167         syslog(LOG_ERR, "I_POP of timod failed: %m");
168         (void) t_close(fd);
169         return (-1);
170     }

172     /*
173      * Common code for CLTS and COTS transports
174      */
175     if (ioctl(fd, I_PUSH, "rpcmod") < 0) {
176         syslog(LOG_ERR, "I_PUSH of rpcmod failed: %m");
177         (void) t_close(fd);
178         return (-1);
179     }

181     strioc.ic_cmd = RPC_SERVER;
182     strioc.ic_dp = (char *)0;
183     strioc.ic_len = 0;
184     strioc.ic_timeout = -1;

186     /* Tell rpcmod to act like a server stream. */
187     if (ioctl(fd, I_STR, &strioc) < 0) {
188         syslog(LOG_ERR, "rpcmod set-up ioctl failed: %m");
189         (void) t_close(fd);
190         return (-1);
191     }

```

```

193     /*
194      * Re-push timod so that we will still be doing TLI
195      * operations on the descriptor.
196      */
197     if (ioctl(fd, I_PUSH, "timod") < 0) {
198         syslog(LOG_ERR, "I_PUSH of timod failed: %m");
199         (void) t_close(fd);
200         return (-1);
201     }

203     /*
204      * Enable options of returning the ip's for udp.
205      */
206     if (strcmp(nconf->nc_netid, "udp6") == 0)
207         __rpc_tli_set_options(fd, IPPROTO_IPV6, IPV6_RECVPKTINFO, 1);
208     else if (strcmp(nconf->nc_netid, "udp") == 0)
209         __rpc_tli_set_options(fd, IPPROTO_IP, IP_RECVDSTADDR, 1);

211     return (fd);
212 }

_____unchanged_portion_omitted_____

242 static int
243 nfslib_set_sockbuf(int fd, int which, int val)
244 {
245     if ((which != SO_RCVBUF) && (which != SO_SNDBUF))
246         return (-1);

248     syslog(LOG_DEBUG, "Set %s option to %d",
249         ((which == SO_RCVBUF) ? "SO_RCVBUF" : "SO_SNDBUF"), val);

251     if (setopt(fd, SOL_SOCKET, which, val) < 0) {
252         syslog(LOG_ERR, "couldn't set %s to %d - t_errno = %d",
253             ((which == SO_RCVBUF) ? "SO_RCVBUF" : "SO_SNDBUF"),
254             val, t_errno);
255         syslog(LOG_ERR, "Check and increase system-wide tcp_max_buf");
256         return (-1);
257     }
258     return (0);
259 }

261 int
262 nfslib_bindit(struct netconfig *nconf, struct netbuf **addr,
263     struct nd_hostserv *hs, int backlog, int sndbufsz, int rcvbufsz)
264 {
265     int fd;
266     struct t_bind *ntb;
267     struct t_bind tb;
268     struct nd_addrlist *addrlist;
269     struct t_optmgmt req, resp;
270     struct opthdr *opt;
271     char reqbuf[128];
272     bool_t use_any = FALSE;
273     bool_t gzone = TRUE;

275     if ((fd = nfslib_transport_open(nconf)) == -1) {
276         syslog(LOG_ERR, "cannot establish transport service over %s",
277             nconf->nc_device);
278         return (-1);
279     }

281     addrlist = (struct nd_addrlist *)NULL;

283     /* nfs4_callback service does not used a fixed port number */

```

```

285     if (strcmp(hs->h_serv, "nfs4_callback") == 0) {
286         tb.addr.maxlen = 0;
287         tb.addr.len = 0;
288         tb.addr.buf = 0;
289         use_any = TRUE;
290         gzone = (getzoneid() == GLOBAL_ZONEID);
291     } else if (netdir_getbyname(nconf, hs, &addrlist) != 0) {
292
293         syslog(LOG_ERR,
294             "Cannot get address for transport %s host %s service %s",
295             nconf->nc_netid, hs->h_host, hs->h_serv);
296         (void) t_close(fd);
297         return (-1);
298     }
299
300     if (strcmp(nconf->nc_proto, "tcp") == 0) {
301         /*
302          * If we're running over TCP, then set the
303          * SO_REUSEADDR option so that we can bind
304          * to our preferred address even if previously
305          * left connections exist in FIN_WAIT states.
306          * This is somewhat bogus, but otherwise you have
307          * to wait 2 minutes to restart after killing it.
308          */
309         if (reuseaddr(fd) == -1) {
310             syslog(LOG_WARNING,
311                 "couldn't set SO_REUSEADDR option on transport");
312         }
313     } else if (strcmp(nconf->nc_proto, "udp") == 0) {
314         /*
315          * In order to run MLP on UDP, we need to handle creds.
316          */
317         if (recvucred(fd) == -1) {
318             syslog(LOG_WARNING,
319                 "couldn't set SO_RECVUCRED option on transport");
320         }
321     }
322
323     /*
324     * Make non global zone nfs4_callback port MLP
325     */
326     if (use_any && is_system_labeled() && !gzone) {
327         if (anonmlp(fd) == -1) {
328             /*
329              * failing to set this option means nfs4_callback
330              * could fail silently later. So fail it with
331              * with an error message now.
332              */
333             syslog(LOG_ERR,
334                 "couldn't set SO_ANON_MLP option on transport");
335             (void) t_close(fd);
336             return (-1);
337         }
338     }
339
340     if (nconf->nc_semantics == NC_TPI_CLTS)
341         tb.qlen = 0;
342     else
343         tb.qlen = backlog;
344
345     /* LINTED pointer alignment */
346     ntb = (struct t_bind *)t_alloc(fd, T_BIND, T_ALL);
347     if (ntb == (struct t_bind *)NULL) {
348         syslog(LOG_ERR, "t_alloc failed: t_errno %d, %m", t_errno);
349         (void) t_close(fd);

```

```

350         netdir_free((void *)addrlist, ND_ADDRLIST);
351         return (-1);
352     }
353
354     /*
355     * XXX - what about the space tb->addr.buf points to? This should
356     * be either a memcpy() to/from the buf fields, or t_alloc(fd, T_BIND,)
357     * should't be called with T_ALL.
358     */
359     if (addrlist)
360         tb.addr = *(addrlist->n_addrs); /* structure copy */
361
362     if (t_bind(fd, &tb, ntb) == -1) {
363         syslog(LOG_ERR, "t_bind failed: t_errno %d, %m", t_errno);
364         (void) t_free((char *)ntb, T_BIND);
365         netdir_free((void *)addrlist, ND_ADDRLIST);
366         (void) t_close(fd);
367         return (-1);
368     }
369
370     /* make sure we bound to the right address */
371     if (use_any == FALSE &&
372         (tb.addr.len != ntb->addr.len ||
373          memcmp(tb.addr.buf, ntb->addr.buf, tb.addr.len) != 0)) {
374         syslog(LOG_ERR, "t_bind to wrong address");
375         (void) t_free((char *)ntb, T_BIND);
376         netdir_free((void *)addrlist, ND_ADDRLIST);
377         (void) t_close(fd);
378         return (-1);
379     }
380
381     /*
382     * Call nfs4svc_setport so that the kernel can be
383     * informed what port number the daemon is listing
384     * for incoming connection requests.
385     */
386
387     if ((nconf->nc_semantics == NC_TPI_COTS ||
388         nconf->nc_semantics == NC_TPI_COTS_ORD) && Mysvc4 != NULL)
389         (*Mysvc4)(fd, NULL, nconf, NFS4_SETPORT, &ntb->addr);
390
391     *addr = &ntb->addr;
392     netdir_free((void *)addrlist, ND_ADDRLIST);
393
394     if (strcmp(nconf->nc_proto, "tcp") == 0) {
395         /*
396          * Disable the Nagle algorithm on TCP connections.
397          * Connections accepted from this listener will
398          * inherit the listener options.
399          */
400
401         /* LINTED pointer alignment */
402         opt = (struct ophdr *)reqbuf;
403         opt->level = IPPROTO_TCP;
404         opt->name = TCP_NODELAY;
405         opt->len = sizeof (int);
406
407         /* LINTED pointer alignment */
408         *(int *)((char *)opt + sizeof (*opt)) = 1;
409
410         req.flags = T_NEGOTIATE;
411         req.opt.len = sizeof (*opt) + opt->len;
412         req.opt.buf = (char *)opt;
413         resp.flags = 0;
414         resp.opt.buf = reqbuf;
415         resp.opt.maxlen = sizeof (reqbuf);

```

```

417         if (t_optmgmt(fd, &req, &resp) < 0 ||
418             resp.flags != T_SUCCESS) {
419             syslog(LOG_ERR,
420                 "couldn't set NODELAY option for proto %s: t_errno = %d, %m",
421                 nconf->nc_proto, t_errno);
422         }

424         if (sndbufsz > 0)
425             (void) nfslib_set_sockbuf(fd, SO_SNDBUF, sndbufsz);
426         if (rcvbufsz > 0)
427             (void) nfslib_set_sockbuf(fd, SO_RCVBUF, rcvbufsz);
428     }

430     return (fd);
431 }
    unchanged portion omitted

502 /*
503  * Called to set up service over a particular transport also
504  * set send and receive buffer size for transport connection.
505  * Called to set up service over a particular transport.
506 */
507 void
508 do_one_setbuf(char *provider, NETSELDECL(proto), struct protob *protobp0,
509              int (*svc)(int, struct netbuf, struct netconfig *), int use_pmap,
510              int sndbufsz, int rcvbufsz)
511 do_one(char *provider, NETSELDECL(proto), struct protob *protobp0,
512        int (*svc)(int, struct netbuf, struct netconfig *), int use_pmap)
513 {
514     register int sock;
515     struct protob *protobp;
516     struct netbuf *retaddr;
517     struct netconfig *retncnf;
518     struct netbuf addrmask;
519     int vers;
520     int err;
521     int l;

522     if (provider)
523         sock = bind_to_provider(provider, protobp0->serv, &retaddr,
524                                 &retncnf, sndbufsz, rcvbufsz);
525     else
526         sock = bind_to_proto(proto, protobp0->serv, &retaddr,
527                               &retncnf, sndbufsz, rcvbufsz);

528     if (sock == -1) {
529         (void) syslog(LOG_ERR,
530             "Cannot establish %s service over %s: transport setup problem.",
531             protobp0->serv, provider ? provider : proto);
532         return;
533     }

534     if (set_addrmask(sock, retncnf, &addrmask) < 0) {
535         (void) syslog(LOG_ERR,
536             "Cannot set address mask for %s", retncnf->nc_netid);
537         return;
538     }

539     /*
540      * Register all versions of the programs in the protocol block list.
541      */
542     l = strlen(NC_UDP);
543     for (protobp = protobp0; protobp; protobp = protobp->next) {

```

```

545         for (vers = protobp->versmin; vers <= protobp->versmax;
546             vers++) {
547             if ((protobp->program == NFS_PROGRAM ||
548                 protobp->program == NFS_ACL_PROGRAM) &&
549                 vers == NFS_V4 &&
550                 strncasecmp(retncnf->nc_proto, NC_UDP, 1) == 0)
551                 continue;

