Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp.c

```
 1 /*
 2  * INET         An implementation of the TCP/IP protocol suite for the LINUX
 3  *              operating system.  INET is implemented using the  BSD Socket
 4  *              interface as the means of communication with the user level.
 5  *
 6  *              Implementation of the Transmission Control Protocol(TCP).
 7  *
 8  * Authors:     Ross Biro
 9  *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
10  *              Mark Evans, <evansmp@uhura.aston.ac.uk>
11  *              Corey Minyard <wf-rch!minyard@relay.EU.net>
12  *              Florian La Roche, <flla@stud.uni-sb.de>
13  *              Charles Hedrick, <hedrick@klinzhai.rutgers.edu>
14  *              Linus Torvalds, <torvalds@cs.helsinki.fi>
15  *              Alan Cox, <gw4pts@gw4pts.ampr.org>
16  *              Matthew Dillon, <dillon@apollo.west.oic.com>
17  *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
18  *              Jorge Cwik, <jorge@laser.satlink.net>
19  *
20  * Fixes:
21  *              Alan Cox        :       Numerous verify_area() calls
22  *              Alan Cox        :       Set the ACK bit on a reset
23  *              Alan Cox        :       Stopped it crashing if it closed while
24  *                                      sk->inuse=1 and was trying to connect
25  *                                      (tcp_err()).
26  *              Alan Cox        :       All icmp error handling was broken
27  *                                      pointers passed where wrong and the
28  *                                      socket was looked up backwards. Nobody
29  *                                      tested any icmp error code obviously.
30  *              Alan Cox        :       tcp_err() now handled properly. It
31  *                                      wakes people on errors. poll
32  *                                      behaves and the icmp error race
33  *                                      has gone by moving it into sock.c
34  *              Alan Cox        :       tcp_send_reset() fixed to work for
35  *                                      everything not just packets for
36  *                                      unknown sockets.
37  *              Alan Cox        :       tcp option processing.
38  *              Alan Cox        :       Reset tweaked (still not 100%) [Had
39  *                                      syn rule wrong]
40  *              Herp Rosmanith  :       More reset fixes
41  *              Alan Cox        :       No longer acks invalid rst frames.
42  *                                      Acking any kind of RST is right out.
43  *              Alan Cox        :       Sets an ignore me flag on an rst
44  *                                      receive otherwise odd bits of prattle
45  *                                      escape still
46  *              Alan Cox        :       Fixed another acking RST frame bug.
47  *                                      Should stop LAN workplace lockups.
48  *              Alan Cox        :       Some tidyups using the new skb list
49  *                                      facilities
50  *              Alan Cox        :       sk->keepopen now seems to work
51  *              Alan Cox        :       Pulls options out correctly on accepts
52  *              Alan Cox        :       Fixed assorted sk->rqueue->next errors
53  *              Alan Cox        :       PSH doesn't end a TCP read. Switched a
54  *                                      bit to skb ops.
55  *              Alan Cox        :       Tidied tcp_data to avoid a potential
56  *                                      nasty.
57  *              Alan Cox        :       Added some better commenting, as the
58  *                                      tcp is hard to follow
59  *              Alan Cox        :       Removed incorrect check for 20 * psh
60  *      Michael O'Reilly        :       ack < copied bug fix.
61  *      Johannes Stille         :       Misc tcp fixes (not all in yet).
62  *              Alan Cox        :       FIN with no memory -> CRASH
63  *              Alan Cox        :       Added socket option proto entries.
64  *                                      Also added awareness of them to accept.
65  *              Alan Cox        :       Added TCP options (SOL_TCP)
```

```
 66  *              Alan Cox        :       Switched wakeup calls to callbacks,
 67  *                                      so the kernel can layer network
 68  *                                      sockets.
 69  *              Alan Cox        :       Use ip_tos/ip_ttl settings.
 70  *              Alan Cox        :       Handle FIN (more) properly (we hope).
 71  *              Alan Cox        :       RST frames sent on unsynchronised
 72  *                                      state ack error.
 73  *              Alan Cox        :       Put in missing check for SYN bit.
 74  *              Alan Cox        :       Added tcp_select_window() aka NET2E
 75  *                                      window non shrink trick.
 76  *              Alan Cox        :       Added a couple of small NET2E timer
 77  *                                      fixes
 78  *              Charles Hedrick :       TCP fixes
 79  *              Toomas Tamm     :       TCP window fixes
 80  *              Alan Cox        :       Small URG fix to rlogin ^C ack fight
 81  *              Charles Hedrick :       Rewrote most of it to actually work
 82  *              Linus           :       Rewrote tcp_read() and URG handling
 83  *                                      completely
 84  *              Gerhard Koerting:       Fixed some missing timer handling
 85  *              Matthew Dillon  :       Reworked TCP machine states as per RFC
 86  *              Gerhard Koerting:       PC/TCP workarounds
 87  *              Adam Caldwell   :       Assorted timer/timing errors
 88  *              Matthew Dillon  :       Fixed another RST bug
 89  *              Alan Cox        :       Move to kernel side addressing changes.
 90  *              Alan Cox        :       Beginning work on TCP fastpathing
 91  *                                      (not yet usable)
 92  *              Arnt Gulbrandsen:       Turbocharged tcp_check() routine.
 93  *              Alan Cox        :       TCP fast path debugging
 94  *              Alan Cox        :       Window clamping
 95  *              Michael Riepe   :       Bug in tcp_check()
 96  *              Matt Dillon     :       More TCP improvements and RST bug fixes
 97  *              Matt Dillon     :       Yet more small nasties remove from the
 98  *                                      TCP code (Be very nice to this man if
 99  *                                      tcp finally works 100%) 8)
100  *              Alan Cox        :       BSD accept semantics.
101  *              Alan Cox        :       Reset on closedown bug.
102  *      Peter De Schrijver      :       ENOTCONN check missing in tcp_sendto().
103  *              Michael Pall    :       Handle poll() after URG properly in
104  *                                      all cases.
105  *              Michael Pall    :       Undo the last fix in tcp_read_urg()
106  *                                      (multi URG PUSH broke rlogin).
107  *              Michael Pall    :       Fix the multi URG PUSH problem in
108  *                                      tcp_readable(), poll() after URG
109  *                                      works now.
110  *              Michael Pall    :       recv(...,MSG_OOB) never blocks in the
111  *                                      BSD api.
112  *              Alan Cox        :       Changed the semantics of sk->socket to
113  *                                      fix a race and a signal problem with
114  *                                      accept() and async I/O.
115  *              Alan Cox        :       Relaxed the rules on tcp_sendto().
116  *              Yury Shevchuk   :       Really fixed accept() blocking problem.
117  *              Craig I. Hagan  :       Allow for BSD compatible TIME_WAIT for
118  *                                      clients/servers which listen in on
119  *                                      fixed ports.
120  *              Alan Cox        :       Cleaned the above up and shrank it to
121  *                                      a sensible code size.
122  *              Alan Cox        :       Self connect lockup fix.
123  *              Alan Cox        :       No connect to multicast.
124  *              Ross Biro       :       Close unaccepted children on master
125  *                                      socket close.
126  *              Alan Cox        :       Reset tracing code.
127  *              Alan Cox        :       Spurious resets on shutdown.
128  *              Alan Cox        :       Giant 15 minute/60 second timer error
129  *              Alan Cox        :       Small whoops in polling before an
130  *                                      accept.
131  *              Alan Cox        :       Kept the state trace facility since
132  *                                      it's handy for debugging.
133  *              Alan Cox        :       More reset handler fixes.
134  *              Alan Cox        :       Started rewriting the code based on
135  *                                      the RFC's for other useful protocol
136  *                                      references see: Comer, KA9Q NOS, and
137  *                                      for a reference on the difference
138  *                                      between specifications and how BSD
139  *                                      works see the 4.4lite source.
140  *              A.N.Kuznetsov   :       Don't time wait on completion of tidy
141  *                                      close.
142  *              Linus Torvalds  :       Fin/Shutdown & copied_seq changes.
143  *              Linus Torvalds  :       Fixed BSD port reuse to work first syn
144  *              Alan Cox        :       Reimplemented timers as per the RFC
145  *                                      and using multiple timers for sanity.
146  *              Alan Cox        :       Small bug fixes, and a lot of new
147  *                                      comments.
148  *              Alan Cox        :       Fixed dual reader crash by locking
149  *                                      the buffers (much like datagram.c)
150  *              Alan Cox        :       Fixed stuck sockets in probe. A probe
```

```
151  *                                      now gets fed up of retrying without
152  *                                      (even a no space) answer.
153  *              Alan Cox        :       Extracted closing code better
154  *              Alan Cox        :       Fixed the closing state machine to
155  *                                      resemble the RFC.
156  *              Alan Cox        :       More 'per spec' fixes.
157  *              Jorge Cwik      :       Even faster checksumming.
158  *              Alan Cox        :       tcp_data() doesn't ack illegal PSH
159  *                                      only frames. At least one pc tcp stack
160  *                                      generates them.
161  *              Alan Cox        :       Cache last socket.
162  *              Alan Cox        :       Per route irtt.
163  *              Matt Day        :       poll()->select() match BSD precisely on error
164  *              Alan Cox        :       New buffers
165  *              Marc Tamsky     :       Various sk->prot->retransmits and
166  *                                      sk->retransmits misupdating fixed.
167  *                                      Fixed tcp_write_timeout: stuck close,
168  *                                      and TCP syn retries gets used now.
169  *              Mark Yarvis     :       In tcp_read_wakeup(), don't send an
170  *                                      ack if state is TCP_CLOSED.
171  *              Alan Cox        :       Look up device on a retransmit - routes may
172  *                                      change. Doesn't yet cope with MSS shrink right
173  *                                      but it's a start!
174  *              Marc Tamsky     :       Closing in closing fixes.
175  *              Mike Shaver     :       RFC1122 verifications.
176  *              Alan Cox        :       rcv_saddr errors.
177  *              Alan Cox        :       Block double connect().
178  *              Alan Cox        :       Small hooks for enSKIP.
179  *              Alexey Kuznetsov:       Path MTU discovery.
180  *              Alan Cox        :       Support soft errors.
181  *              Alan Cox        :       Fix MTU discovery pathological case
182  *                                      when the remote claims no mtu!
183  *              Marc Tamsky     :       TCP_CLOSE fix.
184  *              Colin (G3TNE)   :       Send a reset on syn ack replies in
185  *                                      window but wrong (fixes NT lpd problems)
186  *              Pedro Roque     :       Better TCP window handling, delayed ack.
187  *              Joerg Reuter    :       No modification of locked buffers in
188  *                                      tcp_do_retransmit()
189  *              Eric Schenk     :       Changed receiver side silly window
190  *                                      avoidance algorithm to BSD style
191  *                                      algorithm. This doubles throughput
192  *                                      against machines running Solaris,
193  *                                      and seems to result in general
194  *                                      improvement.
195  *      Stefan Magdalinski      :       adjusted tcp_readable() to fix FIONREAD
196  *      Willy Konynenberg       :       Transparent proxying support.
197  *      Mike McLagan            :       Routing by source
198  *              Keith Owens     :       Do proper merging with partial SKB's in
199  *                                      tcp_do_sendmsg to avoid burstiness.
200  *              Eric Schenk     :       Fix fast close down bug with
201  *                                      shutdown() followed by close().
202  *              Andi Kleen      :       Make poll agree with SIGIO
203  *      Salvatore Sanfilippo    :       Support SO_LINGER with linger == 1 and
204  *                                      lingertime == 0 (RFC 793 ABORT Call)
205  *      Hirokazu Takahashi      :       Use copy_from_user() instead of
206  *                                      csum_and_copy_from_user() if possible.
207  *
208  *              This program is free software; you can redistribute it and/or
209  *              modify it under the terms of the GNU General Public License
210  *              as published by the Free Software Foundation; either version
211  *              2 of the License, or(at your option) any later version.