552             if (use_pmap) {
553                 /*
554                  * Note that if we're using a portmapper
555                  * instead of rpcbind then we can't do an
556                  * unregister operation here.
557                  *
558                  * The reason is that the portmapper unset
559                  * operation removes all the entries for a
560                  * given program/version regardless of
561                  * transport protocol.
562                  *
563                  * The caller of this routine needs to ensure
564                  * that __pmap_unset() has been called for all
565                  * program/version service pairs they plan
566                  * to support before they start registering
567                  * each program/version/protocol triplet.
568                  */
569                 (void) __pmap_set(protobp->program, vers,
570                                 retncnf, retaddr);
571             } else {
572                 (void) rpcb_unset(protobp->program, vers,
573                                 retncnf);
574                 (void) rpcb_set(protobp->program, vers,
575                                 retncnf, retaddr);
576             }
577         }
578     }
579 }

580 if (retncnf->nc_semantics == NC_TPI_CLTS) {
581     /* Don't drop core if supporting module(s) aren't loaded. */
582     (void) signal(SIGSYS, SIG_IGN);

583 }

584 /*
585  * svc() doesn't block, it returns success or failure.
586  */

587 if (svc == NULL && Mysvc4 != NULL)
588     err = (*Mysvc4)(sock, &addrmask, retncnf,
589                    NFS4_SETPORT|NFS4_KRPC_START, retaddr);
590 else
591     err = (*svc)(sock, addrmask, retncnf);

592 if (err < 0) {
593     (void) syslog(LOG_ERR,
594         "Cannot establish %s service over <file desc."
595         " %d, protocol %s> : %m. Exiting",
596         protobp0->serv, sock, retncnf->nc_proto);
597     exit(1);
598 }

599 }

600 }

601 }

602 }

603 /*
604  * We successfully set up the server over this transport.
605  * Add this descriptor to the one being polled on.
606  */
607 add_to_poll_list(sock, retncnf);
608 }
609 }

```

```

611 /*
612 * Set up the NFS service over all the available transports and
613 * also set send and receive buffer size for transport connection.
614 * Set up the NFS service over all the available transports.
615 * Returns -1 for failure, 0 for success.
616 */
617 int
618 do_all_setbuf(struct protob *protobp,
619              int (*svc)(int, struct netbuf, struct netconfig *), int use_pmap,
620              int sndbufsz, int rcvbufsz)
621 do_all(struct protob *protobp,
622        int (*svc)(int, struct netbuf, struct netconfig *), int use_pmap)
623 {
624     struct netconfig *nconf;
625     NCONF_HANDLE *nc;
626     int l;
627
628     if ((nc = setnetconfig()) == (NCONF_HANDLE *)NULL) {
629         syslog(LOG_ERR, "setnetconfig failed: %m");
630         return (-1);
631     }
632     l = strlen(NC_UDP);
633     while (nconf = getnetconfig(nc)) {
634         if ((nconf->nc_flag & NC_VISIBLE) &&
635             strcmp(nconf->nc_protofmly, NC_LOOPBACK) != 0 &&
636             OK_TPI_TYPE(nconf) &&
637             (protobp->program != NFS4_CALLBACK ||
638             strncasecmp(nconf->nc_proto, NC_UDP, l) != 0))
639             do_one_setbuf(nconf->nc_device, nconf->nc_proto,
640                          protobp, svc, use_pmap, sndbufsz, rcvbufsz);
641             do_one(nconf->nc_device, nconf->nc_proto,
642                  protobp, svc, use_pmap);
643     }
644     (void) endnetconfig(nc);
645     return (0);
646 }
647
648 /*
649 * Called to set up service over a particular transport.
650 */
651 void
652 do_one(char *provider, NETSELDECL(proto), struct protob *protobp0,
653        int (*svc)(int, struct netbuf, struct netconfig *), int use_pmap)
654 {
655     do_one_setbuf(provider, proto, protobp0, svc, use_pmap, 0, 0);
656 }
657
658 /*
659 * Set up the NFS service over all the available transports.
660 * Returns -1 for failure, 0 for success.
661 */
662 int
663 do_all(struct protob *protobp,
664        int (*svc)(int, struct netbuf, struct netconfig *), int use_pmap)
665 {
666     return (do_all_setbuf(protobp, svc, use_pmap, 0, 0));
667 }
668
669 /*
670 * poll on the open transport descriptors for events and errors.
671 */
672 void
673 poll_for_action(void)
674 {
675     int nfd;
676     int i;

```

```

677     /*
678     * Keep polling until all transports have been closed. When this
679     * happens, we return.
680     */
681     while ((int)num_fds > 0) {
682         nfd = poll(poll_array, num_fds, INFTIM);
683         switch (nfd) {
684             case 0:
685                 continue;
686
687             case -1:
688                 /*
689                 * Some errors from poll could be
690                 * due to temporary conditions, and we try to
691                 * be robust in the face of them. Other
692                 * errors (should never happen in theory)
693                 * are fatal (eg. EINVAL, EFAULT).
694                 */
695                 switch (errno) {
696                     case EINTR:
697                         continue;
698
699                     case EAGAIN:
700                     case ENOMEM:
701                         (void) sleep(10);
702                         continue;
703
704                     default:
705                         (void) syslog(LOG_ERR,
706                                     "poll failed: %m. Exiting");
707                         exit(1);
708                 }
709                 break;
710             }
711
712     /*
713     * Go through the poll list looking for events.
714     */
715     for (i = 0; i < num_fds && nfd > 0; i++) {
716         if (poll_array[i].revents) {
717             nfd--;
718             /*
719             * We have a message, so try to read it.
720             * Record the error return in errno,
721             * so that syslog(LOG_ERR, "...%m")
722             * dumps the corresponding error string.
723             */
724             if (conn_polled[i].nc.nc_semantics ==
725                 NC_TPI_CLTS) {
726                 errno = do_poll_clts_action(
727                     poll_array[i].fd, i);
728             } else {
729                 errno = do_poll_cots_action(
730                     poll_array[i].fd, i);
731             }
732
733             if (errno == 0)
734                 continue;
735
736             /*
737             * Most returned error codes mean that there is
738             * a fatal condition which we can only deal with
739             * by closing the transport.
740             */
741             if (errno != EAGAIN && errno != ENOMEM) {

```

```

738         (void) syslog(LOG_ERR,
739             "Error (%m) reading descriptor %d/transport %s. Closing it.",
740             poll_array[i].fd,
741             conn_polled[i].nc.nc_proto);
742         (void) t_close(poll_array[i].fd);
743         remove_from_poll_list(poll_array[i].fd);
744     } else if (errno == ENOMEM)
745         (void) sleep(5);
746     }
747 }
748
751 (void) syslog(LOG_ERR,
752     "All transports have been closed with errors. Exiting.");
753 }

```

unchanged portion omitted

```

1627 static int
1628 bind_to_provider(char *provider, char *serv, struct netbuf **addr,
1629                 struct netconfig **retncnf, int sndbufsz, int rcvbufsz)
1630                 struct netconfig **retncnf)
1631 {
1632     struct netconfig *nconf;
1633     NCONF_HANDLE *nc;
1634     struct nd_hostserv hs;
1635
1636     hs.h_host = HOST_SELF;
1637     hs.h_serv = serv_name_to_port_name(serv);
1638
1639     if ((nc = setnetconfig()) == (NCONF_HANDLE *)NULL) {
1640         syslog(LOG_ERR, "setnetconfig failed: %m");
1641         return (-1);
1642     }
1643     while (nconf = getnetconfig(nc)) {
1644         if (OK_TPI_TYPE(nconf) &&
1645             strcmp(nconf->nc_device, provider) == 0) {
1646             *retncnf = nconf;
1647             return (nfslib_bindit(nconf, addr, &hs,
1648                 listen_backlog, sndbufsz, rcvbufsz));
1649             listen_backlog);
1650         }
1651     }
1652     (void) endnetconfig(nc);
1653
1654     syslog(LOG_ERR, "couldn't find netconfig entry for provider %s",
1655         provider);
1656     return (-1);
1657 }

```

```

1657 static int
1658 bind_to_proto(NETSELDECL(proto), char *serv, struct netbuf **addr,
1659              struct netconfig **retncnf, int sndbufsz, int rcvbufsz)
1660              struct netconfig **retncnf)
1661 {
1662     struct netconfig *nconf;
1663     NCONF_HANDLE *nc = NULL;
1664     struct nd_hostserv hs;
1665
1666     hs.h_host = HOST_SELF;
1667     hs.h_serv = serv_name_to_port_name(serv);
1668
1669     if ((nc = setnetconfig()) == (NCONF_HANDLE *)NULL) {
1670         syslog(LOG_ERR, "setnetconfig failed: %m");
1671         return (-1);
1672     }

```

```

1672     while (nconf = getnetconfig(nc)) {
1673         if (OK_TPI_TYPE(nconf) && NETSELEQ(nconf->nc_proto, proto)) {
1674             *retncnf = nconf;
1675             return (nfslib_bindit(nconf, addr, &hs,
1676                 listen_backlog, sndbufsz, rcvbufsz));
1677             listen_backlog);
1678         }
1679     }
1680     (void) endnetconfig(nc);
1681
1682     syslog(LOG_ERR, "couldn't find netconfig entry for protocol %s",
1683         proto);
1684     return (-1);
1685 }

```

unchanged portion omitted

```

*****
7961 Wed Jun 17 00:15:43 2009
new/usr/src/cmd/fs.d/nfs/nfs4cbd/nfs4cbd.c
4953763 Need way to configure NFS window sizes without changing system wide defa
6216670 NFS server needs a bigger transmit buffer
*****
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  * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25 */

26 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T      */
27 /*      All Rights Reserved      */

29 /*
30  * Portions of this source code were derived from Berkeley 4.3 BSD
31  * under license from the Regents of the University of California.
32  */

34 #pragma ident      "%Z%M% %I%      %E% SMI"

34 /*
35  * This module provides the user level support for the NFSv4
36  * callback program. It is modeled after nfsd. When a nfsv4
37  * mount occurs, the mount command forks and the child runs
38  * start_nfs4_callback. If this is the first mount, then the
39  * process will hang around listening for incoming connection
40  * requests from the nfsv4 server.
41  *
42  * For connection-less protocols, the krpc is started immediately.
43  * For connection oriented protocols, the kernel module is informed
44  * of netid and universal address that it can give this
45  * information to the server during setclientid.
46  */

48 #include <sys/param.h>
49 #include <sys/types.h>
50 #include <syslog.h>
51 #include <tiuser.h>
52 #include <rpc/rpc.h>
53 #include <errno.h>
54 #include <thread.h>
55 #include <sys/resource.h>
56 #include <sys/file.h>
57 #include <nfs/nfs.h>

```