212  *
213  * Description of States:
214  *
215  *      TCP_SYN_SENT            sent a connection request, waiting for ack
216  *
217  *      TCP_SYN_RECV            received a connection request, sent ack,
218  *                              waiting for final ack in three-way handshake.
219  *
220  *      TCP_ESTABLISHED         connection established
221  *
222  *      TCP_FIN_WAIT1           our side has shutdown, waiting to complete
223  *                              transmission of remaining buffered data
224  *
225  *      TCP_FIN_WAIT2           all buffered data sent, waiting for remote
226  *                              to shutdown
227  *
228  *      TCP_CLOSING             both sides have shutdown but we still have
229  *                              data we have to finish sending
230  *
231  *      TCP_TIME_WAIT           timeout to catch resent junk before entering
232  *                              closed, can only be entered from FIN_WAIT2
233  *                              or CLOSING.  Required because the other end
234  *                              may not have gotten our last ACK causing it
235  *                              to retransmit the data packet (which we ignore)
```

```
236  *
237  *      TCP_CLOSE_WAIT          remote side has shutdown and is waiting for
238  *                              us to finish writing our data and to shutdown
239  *                              (we have to close() to move on to LAST_ACK)
240  *
241  *      TCP_LAST_ACK            out side has shutdown after remote has
242  *                              shutdown.  There may still be data in our
243  *                              buffer that we have to finish sending
244  *
245  *      TCP_CLOSE               socket is finished
246  */
247
248 #define pr_fmt(fmt) "TCP: " fmt
249
250 #include <linux/kernel.h>
251 #include <linux/module.h>
252 #include <linux/types.h>
253 #include <linux/fcntl.h>
254 #include <linux/poll.h>
255 #include <linux/init.h>
256 #include <linux/fs.h>
257 #include <linux/skbuff.h>
258 #include <linux/scatterlist.h>
259 #include <linux/splice.h>
260 #include <linux/net.h>
261 #include <linux/socket.h>
262 #include <linux/random.h>
263 #include <linux/bootmem.h>
264 #include <linux/highmem.h>
265 #include <linux/swap.h>
266 #include <linux/cache.h>
267 #include <linux/err.h>
268 #include <linux/crypto.h>
269 #include <linux/time.h>
270 #include <linux/slab.h>
271
272 #include <net/icmp.h>
273 #include <net/inet_common.h>
274 #include <net/tcp.h>
275 #include <net/xfrm.h>
276 #include <net/ip.h>
277 #include <net/netdma.h>
278 #include <net/sock.h>
279
280 #include <asm/uaccess.h>
281 #include <asm/ioctls.h>
282 #include <net/busy_poll.h>
283
284 int sysctl_tcp_fin_timeout __read_mostly = TCP_FIN_TIMEOUT;
285
286 int sysctl_tcp_min_tso_segs __read_mostly = 2;
287
288 int sysctl_tcp_autocorking __read_mostly = 1;
289
290 struct percpu_counter tcp_orphan_count;
291 EXPORT_SYMBOL_GPL(tcp_orphan_count);
292
293 long sysctl_tcp_mem[3] __read_mostly;
294 int sysctl_tcp_wmem[3] __read_mostly;
295 int sysctl_tcp_rmem[3] __read_mostly;
296
297 EXPORT_SYMBOL(sysctl_tcp_mem);
298 EXPORT_SYMBOL(sysctl_tcp_rmem);
299 EXPORT_SYMBOL(sysctl_tcp_wmem);
300
301 atomic_long_t tcp_memory_allocated;     /* Current allocated memory. */
302 EXPORT_SYMBOL(tcp_memory_allocated);
303
304 /*
305  * Current number of TCP sockets.
306  */
307 struct percpu_counter tcp_sockets_allocated;
308 EXPORT_SYMBOL(tcp_sockets_allocated);
309
310 /*
311  * TCP splice context
312  */
313 struct tcp_splice_state {
314         struct pipe_inode_info *pipe;
315         size_t len;
316         unsigned int flags;
317 };
318
319 /*
320  * Pressure flag: try to collapse.
```

```
321  * Technical note: it is used by multiple contexts non atomically.
322  * All the __sk_mem_schedule() is of this nature: accounting
323  * is strict, actions are advisory and have some latency.
324  */
325 int tcp_memory_pressure __read_mostly;
326 EXPORT_SYMBOL(tcp_memory_pressure);
327
328 void tcp_enter_memory_pressure(struct sock *sk)
329 {
330         if (!tcp_memory_pressure) {
331                 NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPMEMORYPRESSURES);
332                 tcp_memory_pressure = 1;
333         }
334 }
335 EXPORT_SYMBOL(tcp_enter_memory_pressure);
336
337 /* Convert seconds to retransmits based on initial and max timeout */
338 static u8 secs_to_retrans(int seconds, int timeout, int rto_max)
339 {
340         u8 res = 0;
341
342         if (seconds > 0) {
343                 int period = timeout;
344
345                 res = 1;
346                 while (seconds > period && res < 255) {
347                         res++;
348                         timeout <<= 1;
349                         if (timeout > rto_max)
350                                 timeout = rto_max;
351                         period += timeout;
352                 }
353         }
354         return res;
355 }
356
357 /* Convert retransmits to seconds based on initial and max timeout */
358 static int retrans_to_secs(u8 retrans, int timeout, int rto_max)
359 {
360         int period = 0;
361
362         if (retrans > 0) {
363                 period = timeout;
364                 while (--retrans) {
365                         timeout <<= 1;
366                         if (timeout > rto_max)
367                                 timeout = rto_max;
368                         period += timeout;
369                 }
370         }
371         return period;
372 }
373
374 /* Address-family independent initialization for a tcp_sock.
375  *
376  * NOTE: A lot of things set to zero explicitly by call to
377  *       sk_alloc() so need not be done here.
378  */
379 void tcp_init_sock(struct sock *sk)
380 {
381         struct inet_connection_sock *icsk = inet_csk(sk);
382         struct tcp_sock *tp = tcp_sk(sk);
383
384         __skb_queue_head_init(&tp->out_of_order_queue);
385         tcp_init_xmit_timers(sk);
386         tcp_prequeue_init(tp);
387         INIT_LIST_HEAD(&tp->tsq_node);
388
389         icsk->icsk_rto = TCP_TIMEOUT_INIT;
390         tp->mdev_us = jiffies_to_usecs(TCP_TIMEOUT_INIT);
391
392         /* So many TCP implementations out there (incorrectly) count the
393          * initial SYN frame in their delayed-ACK and congestion control
394          * algorithms that we must have the following bandaid to talk
395          * efficiently to them.  -DaveM
396          */
397         tp->snd_cwnd = TCP_INIT_CWND;
398
399         /* See draft-stevens-tcpca-spec-01 for discussion of the
400          * initialization of these values.
401          */
402         tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
403         tp->snd_cwnd_clamp = ~0;
404         tp->mss_cache = TCP_MSS_DEFAULT;
405
```

```
406         tp->reordering = sysctl_tcp_reordering;
407         tcp_enable_early_retrans(tp);
408         icsk->icsk_ca_ops = &tcp_init_congestion_ops;
409
410         tp->tsoffset = 0;
411
412         sk->sk_state = TCP_CLOSE;
413
414         sk->sk_write_space = sk_stream_write_space;
415         sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);
416
417         icsk->icsk_sync_mss = tcp_sync_mss;
418
419         sk->sk_sndbuf = sysctl_tcp_wmem[1];
420         sk->sk_rcvbuf = sysctl_tcp_rmem[1];
421
422         local_bh_disable();
423         sock_update_memcg(sk);
424         sk_sockets_allocated_inc(sk);
425         local_bh_enable();
426 }
427 EXPORT_SYMBOL(tcp_init_sock);
428
429 static void tcp_tx_timestamp(struct sock *sk, struct sk_buff *skb)
430 {
431         if (sk->sk_tsflags) {
432                 struct skb_shared_info *shinfo = skb_shinfo(skb);
433
434                 sock_tx_timestamp(sk, &shinfo->tx_flags);
435                 if (shinfo->tx_flags & SKBTX_ANY_TSTAMP)
436                         shinfo->tskey = TCP_SKB_CB(skb)->seq + skb->len - 1;
437         }
438 }
439
440 /*
441  *      Wait for a TCP event.
442  *
443  *      Note that we don't need to lock the socket, as the upper poll layers
444  *      take care of normal races (between the test and the event) and we don't
445  *      go look at any of the socket buffers directly.
446  */
447 unsigned int tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
448 {
449         unsigned int mask;
450         struct sock *sk = sock->sk;
451         const struct tcp_sock *tp = tcp_sk(sk);
452
453         sock_rps_record_flow(sk);
454
455         sock_poll_wait(file, sk_sleep(sk), wait);
456         if (sk->sk_state == TCP_LISTEN)
457                 return inet_csk_listen_poll(sk);
458
459         /* Socket is not locked. We are protected from async events
460          * by poll logic and correct handling of state changes
461          * made by other threads is impossible in any case.
462          */
463
464         mask = 0;
465
466         /*
467          * POLLHUP is certainly not done right. But poll() doesn't
468          * have a notion of HUP in just one direction, and for a
469          * socket the read side is more interesting.
470          *
471          * Some poll() documentation says that POLLHUP is incompatible
472          * with the POLLOUT/POLLWR flags, so somebody should check this
473          * all. But careful, it tends to be safer to return too many
474          * bits than too few, and you can easily break real applications
475          * if you don't tell them that something has hung up!
476          *
477          * Check-me.
478          *
479          * Check number 1. POLLHUP is _UNMASKABLE_ event (see UNIX98 and
480          * our fs/select.c). It means that after we received EOF,
481          * poll always returns immediately, making impossible poll() on write()
482          * in state CLOSE_WAIT. One solution is evident --- to set POLLHUP
483          * if and only if shutdown has been made in both directions.
484          * Actually, it is interesting to look how Solaris and DUX
485          * solve this dilemma. I would prefer, if POLLHUP were maskable,
486          * then we could set it on SND_SHUTDOWN. BTW examples given
487          * in Stevens' books assume exactly this behaviour, it explains
488          * why POLLHUP is incompatible with POLLOUT.    --ANK
489          *
490          * NOTE. Check for TCP_CLOSE is added. The goal is to prevent
```

```
491          * blocking on fresh not-connected or disconnected socket. --ANK
492          */
493         if (sk->sk_shutdown == SHUTDOWN_MASK || sk->sk_state == TCP_CLOSE)
494                 mask |= POLLHUP;
495         if (sk->sk_shutdown & RCV_SHUTDOWN)
496                 mask |= POLLIN | POLLRDNORM | POLLRDHUP;
497
498         /* Connected or passive Fast Open socket? */
499         if (sk->sk_state != TCP_SYN_SENT &&
500             (sk->sk_state != TCP_SYN_RECV || tp->fastopen_rsk != NULL)) {
501                 int target = sock_rcvlowat(sk, 0, INT_MAX);
502
503                 if (tp->urg_seq == tp->copied_seq &&
504                     !sock_flag(sk, SOCK_URGINLINE) &&
505                     tp->urg_data)
506                         target++;
507
508                 /* Potential race condition. If read of tp below will
509                  * escape above sk->sk_state, we can be illegally awaken
510                  * in SYN_* states. */
511                 if (tp->rcv_nxt - tp->copied_seq >= target)
512                         mask |= POLLIN | POLLRDNORM;
513
514                 if (!(sk->sk_shutdown & SEND_SHUTDOWN)) {
515                         if (sk_stream_is_writeable(sk)) {
516                                 mask |= POLLOUT | POLLWRNORM;
517                         } else {  /* send SIGIO later */
518                                 set_bit(SOCK_ASYNC_NOSPACE,
519                                         &sk->sk_socket->flags);
520                                 set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
521
522                                 /* Race breaker. If space is freed after
523                                  * wspace test but before the flags are set,
524                                  * IO signal will be lost.