```

58 #include <nfs/nfssys.h>
59 #include <stdio.h>
60 #include <stdlib.h>
61 #include <netconfig.h>
62 #include <netdir.h>
63 #include <string.h>
64 #include <unistd.h>
65 #include <stropts.h>
66 #include <sys/tihdr.h>
67 #include <netinet/tcp.h>
68 #include "nfs_tbind.h"
69 #include "thrpool.h"
70 #include <rpcsvc/nfs4_prot.h>
71 #include <netdb.h>
72 #include <signal.h>
73 #include <strings.h>
74 #include <priv_utils.h>
75 #include <rpcsvc/daemon_utils.h>

77 static int      nfs4svc(int, struct netbuf *, struct netconfig *, int,
78                      struct netbuf *);
79 extern int      _nfssys(int, void *);

81 static char      *MyName;

83 /*
84  * The following are all globals used by routines in nfs_tbind.c.
85  */
86 size_t end_listen_fds;      /* used by conn_close_oldest() */
87 size_t num_fds = 0;      /* used by multiple routines */
88 int listen_backlog = 32;      /* used by bind_to_{provider,proto}() */
89 int num_servers;      /* used by cots_listen_event() */
90 int (*Mysvc)(int, struct netbuf, struct netconfig *) = NULL;
91      /* used by cots_listen_event() */
92 int max_conns_allowed = -1; /* used by cots_listen_event() */

94 int
95 main(int argc, char *argv[])
96 {
97     int pid;
98     int i;
99     struct protob *protobp;
100    struct flock f;
101    pid_t pi;
102    struct svcpool_args cb_svcpool;

104    MyName = "nfs4cbd";
105    Mysvc4 = nfs4svc;

107 #ifndef DEBUG
108     /*
109      * Close existing file descriptors, open "/dev/null" as
110      * standard input, output, and error, and detach from
111      * controlling terminal.
112      */
113     closefrom(0);
114     (void) open("/dev/null", O_RDONLY);
115     (void) open("/dev/null", O_WRONLY);
116     (void) dup(1);
117     (void) setsid();
118 #endif

120     /*
121      * create a child to continue our work
122      * Parent's exit will tell mount command we're ready to go
123      */

```

```

124     if ((pi = fork()) > 0) {
125         exit(0);
126     }
127
128     if (pi == -1) {
129         (void) syslog(LOG_ERR,
130             "Could not start NFS4_CALLBACK service");
131         exit(1);
132     }
133
134     (void) _create_daemon_lock(NFS4CBD, DAEMON_UID, DAEMON_GID);
135
136     svcsetprio();
137
138     if (__init_daemon_priv(PU_RESETGROUPS|PU_CLEARLIMITSET,
139         DAEMON_UID, DAEMON_GID, PRIV_SYS_NFS, (char *)NULL) == -1) {
140         (void) fprintf(stderr, "%s must be run with sufficient"
141             " privileges\n", argv[0]);
142         exit(1);
143     }
144     /* Basic privileges we don't need, remove from E/P. */
145     __fini_daemon_priv(PRIV_PROC_EXEC, PRIV_PROC_FORK, PRIV_FILE_LINK_ANY,
146         PRIV_PROC_SESSION, PRIV_PROC_INFO, (char *)NULL);
147
148     /*
149     * establish our lock on the lock file and write our pid to it.
150     * exit if some other process holds the lock, or if there's any
151     * error in writing/locking the file.
152     */
153     pid = _enter_daemon_lock(NFS4CBD);
154     switch (pid) {
155     case 0:
156         break;
157     case -1:
158         syslog(LOG_ERR, "error locking for %s: %s", NFS4CBD,
159             strerror(errno));
160         exit(2);
161     default:
162         /* daemon was already running */
163         exit(0);
164     }
165
166     openlog(MyName, LOG_PID | LOG_NDELAY, LOG_DAEMON);
167
168     cb_svcpool.id = NFS_CB_SVCPPOOL_ID;
169     cb_svcpool.maxthreads = 0;
170     cb_svcpool.redline = 0;
171     cb_svcpool.qsize = 0;
172     cb_svcpool.timeout = 0;
173     cb_svcpool.stksize = 0;
174     cb_svcpool.max_same_xprt = 0;
175
176     /* create a SVC_POOL for the nfsv4 callback daemon */
177     if (__nfssys(SVCPPOOL_CREATE, &cb_svcpool)) {
178         (void) syslog(LOG_ERR, "can't setup NFS_CB SVCPPOOL: Exiting");
179         exit(1);
180     }
181
182     /*
183     * Set up blocked thread to do LWP creation on behalf of the kernel.
184     */
185     if (svcwait(NFS_CB_SVCPPOOL_ID)) {
186         (void) syslog(LOG_ERR,
187             "Can't set up NFS_CB LWP creator: Exiting");
188         exit(1);
189     }

```

```

192     /*
193     * Build a protocol block list for registration.
194     */
195     protobp = (struct protob *)malloc(sizeof (struct protob));
196     protobp->serv = "NFS4_CALLBACK";
197     protobp->versmin = NFS_CB;
198     protobp->versmax = NFS_CB;
199     protobp->program = NFS4_CALLBACK;
200     protobp->next = NULL;
201
202     if (do_all(protobp, NULL, 0) == -1) {
203         exit(1);
204     }
205
206     free(protobp);
207
208     if (num_fds == 0) {
209         (void) syslog(LOG_ERR,
210             "Could not start NFS4_CALLBACK service for any protocol");
211         exit(1);
212     }
213
214     end_listen_fds = num_fds;
215     /*
216     * Poll for non-data control events on the transport descriptors.
217     */
218     poll_for_action();
219
220     /*
221     * If we get here, something failed in poll_for_action().
222     */
223     return (1);
224 }

```

unchanged portion omitted

```

*****
25416 Wed Jun 17 00:15:43 2009
new/usr/src/cmd/fs.d/nfs/nfsd/nfsd.c
4953763 Need way to configure NFS window sizes without changing system wide defa
6216670 NFS server needs a bigger transmit buffer
*****
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  */

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 /* LINTLIBRARY */
40 /* PROTOLIB1 */

42 /* NFS server */

44 #include <sys/param.h>
45 #include <sys/types.h>
46 #include <sys/stat.h>
47 #include <syslog.h>
48 #include <tiuser.h>
49 #include <rpc/rpc.h>
50 #include <errno.h>
51 #include <thread.h>
52 #include <sys/resource.h>
53 #include <sys/time.h>
54 #include <sys/file.h>
55 #include <nfs/nfs.h>
56 #include <nfs/nfs_acl.h>
57 #include <nfs/nfssys.h>
58 #include <stdio.h>
59 #include <stdio_ext.h>
60 #include <stdlib.h>

```

```

61 #include <signal.h>
62 #include <netconfig.h>
63 #include <netdir.h>
64 #include <string.h>
65 #include <unistd.h>
66 #include <stropts.h>
67 #include <sys/tihdr.h>
68 #include <sys/wait.h>
69 #include <poll.h>
70 #include <priv_utils.h>
71 #include <sys/tiuser.h>
72 #include <netinet/tcp.h>
73 #include <deflt.h>
74 #include <rpcsvc/daemon_utils.h>
75 #include <rpcsvc/nfs4_prot.h>
76 #include <libnvpair.h>
77 #include "nfs_tbind.h"
78 #include "thrpool.h"

80 /* quiesce requests will be ignored if nfs_server_vers_max < QUIESCE_VERSMIN */
81 #define QUIESCE_VERSMIN 4
82 /* DSS: distributed stable storage */
83 #define DSS_VERSMIN 4

85 static int      nfssvc(int, struct netbuf, struct netconfig *);
86 static int      nfssvcpool(int maxservers);
87 static int      dss_init(uint_t npaths, char **pathnames);
88 static void     dss_mkleafdirs(uint_t npaths, char **pathnames);
89 static void     dss_mkleafdir(char *dir, char *leaf, char *path);
90 static void     usage(void);
91 int             qstrcmp(const void *s1, const void *s2);

93 extern int      _nfssys(int, void *);

95 extern int      daemonize_init(void);
96 extern void     daemonize_fini(int fd);

98 /* signal handlers */
99 static void     sigflush(int);
100 static void     quiesce(int);

102 static char     *MyName;
103 static NETSELDECL(defaultproviders)[] = { "/dev/tcp6", "/dev/tcp", "/dev/udp",
104                                           "/dev/udp6", NULL };
105 /* static      NETSELDECL(defaultprotos)[] = { NC_UDP, NC_TCP, NULL }; */
106 /*
107  * The following are all globals used by routines in nfs_tbind.c.
108  */
109 size_t         end_listen_fds;          /* used by conn_close_oldest() */
110 size_t         num_fds = 0;             /* used by multiple routines */
111 int            listen_backlog = 32;     /* used by bind_to_{provider,proto}() */
112 int            num_servers;            /* used by cots_listen_event() */
113 int            (*Mysvc)(int, struct netbuf, struct netconfig *) = nfssvc;
114                                                       /* used by cots_listen_event() */
115 int            max_conns_allowed = -1; /* used by cots_listen_event() */

117 /*
118  * Keep track of min/max versions of NFS protocol to be started.
119  * Start with the defaults (min == 2, max == 3). We have the
120  * capability of starting vers=4 but only if the user requests it.
121  */
122 int            nfs_server_vers_min = NFS_VERSMIN_DEFAULT;
123 int            nfs_server_vers_max = NFS_VERSMAX_DEFAULT;

125 /*
126  * Set the default for server delegation enablement and set per

```

```

127 * /etc/default/nfs configuration (if present).
128 */
129 int     nfs_server_delegation = NFS_SERVER_DELEGATION_DEFAULT;

131 /*
132 * Default values for TCP send and receive buffer size of NFS server
133 * connections.
134 *
135 * These values can be tuned by user via /etc/default/nfs configuration
136 * file by setting NFS_SERVER_SNDBUFSZ and NFS_SERVER_RCVBUFSZ.
137 *
138 * To force NFS connections to use system-wide default for TCP send and
139 * receive buffer, set NFS_SERVER_SNDBUFSZ and NFS_SERVER_RCVBUFSZ to 0.
140 */
141 int     nfs_server_sndbufsz = 1048576;
142 int     nfs_server_rcvbufsz = 1048576;

144 int
145 main(int ac, char *av[])
146 {
147     char *dir = "/";
148     int allflag = 0;
149     int df_allflag = 0;
150     int opt_cnt = 0;
151     int maxservers = 1; /* zero allows infinite number of threads */
152     int maxservers_set = 0;
153     int logmaxservers = 0;
154     int pid;
155     int i, bufksz;
156     char *provider = (char *)NULL;
157     char *df_provider = (char *)NULL;
158     struct protob *protobp0, *protobp;
159     NETSELDECL(proto) = NULL;
160     NETSELDECL(df_proto) = NULL;
161     NETSELPDECL(providerp);
162     char *defval;
163     boolean_t can_do_mlp;
164     uint_t dss_npaths = 0;
165     char **dss_pathnames = NULL;
166     sigset_t sgset;

168     int pipe_fd = -1;

170     MyName = *av;

172     /*
173     * Initializations that require more privileges than we need to run.
174     */
175     (void) _create_daemon_lock(NFSD, DAEMON_UID, DAEMON_GID);
176     svcsetprio();

178     can_do_mlp = priv_ineffect(PRIV_NET_BINDMLP);
179     if ((__init_daemon_priv(PU_RESETPROGS|PU_CLEARLIMITSET,
180     DAEMON_UID, DAEMON_GID, PRIV_SYS_NFS,
181     can_do_mlp ? PRIV_NET_BINDMLP : NULL, NULL) == -1) {
182         (void) fprintf(stderr, "%s should be run with"
183         " sufficient privileges\n", av[0]);
184         exit(1);
185     }

187     (void) enable_extended_FILE_stdio(-1, -1);

189     /*
190     * Read in the values from config file first before we check
191     * command line options so the options override the file.

```

```

192     */
193     if ((defopen(NFSADMIN)) == 0) {
194         if ((defval = defread("NFSD_MAX_CONNECTIONS=")) != NULL) {
195             errno = 0;
196             max_conns_allowed = strtol(defval, (char **)NULL, 10);
197             if (errno != 0) {
198                 max_conns_allowed = -1;
199             }
200         }
201         if ((defval = defread("NFSD_LISTEN_BACKLOG=")) != NULL) {
202             errno = 0;
203             listen_backlog = strtol(defval, (char **)NULL, 10);
204             if (errno != 0) {
205                 listen_backlog = 32;
206             }
207         }
208         if ((defval = defread("NFSD_PROTOCOL=")) != NULL) {
209             df_proto = strdup(defval);
210             opt_cnt++;
211             if (strncasecmp("ALL", defval, 3) == 0) {
212                 free(df_proto);
213                 df_proto = NULL;
214                 df_allflag = 1;
215             }
216         }
217         if ((defval = defread("NFSD_DEVICE=")) != NULL) {
218             df_provider = strdup(defval);
219             opt_cnt++;
220         }
221         if ((defval = defread("NFSD_SERVERS=")) != NULL) {
222             errno = 0;
223             maxservers = strtol(defval, (char **)NULL, 10);
224             if (errno != 0) {
225                 maxservers = 1;
226             } else {
227                 maxservers_set = 1;
228             }
229         }
230         if ((defval = defread("NFS_SERVER_VERSMIN=")) != NULL) {
231             errno = 0;
232             nfs_server_vers_min =
233                 strtol(defval, (char **)NULL, 10);
234             if (errno != 0) {
235                 nfs_server_vers_min = NFS_VERSMIN_DEFAULT;
236             }
237         }
238         if ((defval = defread("NFS_SERVER_VERSMAX=")) != NULL) {
239             errno = 0;
240             nfs_server_vers_max =
241                 strtol(defval, (char **)NULL, 10);
242             if (errno != 0) {
243                 nfs_server_vers_max = NFS_VERSMAX_DEFAULT;
244             }
245         }
246         if ((defval = defread("NFS_SERVER_DELEGATION=")) != NULL) {
247             if (strcmp(defval, "off") == 0) {
248                 nfs_server_delegation = FALSE;
249             }
250         }
251         if ((defval = defread("NFS_SERVER_SNDBUFSZ=")) != NULL) {
252             errno = 0;
253             bufksz = strtol(defval, (char **)NULL, 10);
254             if (errno == 0)
255                 nfs_server_sndbufsz = bufksz;
256         }
257         if ((defval = defread("NFS_SERVER_RCVBUFSZ=")) != NULL) {

```

```

258         errno = 0;
259         bufksz = strtol(defval, (char **)NULL, 10);
260         if (errno == 0)
261             nfs_server_rcvbufsz = bufksz;
262     }
263
264     /* close defaults file */
265     defopen(NULL);
266 }
267
268 /*
269 * Conflict options error messages.
270 */
271 if (opt_cnt > 1) {
272     (void) fprintf(stderr, "\nConflicting options, only one of "
273         "the following options can be specified\n"
274         "in " NFSADMIN "::\n"
275         "\tNFSD_PROTOCOL=ALL\n"
276         "\tNFSD_PROTOCOL=protocol\n"
277         "\tNFSD_DEVICE=device\n\n");
278     usage();
279 }
280 opt_cnt = 0;
281
282 while ((i = getopt(ac, av, "ac:p:s:t:l:")) != EOF) {
283     switch (i) {
284     case 'a':
285         free(df_proto);
286         df_proto = NULL;
287         free(df_provider);
288         df_provider = NULL;
289
290         allflag = 1;
291         opt_cnt++;
292         break;
293
294     case 'c':
295         max_conns_allowed = atoi(optarg);
296         break;
297
298     case 'p':
299         proto = optarg;
300         df_allflag = 0;
301         opt_cnt++;
302         break;
303
304     /*
305     * DSS: NFSv4 distributed stable storage.
306     *
307     * This is a Contracted Project Private interface, for
308     * the sole use of Sun Cluster HA-NFS. See PSARC/2006/313.
309     */
310     case 's':
311         if (strlen(optarg) < MAXPATHLEN) {
312             /* first "-s" option encountered? */
313             if (dss_pathnames == NULL) {
314                 /*
315                  * Allocate maximum possible space
316                  * required given cmdline arg count;
317                  * "-s <path>" consumes two args.
318                  */
319                 size_t sz = (ac / 2) * sizeof(char *);
320                 dss_pathnames = (char **)malloc(sz);
321                 if (dss_pathnames == NULL) {
322                     (void) fprintf(stderr, "%s: "
323                         "dss paths malloc failed\n",

```

```

324             av[0]);
325             exit(1);
326         }
327         (void) memset(dss_pathnames, 0, sz);
328     }
329     dss_pathnames[dss_npaths] = optarg;
330     dss_npaths++;
331     } else {
332         (void) fprintf(stderr,
333             "%s: -s pathname too long.\n", av[0]);
334     }
335     break;
336
337     case 't':
338         provider = optarg;
339         df_allflag = 0;
340         opt_cnt++;
341         break;
342
343     case 'l':
344         listen_backlog = atoi(optarg);
345         break;
346
347     case '?':
348         usage();
349         /* NOTREACHED */
350     }
351 }
352
353 allflag = df_allflag;
354 if (proto == NULL)
355     proto = df_proto;
356 if (provider == NULL)
357     provider = df_provider;
358
359 /*
360 * Conflict options error messages.
361 */
362 if (opt_cnt > 1) {
363     (void) fprintf(stderr, "\nConflicting options, only one of "
364         "the following options can be specified\n"
365         "on the command line:\n"
366         "\t-a\n"
367         "\t-p protocol\n"
368         "\t-t transport\n\n");
369     usage();
370 }
371
372 if (proto != NULL &&
373     strncasecmp(proto, NC_UDP, strlen(NC_UDP)) == 0) {
374     if (nfs_server_vers_max == NFS_V4) {
375         if (nfs_server_vers_min == NFS_V4) {
376             fprintf(stderr,
377                 "NFS version 4 is not supported "
378                 "with the UDP protocol. Exiting\n");
379             exit(3);
380         } else {
381             fprintf(stderr,
382                 "NFS version 4 is not supported "
383                 "with the UDP protocol.\n");
384         }
385     }
386 }
387
388 /*
389 * If there is exactly one more argument, it is the number of

```

```

390     * servers.
391     */
392     if (optind == ac - 1) {
393         maxservers = atoi(av[optind]);
394         maxservers_set = 1;
395     }
396     /*
397     * If there are two or more arguments, then this is a usage error.
398     */
399     else if (optind < ac - 1)
400         usage();
401     /*
402     * Check the ranges for min/max version specified
403     */
404     else if ((nfs_server_vers_min > nfs_server_vers_max) ||
405             (nfs_server_vers_min < NFS_VERSMIN) ||
406             (nfs_server_vers_max > NFS_VERSMAX))
407         usage();
408     /*
409     * There are no additional arguments, and we haven't set maxservers
410     * explicitly via the config file, we use a default number of
411     * servers. We will log this.
412     */
413     else if (maxservers_set == 0)
414         logmaxservers = 1;
415
416     /*
417     * Basic Sanity checks on options
418     *
419     * max_conns_allowed must be positive, except for the special
420     * value of -1 which is used internally to mean unlimited, -1 isn't
421     * documented but we allow it anyway.
422     *
423     * maxservers must be positive
424     * listen_backlog must be positive or zero
425     */
426     if (((max_conns_allowed != -1) && (max_conns_allowed <= 0)) ||
427         (listen_backlog < 0) || (maxservers <= 0)) {
428         usage();
429     }
430
431     /*
432     * Set current dir to server root
433     */
434     if (chdir(dir) < 0) {
435         (void) fprintf(stderr, "%s: ", MyName);
436         perror(dir);
437         exit(1);
438     }
439
440 #ifndef DEBUG
441     pipe_fd = daemonize_init();
442 #endif
443
444     openlog(MyName, LOG_PID | LOG_NDELAY, LOG_DAEMON);
445
446     /*
447     * establish our lock on the lock file and write our pid to it.
448     * exit if some other process holds the lock, or if there's any
449     * error in writing/locking the file.
450     */
451     pid = _enter_daemon_lock(NFSD);
452     switch (pid) {
453     case 0:
454         break;
455     case -1:

```

```

456         fprintf(stderr, "error locking for %s: %s", NFSD,
457                 strerror(errno));
458         exit(2);
459     default:
460         /* daemon was already running */
461         exit(0);
462     }
463
464     /*
465     * If we've been given a list of paths to be used for distributed
466     * stable storage, and provided we're going to run a version
467     * that supports it, setup the DSS paths.
468     */
469     if (dss_pathnames != NULL && nfs_server_vers_max >= DSS_VERSMIN) {
470         if (dss_init(dss_npaths, dss_pathnames) != 0) {
471             fprintf(stderr, "%s", "dss_init failed. Exiting.");
472             exit(1);
473         }
474     }
475
476     /*
477     * Block all signals till we spawn other
478     * threads.
479     */
480     (void) sigfillset(&sgset);
481     (void) thr_sigsetmask(SIG_BLOCK, &sgset, NULL);
482
483     if (logmaxservers) {
484         fprintf(stderr,
485                 "Number of servers not specified. Using default of %d.",
486                 maxservers);
487     }
488
489     /*
490     * Make sure to unregister any previous versions in case the
491     * user is reconfiguring the server in interesting ways.
492     */
493     svc_unreg(NFS_PROGRAM, NFS_VERSION);
494     svc_unreg(NFS_PROGRAM, NFS_V3);
495     svc_unreg(NFS_PROGRAM, NFS_V4);
496     svc_unreg(NFS_ACL_PROGRAM, NFS_ACL_V2);
497     svc_unreg(NFS_ACL_PROGRAM, NFS_ACL_V3);
498
499     /*
500     * Set up kernel RPC thread pool for the NFS server.
501     */
502     if (nfssvcpool(maxservers)) {
503         fprintf(stderr, "Can't set up kernel NFS service: %s. Exiting",
504                 strerror(errno));
505         exit(1);
506     }
507
508     /*
509     * Set up blocked thread to do LWP creation on behalf of the kernel.
510     */
511     if (svcwait(NFS_SVCPOOL_ID)) {
512         fprintf(stderr, "Can't set up NFS pool creator: %s. Exiting",
513                 strerror(errno));
514         exit(1);
515     }
516
517     /*
518     * RDMA start and stop thread.
519     * Per pool RDMA listener creation and
520     * destructor thread.
521     */

```