525                                  */
526                                 if (sk_stream_is_writeable(sk))
527                                         mask |= POLLOUT | POLLWRNORM;
528                         }
529                 } else
530                         mask |= POLLOUT | POLLWRNORM;
531
532                 if (tp->urg_data & TCP_URG_VALID)
533                         mask |= POLLPRI;
534         }
535         /* This barrier is coupled with smp_wmb() in tcp_reset() */
536         smp_rmb();
537         if (sk->sk_err || !skb_queue_empty(&sk->sk_error_queue))
538                 mask |= POLLERR;
539
540         return mask;
541 }
542 EXPORT_SYMBOL(tcp_poll);
543
544 int tcp_ioctl(struct sock *sk, int cmd, unsigned long arg)
545 {
546         struct tcp_sock *tp = tcp_sk(sk);
547         int answ;
548         bool slow;
549
550         switch (cmd) {
551         case SIOCINQ:
552                 if (sk->sk_state == TCP_LISTEN)
553                         return -EINVAL;
554
555                 slow = lock_sock_fast(sk);
556                 if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))
557                         answ = 0;
558                 else if (sock_flag(sk, SOCK_URGINLINE) ||
559                          !tp->urg_data ||
560                          before(tp->urg_seq, tp->copied_seq) ||
561                          !before(tp->urg_seq, tp->rcv_nxt)) {
562
563                         answ = tp->rcv_nxt - tp->copied_seq;
564
565                         /* Subtract 1, if FIN was received */
566                         if (answ && sock_flag(sk, SOCK_DONE))
567                                 answ--;
568                 } else
569                         answ = tp->urg_seq - tp->copied_seq;
570                 unlock_sock_fast(sk, slow);
571                 break;
572         case SIOCATMARK:
573                 answ = tp->urg_data && tp->urg_seq == tp->copied_seq;
574                 break;
575         case SIOCOUTQ:
```

```
576                 if (sk->sk_state == TCP_LISTEN)
577                         return -EINVAL;
578
579                 if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))
580                         answ = 0;
581                 else
582                         answ = tp->write_seq - tp->snd_una;
583                 break;
584         case SIOCOUTQNSD:
585                 if (sk->sk_state == TCP_LISTEN)
586                         return -EINVAL;
587
588                 if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))
589                         answ = 0;
590                 else
591                         answ = tp->write_seq - tp->snd_nxt;
592                 break;
593         default:
594                 return -ENOIOCTLCMD;
595         }
596
597         return put_user(answ, (int __user *)arg);
598 }
599 EXPORT_SYMBOL(tcp_ioctl);
600
601 static inline void tcp_mark_push(struct tcp_sock *tp, struct sk_buff *skb)
602 {
603         TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_PSH;
604         tp->pushed_seq = tp->write_seq;
605 }
606
607 static inline bool forced_push(const struct tcp_sock *tp)
608 {
609         return after(tp->write_seq, tp->pushed_seq + (tp->max_window >> 1));
610 }
611
612 static inline void skb_entail(struct sock *sk, struct sk_buff *skb)
613 {
614         struct tcp_sock *tp = tcp_sk(sk);
615         struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);
616
617         skb->csum    = 0;
618         tcb->seq     = tcb->end_seq = tp->write_seq;
619         tcb->tcp_flags = TCPHDR_ACK;
620         tcb->sacked  = 0;
621         skb_header_release(skb);
622         tcp_add_write_queue_tail(sk, skb);
623         sk->sk_wmem_queued += skb->truesize;
624         sk_mem_charge(sk, skb->truesize);
625         if (tp->nonagle & TCP_NAGLE_PUSH)
626                 tp->nonagle &= ~TCP_NAGLE_PUSH;
627 }
628
629 static inline void tcp_mark_urg(struct tcp_sock *tp, int flags)
630 {
631         if (flags & MSG_OOB)
632                 tp->snd_up = tp->write_seq;
633 }
634
635 /* If a not yet filled skb is pushed, do not send it if
636  * we have data packets in Qdisc or NIC queues :
637  * Because TX completion will happen shortly, it gives a chance
638  * to coalesce future sendmsg() payload into this skb, without
639  * need for a timer, and with no latency trade off.
640  * As packets containing data payload have a bigger truesize
641  * than pure acks (dataless) packets, the last checks prevent
642  * autocorking if we only have an ACK in Qdisc/NIC queues,
643  * or if TX completion was delayed after we processed ACK packet.
644  */
645 static bool tcp_should_autocork(struct sock *sk, struct sk_buff *skb,
646                                 int size_goal)
647 {
648         return skb->len < size_goal &&
649                sysctl_tcp_autocorking &&
650                skb != tcp_write_queue_head(sk) &&
651                atomic_read(&sk->sk_wmem_alloc) > skb->truesize;
652 }
653
654 static void tcp_push(struct sock *sk, int flags, int mss_now,
655                      int nonagle, int size_goal)
656 {
657         struct tcp_sock *tp = tcp_sk(sk);
658         struct sk_buff *skb;
659
660         if (!tcp_send_head(sk))
```

```
661                 return;
662
663         skb = tcp_write_queue_tail(sk);
664         if (!(flags & MSG_MORE) || forced_push(tp))
665                 tcp_mark_push(tp, skb);
666
667         tcp_mark_urg(tp, flags);
668
669         if (tcp_should_autocork(sk, skb, size_goal)) {
670
671                 /* avoid atomic op if TSQ_THROTTLED bit is already set */
672                 if (!test_bit(TSQ_THROTTLED, &tp->tsq_flags)) {
673                         NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPAUTOCORKING);
674                         set_bit(TSQ_THROTTLED, &tp->tsq_flags);
675                 }
676                 /* It is possible TX completion already happened
677                  * before we set TSQ_THROTTLED.
678                  */
679                 if (atomic_read(&sk->sk_wmem_alloc) > skb->truesize)
680                         return;
681         }
682
683         if (flags & MSG_MORE)
684                 nonagle = TCP_NAGLE_CORK;
685
686         __tcp_push_pending_frames(sk, mss_now, nonagle);
687 }
688
689 static int tcp_splice_data_recv(read_descriptor_t *rd_desc, struct sk_buff *skb,
690                                 unsigned int offset, size_t len)
691 {
692         struct tcp_splice_state *tss = rd_desc->arg.data;
693         int ret;
694
695         ret = skb_splice_bits(skb, offset, tss->pipe, min(rd_desc->count, len),
696                               tss->flags);
697         if (ret > 0)
698                 rd_desc->count -= ret;
699         return ret;
700 }
701
702 static int __tcp_splice_read(struct sock *sk, struct tcp_splice_state *tss)
703 {
704         /* Store TCP splice context information in read_descriptor_t. */
705         read_descriptor_t rd_desc = {
706                 .arg.data = tss,
707                 .count    = tss->len,
708         };
709
710         return tcp_read_sock(sk, &rd_desc, tcp_splice_data_recv);
711 }
712
713 /**
714  *  tcp_splice_read - splice data from TCP socket to a pipe
715  * @sock:       socket to splice from
716  * @ppos:       position (not valid)
717  * @pipe:       pipe to splice to
718  * @len:        number of bytes to splice
719  * @flags:      splice modifier flags
720  *
721  * Description:
722  *    Will read pages from given socket and fill them into a pipe.
723  *
724  **/
725 ssize_t tcp_splice_read(struct socket *sock, loff_t *ppos,
726                         struct pipe_inode_info *pipe, size_t len,
727                         unsigned int flags)
728 {
729         struct sock *sk = sock->sk;
730         struct tcp_splice_state tss = {
731                 .pipe = pipe,
732                 .len = len,
733                 .flags = flags,
734         };
735         long timeo;
736         ssize_t spliced;
737         int ret;
738
739         sock_rps_record_flow(sk);
740         /*
741          * We can't seek on a socket input
742          */
743         if (unlikely(*ppos))
744                 return -ESPIPE;
745
```

```
746         ret = spliced = 0;
747
748         lock_sock(sk);
749
750         timeo = sock_rcvtimeo(sk, sock->file->f_flags & O_NONBLOCK);
751         while (tss.len) {
752                 ret = __tcp_splice_read(sk, &tss);
753                 if (ret < 0)
754                         break;
755                 else if (!ret) {
756                         if (spliced)
757                                 break;
758                         if (sock_flag(sk, SOCK_DONE))
759                                 break;
760                         if (sk->sk_err) {
761                                 ret = sock_error(sk);
762                                 break;
763                         }
764                         if (sk->sk_shutdown & RCV_SHUTDOWN)
765                                 break;
766                         if (sk->sk_state == TCP_CLOSE) {
767                                 /*
768                                  * This occurs when user tries to read
769                                  * from never connected socket.
770                                  */
771                                 if (!sock_flag(sk, SOCK_DONE))
772                                         ret = -ENOTCONN;
773                                 break;
774                         }
775                         if (!timeo) {
776                                 ret = -EAGAIN;
777                                 break;
778                         }
779                         sk_wait_data(sk, &timeo);
780                         if (signal_pending(current)) {
781                                 ret = sock_intr_errno(timeo);
782                                 break;
783                         }
784                         continue;
785                 }
786                 tss.len -= ret;
787                 spliced += ret;
788
789                 if (!timeo)
790                         break;
791                 release_sock(sk);
792                 lock_sock(sk);
793
794                 if (sk->sk_err || sk->sk_state == TCP_CLOSE ||
795                     (sk->sk_shutdown & RCV_SHUTDOWN) ||
796                     signal_pending(current))
797                         break;
798         }
799
800         release_sock(sk);
801
802         if (spliced)
803                 return spliced;
804
805         return ret;
806 }
807 EXPORT_SYMBOL(tcp_splice_read);
808
809 struct sk_buff *sk_stream_alloc_skb(struct sock *sk, int size, gfp_t gfp)
810 {
811         struct sk_buff *skb;
812
813         /* The TCP header must be at least 32-bit aligned.  */
814         size = ALIGN(size, 4);
815
816         skb = alloc_skb_fclone(size + sk->sk_prot->max_header, gfp);
817         if (skb) {
818                 if (sk_wmem_schedule(sk, skb->truesize)) {
819                         skb_reserve(skb, sk->sk_prot->max_header);
820                         /*
821                          * Make sure that we have exactly size bytes
822                          * available to the caller, no more, no less.
823                          */
824                         skb->reserved_tailroom = skb->end - skb->tail - size;
825                         return skb;
826                 }
827                 __kfree_skb(skb);
828         } else {
829                 sk->sk_prot->enter_memory_pressure(sk);
830                 sk_stream_moderate_sndbuf(sk);
```

```
831         }
832         return NULL;
833 }
834
835 static unsigned int tcp_xmit_size_goal(struct sock *sk, u32 mss_now,
836                                        int large_allowed)
837 {
838         struct tcp_sock *tp = tcp_sk(sk);
839         u32 xmit_size_goal, old_size_goal;
840
841         xmit_size_goal = mss_now;
842
843         if (large_allowed && sk_can_gso(sk)) {
844                 u32 gso_size, hlen;
845
846                 /* Maybe we should/could use sk->sk_prot->max_header here ? */
847                 hlen = inet_csk(sk)->icsk_af_ops->net_header_len +
848                        inet_csk(sk)->icsk_ext_hdr_len +
849                        tp->tcp_header_len;
850
851                 /* Goal is to send at least one packet per ms,
852                  * not one big TSO packet every 100 ms.
853                  * This preserves ACK clocking and is consistent
854                  * with tcp_tso_should_defer() heuristic.