```

522  * start rdma services and block in the kernel.
523  * (only if proto or provider is not set to TCP or UDP)
524  */
525  if ((proto == NULL) && (provider == NULL)) {
526      if (svcrdma(NFS_SVCPOOL_ID, nfs_server_vers_min,
527                nfs_server_vers_max, nfs_server_delegation)) {
528          fprintf(stderr,
529                 "Can't set up RDMA creator thread : %s",
530                 strerror(errno));
531      }
532  }

534  /*
535  * Now open up for signal delivery
536  */

538  (void) thr_sigsetmask(SIG_UNBLOCK, &sgset, NULL);
539  sigset(SIGTERM, sigflush);
540  sigset(SIGUSR1, quiesce);

542  /*
543  * Build a protocol block list for registration.
544  */
545  protobp0 = protobp = (struct protob *)malloc(sizeof (struct protob));
546  protobp->serv = "NFS";
547  protobp->versmin = nfs_server_vers_min;
548  protobp->versmax = nfs_server_vers_max;
549  protobp->program = NFS_PROGRAM;

551  protobp->next = (struct protob *)malloc(sizeof (struct protob));
552  protobp = protobp->next;
553  protobp->serv = "NFS_ACL"; /* not used */
554  protobp->versmin = nfs_server_vers_min;
555  /* XXX - this needs work to get the version just right */
556  protobp->versmax = (nfs_server_vers_max > NFS_ACL_V3) ?
557                    NFS_ACL_V3 : nfs_server_vers_max;
558  protobp->program = NFS_ACL_PROGRAM;
559  protobp->next = (struct protob *)NULL;

561  if (allflag) {
562      if (do_all_setbuf(protobp0, nfssvc, 0, nfs_server_sndbufsz,
563                      nfs_server_rcvbufsz) == -1) {
564          if (do_all(protobp0, nfssvc, 0) == -1) {
565              fprintf(stderr, "setnetconfig failed : %s",
566                         strerror(errno));
567              exit(1);
568          }
569      } else if (proto) {
570          /* there's more than one match for the same protocol */
571          struct netconfig *nconf;
572          NCONF_HANDLE *nc;
573          bool_t protoFound = FALSE;
574          if ((nc = setnetconfig()) == (NCONF_HANDLE *) NULL) {
575              fprintf(stderr, "setnetconfig failed : %s",
576                         strerror(errno));
577              goto done;
578          }
579          while (nconf = getnetconfig(nc)) {
580              if (strcmp(nconf->nc_proto, proto) == 0) {
581                  protoFound = TRUE;
582                  do_one_setbuf(nconf->nc_device, NULL,
583                               protobp0, nfssvc, 0,
584                               nfs_server_sndbufsz, nfs_server_rcvbufsz);
585                  do_one(nconf->nc_device, NULL,
586                        protobp0, nfssvc, 0);
587              }
588          }
589      }
590  }

```

```

585  }
586  (void) endnetconfig(nc);
587  if (protoFound == FALSE) {
588      fprintf(stderr,
589             "couldn't find netconfig entry for protocol %s",
590             proto);
591  }
592  } else if (provider)
593      do_one_setbuf(provider, proto, protobp0, nfssvc, 0,
594                  nfs_server_sndbufsz, nfs_server_rcvbufsz);
595  do_one(provider, proto, protobp0, nfssvc, 0);
596  else {
597      for (providerp = defaultproviders;
598           *providerp != NULL; providerp++) {
599          provider = *providerp;
600          do_one_setbuf(provider, NULL, protobp0, nfssvc, 0,
601                      nfs_server_sndbufsz, nfs_server_rcvbufsz);
602          do_one(provider, NULL, protobp0, nfssvc, 0);
603      }
604  }
605  done:
606
607  free(protobp);
608  free(protobp0);
609
610  if (num_fds == 0) {
611      fprintf(stderr, "Could not start NFS service for any protocol."
612              " Exiting");
613      exit(1);
614  }
615
616  end_listen_fds = num_fds;
617
618  /*
619  * nfsd is up and running as far as we are concerned.
620  */
621  daemonize_fini(pipe_fd);
622
623  /*
624  * Get rid of unneeded privileges.
625  */
626  __fini_daemon_priv(PRIV_PROC_FORK, PRIV_PROC_EXEC, PRIV_PROC_SESSION,
627                   PRIV_FILE_LINK_ANY, PRIV_PROC_INFO, (char *)NULL);
628
629  /*
630  * Poll for non-data control events on the transport descriptors.
631  */
632  poll_for_action();
633
634  /*
635  * If we get here, something failed in poll_for_action().
636  */
637  return (1);
638 }

```

unchanged_portion_omitted

```

*****
105719 Wed Jun 17 00:15:44 2009
new/usr/src/uts/common/rpc/clnt_cots.c
4953763 Need way to configure NFS window sizes without changing system wide defa
6216670 NFS server needs a bigger transmit buffer
*****
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  */

26 /*
27  * Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T
28  * All Rights Reserved
29  */

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 /*
38  * Implements a kernel based, client side RPC over Connection Oriented
39  * Transports (COTS).
40  */

42 /*
43  * Much of this file has been re-written to let NFS work better over slow
44  * transports. A description follows.
45  *
46  * One of the annoying things about kRPC/COTS is that it will temporarily
47  * create more than one connection between a client and server. This
48  * happens because when a connection is made, the end-points entry in the
49  * linked list of connections (headed by cm_hd), is removed so that other
50  * threads don't mess with it. Went ahead and bit the bullet by keeping
51  * the endpoint on the connection list and introducing state bits,
52  * condition variables etc. to the connection entry data structure (struct
53  * cm_xprt).
54  *
55  * Here is a summary of the changes to cm-xprt:
56  *
57  * x_ctime is the timestamp of when the endpoint was last
58  * connected or disconnected. If an end-point is ever disconnected
59  * or re-connected, then any outstanding RPC request is presumed
60  * lost, telling clnt_cots_kcallit that it needs to re-send the

```

```

61  * request, not just wait for the original request's reply to
62  * arrive.
63  *
64  * x_thread flag which tells us if a thread is doing a connection attempt.
65  *
66  * x_waitdis flag which tells us we are waiting a disconnect ACK.
67  *
68  * x_needdis flag which tells us we need to send a T_DISCONN_REQ
69  * to kill the connection.
70  *
71  * x_needrel flag which tells us we need to send a T_ORDREL_REQ to
72  * gracefully close the connection.
73  *
74  * #defined bitmasks for the all the b_* bits so that more
75  * efficient (and at times less clumsy) masks can be used to
76  * manipulated state in cases where multiple bits have to
77  * set/cleared/checked in the same critical section.
78  *
79  * x_conn_cv and x_dis_cv are new condition variables to let
80  * threads know when the connection attempt is done, and to let
81  * the connecting thread know when the disconnect handshake is
82  * done.
83  *
84  * Added the CONN_HOLD() macro so that all reference holds have the same
85  * look and feel.
86  *
87  * In the private (cku_private) portion of the client handle,
88  *
89  * cku_flags replaces the cku_sent a boolean. cku_flags keeps
90  * track of whether a request as been sent, and whether the
91  * client's handles call record is on the dispatch list (so that
92  * the reply can be matched by XID to the right client handle).
93  * The idea of CKU_ONQUEUE is that we can exit clnt_cots_kcallit()
94  * and still have the response find the right client handle so
95  * that the retry of CLNT_CALL() gets the result. Testing found
96  * situations where if the timeout was increased, performance
97  * degraded. This was due to us hitting a window where the thread
98  * was back in rfscall() (probably printing server not responding)
99  * while the response came back but no place to put it.
100 *
101 * cku_ctime is just a cache of x_ctime. If they match,
102 * clnt_cots_kcallit() won't to send a retry (unless the maximum
103 * receive count limit as been reached). If the don't match, then
104 * we assume the request has been lost, and a retry of the request
105 * is needed.
106 *
107 * cku_recv_attempts counts the number of receive count attempts
108 * after one try is sent on the wire.
109 *
110 * Added the clnt_delay() routine so that interruptible and
111 * noninterruptible delays are possible.
112 *
113 * CLNT_MIN_TIMEOUT has been bumped to 10 seconds from 3. This is used to
114 * control how long the client delays before returned after getting
115 * ECONNREFUSED. At 3 seconds, 8 client threads per mount really does bash
116 * a server that may be booting and not yet started nfsd.
117 *
118 * CLNT_MAXRECV_WITHOUT_RETRY is a new macro (value of 3) (with a tunable)
119 * Why don't we just wait forever (receive an infinite # of times)?
120 * Because the server may have rebooted. More insidious is that some
121 * servers (ours) will drop NFS/TCP requests in some cases. This is bad,
122 * but it is a reality.
123 *
124 * The case of a server doing orderly release really messes up the
125 * client's recovery, especially if the server's TCP implementation is
126 * buggy. It was found was that the kRPC/COTS client was breaking some

```

```

127 * TPI rules, such as not waiting for the acknowledgement of a
128 * T_DISCON_REQ (hence the added case statements T_ERROR_ACK, T_OK_ACK and
129 * T_DISCON_REQ in clnt_dispatch_notifyall()).
130 *
131 * One of things that we've seen is that a kRPC TCP endpoint goes into
132 * TIMEWAIT and a thus a reconnect takes a long time to satisfy because
133 * that the TIMEWAIT state takes a while to finish. If a server sends a
134 * T_ORDREL_IND, there is little point in an RPC client doing a
135 * T_ORDREL_REQ, because the RPC request isn't going to make it (the
136 * server is saying that it won't accept any more data). So kRPC was
137 * changed to send a T_DISCON_REQ when we get a T_ORDREL_IND. So now the
138 * connection skips the TIMEWAIT state and goes straight to a bound state
139 * that kRPC can quickly switch to connected.
140 *
141 * Code that issues TPI request must use waitforack() to wait for the
142 * corresponding ack (assuming there is one) in any future modifications.
143 * This works around problems that may be introduced by breaking TPI rules
144 * (by submitting new calls before earlier requests have been acked) in the
145 * case of a signal or other early return. waitforack() depends on
146 * clnt_dispatch_notifyconn() to issue the wakeup when the ack
147 * arrives, so adding new TPI calls may require corresponding changes
148 * to clnt_dispatch_notifyconn(). Presently, the timeout period is based on
149 * CLNT_MIN_TIMEOUT which is 10 seconds. If you modify this value, be sure
150 * not to set it too low or TPI ACKS will be lost.
151 */

153 #include <sys/param.h>
154 #include <sys/types.h>
155 #include <sys/user.h>
156 #include <sys/system.h>
157 #include <sys/sysmacros.h>
158 #include <sys/proc.h>
159 #include <sys/socket.h>
160 #include <sys/file.h>
161 #include <sys/stream.h>
162 #include <sys/strsubr.h>
163 #include <sys/stropts.h>
164 #include <sys/strsun.h>
165 #include <sys/timod.h>
166 #include <sys/tiuser.h>
167 #include <sys/tihdr.h>
168 #include <sys/t_kuser.h>
169 #include <sys/fcntl.h>
170 #include <sys/errno.h>
171 #include <sys/kmem.h>
172 #include <sys/debug.h>
173 #include <sys/system.h>
174 #include <sys/kstat.h>
175 #include <sys/t_lock.h>
176 #include <sys/ddi.h>
177 #include <sys/cmn_err.h>
178 #include <sys/time.h>
179 #include <sys/isa_defs.h>
180 #include <sys/callb.h>
181 #include <sys/sunddi.h>
182 #include <sys/atomic.h>
183 #include <sys/sdt.h>

185 #include <netinet/in.h>
186 #include <netinet/tcp.h>

188 #include <rpc/types.h>
189 #include <rpc/xdr.h>
190 #include <rpc/auth.h>
191 #include <rpc/clnt.h>
192 #include <rpc/rpc_msg.h>

```

```

193 #include <nfs/nfs.h>

195 #define COTS_DEFAULT_ALLOCSIZE 2048

197 #define WIRE_HDR_SIZE 20 /* serialized call header, sans proc number */
198 #define MSG_OFFSET 128 /* offset of call into the mblk */

200 const char *kinet_ntop6(uchar_t *, char *, size_t);

202 static int clnt_cots_ksettimers(CLIENT *, struct rpc_timers *,
203 struct rpc_timers *, int, void*)(int, int, caddr_t), caddr_t, uint32_t);
204 static enum clnt_stat clnt_cots_kcallit(CLIENT *, rpcproc_t, xdrproc_t,
205 caddr_t, xdrproc_t, caddr_t, struct timeval);
206 static void clnt_cots_kabort(CLIENT *);
207 static void clnt_cots_kerror(CLIENT *, struct rpc_err *);
208 static bool_t clnt_cots_kfreeres(CLIENT *, xdrproc_t, caddr_t);
209 static void clnt_cots_kdestroy(CLIENT *);
210 static bool_t clnt_cots_kcontrol(CLIENT *, int, char *);

213 /* List of transports managed by the connection manager. */
214 struct cm_xprt {
215 TIUSER *x_tiptr; /* transport handle */
216 queue_t *x_wq; /* send queue */
217 clock_t x_time; /* last time we handed this xprt out */
218 clock_t x_ctime; /* time we went to CONNECTED */
219 int x_tidu_size; /* TIDU size of this transport */
220 union {
221 struct {
222 unsigned int
223 #ifdef _BIT_FIELDS_HTOH
224 b_closing: 1, /* we've sent a ord rel on this conn */
225 b_dead: 1, /* transport is closed or disconn */
226 b_doomed: 1, /* too many conns, let this go idle */
227 b_connected: 1, /* this connection is connected */
228
229 b_ordrel: 1, /* do an orderly release? */
230 b_thread: 1, /* thread doing connect */
231 b_waitdis: 1, /* waiting for disconnect ACK */
232 b_needdis: 1, /* need T_DISCON_REQ */
233
234 b_needrel: 1, /* need T_ORDREL_REQ */
235 b_early_disc: 1, /* got a T_ORDREL_IND or T_DISCON_IND */
236
237
238 b_pad: 22;
239
240 #endif
241 #ifdef _BIT_FIELDS_LTOH
242 b_pad: 22,
243
244 b_early_disc: 1, /* got a T_ORDREL_IND or T_DISCON_IND */
245
246 b_needrel: 1, /* disconnect during connect */
247
248
249 b_needdis: 1, /* need T_DISCON_REQ */
250 b_waitdis: 1, /* waiting for disconnect ACK */
251 b_thread: 1, /* thread doing connect */
252 b_ordrel: 1, /* do an orderly release? */
253
254 b_connected: 1, /* this connection is connected */
255 b_doomed: 1, /* too many conns, let this go idle */
256 b_dead: 1, /* transport is closed or disconn */
257 b_closing: 1; /* we've sent a ord rel on this conn */
258 #endif

```

```

259         } bit;           unsigned int word;

261 #define x_closing      x_state.bit.b_closing
262 #define x_dead         x_state.bit.b_dead
263 #define x_doomed      x_state.bit.b_doomed
264 #define x_connected   x_state.bit.b_connected

266 #define x_ordrel       x_state.bit.b_ordrel
267 #define x_thread       x_state.bit.b_thread
268 #define x_waitdis     x_state.bit.b_waitdis
269 #define x_needdis     x_state.bit.b_needdis

271 #define x_needrel     x_state.bit.b_needrel
272 #define x_early_disc  x_state.bit.b_early_disc

274 #define x_state_flags x_state.word

276 #define X_CLOSING      0x80000000
277 #define X_DEAD         0x40000000
278 #define X_DOOMED      0x20000000
279 #define X_CONNECTED   0x10000000

281 #define X_ORDREL       0x08000000
282 #define X_THREAD       0x04000000
283 #define X_WAITDIS     0x02000000
284 #define X_NEEDDIS     0x01000000

286 #define X_NEEDREL     0x00800000
287 #define X_EARLYDISC  0x00400000

289 #define X_BADSTATES   (X_CLOSING | X_DEAD | X_DOOMED)

291     }           x_state;
292     int         x_ref;           /* number of users of this xpirt */
293     int         x_family;       /* address family of transport */
294     dev_t       x_rdev;        /* device number of transport */
295     struct cm_xprt *x_next;

297     struct netbuf x_server;     /* destination address */
298     struct netbuf x_src;        /* src address (for retries) */
299     kmutex_t     x_lock;       /* lock on this entry */
300     kcondvar_t   x_cv;         /* to signal when can be closed */
301     kcondvar_t   x_conn_cv;    /* to signal when connection attempt */
302                                     /* is complete */
303     kstat_t      *x_ksp;

305     kcondvar_t   x_dis_cv;     /* to signal when disconnect attempt */
306                                     /* is complete */
307     zoneid_t     x_zoneid;     /* zone this xpirt belongs to */
308 };
    unchanged portion omitted

377 static struct cm_xprt *connmgr_wrapconnect(struct cm_xprt *,
378     const struct timeval *, struct netbuf *, int, struct netbuf *,
379     struct rpc_err *, bool_t, bool_t, cred_t *);

381 static bool_t   connmgr_connect(struct cm_xprt *, queue_t *, struct netbuf *,
382     int, calllist_t *, int *, bool_t reconnect,
383     const struct timeval *, bool_t, cred_t *);

385 static bool_t   connmgr_getopt_int(queue_t *wq, int level, int name, int *val,
386     calllist_t *e, cred_t *cr);
387 static bool_t   connmgr_setopt_int(queue_t *, int, int, int,
388     calllist_t *, cred_t *cr);
389 static bool_t   connmgr_setopt(queue_t *, int, int, calllist_t *, cred_t *cr);
390 static void     connmgr_sndrel(struct cm_xprt *);

```

```

391 static void     connmgr_snddis(struct cm_xprt *);
392 static void     connmgr_close(struct cm_xprt *);
393 static void     connmgr_release(struct cm_xprt *);
394 static struct cm_xprt *connmgr_wrapget(struct netbuf *, const struct timeval *,
395     cku_private_t *);

397 static struct cm_xprt *connmgr_get(struct netbuf *, const struct timeval *,
398     struct netbuf *, int, struct netbuf *, struct rpc_err *, dev_t,
399     bool_t, int, cred_t *);

401 static void     connmgr_cancelconn(struct cm_xprt *);
402 static enum clnt_stat connmgr_cwait(struct cm_xprt *, const struct timeval *,
403     bool_t);
404 static void     connmgr_dis_and_wait(struct cm_xprt *);

406 static int      clnt_dispatch_send(queue_t *, mblk_t *, calllist_t *, uint_t,
407     uint_t);

409 static int      clnt_delay(clock_t, bool_t);

411 static int      waitforack(calllist_t *, t_scalar_t, const struct timeval *, bool_t);

413 /*
414  * Operations vector for TCP/IP based RPC
415  */
416 static struct clnt_ops tcp_ops = {
417     clnt_cots_kcallit,    /* do rpc call */
418     clnt_cots_kabort,    /* abort call */
419     clnt_cots_kerror,    /* return error status */
420     clnt_cots_kfreeres,  /* free results */
421     clnt_cots_kdestroy,  /* destroy rpc handle */
422     clnt_cots_kcontrol,  /* the ioctl() of rpc */
423     clnt_cots_ksettimers, /* set retry timers */
424 };
    unchanged portion omitted

465 #define COTSRSTAT_INCR(p, x) \
466     atomic_add_64(&(p)->x.value.ui64, 1)

468 #define CLNT_MAX_CONNS 1      /* concurrent connections between clnt/srvr */
469 int clnt_max_conns = CLNT_MAX_CONNS;

471 #define CLNT_MIN_TIMEOUT 10   /* seconds to wait after we get a */
472                                     /* connection reset */
473 #define CLNT_MIN_CONNTIMEOUT 5 /* seconds to wait for a connection */

476 int clnt_cots_min_tout = CLNT_MIN_TIMEOUT;
477 int clnt_cots_min_conntout = CLNT_MIN_CONNTIMEOUT;

479 /*
480  * Limit the number of times we will attempt to receive a reply without
481  * re-sending a response.
482  */
483 #define CLNT_MAXRECV_WITHOUT_RETRY 3
484 uint_t clnt_cots_maxrecv = CLNT_MAXRECV_WITHOUT_RETRY;

486 uint_t *clnt_max_msg_sizep;
487 void (*clnt_stop_idle)(queue_t *wq);

489 #define ptoh(p)          (&((p)->cku_client))
490 #define htop(h)          ((cku_private_t *) (h)->cl_private)

492 /*
493  * Times to retry
494  */

```

```

495 #define REFRESHES      2      /* authentication refreshes */
497 /*
498 * The following is used to determine the global default behavior for
499 * COTS when binding to a local port.
500 *
501 * If the value is set to 1 the default will be to select a reserved
502 * (aka privileged) port, if the value is zero the default will be to
503 * use non-reserved ports. Users of kRPC may override this by using
504 * CLNT_CONTROL() and CLSET_BINDRESVPORT.
505 */
506 int clnt_cots_do_bindresvport = 1;

508 static zone_key_t zone_cots_key;

510 #define TWO_GIGB      0x80000000
511 int nfsd_port = NFS_PORT;
512 /*
513 * Defaults TCP send and receive buffer size for NFS connections.
514 * These values can be tuned by /etc/default.
515 */
516 int nfs_send_bufsz = 1024*1024;
517 int nfs_rcv_bufsz = 1024*1024;
518 /*
519 * To use system-wide default for TCP send and receive buffer size,
520 * use /etc/system to set nfs_default_bufsz to 1:
521 *
522 * set rpcmod:nfs_default_bufsz=1
523 */
524 int nfs_default_bufsz = 0;