855                  */
856                 gso_size = sk->sk_pacing_rate / (2 * MSEC_PER_SEC);
857                 gso_size = max_t(u32, gso_size,
858                                  sysctl_tcp_min_tso_segs * mss_now);
859
860                 xmit_size_goal = min_t(u32, gso_size,
861                                        sk->sk_gso_max_size - 1 - hlen);
862
863                 xmit_size_goal = tcp_bound_to_half_wnd(tp, xmit_size_goal);
864
865                 /* We try hard to avoid divides here */
866                 old_size_goal = tp->xmit_size_goal_segs * mss_now;
867
868                 if (likely(old_size_goal <= xmit_size_goal &&
869                            old_size_goal + mss_now > xmit_size_goal)) {
870                         xmit_size_goal = old_size_goal;
871                 } else {
872                         tp->xmit_size_goal_segs =
873                                 min_t(u16, xmit_size_goal / mss_now,
874                                       sk->sk_gso_max_segs);
875                         xmit_size_goal = tp->xmit_size_goal_segs * mss_now;
876                 }
877         }
878
879         return max(xmit_size_goal, mss_now);
880 }
881
882 static int tcp_send_mss(struct sock *sk, int *size_goal, int flags)
883 {
884         int mss_now;
885
886         mss_now = tcp_current_mss(sk);
887         *size_goal = tcp_xmit_size_goal(sk, mss_now, !(flags & MSG_OOB));
888
889         return mss_now;
890 }
891
892 static ssize_t do_tcp_sendpages(struct sock *sk, struct page *page, int offset,
893                                 size_t size, int flags)
894 {
895         struct tcp_sock *tp = tcp_sk(sk);
896         int mss_now, size_goal;
897         int err;
898         ssize_t copied;
899         long timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
900
901         /* Wait for a connection to finish. One exception is TCP Fast Open
902          * (passive side) where data is allowed to be sent before a connection
903          * is fully established.
904          */
905         if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
906             !tcp_passive_fastopen(sk)) {
907                 if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
908                         goto out_err;
909         }
910
911         clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
912
913         mss_now = tcp_send_mss(sk, &size_goal, flags);
914         copied = 0;
915
```

```
916         err = -EPIPE;
917         if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
918                 goto out_err;
919
920         while (size > 0) {
921                 struct sk_buff *skb = tcp_write_queue_tail(sk);
922                 int copy, i;
923                 bool can_coalesce;
924
925                 if (!tcp_send_head(sk) || (copy = size_goal - skb->len) <= 0) {
926 new_segment:
927                         if (!sk_stream_memory_free(sk))
928                                 goto wait_for_sndbuf;
929
930                         skb = sk_stream_alloc_skb(sk, 0, sk->sk_allocation);
931                         if (!skb)
932                                 goto wait_for_memory;
933
934                         skb_entail(sk, skb);
935                         copy = size_goal;
936                 }
937
938                 if (copy > size)
939                         copy = size;
940
941                 i = skb_shinfo(skb)->nr_frags;
942                 can_coalesce = skb_can_coalesce(skb, i, page, offset);
943                 if (!can_coalesce && i >= MAX_SKB_FRAGS) {
944                         tcp_mark_push(tp, skb);
945                         goto new_segment;
946                 }
947                 if (!sk_wmem_schedule(sk, copy))
948                         goto wait_for_memory;
949
950                 if (can_coalesce) {
951                         skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
952                 } else {
953                         get_page(page);
954                         skb_fill_page_desc(skb, i, page, offset, copy);
955                 }
956                 skb_shinfo(skb)->tx_flags |= SKBTX_SHARED_FRAG;
957
958                 skb->len += copy;
959                 skb->data_len += copy;
960                 skb->truesize += copy;
961                 sk->sk_wmem_queued += copy;
962                 sk_mem_charge(sk, copy);
963                 skb->ip_summed = CHECKSUM_PARTIAL;
964                 tp->write_seq += copy;
965                 TCP_SKB_CB(skb)->end_seq += copy;
966                 skb_shinfo(skb)->gso_segs = 0;
967
968                 if (!copied)
969                         TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;
970
971                 copied += copy;
972                 offset += copy;
973                 if (!(size -= copy)) {
974                         tcp_tx_timestamp(sk, skb);
975                         goto out;
976                 }
977
978                 if (skb->len < size_goal || (flags & MSG_OOB))
979                         continue;
980
981                 if (forced_push(tp)) {
982                         tcp_mark_push(tp, skb);
983                         __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
984                 } else if (skb == tcp_send_head(sk))
985                         tcp_push_one(sk, mss_now);
986                 continue;
987
988 wait_for_sndbuf:
989                 set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
990 wait_for_memory:
991                 tcp_push(sk, flags & ~MSG_MORE, mss_now,
992                          TCP_NAGLE_PUSH, size_goal);
993
994                 if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
995                         goto do_error;
996
997                 mss_now = tcp_send_mss(sk, &size_goal, flags);
998         }
999
1000 out:
```

```
1001         if (copied && !(flags & MSG_SENDPAGE_NOTLAST))
1002                 tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
1003         return copied;
1004
1005 do_error:
1006         if (copied)
1007                 goto out;
1008 out_err:
1009         return sk_stream_error(sk, flags, err);
1010 }
1011
1012 int tcp_sendpage(struct sock *sk, struct page *page, int offset,
1013                  size_t size, int flags)
1014 {
1015         ssize_t res;
1016
1017         if (!(sk->sk_route_caps & NETIF_F_SG) ||
1018             !(sk->sk_route_caps & NETIF_F_ALL_CSUM))
1019                 return sock_no_sendpage(sk->sk_socket, page, offset, size,
1020                                         flags);
1021
1022         lock_sock(sk);
1023         res = do_tcp_sendpages(sk, page, offset, size, flags);
1024         release_sock(sk);
1025         return res;
1026 }
1027 EXPORT_SYMBOL(tcp_sendpage);
1028
1029 static inline int select_size(const struct sock *sk, bool sg)
1030 {
1031         const struct tcp_sock *tp = tcp_sk(sk);
1032         int tmp = tp->mss_cache;
1033
1034         if (sg) {
1035                 if (sk_can_gso(sk)) {
1036                         /* Small frames wont use a full page:
1037                          * Payload will immediately follow tcp header.
1038                          */
1039                         tmp = SKB_WITH_OVERHEAD(2048 - MAX_TCP_HEADER);
1040                 } else {
1041                         int pgbreak = SKB_MAX_HEAD(MAX_TCP_HEADER);
1042
1043                         if (tmp >= pgbreak &&
1044                             tmp <= pgbreak + (MAX_SKB_FRAGS - 1) * PAGE_SIZE)
1045                                 tmp = pgbreak;
1046                 }
1047         }
1048
1049         return tmp;
1050 }
1051
1052 void tcp_free_fastopen_req(struct tcp_sock *tp)
1053 {
1054         if (tp->fastopen_req != NULL) {
1055                 kfree(tp->fastopen_req);
1056                 tp->fastopen_req = NULL;
1057         }
1058 }
1059
1060 static int tcp_sendmsg_fastopen(struct sock *sk, struct msghdr *msg,
1061                                 int *copied, size_t size)
1062 {
1063         struct tcp_sock *tp = tcp_sk(sk);
1064         int err, flags;
1065
1066         if (!(sysctl_tcp_fastopen & TFO_CLIENT_ENABLE))
1067                 return -EOPNOTSUPP;
1068         if (tp->fastopen_req != NULL)
1069                 return -EALREADY; /* Another Fast Open is in progress */
1070
1071         tp->fastopen_req = kzalloc(sizeof(struct tcp_fastopen_request),
1072                                    sk->sk_allocation);
1073         if (unlikely(tp->fastopen_req == NULL))
1074                 return -ENOBUFS;
1075         tp->fastopen_req->data = msg;
1076         tp->fastopen_req->size = size;
1077
1078         flags = (msg->msg_flags & MSG_DONTWAIT) ? O_NONBLOCK : 0;
1079         err = __inet_stream_connect(sk->sk_socket, msg->msg_name,
1080                                     msg->msg_namelen, flags);
1081         *copied = tp->fastopen_req->copied;
1082         tcp_free_fastopen_req(tp);
1083         return err;
1084 }
1085
```

```
1086 int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
1087                 size_t size)
1088 {
1089         struct iovec *iov;
1090         struct tcp_sock *tp = tcp_sk(sk);
1091         struct sk_buff *skb;
1092         int iovlen, flags, err, copied = 0;
1093         int mss_now = 0, size_goal, copied_syn = 0, offset = 0;
1094         bool sg;
1095         long timeo;
1096
1097         lock_sock(sk);
1098
1099         flags = msg->msg_flags;
1100         if (flags & MSG_FASTOPEN) {
1101                 err = tcp_sendmsg_fastopen(sk, msg, &copied_syn, size);
1102                 if (err == -EINPROGRESS && copied_syn > 0)
1103                         goto out;
1104                 else if (err)
1105                         goto out_err;
1106                 offset = copied_syn;
1107         }
1108
1109         timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
1110
1111         /* Wait for a connection to finish. One exception is TCP Fast Open
1112          * (passive side) where data is allowed to be sent before a connection
1113          * is fully established.
1114          */
1115         if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
1116             !tcp_passive_fastopen(sk)) {
1117                 if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
1118                         goto do_error;
1119         }
1120
1121         if (unlikely(tp->repair)) {
1122                 if (tp->repair_queue == TCP_RECV_QUEUE) {
1123                         copied = tcp_send_rcvq(sk, msg, size);
1124                         goto out_nopush;
1125                 }
1126
1127                 err = -EINVAL;
1128                 if (tp->repair_queue == TCP_NO_QUEUE)
1129                         goto out_err;
1130
1131                 /* 'common' sending to sendq */
1132         }
1133
1134         /* This should be in poll */
1135         clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
1136
1137         mss_now = tcp_send_mss(sk, &size_goal, flags);
1138
1139         /* Ok commence sending. */
1140         iovlen = msg->msg_iovlen;
1141         iov = msg->msg_iov;
1142         copied = 0;
1143
1144         err = -EPIPE;
1145         if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
1146                 goto out_err;
1147
1148         sg = !!(sk->sk_route_caps & NETIF_F_SG);
1149
1150         while (--iovlen >= 0) {
1151                 size_t seglen = iov->iov_len;
1152                 unsigned char __user *from = iov->iov_base;
1153
1154                 iov++;
1155                 if (unlikely(offset > 0)) {  /* Skip bytes copied in SYN */
1156                         if (offset >= seglen) {
1157                                 offset -= seglen;
1158                                 continue;
1159                         }
1160                         seglen -= offset;
1161                         from += offset;
1162                         offset = 0;
1163                 }
1164
1165                 while (seglen > 0) {
1166                         int copy = 0;
1167                         int max = size_goal;
1168
1169                         skb = tcp_write_queue_tail(sk);
1170                         if (tcp_send_head(sk)) {
```

```
1171                            if (skb->ip_summed == CHECKSUM_NONE)
1172                                    max = mss_now;
1173                            copy = max - skb->len;
1174                    }
1175
1176                    if (copy <= 0) {
1177 new_segment:
1178                            /* Allocate new segment. If the interface is SG,
1179                             * allocate skb fitting to single page.
1180                             */
1181                            if (!sk_stream_memory_free(sk))
1182                                    goto wait_for_sndbuf;
1183
1184                            skb = sk_stream_alloc_skb(sk,
1185                                                      select_size(sk, sg),
1186                                                      sk->sk_allocation);
1187                            if (!skb)
1188                                    goto wait_for_memory;
1189
1190                            /*
1191                             * Check whether we can use HW checksum.
1192                             */
1193                            if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
1194                                    skb->ip_summed = CHECKSUM_PARTIAL;
1195
1196                            skb_entail(sk, skb);
1197                            copy = size_goal;
1198                            max = size_goal;
1199
1200                            /* All packets are restored as if they have
1201                             * already been sent. skb_mstamp isn't set to
1202                             * avoid wrong rtt estimation.
1203                             */
1204                            if (tp->repair)
1205                                    TCP_SKB_CB(skb)->sacked |= TCPCB_REPAIRED;
1206                    }
1207
1208                    /* Try to append data to the end of skb. */
1209                    if (copy > seglen)
1210                            copy = seglen;
1211
1212                    /* Where to copy to? */
1213                    if (skb_availroom(skb) > 0) {
1214                            /* We have some space in skb head. Superb! */
1215                            copy = min_t(int, copy, skb_availroom(skb));
1216                            err = skb_add_data_nocache(sk, skb, from, copy);
1217                            if (err)
1218                                    goto do_fault;
1219                    } else {
1220                            bool merge = true;
1221                            int i = skb_shinfo(skb)->nr_frags;
1222                            struct page_frag *pfrag = sk_page_frag(sk);
1223
1224                            if (!sk_page_frag_refill(sk, pfrag))
1225                                    goto wait_for_memory;
1226
1227                            if (!skb_can_coalesce(skb, i, pfrag->page,
1228                                                  pfrag->offset)) {
1229                                    if (i == MAX_SKB_FRAGS || !sg) {
1230                                            tcp_mark_push(tp, skb);
1231                                            goto new_segment;
1232                                    }
1233                                    merge = false;
1234                            }
1235
1236                            copy = min_t(int, copy, pfrag->size - pfrag->offset);
1237
1238                            if (!sk_wmem_schedule(sk, copy))
1239                                    goto wait_for_memory;
1240
1241                            err = skb_copy_to_page_nocache(sk, from, skb,
1242                                                           pfrag->page,
1243                                                           pfrag->offset,
1244                                                           copy);
1245                            if (err)
1246                                    goto do_error;
1247
1248                            /* Update the skb. */
1249                            if (merge) {
1250                                    skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
1251                            } else {
1252                                    skb_fill_page_desc(skb, i, pfrag->page,
1253                                                       pfrag->offset, copy);
1254                                    get_page(pfrag->page);
1255                            }
```

```
1256                            pfrag->offset += copy;
1257                    }
1258
1259                    if (!copied)
1260                            TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;
1261
1262                    tp->write_seq += copy;
1263                    TCP_SKB_CB(skb)->end_seq += copy;
1264                    skb_shinfo(skb)->gso_segs = 0;
1265
1266                    from += copy;
1267                    copied += copy;
1268                    if ((seglen -= copy) == 0 && iovlen == 0) {
1269                            tcp_tx_timestamp(sk, skb);
1270                            goto out;
1271                    }
1272
1273                    if (skb->len < max || (flags & MSG_OOB) || unlikely(tp->repair))
1274                            continue;
1275
1276                    if (forced_push(tp)) {
1277                            tcp_mark_push(tp, skb);
1278                            __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
1279                    } else if (skb == tcp_send_head(sk))
1280                            tcp_push_one(sk, mss_now);
1281                    continue;
1282
1283 wait_for_sndbuf:
1284                    set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
1285 wait_for_memory:
1286                    if (copied)
1287                            tcp_push(sk, flags & ~MSG_MORE, mss_now,
1288                                     TCP_NAGLE_PUSH, size_goal);
1289
1290                    if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
1291                            goto do_error;
1292
1293                    mss_now = tcp_send_mss(sk, &size_goal, flags);
1294            }
1295    }
1296
1297 out:
1298    if (copied)
1299            tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
1300 out_nopush:
1301    release_sock(sk);
1302    return copied + copied_syn;
1303
1304 do_fault:
1305    if (!skb->len) {
1306            tcp_unlink_write_queue(skb, sk);
1307            /* It is the one place in all of TCP, except connection
1308             * reset, where we can be unlinking the send_head.
1309             */
1310            tcp_check_send_head(sk, skb);
1311            sk_wmem_free_skb(sk, skb);
1312    }
1313
1314 do_error:
1315    if (copied + copied_syn)
1316            goto out;
1317 out_err:
1318    err = sk_stream_error(sk, flags, err);
1319    release_sock(sk);
1320    return err;
1321 }
1322 EXPORT_SYMBOL(tcp_sendmsg);
1323
1324 /*
1325  *      Handle reading urgent data. BSD has very simple semantics for
1326  *      this, no blocking and very strange errors 8)
1327  */
1328
1329 static int tcp_recv_urg(struct sock *sk, struct msghdr *msg, int len, int flags)
1330 {
1331    struct tcp_sock *tp = tcp_sk(sk);
1332
1333    /* No URG data to read. */
1334    if (sock_flag(sk, SOCK_URGINLINE) || !tp->urg_data ||
1335        tp->urg_data == TCP_URG_READ)
1336            return -EINVAL; /* Yes this is right ! */
1337
1338    if (sk->sk_state == TCP_CLOSE && !sock_flag(sk, SOCK_DONE))
1339            return -ENOTCONN;
1340
```

```
1341         if (tp->urg_data & TCP_URG_VALID) {
1342                 int err = 0;
1343                 char c = tp->urg_data;
1344
1345                 if (!(flags & MSG_PEEK))
1346                         tp->urg_data = TCP_URG_READ;
1347
1348                 /* Read urgent data. */
1349                 msg->msg_flags |= MSG_OOB;
1350
1351                 if (len > 0) {
1352                         if (!(flags & MSG_TRUNC))
1353                                 err = memcpy_toiovec(msg->msg_iov, &c, 1);
1354                         len = 1;
1355                 } else
1356                         msg->msg_flags |= MSG_TRUNC;
1357
1358                 return err ? -EFAULT : len;
1359         }
1360
1361         if (sk->sk_state == TCP_CLOSE || (sk->sk_shutdown & RCV_SHUTDOWN))
1362                 return 0;
1363
1364         /* Fixed the recv(..., MSG_OOB) behaviour.  BSD docs and
1365          * the available implementations agree in this case:
1366          * this call should never block, independent of the
1367          * blocking state of the socket.
1368          * Mike <pall@rz.uni-karlsruhe.de>
1369          */
1370         return -EAGAIN;
1371 }
1372
1373 static int tcp_peek_sndq(struct sock *sk, struct msghdr *msg, int len)
1374 {
1375         struct sk_buff *skb;
1376         int copied = 0, err = 0;
1377
1378         /* XXX -- need to support SO_PEEK_OFF */
1379
1380         skb_queue_walk(&sk->sk_write_queue, skb) {
1381                 err = skb_copy_datagram_iovec(skb, 0, msg->msg_iov, skb->len);
1382                 if (err)
1383                         break;
1384
1385                 copied += skb->len;
1386         }
1387
1388         return err ?: copied;
1389 }
1390
1391 /* Clean up the receive buffer for full frames taken by the user,
1392  * then send an ACK if necessary.  COPIED is the number of bytes
1393  * tcp_recvmsg has given to the user so far, it speeds up the
1394  * calculation of whether or not we must ACK for the sake of
1395  * a window update.
1396  */
1397 void tcp_cleanup_rbuf(struct sock *sk, int copied)
1398 {
1399         struct tcp_sock *tp = tcp_sk(sk);
1400         bool time_to_ack = false;
1401
1402         struct sk_buff *skb = skb_peek(&sk->sk_receive_queue);
1403
1404         WARN(skb && !before(tp->copied_seq, TCP_SKB_CB(skb)->end_seq),
1405              "cleanup rbuf bug: copied %X seq %X rcvnxt %X\n",
1406              tp->copied_seq, TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt);
1407
1408         if (inet_csk_ack_scheduled(sk)) {
1409                 const struct inet_connection_sock *icsk = inet_csk(sk);
1410                    /* Delayed ACKs frequently hit locked sockets during bulk
1411                     * receive. */
1412                 if (icsk->icsk_ack.blocked ||
1413                     /* Once-per-two-segments ACK was not sent by tcp_input.c */
1414                     tp->rcv_nxt - tp->rcv_wup > icsk->icsk_ack.rcv_mss ||
1415                     /*
1416                      * If this read emptied read buffer, we send ACK, if
1417                      * connection is not bidirectional, user drained
1418                      * receive buffer and there was a small segment
1419                      * in queue.
1420                      */
1421                     (copied > 0 &&
1422                      ((icsk->icsk_ack.pending & ICSK_ACK_PUSHED2) ||
1423                       ((icsk->icsk_ack.pending & ICSK_ACK_PUSHED) &&
1424                        !icsk->icsk_ack.pingpong)) &&
1425                       !atomic_read(&sk->sk_rmem_alloc)))
```

```
1426                        time_to_ack = true;
1427        }
1428
1429        /* We send an ACK if we can now advertise a non-zero window
1430         * which has been raised "significantly".
1431         *
1432         * Even if window raised up to infinity, do not send window open ACK
1433         * in states, where we will not receive more. It is useless.
1434         */
1435        if (copied > 0 && !time_to_ack && !(sk->sk_shutdown & RCV_SHUTDOWN)) {
1436                __u32 rcv_window_now = tcp_receive_window(tp);
1437
1438                /* Optimize, __tcp_select_window() is not cheap. */
1439                if (2*rcv_window_now <= tp->window_clamp) {
1440                        __u32 new_window = __tcp_select_window(sk);
1441
1442                        /* Send ACK now, if this read freed lots of space
1443                         * in our buffer. Certainly, new_window is new window.
1444                         * We can advertise it now, if it is not less than current one.
1445                         * "Lots" means "at least twice" here.
1446                         */
1447                        if (new_window && new_window >= 2 * rcv_window_now)
1448                                time_to_ack = true;
1449                }
1450        }
1451        if (time_to_ack)
1452                tcp_send_ack(sk);
1453 }
1454
1455 static void tcp_prequeue_process(struct sock *sk)
1456 {
1457        struct sk_buff *skb;
1458        struct tcp_sock *tp = tcp_sk(sk);
1459
1460        NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPPREQUEUED);
1461
1462        /* RX process wants to run with disabled BHs, though it is not
1463         * necessary */
1464        local_bh_disable();
1465        while ((skb = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)
1466                sk_backlog_rcv(sk, skb);
1467        local_bh_enable();
1468
1469        /* Clear memory counter. */
1470        tp->ucopy.memory = 0;
1471 }
1472
1473 #ifdef CONFIG_NET_DMA
1474 static void tcp_service_net_dma(struct sock *sk, bool wait)
1475 {
1476        dma_cookie_t done, used;
1477        dma_cookie_t last_issued;
1478        struct tcp_sock *tp = tcp_sk(sk);
1479
1480        if (!tp->ucopy.dma_chan)
1481                return;
1482
1483        last_issued = tp->ucopy.dma_cookie;
1484        dma_async_issue_pending(tp->ucopy.dma_chan);
1485
1486        do {
1487                if (dma_async_is_tx_complete(tp->ucopy.dma_chan,
1488                                             last_issued, &done,
1489                                             &used) == DMA_COMPLETE) {
1490                        /* Safe to free early-copied skbs now */
1491                        __skb_queue_purge(&sk->sk_async_wait_queue);
1492                        break;
1493                } else {
1494                        struct sk_buff *skb;
1495                        while ((skb = skb_peek(&sk->sk_async_wait_queue)) &&
1496                               (dma_async_is_complete(skb->dma_cookie, done,
1497                                                      used) == DMA_COMPLETE)) {
1498                                __skb_dequeue(&sk->sk_async_wait_queue);
1499                                kfree_skb(skb);
1500                        }
1501                }
1502        } while (wait);
1503 }
1504 #endif
1505
1506 static struct sk_buff *tcp_recv_skb(struct sock *sk, u32 seq, u32 *off)
1507 {
1508        struct sk_buff *skb;
1509        u32 offset;
1510
```

```
1511         while ((skb = skb_peek(&sk->sk_receive_queue)) != NULL) {
1512                 offset = seq - TCP_SKB_CB(skb)->seq;
1513                 if (tcp_hdr(skb)->syn)
1514                         offset--;
1515                 if (offset < skb->len || tcp_hdr(skb)->fin) {
1516                         *off = offset;
1517                         return skb;
1518                 }
1519                 /* This looks weird, but this can happen if TCP collapsing
1520                  * splitted a fat GRO packet, while we released socket lock
1521                  * in skb_splice_bits()
1522                  */
1523                 sk_eat_skb(sk, skb, false);
1524         }
1525         return NULL;
1526 }
1527
1528 /*
1529  * This routine provides an alternative to tcp_recvmsg() for routines
1530  * that would like to handle copying from skbuffs directly in 'sendfile'
1531  * fashion.