526 /*
527 * We need to do this after all kernel threads in the zone have exited.
528 */
529 /* ARGSUSED */
530 static void
531 clnt_zone_destroy(zoneid_t zoneid, void *unused)
532 {
533     struct cm_xprt **cmp;
534     struct cm_xprt *cm_entry;
535     struct cm_xprt *freelist = NULL;

537     mutex_enter(&connmgr_lock);
538     cmp = &cm_hd;
539     while ((cm_entry = *cmp) != NULL) {
540         if (cm_entry->x_zoneid == zoneid) {
541             *cmp = cm_entry->x_next;
542             cm_entry->x_next = freelist;
543             freelist = cm_entry;
544         } else {
545             *cmp = &cm_entry->x_next;
546         }
547     }
548     mutex_exit(&connmgr_lock);
549     while ((cm_entry = freelist) != NULL) {
550         freelist = cm_entry->x_next;
551         connmgr_close(cm_entry);
552     }
553 }

```

unchanged_portion_omitted

```

2581 /*
2582 * Set TCP receive and xmit buffer size for NFS connections.
2583 */
2584 static bool_t
2585 connmgr_nfs_setbufsz(calllist_t *e, int addrfamily, struct netbuf *addr,

```

```

2586     queue_t *wq, cred_t *cr)
2587 {
2588     struct sockaddr_in *sa;
2589     int ok = FALSE;
2590     int val;
2591     uint32_t sbufsz, rbufsz;

2593     if (nfs_default_bufsz ||
2594         (addrfamily != AF_INET && addrfamily != AF_INET6))
2595         return (FALSE);

2597     sa = (struct sockaddr_in *)addr->buf;
2598     if (ntohs(sa->sin_port) != nfsd_port)
2599         return (FALSE);
2600     /*
2601     * For system with 2GB, or less, of physical memory set send
2602     * and receive buffer size to half of nfs_send_bufsz and
2603     * nfs_rcv_bufsz respectively.
2604     */
2605     if (ptob(physmem) <= TWO_GIGB) {
2606         sbufsz = nfs_send_bufsz >> 1;
2607         rbufsz = nfs_rcv_bufsz >> 1;
2608     } else {
2609         sbufsz = nfs_send_bufsz;
2610         rbufsz = nfs_rcv_bufsz;
2611     }
2612     /*
2613     * Only set new buffer size if it's larger than the system
2614     * default buffer size. If smaller buffer size is needed
2615     * then use /etc/system to set nfs_default_bufsz to 1.
2616     */
2617     ok = connmgr_getopt_int(wq, SOL_SOCKET, SO_RCVBUF, &val, e, cr);
2618     if ((ok == TRUE) && (val < sbufsz)) {
2619         ok = connmgr_setopt_int(wq, SOL_SOCKET, SO_RCVBUF,
2620             sbufsz, e, cr);
2621         DTRACE_PROBE2(connmgr_nfs_rcvbufsz_setopt,
2622             int, ok, calllist_t *, e);
2623     }

2625     ok = connmgr_getopt_int(wq, SOL_SOCKET, SO_SNDBUF, &val, e, cr);
2626     if ((ok == TRUE) && (val < rbufsz)) {
2627         ok = connmgr_setopt_int(wq, SOL_SOCKET, SO_SNDBUF,
2628             rbufsz, e, cr);
2629         DTRACE_PROBE2(connmgr_nfs_sndbufsz_setopt,
2630             int, ok, calllist_t *, e);
2631     }
2632     return (TRUE);
2633 }

2635 /*
2636 * Given an open stream, connect to the remote. Returns true if connected,
2637 * false otherwise.
2638 */
2639 static bool_t
2640 connmgr_connect(
2641     struct cm_xprt *cm_entry,
2642     queue_t *wq,
2643     struct netbuf *addr,
2644     int addrfamily,
2645     calllist_t *e,
2646     int *tidu_ptr,
2647     bool_t reconnect,
2648     const struct timeval *waitp,
2649     bool_t nosignal,
2650     cred_t *cr)
2651 {

```

```

2652     mblk_t *mp;
2653     struct T_conn_req *tcr;
2654     struct T_info_ack *tinfo;
2655     int interrupted, error;
2656     int tidu_size, kstat_instance;

2658     /* if it's a reconnect, flush any lingering data messages */
2659     if (reconnect)
2660         (void) putctl1(wq, M_FLUSH, FLUSHRW);

2662     /*
2663     * Note: if the receiver uses SCM_UCRED/getpeerucred the pid will
2664     * appear as -1.
2665     */
2666     mp = allocb_cred(sizeof (*tcr) + addr->len, cr, NOPID);
2667     if (mp == NULL) {
2668         /*
2669         * This is unfortunate, but we need to look up the stats for
2670         * this zone to increment the "memory allocation failed"
2671         * counter. curproc->p_zone is safe since we're initiating a
2672         * connection and not in some strange streams context.
2673         */
2674         struct rpcstat *rpcstat;

2676         rpcstat = zone_getspecific(rpcstat_zone_key, rpc_zone());
2677         ASSERT(rpcstat != NULL);

2679         RPCLOG(1, "connmgr_connect: cannot alloc mp for "
2680             "sending conn request\n");
2681         COTSRCSTAT_INCR(rpcstat->rpc_cots_client, rcnomem);
2682         e->call_status = RPC_SYSTEMERROR;
2683         e->call_reason = ENOSR;
2684         return (FALSE);
2685     }

2687     /* Set TCP buffer size for NFS connections if needed */
2688     (void) connmgr_nfs_setbufsz(e, addrfmlly, addr, wq, cr);

2690     mp->b_datap->db_type = M_PROTO;
2691     tcr = (struct T_conn_req *)mp->b_rptr;
2692     bzero(tcr, sizeof (*tcr));
2693     tcr->PRIM_type = T_CONN_REQ;
2694     tcr->DEST_length = addr->len;
2695     tcr->DEST_offset = sizeof (struct T_conn_req);
2696     mp->b_wptr = mp->b_rptr + sizeof (*tcr);

2698     bcopy(addr->buf, mp->b_wptr, tcr->DEST_length);
2699     mp->b_wptr += tcr->DEST_length;

2701     RPCLOG(8, "connmgr_connect: sending conn request on queue "
2702         "%p", (void *)wq);
2703     RPCLOG(8, " call %p\n", (void *)wq);
2704     /*
2705     * We use the entry in the handle that is normally used for
2706     * waiting for RPC replies to wait for the connection accept.
2707     */
2708     if (clnt_dispatch_send(wq, mp, e, 0, 0) != RPC_SUCCESS) {
2709         DTRACE_PROBE(krpc__e__connmgr__connect__cantsend);
2710         freemsg(mp);
2711         return (FALSE);
2712     }

2714     mutex_enter(&clnt_pending_lock);

2716     /*
2717     * We wait for the transport connection to be made, or an

```

```

2718     * indication that it could not be made.
2719     */
2720     interrupted = 0;

2722     /*
2723     * waitforack should have been called with T_OK_ACK, but the
2724     * present implementation needs to be passed T_INFO_ACK to
2725     * work correctly.
2726     */
2727     error = waitforack(e, T_INFO_ACK, waitp, nosignal);
2728     if (error == EINTR)
2729         interrupted = 1;
2730     if (zone_status_get(curproc->p_zone) >= ZONE_IS_EMPTY) {
2731         /*
2732         * No time to lose; we essentially have been signaled to
2733         * quit.
2734         */
2735         interrupted = 1;
2736     }
2737     #ifdef RPCDEBUG
2738     if (error == ETIME)
2739         RPCLOG(8, "connmgr_connect: giving up "
2740             "on connection attempt; "
2741             "clnt_dispatch notifyconn "
2742             "diagnostic 'no one waiting for "
2743             "connection' should not be "
2744             "unexpected\n");
2745     #endif
2746     if (e->call_prev)
2747         e->call_prev->call_next = e->call_next;
2748     else
2749         clnt_pending = e->call_next;
2750     if (e->call_next)
2751         e->call_next->call_prev = e->call_prev;
2752     mutex_exit(&clnt_pending_lock);

2754     if (e->call_status != RPC_SUCCESS || error != 0) {
2755         if (interrupted)
2756             e->call_status = RPC_INTR;
2757         else if (error == ETIME)
2758             e->call_status = RPC_TIMEDOUT;
2759         else if (error == EPROTO) {
2760             e->call_status = RPC_SYSTEMERROR;
2761             e->call_reason = EPROTO;
2762         }

2764         RPCLOG(8, "connmgr_connect: can't connect, status: "
2765             "%s\n", clnt_sperrno(e->call_status));

2767         if (e->call_reply) {
2768             freemsg(e->call_reply);
2769             e->call_reply = NULL;
2770         }

2772         return (FALSE);
2773     }
2774     /*
2775     * The result of the "connection accept" is a T_info_ack
2776     * in the call_reply field.
2777     */
2778     ASSERT(e->call_reply != NULL);
2779     mp = e->call_reply;
2780     e->call_reply = NULL;
2781     tinfo = (struct T_info_ack *)mp->b_rptr;

2783     tidu_size = tinfo->TIDU_size;

```

```

2784 tidu_size -= (tidu_size % BYTES_PER_XDR_UNIT);
2785 if (tidu_size > COTS_DEFAULT_ALLOCSIZE || (tidu_size <= 0))
2786     tidu_size = COTS_DEFAULT_ALLOCSIZE;
2787 *tidu_ptr = tidu_size;

2789 freemsg(mp);

2791 /*
2792  * Set up the pertinent options. NODELAY is so the transport doesn't
2793  * buffer up RPC messages on either end. This may not be valid for
2794  * all transports. Failure to set this option is not cause to
2795  * bail out so we return success anyway. Note that lack of NODELAY
2796  * or some other way to flush the message on both ends will cause
2797  * lots of retries and terrible performance.
2798  */
2799 if (addrfmly == AF_INET || addrfmly == AF_INET6) {
2800     (void) connmgr_setopt(wq, IPPROTO_TCP, TCP_NODELAY, e, cr);
2801     if (e->call_status == RPC_XPRTFAILED)
2802         return (FALSE);
2803 }

2805 /*
2806  * Since we have a connection, we now need to figure out if
2807  * we need to create a kstat. If x_ksp is not NULL then we
2808  * are reusing a connection and so we do not need to create
2809  * another kstat -- lets just return.
2810  */
2811 if (cm_entry->x_ksp != NULL)
2812     return (TRUE);

2814 /*
2815  * We need to increment rpc_kstat_instance atomically to prevent
2816  * two kstats being created with the same instance.
2817  */
2818 kstat_instance = atomic_add_32_nv((uint32_t *)&rpc_kstat_instance, 1);

2820 if ((cm_entry->x_ksp = kstat_create_zone("unix", kstat_instance,
2821     "rpc_cots_connections", "rpc", KSTAT_TYPE_NAMED,
2822     (uint_t)(sizeof (cm_kstat_xprt_t) / sizeof (kstat_named_t)),
2823     KSTAT_FLAG_VIRTUAL, cm_entry->x_zoneid)) == NULL) {
2824     return (TRUE);
2825 }