1532  * Note:
1533  *      - It is assumed that the socket was locked by the caller.
1534  *      - The routine does not block.
1535  *      - At present, there is no support for reading OOB data
1536  *        or for 'peeking' the socket using this routine
1537  *        (although both would be easy to implement).
1538  */
1539 int tcp_read_sock(struct sock *sk, read_descriptor_t *desc,
1540                   sk_read_actor_t recv_actor)
1541 {
1542         struct sk_buff *skb;
1543         struct tcp_sock *tp = tcp_sk(sk);
1544         u32 seq = tp->copied_seq;
1545         u32 offset;
1546         int copied = 0;
1547
1548         if (sk->sk_state == TCP_LISTEN)
1549                 return -ENOTCONN;
1550         while ((skb = tcp_recv_skb(sk, seq, &offset)) != NULL) {
1551                 if (offset < skb->len) {
1552                         int used;
1553                         size_t len;
1554
1555                         len = skb->len - offset;
1556                         /* Stop reading if we hit a patch of urgent data */
1557                         if (tp->urg_data) {
1558                                 u32 urg_offset = tp->urg_seq - seq;
1559                                 if (urg_offset < len)
1560                                         len = urg_offset;
1561                                 if (!len)
1562                                         break;
1563                         }
1564                         used = recv_actor(desc, skb, offset, len);
1565                         if (used <= 0) {
1566                                 if (!copied)
1567                                         copied = used;
1568                                 break;
1569                         } else if (used <= len) {
1570                                 seq += used;
1571                                 copied += used;
1572                                 offset += used;
1573                         }
1574                         /* If recv_actor drops the lock (e.g. TCP splice
1575                          * receive) the skb pointer might be invalid when
1576                          * getting here: tcp_collapse might have deleted it
1577                          * while aggregating skbs from the socket queue.
1578                          */
1579                         skb = tcp_recv_skb(sk, seq - 1, &offset);
1580                         if (!skb)
1581                                 break;
1582                         /* TCP coalescing might have appended data to the skb.
1583                          * Try to splice more frags
1584                          */
1585                         if (offset + 1 != skb->len)
1586                                 continue;
1587                 }
1588                 if (tcp_hdr(skb)->fin) {
1589                         sk_eat_skb(sk, skb, false);
1590                         ++seq;
1591                         break;
1592                 }
1593                 sk_eat_skb(sk, skb, false);
1594                 if (!desc->count)
1595                         break;
```

```
1596                    tp->copied_seq = seq;
1597            }
1598            tp->copied_seq = seq;
1599
1600            tcp_rcv_space_adjust(sk);
1601
1602            /* Clean up data we have read: This will do ACK frames. */
1603            if (copied > 0) {
1604                    tcp_recv_skb(sk, seq, &offset);
1605                    tcp_cleanup_rbuf(sk, copied);
1606            }
1607            return copied;
1608    }
1609    EXPORT_SYMBOL(tcp_read_sock);
1610
1611    /*
1612     *      This routine copies from a sock struct into the user buffer.
1613     *
1614     *      Technical note: in 2.3 we work on _locked_ socket, so that
1615     *      tricks with *seq access order and skb->users are not required.
1616     *      Probably, code can be easily improved even more.
1617     */
1618
1619    int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
1620                    size_t len, int nonblock, int flags, int *addr_len)
1621    {
1622            struct tcp_sock *tp = tcp_sk(sk);
1623            int copied = 0;
1624            u32 peek_seq;
1625            u32 *seq;
1626            unsigned long used;
1627            int err;
1628            int target;             /* Read at least this many bytes */
1629            long timeo;
1630            struct task_struct *user_recv = NULL;
1631            bool copied_early = false;
1632            struct sk_buff *skb;
1633            u32 urg_hole = 0;
1634
1635            if (unlikely(flags & MSG_ERRQUEUE))
1636                    return ip_recv_error(sk, msg, len, addr_len);
1637
1638            if (sk_can_busy_loop(sk) && skb_queue_empty(&sk->sk_receive_queue) &&
1639                (sk->sk_state == TCP_ESTABLISHED))
1640                sk_busy_loop(sk, nonblock);
1641
1642            lock_sock(sk);
1643
1644            err = -ENOTCONN;
1645            if (sk->sk_state == TCP_LISTEN)
1646                    goto out;
1647
1648            timeo = sock_rcvtimeo(sk, nonblock);
1649
1650            /* Urgent data needs to be handled specially. */
1651            if (flags & MSG_OOB)
1652                    goto recv_urg;
1653
1654            if (unlikely(tp->repair)) {
1655                    err = -EPERM;
1656                    if (!(flags & MSG_PEEK))
1657                            goto out;
1658
1659                    if (tp->repair_queue == TCP_SEND_QUEUE)
1660                            goto recv_sndq;
1661
1662                    err = -EINVAL;
1663                    if (tp->repair_queue == TCP_NO_QUEUE)
1664                            goto out;
1665
1666                    /* 'common' recv queue MSG_PEEK-ing */
1667            }
1668
1669            seq = &tp->copied_seq;
1670            if (flags & MSG_PEEK) {
1671                    peek_seq = tp->copied_seq;
1672                    seq = &peek_seq;
1673            }
1674
1675            target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
1676
1677    #ifdef CONFIG_NET_DMA
1678            tp->ucopy.dma_chan = NULL;
1679            preempt_disable();
1680            skb = skb_peek_tail(&sk->sk_receive_queue);
```

```
1681        {
1682                int available = 0;
1683
1684                if (skb)
1685                        available = TCP_SKB_CB(skb)->seq + skb->len - (*seq);
1686                if ((available < target) &&
1687                    (len > sysctl_tcp_dma_copybreak) && !(flags & MSG_PEEK) &&
1688                    !sysctl_tcp_low_latency &&
1689                    net_dma_find_channel()) {
1690                        preempt_enable();
1691                        tp->ucopy.pinned_list =
1692                                        dma_pin_iovec_pages(msg->msg_iov, len);
1693                } else {
1694                        preempt_enable();
1695                }
1696        }
1697 #endif
1698
1699        do {
1700                u32 offset;
1701
1702                /* Are we at urgent data? Stop if we have read anything or have SIGURG pending. */
1703                if (tp->urg_data && tp->urg_seq == *seq) {
1704                        if (copied)
1705                                break;
1706                        if (signal_pending(current)) {
1707                                copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
1708                                break;
1709                        }
1710                }
1711
1712                /* Next get a buffer. */
1713
1714                skb_queue_walk(&sk->sk_receive_queue, skb) {
1715                        /* Now that we have two receive queues this
1716                         * shouldn't happen.
1717                         */
1718                        if (WARN(before(*seq, TCP_SKB_CB(skb)->seq),
1719                                 "recvmsg bug: copied %X seq %X rcvnxt %X fl %X\n",
1720                                 *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt,
1721                                 flags))
1722                                break;
1723
1724                        offset = *seq - TCP_SKB_CB(skb)->seq;
1725                        if (tcp_hdr(skb)->syn)
1726                                offset--;
1727                        if (offset < skb->len)
1728                                goto found_ok_skb;
1729                        if (tcp_hdr(skb)->fin)
1730                                goto found_fin_ok;
1731                        WARN(!(flags & MSG_PEEK),
1732                             "recvmsg bug 2: copied %X seq %X rcvnxt %X fl %X\n",
1733                             *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt, flags);
1734                }
1735
1736                /* Well, if we have backlog, try to process it now yet. */
1737
1738                if (copied >= target && !sk->sk_backlog.tail)
1739                        break;
1740
1741                if (copied) {
1742                        if (sk->sk_err ||
1743                            sk->sk_state == TCP_CLOSE ||
1744                            (sk->sk_shutdown & RCV_SHUTDOWN) ||
1745                            !timeo ||
1746                            signal_pending(current))
1747                                break;
1748                } else {
1749                        if (sock_flag(sk, SOCK_DONE))
1750                                break;
1751
1752                        if (sk->sk_err) {
1753                                copied = sock_error(sk);
1754                                break;
1755                        }
1756
1757                        if (sk->sk_shutdown & RCV_SHUTDOWN)
1758                                break;
1759
1760                        if (sk->sk_state == TCP_CLOSE) {
1761                                if (!sock_flag(sk, SOCK_DONE)) {
1762                                        /* This occurs when user tries to read
1763                                         * from never connected socket.
1764                                         */
1765                                        copied = -ENOTCONN;
```

```
1766                                    break;
1767                            }
1768                            break;
1769                    }
1770
1771                    if (!timeo) {
1772                            copied = -EAGAIN;
1773                            break;
1774                    }
1775
1776                    if (signal_pending(current)) {
1777                            copied = sock_intr_errno(timeo);
1778                            break;
1779                    }
1780            }
1781
1782            tcp_cleanup_rbuf(sk, copied);
1783
1784            if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
1785                    /* Install new reader */
1786                    if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
1787                            user_recv = current;
1788                            tp->ucopy.task = user_recv;
1789                            tp->ucopy.iov = msg->msg_iov;
1790                    }
1791
1792                    tp->ucopy.len = len;
1793
1794                    WARN_ON(tp->copied_seq != tp->rcv_nxt &&
1795                            !(flags & (MSG_PEEK | MSG_TRUNC)));
1796
1797                    /* Ugly... If prequeue is not empty, we have to
1798                     * process it before releasing socket, otherwise
1799                     * order will be broken at second iteration.
1800                     * More elegant solution is required!!!
1801                     *
1802                     * Look: we have the following (pseudo)queues:
1803                     *
1804                     * 1. packets in flight
1805                     * 2. backlog
1806                     * 3. prequeue
1807                     * 4. receive_queue
1808                     *
1809                     * Each queue can be processed only if the next ones
1810                     * are empty. At this point we have empty receive_queue.
1811                     * But prequeue _can_ be not empty after 2nd iteration,
1812                     * when we jumped to start of loop because backlog
1813                     * processing added something to receive_queue.
1814                     * We cannot release_sock(), because backlog contains
1815                     * packets arrived _after_ prequeued ones.
1816                     *
1817                     * Shortly, algorithm is clear --- to process all
1818                     * the queues in order. We could make it more directly,
1819                     * requeueing packets from backlog to prequeue, if
1820                     * is not empty. It is more elegant, but eats cycles,
1821                     * unfortunately.
1822                     */
1823                    if (!skb_queue_empty(&tp->ucopy.prequeue))
1824                            goto do_prequeue;
1825
1826                    /* __ Set realtime policy in scheduler __ */
1827            }
1828
1829 #ifdef CONFIG_NET_DMA
1830            if (tp->ucopy.dma_chan) {
1831                    if (tp->rcv_wnd == 0 &&
1832                        !skb_queue_empty(&sk->sk_async_wait_queue)) {
1833                            tcp_service_net_dma(sk, true);
1834                            tcp_cleanup_rbuf(sk, copied);
1835                    } else
1836                            dma_async_issue_pending(tp->ucopy.dma_chan);
1837            }
1838 #endif
1839            if (copied >= target) {
1840                    /* Do not sleep, just process backlog. */
1841                    release_sock(sk);
1842                    lock_sock(sk);
1843            } else
1844                    sk_wait_data(sk, &timeo);
1845
1846 #ifdef CONFIG_NET_DMA
1847            tcp_service_net_dma(sk, false);  /* Don't block */
1848            tp->ucopy.wakeup = 0;
1849 #endif
1850
```

```
1851                if (user_recv) {
1852                        int chunk;
1853
1854                        /* __ Restore normal policy in scheduler __ */
1855
1856                        if ((chunk = len - tp->ucopy.len) != 0) {
1857                                NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, chunk);
1858                                len -= chunk;
1859                                copied += chunk;
1860                        }
1861
1862                        if (tp->rcv_nxt == tp->copied_seq &&
1863                            !skb_queue_empty(&tp->ucopy.prequeue)) {
1864 do_prequeue:
1865                                tcp_prequeue_process(sk);
1866
1867                                if ((chunk = len - tp->ucopy.len) != 0) {
1868                                        NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
1869                                        len -= chunk;
1870                                        copied += chunk;
1871                                }
1872                        }
1873                }
1874                if ((flags & MSG_PEEK) &&
1875                    (peek_seq - copied - urg_hole != tp->copied_seq)) {
1876                        net_dbg_ratelimited("TCP(%s:%d): Application bug, race in MSG_PEEK\n",
1877                                            current->comm,
1878                                            task_pid_nr(current));
1879                        peek_seq = tp->copied_seq;
1880                }
1881                continue;
1882
1883        found_ok_skb:
1884                /* Ok so how much can we use? */
1885                used = skb->len - offset;
1886                if (len < used)
1887                        used = len;
1888
1889                /* Do we have urgent data here? */
1890                if (tp->urg_data) {
1891                        u32 urg_offset = tp->urg_seq - *seq;
1892                        if (urg_offset < used) {
1893                                if (!urg_offset) {
1894                                        if (!sock_flag(sk, SOCK_URGINLINE)) {
1895                                                ++*seq;
1896                                                urg_hole++;
1897                                                offset++;
1898                                                used--;
1899                                                if (!used)
1900                                                        goto skip_copy;
1901                                        }
1902                                } else
1903                                        used = urg_offset;
1904                        }
1905                }
1906
1907                if (!(flags & MSG_TRUNC)) {
1908 #ifdef CONFIG_NET_DMA
1909                        if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
1910                                tp->ucopy.dma_chan = net_dma_find_channel();
1911
1912                        if (tp->ucopy.dma_chan) {
1913                                tp->ucopy.dma_cookie = dma_skb_copy_datagram_iovec(
1914                                        tp->ucopy.dma_chan, skb, offset,
1915                                        msg->msg_iov, used,
1916                                        tp->ucopy.pinned_list);
1917
1918                                if (tp->ucopy.dma_cookie < 0) {
1919
1920                                        pr_alert("%s: dma_cookie < 0\n",
1921                                                 __func__);
1922
1923                                        /* Exception. Bailout! */
1924                                        if (!copied)
1925                                                copied = -EFAULT;
1926                                        break;
1927                                }
1928
1929                                dma_async_issue_pending(tp->ucopy.dma_chan);
1930
1931                                if ((offset + used) == skb->len)
1932                                        copied_early = true;
1933
1934                        } else
1935 #endif
```

```
1936                        {
1937                                err = skb_copy_datagram_iovec(skb, offset,
1938                                                msg->msg_iov, used);
1939                                if (err) {
1940                                        /* Exception. Bailout! */
1941                                        if (!copied)
1942                                                copied = -EFAULT;
1943                                        break;
1944                                }
1945                        }
1946                }
1947
1948                *seq += used;
1949                copied += used;
1950                len -= used;
1951
1952                tcp_rcv_space_adjust(sk);
1953
1954 skip_copy:
1955                if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
1956                        tp->urg_data = 0;
1957                        tcp_fast_path_check(sk);
1958                }
1959                if (used + offset < skb->len)
1960                        continue;
1961
1962                if (tcp_hdr(skb)->fin)
1963                        goto found_fin_ok;
1964                if (!(flags & MSG_PEEK)) {
1965                        sk_eat_skb(sk, skb, copied_early);
1966                        copied_early = false;
1967                }
1968                continue;
1969
1970        found_fin_ok:
1971                /* Process the FIN. */
1972                ++*seq;
1973                if (!(flags & MSG_PEEK)) {
1974                        sk_eat_skb(sk, skb, copied_early);
1975                        copied_early = false;
1976                }
1977                break;
1978        } while (len > 0);
1979
1980        if (user_recv) {
1981                if (!skb_queue_empty(&tp->ucopy.prequeue)) {
1982                        int chunk;
1983
1984                        tp->ucopy.len = copied > 0 ? len : 0;
1985
1986                        tcp_prequeue_process(sk);
1987
1988                        if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
1989                                NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
1990                                len -= chunk;
1991                                copied += chunk;
1992                        }
1993                }
1994
1995                tp->ucopy.task = NULL;
1996                tp->ucopy.len = 0;
1997        }
1998
1999 #ifdef CONFIG_NET_DMA
2000        tcp_service_net_dma(sk, true);  /* Wait for queue to drain */
2001        tp->ucopy.dma_chan = NULL;
2002
2003        if (tp->ucopy.pinned_list) {
2004                dma_unpin_iovec_pages(tp->ucopy.pinned_list);
2005                tp->ucopy.pinned_list = NULL;
2006        }
2007 #endif
2008
2009        /* According to UNIX98, msg_name/msg_namelen are ignored
2010         * on connected socket. I was just happy when found this 8) --ANK
2011         */
2012
2013        /* Clean up data we have read: This will do ACK frames. */
2014        tcp_cleanup_rbuf(sk, copied);
2015
2016        release_sock(sk);
2017        return copied;
2018
2019 out:
2020        release_sock(sk);
```

```
2021         return err;
2022
2023 recv_urg:
2024         err = tcp_recv_urg(sk, msg, len, flags);
2025         goto out;
2026
2027 recv_sndq:
2028         err = tcp_peek_sndq(sk, msg, len);
2029         goto out;
2030 }
2031 EXPORT_SYMBOL(tcp_recvmsg);
2032
2033 void tcp_set_state(struct sock *sk, int state)
2034 {
2035         int oldstate = sk->sk_state;
2036
2037         switch (state) {
2038         case TCP_ESTABLISHED:
2039                 if (oldstate != TCP_ESTABLISHED)
2040                         TCP_INC_STATS(sock_net(sk), TCP_MIB_CURRESTAB);
2041                 break;
2042
2043         case TCP_CLOSE:
2044                 if (oldstate == TCP_CLOSE_WAIT || oldstate == TCP_ESTABLISHED)
2045                         TCP_INC_STATS(sock_net(sk), TCP_MIB_ESTABRESETS);
2046
2047                 sk->sk_prot->unhash(sk);
2048                 if (inet_csk(sk)->icsk_bind_hash &&
2049                     !(sk->sk_userlocks & SOCK_BINDPORT_LOCK))
2050                         inet_put_port(sk);
2051                 /* fall through */
2052         default:
2053                 if (oldstate == TCP_ESTABLISHED)
2054                         TCP_DEC_STATS(sock_net(sk), TCP_MIB_CURRESTAB);
2055         }
2056
2057         /* Change state AFTER socket is unhashed to avoid closed
2058          * socket sitting in hash tables.
2059          */
2060         sk->sk_state = state;
2061
2062 #ifdef STATE_TRACE
2063         SOCK_DEBUG(sk, "TCP sk=%p, State %s -> %s\n", sk, statename[oldstate], statename[state]);
2064 #endif
2065 }
2066 EXPORT_SYMBOL_GPL(tcp_set_state);
2067
2068 /*
2069  *      State processing on a close. This implements the state shift for
2070  *      sending our FIN frame. Note that we only send a FIN for some
2071  *      states. A shutdown() may have already sent the FIN, or we may be
2072  *      closed.
2073  */
2074
2075 static const unsigned char new_state[16] = {
2076   /* current state:        new state:      action:      */
2077   /* (Invalid)          */ TCP_CLOSE,
2078   /* TCP_ESTABLISHED    */ TCP_FIN_WAIT1 | TCP_ACTION_FIN,
2079   /* TCP_SYN_SENT       */ TCP_CLOSE,
2080   /* TCP_SYN_RECV       */ TCP_FIN_WAIT1 | TCP_ACTION_FIN,
2081   /* TCP_FIN_WAIT1      */ TCP_FIN_WAIT1,
2082   /* TCP_FIN_WAIT2      */ TCP_FIN_WAIT2,
2083   /* TCP_TIME_WAIT      */ TCP_CLOSE,
2084   /* TCP_CLOSE          */ TCP_CLOSE,
2085   /* TCP_CLOSE_WAIT     */ TCP_LAST_ACK  | TCP_ACTION_FIN,
2086   /* TCP_LAST_ACK       */ TCP_LAST_ACK,
2087   /* TCP_LISTEN         */ TCP_CLOSE,
2088   /* TCP_CLOSING        */ TCP_CLOSING,
2089 };
2090
2091 static int tcp_close_state(struct sock *sk)
2092 {
2093         int next = (int)new_state[sk->sk_state];
2094         int ns = next & TCP_STATE_MASK;
2095
2096         tcp_set_state(sk, ns);
2097
2098         return next & TCP_ACTION_FIN;
2099 }
2100
2101 /*
2102  *      Shutdown the sending side of a connection. Much like close except
2103  *      that we don't receive shut down or sock_set_flag(sk, SOCK_DEAD).
2104  */
2105
```

```
2106 void tcp_shutdown(struct sock *sk, int how)
2107 {
2108         /*      We need to grab some memory, and put together a FIN,
2109          *      and then put it into the queue to be sent.
2110          *              Tim MacKenzie(tym@dibbler.cs.monash.edu.au) 4 Dec '92.
2111          */
2112         if (!(how & SEND_SHUTDOWN))
2113                 return;
2114
2115         /* If we've already sent a FIN, or it's a closed state, skip this. */
2116         if ((1 << sk->sk_state) &
2117             (TCPF_ESTABLISHED | TCPF_SYN_SENT |
2118              TCPF_SYN_RECV | TCPF_CLOSE_WAIT)) {
2119                 /* Clear out any half completed packets.  FIN if needed. */
2120                 if (tcp_close_state(sk))
2121                         tcp_send_fin(sk);
2122         }
2123 }
2124 EXPORT_SYMBOL(tcp_shutdown);
2125
2126 bool tcp_check_oom(struct sock *sk, int shift)
2127 {
2128         bool too_many_orphans, out_of_socket_memory;
2129
2130         too_many_orphans = tcp_too_many_orphans(sk, shift);
2131         out_of_socket_memory = tcp_out_of_memory(sk);
2132
2133         if (too_many_orphans)
2134                 net_info_ratelimited("too many orphaned sockets\n");
2135         if (out_of_socket_memory)
2136                 net_info_ratelimited("out of memory -- consider tuning tcp_mem\n");
2137         return too_many_orphans || out_of_socket_memory;
2138 }
2139
2140 void tcp_close(struct sock *sk, long timeout)
2141 {
2142         struct sk_buff *skb;
2143         int data_was_unread = 0;
2144         int state;
2145
2146         lock_sock(sk);
2147         sk->sk_shutdown = SHUTDOWN_MASK;
2148
2149         if (sk->sk_state == TCP_LISTEN) {
2150                 tcp_set_state(sk, TCP_CLOSE);
2151
2152                 /* Special case. */
2153                 inet_csk_listen_stop(sk);
2154
2155                 goto adjudge_to_death;
2156         }
2157
2158         /*  We need to flush the recv. buffs.  We do this only on the
2159          *  descriptor close, not protocol-sourced closes, because the
2160          *  reader process may not have drained the data yet!
2161          */
2162         while ((skb = __skb_dequeue(&sk->sk_receive_queue)) != NULL) {
2163                 u32 len = TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq -
2164                           tcp_hdr(skb)->fin;
2165                 data_was_unread += len;
2166                 __kfree_skb(skb);
2167         }
2168
2169         sk_mem_reclaim(sk);
2170
2171         /* If socket has been already reset (e.g. in tcp_reset()) - kill it. */
2172         if (sk->sk_state == TCP_CLOSE)
2173                 goto adjudge_to_death;
2174
2175         /* As outlined in RFC 2525, section 2.17, we send a RST here because
2176          * data was lost. To witness the awful effects of the old behavior of
2177          * always doing a FIN, run an older 2.1.x kernel or 2.0.x, start a bulk
2178          * GET in an FTP client, suspend the process, wait for the client to
2179          * advertise a zero window, then kill -9 the FTP client, wheee...
2180          * Note: timeout is always zero in such a case.