2827 cm_entry->x_ksp->ks_lock = &connmgr_lock;
2828 cm_entry->x_ksp->ks_private = cm_entry;
2829 cm_entry->x_ksp->ks_data_size = ((INET6_ADDRSTRLEN * sizeof (char))
2830     + sizeof (cm_kstat_template));
2831 cm_entry->x_ksp->ks_data = kmem_alloc(cm_entry->x_ksp->ks_data_size,
2832     KM_SLEEP);
2833 bcopy(&cm_kstat_template, cm_entry->x_ksp->ks_data,
2834     cm_entry->x_ksp->ks_data_size);
2835 ((struct cm_kstat_xprt *) (cm_entry->x_ksp->ks_data))->
2836     x_server.value.str.addr.ptr =
2837     kmem_alloc(INET6_ADDRSTRLEN, KM_SLEEP);

2839 cm_entry->x_ksp->ks_update = conn_kstat_update;
2840 kstat_install(cm_entry->x_ksp);
2841 return (TRUE);
2842 }

2844 /*
2845  * Verify that the specified offset falls within the mblk and
2846  * that the resulting pointer is aligned.
2847  * Returns NULL if not.
2848  *
2849  * code from fs/sockfs/socksubr.c

```

```

2850 */
2851 static void *
2852 connmgr_opt_getoff(mblk_t *mp, t_uscalar_t offset,
2853     t_uscalar_t length, uint_t align_size)
2854 {
2855     uintptr_t ptr1, ptr2;

2857     ASSERT(mp && mp->b_wptr >= mp->b_rptr);
2858     ptr1 = (uintptr_t)mp->b_rptr + offset;
2859     ptr2 = (uintptr_t)ptr1 + length;
2860     if (ptr1 < (uintptr_t)mp->b_rptr || ptr2 > (uintptr_t)mp->b_wptr) {
2861         return (NULL);
2862     }
2863     if ((ptr1 & (align_size - 1)) != 0) {
2864         return (NULL);
2865     }
2866     return ((void *)ptr1);
2867 }

2869 static bool_t
2870 connmgr_getopt_int(queue_t *wq, int level, int name, int *val,
2871     calllist_t *e, cred_t *cr)
2872 {
2873     mblk_t *mp;
2874     struct ophdr *opt, *opt_res;
2875     struct T_optmgmt_req *tor;
2876     struct T_optmgmt_ack *opt_ack;
2877     struct timeval waitp;
2878     int error;

2880     mp = allocb_cred(sizeof (struct T_optmgmt_req) +
2881         sizeof (struct ophdr) + sizeof (int), cr, NOPID);
2882     if (mp == NULL) {
2883         RPCLOG0(1, "connmgr_getopt: cannot alloc mp for option "
2884             "request\n");
2885         return (FALSE);
2886     }

2888     mp->b_datap->db_type = M_PROTO;
2889     tor = (struct T_optmgmt_req *) (mp->b_rptr);
2890     tor->PRIM_type = T_SVR4_OPTMGMT_REQ;
2891     tor->MGMT_flags = T_CURRENT;
2892     tor->OPT_length = sizeof (struct ophdr) + sizeof (int);
2893     tor->OPT_offset = sizeof (struct T_optmgmt_req);

2895     opt = (struct ophdr *) (mp->b_rptr + sizeof (struct T_optmgmt_req));
2896     opt->level = level;
2897     opt->name = name;
2898     opt->len = sizeof (int);
2899     mp->b_wptr += sizeof (struct T_optmgmt_req) + sizeof (struct ophdr) +
2900         sizeof (int);

2902     /*
2903      * We will use this connection regardless
2904      * of whether or not the option is readable.
2905      */
2906     if (clnt_dispatch_send(wq, mp, e, 0, 0) != RPC_SUCCESS) {
2907         DTRACE_PROBE(krpc_e_connmgr_getopt_cantsend);
2908         freemsg(mp);
2909         return (FALSE);
2910     }

2912     mutex_enter(&clnt_pending_lock);

2914     waitp.tv_sec = clnt_cots_min_comntout;
2915     waitp.tv_usec = 0;

```

```

2916     error = waitforack(e, T_OPTMGMT_ACK, &waitp, 1);
2918     if (e->call_prev)
2919         e->call_prev->call_next = e->call_next;
2920     else
2921         clnt_pending = e->call_next;
2922     if (e->call_next)
2923         e->call_next->call_prev = e->call_prev;
2924     mutex_exit(&clnt_pending_lock);

2926     /* get reply message */
2927     mp = e->call_reply;
2928     e->call_reply = NULL;

2930     if ((!mp) || (e->call_status != RPC_SUCCESS) || (error != 0)) {
2932         DTRACE_PROBE4(connmgr_getopt_failed, int, name,
2933             int, e->call_status, int, error, mblk_t *, mp);

2935         if (mp)
2936             freemsg(mp);
2937         return (FALSE);
2938     }

2940     opt_ack = (struct T_optmgmt_ack *)mp->b_rptr;
2941     opt_res = (struct opthdr *)connmgr_opt_getoff(mp, opt_ack->OPT_offset,
2942         opt_ack->OPT_length, __TPI_ALIGN_SIZE);

2944     if (!opt_res) {
2945         DTRACE_PROBE4(connmgr_getopt_optres, mblk_t *, mp, int, name,
2946             int, opt_ack->OPT_offset, int, opt_ack->OPT_length);
2947         freemsg(mp);
2948         return (FALSE);
2949     }
2950     *val = *(int *)&opt_res[1];

2952     DTRACE_PROBE2(connmgr_getopt_ok, int, name, int, *val);

2954     freemsg(mp);
2955     return (TRUE);
2956 }

2958 /*
2959  * Called by connmgr_connect to set an option on the new stream.
2960  */
2961 static bool_t
2962 connmgr_setopt_int(queue_t *wq, int level, int name, int val,
2963     calllist_t *e, cred_t *cr)
2964 connmgr_setopt(queue_t *wq, int level, int name, calllist_t *e, cred_t *cr)
2965 {
2966     mblk_t *mp;
2967     struct opthdr *opt;
2968     struct T_optmgmt_req *tor;
2969     struct timeval waitp;
2970     int error;

2971     mp = allocb_cred(sizeof (struct T_optmgmt_req) +
2972         sizeof (struct opthdr) + sizeof (int), cr, NOPID);
2973     if (mp == NULL) {
2974         RPCLOG0(1, "connmgr_setopt: cannot alloc mp for option "
2975             "request\n");
2976         return (FALSE);
2977     }

2979     mp->b_datap->db_type = M_PROTO;
2980     tor = (struct T_optmgmt_req *) (mp->b_rptr);

```

```

2981     tor->PRIM_type = T_SVR4_OPTMGMT_REQ;
2982     tor->MGMT_flags = T_NEGOTIATE;
2983     tor->OPT_length = sizeof (struct opthdr) + sizeof (int);
2984     tor->OPT_offset = sizeof (struct T_optmgmt_req);

2986     opt = (struct opthdr *) (mp->b_rptr + sizeof (struct T_optmgmt_req));
2987     opt->level = level;
2988     opt->name = name;
2989     opt->len = sizeof (int);
2990     *(int *) ((char *) opt + sizeof (*opt)) = val;
2991     *(int *) ((char *) opt + sizeof (*opt)) = 1;
2992     mp->b_wptr += sizeof (struct T_optmgmt_req) + sizeof (struct opthdr) +
2993         sizeof (int);

2994     /*
2995      * We will use this connection regardless
2996      * of whether or not the option is settable.
2997      */
2998     if (clnt_dispatch_send(wq, mp, e, 0, 0) != RPC_SUCCESS) {
2999         DTRACE_PROBE(krpc_e_connmgr_setopt_cantsend);
3000         freemsg(mp);
3001         return (FALSE);
3002     }

3004     mutex_enter(&clnt_pending_lock);

3006     waitp.tv_sec = clnt_cots_min_conntout;
3007     waitp.tv_usec = 0;
3008     error = waitforack(e, T_OPTMGMT_ACK, &waitp, 1);

3010     if (e->call_prev)
3011         e->call_prev->call_next = e->call_next;
3012     else
3013         clnt_pending = e->call_next;
3014     if (e->call_next)
3015         e->call_next->call_prev = e->call_prev;
3016     mutex_exit(&clnt_pending_lock);

3018     if (e->call_reply != NULL) {
3019         freemsg(e->call_reply);
3020         e->call_reply = NULL;
3021     }

3023     if (e->call_status != RPC_SUCCESS || error != 0) {
3024         RPCLOG(1, "connmgr_setopt: can't set option: %d\n", name);
3025         return (FALSE);
3026     }
3027     RPCLOG(8, "connmgr_setopt: successfully set option: %d\n", name);
3028     return (TRUE);
3029 }

3031 static bool_t
3032 connmgr_setopt(queue_t *wq, int level, int name, calllist_t *e, cred_t *cr)
3033 {
3034     return (connmgr_setopt_int(wq, level, name, 1, e, cr));
3035 }

3037 #ifdef DEBUG

3039 /*
3040  * This is a knob to let us force code coverage in allocation failure
3041  * case.
3042  */
3043 static int connmgr_failsnd;
3044 #define CONN_SND_ALLOC(Size, Pri) \
3045     ((connmgr_failsnd-- > 0) ? NULL : allocb(Size, Pri))

```

```
3047 #else
3049 #define CONN_SND_ALLOC(Size, Pri)      allocb(Size, Pri)
3051 #endif
3053 /*
3054  * Sends an orderly release on the specified queue.
3055  * Entered with connmgr_lock. Exited without connmgr_lock
3056  */
3057 static void
3058 connmgr_sndrel(struct cm_xprt *cm_entry)
3059 {
3060     struct T_ordrel_req *torr;
3061     mblk_t *mp;
3062     queue_t *q = cm_entry->x_wq;
3063     ASSERT(MUTEX_HELD(&connmgr_lock));
3064     mp = CONN_SND_ALLOC(sizeof (struct T_ordrel_req), BPRI_LO);
3065     if (mp == NULL) {
3066         cm_entry->x_needrel = TRUE;
3067         mutex_exit(&connmgr_lock);
3068         RPCLOG(1, "connmgr_sndrel: cannot alloc mp for sending ordrel "
3069             "to queue %p\n", (void *)q);
3070         return;
3071     }
3072     mutex_exit(&connmgr_lock);
3074     mp->b_datap->db_type = M_PROTO;
3075     torr = (struct T_ordrel_req *) (mp->b_rptr);
3076     torr->PRIM_type = T_ORDREL_REQ;
3077     mp->b_wptr = mp->b_rptr + sizeof (struct T_ordrel_req);
3079     RPCLOG(8, "connmgr_sndrel: sending ordrel to queue %p\n", (void *)q);
3080     put(q, mp);
3081 }
unchanged portion omitted
```