2181          */
2182         if (unlikely(tcp_sk(sk)->repair)) {
2183                 sk->sk_prot->disconnect(sk, 0);
2184         } else if (data_was_unread) {
2185                 /* Unread data was tossed, zap the connection. */
2186                 NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONCLOSE);
2187                 tcp_set_state(sk, TCP_CLOSE);
2188                 tcp_send_active_reset(sk, sk->sk_allocation);
2189         } else if (sock_flag(sk, SOCK_LINGER) && !sk->sk_lingertime) {
2190                 /* Check zero linger _after_ checking for unread data. */
```

```
2191                    sk->sk_prot->disconnect(sk, 0);
2192                    NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
2193            } else if (tcp_close_state(sk)) {
2194                    /* We FIN if the application ate all the data before
2195                     * zapping the connection.
2196                     */
2197
2198                    /* RED-PEN. Formally speaking, we have broken TCP state
2199                     * machine. State transitions:
2200                     *
2201                     * TCP_ESTABLISHED -> TCP_FIN_WAIT1
2202                     * TCP_SYN_RECV -> TCP_FIN_WAIT1 (forget it, it's impossible)
2203                     * TCP_CLOSE_WAIT -> TCP_LAST_ACK
2204                     *
2205                     * are legal only when FIN has been sent (i.e. in window),
2206                     * rather than queued out of window. Purists blame.
2207                     *
2208                     * F.e. "RFC state" is ESTABLISHED,
2209                     * if Linux state is FIN-WAIT-1, but FIN is still not sent.
2210                     *
2211                     * The visible declinations are that sometimes
2212                     * we enter time-wait state, when it is not required really
2213                     * (harmless), do not send active resets, when they are
2214                     * required by specs (TCP_ESTABLISHED, TCP_CLOSE_WAIT, when
2215                     * they look as CLOSING or LAST_ACK for Linux)
2216                     * Probably, I missed some more holelets.
2217                     *                                              --ANK
2218                     * XXX (TFO) - To start off we don't support SYN+ACK+FIN
2219                     * in a single packet! (May consider it later but will
2220                     * probably need API support or TCP_CORK SYN-ACK until
2221                     * data is written and socket is closed.)
2222                     */
2223                    tcp_send_fin(sk);
2224            }
2225
2226            sk_stream_wait_close(sk, timeout);
2227
2228    adjudge_to_death:
2229            state = sk->sk_state;
2230            sock_hold(sk);
2231            sock_orphan(sk);
2232
2233            /* It is the last release_sock in its life. It will remove backlog. */
2234            release_sock(sk);
2235
2236
2237            /* Now socket is owned by kernel and we acquire BH lock
2238               to finish close. No need to check for user refs.
2239             */
2240            local_bh_disable();
2241            bh_lock_sock(sk);
2242            WARN_ON(sock_owned_by_user(sk));
2243
2244            percpu_counter_inc(sk->sk_prot->orphan_count);
2245
2246            /* Have we already been destroyed by a softirq or backlog? */
2247            if (state != TCP_CLOSE && sk->sk_state == TCP_CLOSE)
2248                    goto out;
2249
2250            /*      This is a (useful) BSD violating of the RFC. There is a
2251             *      problem with TCP as specified in that the other end could
2252             *      keep a socket open forever with no application left this end.
2253             *      We use a 1 minute timeout (about the same as BSD) then kill
2254             *      our end. If they send after that then tough - BUT: long enough
2255             *      that we won't make the old 4*rto = almost no time - whoops
2256             *      reset mistake.
2257             *
2258             *      Nope, it was not mistake. It is really desired behaviour
2259             *      f.e. on http servers, when such sockets are useless, but
2260             *      consume significant resources. Let's do it with special
2261             *      linger2 option.                                 --ANK
2262             */
2263
2264            if (sk->sk_state == TCP_FIN_WAIT2) {
2265                    struct tcp_sock *tp = tcp_sk(sk);
2266                    if (tp->linger2 < 0) {
2267                            tcp_set_state(sk, TCP_CLOSE);
2268                            tcp_send_active_reset(sk, GFP_ATOMIC);
2269                            NET_INC_STATS_BH(sock_net(sk),
2270                                             LINUX_MIB_TCPABORTONLINGER);
2271                    } else {
2272                            const int tmo = tcp_fin_time(sk);
2273
2274                            if (tmo > TCP_TIMEWAIT_LEN) {
2275                                    inet_csk_reset_keepalive_timer(sk,
```

```
2276                                                tmo - TCP_TIMEWAIT_LEN);
2277                        } else {
2278                                tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
2279                                goto out;
2280                        }
2281                }
2282        }
2283        if (sk->sk_state != TCP_CLOSE) {
2284                sk_mem_reclaim(sk);
2285                if (tcp_check_oom(sk, 0)) {
2286                        tcp_set_state(sk, TCP_CLOSE);
2287                        tcp_send_active_reset(sk, GFP_ATOMIC);
2288                        NET_INC_STATS_BH(sock_net(sk),
2289                                         LINUX_MIB_TCPABORTONMEMORY);
2290                }
2291        }
2292
2293        if (sk->sk_state == TCP_CLOSE) {
2294                struct request_sock *req = tcp_sk(sk)->fastopen_rsk;
2295                /* We could get here with a non-NULL req if the socket is
2296                 * aborted (e.g., closed with unread data) before 3WHS
2297                 * finishes.
2298                 */
2299                if (req != NULL)
2300                        reqsk_fastopen_remove(sk, req, false);
2301                inet_csk_destroy_sock(sk);
2302        }
2303        /* Otherwise, socket is reprieved until protocol close. */
2304
2305 out:
2306        bh_unlock_sock(sk);
2307        local_bh_enable();
2308        sock_put(sk);
2309 }
2310 EXPORT_SYMBOL(tcp_close);
2311
2312 /* These states need RST on ABORT according to RFC793 */
2313
2314 static inline bool tcp_need_reset(int state)
2315 {
2316        return (1 << state) &
2317               (TCPF_ESTABLISHED | TCPF_CLOSE_WAIT | TCPF_FIN_WAIT1 |
2318                TCPF_FIN_WAIT2 | TCPF_SYN_RECV);
2319 }
2320
2321 int tcp_disconnect(struct sock *sk, int flags)
2322 {
2323        struct inet_sock *inet = inet_sk(sk);
2324        struct inet_connection_sock *icsk = inet_csk(sk);
2325        struct tcp_sock *tp = tcp_sk(sk);
2326        int err = 0;
2327        int old_state = sk->sk_state;
2328
2329        if (old_state != TCP_CLOSE)
2330                tcp_set_state(sk, TCP_CLOSE);
2331
2332        /* ABORT function of RFC793 */
2333        if (old_state == TCP_LISTEN) {
2334                inet_csk_listen_stop(sk);
2335        } else if (unlikely(tp->repair)) {
2336                sk->sk_err = ECONNABORTED;
2337        } else if (tcp_need_reset(old_state) ||
2338                   (tp->snd_nxt != tp->write_seq &&
2339                    (1 << old_state) & (TCPF_CLOSING | TCPF_LAST_ACK))) {
2340                /* The last check adjusts for discrepancy of Linux wrt. RFC
2341                 * states
2342                 */
2343                tcp_send_active_reset(sk, gfp_any());
2344                sk->sk_err = ECONNRESET;
2345        } else if (old_state == TCP_SYN_SENT)
2346                sk->sk_err = ECONNRESET;
2347
2348        tcp_clear_xmit_timers(sk);
2349        __skb_queue_purge(&sk->sk_receive_queue);
2350        tcp_write_queue_purge(sk);
2351        __skb_queue_purge(&tp->out_of_order_queue);
2352 #ifdef CONFIG_NET_DMA
2353        __skb_queue_purge(&sk->sk_async_wait_queue);
2354 #endif
2355
2356        inet->inet_dport = 0;
2357
2358        if (!(sk->sk_userlocks & SOCK_BINDADDR_LOCK))
2359                inet_reset_saddr(sk);
2360
```

```
2361         sk->sk_shutdown = 0;
2362         sock_reset_flag(sk, SOCK_DONE);
2363         tp->srtt_us = 0;
2364         if ((tp->write_seq += tp->max_window + 2) == 0)
2365                 tp->write_seq = 1;
2366         icsk->icsk_backoff = 0;
2367         tp->snd_cwnd = 2;
2368         icsk->icsk_probes_out = 0;
2369         tp->packets_out = 0;
2370         tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
2371         tp->snd_cwnd_cnt = 0;
2372         tp->window_clamp = 0;
2373         tcp_set_ca_state(sk, TCP_CA_Open);
2374         tcp_clear_retrans(tp);
2375         inet_csk_delack_init(sk);
2376         tcp_init_send_head(sk);
2377         memset(&tp->rx_opt, 0, sizeof(tp->rx_opt));
2378         __sk_dst_reset(sk);
2379
2380         WARN_ON(inet->inet_num && !icsk->icsk_bind_hash);
2381
2382         sk->sk_error_report(sk);
2383         return err;
2384 }
2385 EXPORT_SYMBOL(tcp_disconnect);
2386
2387 void tcp_sock_destruct(struct sock *sk)
2388 {
2389         inet_sock_destruct(sk);
2390
2391         kfree(inet_csk(sk)->icsk_accept_queue.fastopenq);
2392 }
2393
2394 static inline bool tcp_can_repair_sock(const struct sock *sk)
2395 {
2396         return ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN) &&
2397                 ((1 << sk->sk_state) & (TCPF_CLOSE | TCPF_ESTABLISHED));
2398 }
2399
2400 static int tcp_repair_options_est(struct tcp_sock *tp,
2401                 struct tcp_repair_opt __user *optbuf, unsigned int len)
2402 {
2403         struct tcp_repair_opt opt;
2404
2405         while (len >= sizeof(opt)) {
2406                 if (copy_from_user(&opt, optbuf, sizeof(opt)))
2407                         return -EFAULT;
2408
2409                 optbuf++;
2410                 len -= sizeof(opt);
2411
2412                 switch (opt.opt_code) {
2413                 case TCPOPT_MSS:
2414                         tp->rx_opt.mss_clamp = opt.opt_val;
2415                         break;
2416                 case TCPOPT_WINDOW:
2417                         {
2418                                 u16 snd_wscale = opt.opt_val & 0xFFFF;
2419                                 u16 rcv_wscale = opt.opt_val >> 16;
2420
2421                                 if (snd_wscale > 14 || rcv_wscale > 14)
2422                                         return -EFBIG;
2423
2424                                 tp->rx_opt.snd_wscale = snd_wscale;
2425                                 tp->rx_opt.rcv_wscale = rcv_wscale;
2426                                 tp->rx_opt.wscale_ok = 1;
2427                         }
2428                         break;
2429                 case TCPOPT_SACK_PERM:
2430                         if (opt.opt_val != 0)
2431                                 return -EINVAL;
2432
2433                         tp->rx_opt.sack_ok |= TCP_SACK_SEEN;
2434                         if (sysctl_tcp_fack)
2435                                 tcp_enable_fack(tp);
2436                         break;
2437                 case TCPOPT_TIMESTAMP:
2438                         if (opt.opt_val != 0)
2439                                 return -EINVAL;
2440
2441                         tp->rx_opt.tstamp_ok = 1;
2442                         break;
2443                 }
2444         }
2445
```

```
2446         return 0;
2447 }
2448
2449 /*
2450  *      Socket option code for TCP.
2451  */
2452 static int do_tcp_setsockopt(struct sock *sk, int level,
2453                 int optname, char __user *optval, unsigned int optlen)
2454 {
2455         struct tcp_sock *tp = tcp_sk(sk);
2456         struct inet_connection_sock *icsk = inet_csk(sk);
2457         int val;
2458         int err = 0;
2459
2460         /* These are data/string values, all the others are ints */
2461         switch (optname) {
2462         case TCP_CONGESTION: {
2463                 char name[TCP_CA_NAME_MAX];
2464
2465                 if (optlen < 1)
2466                         return -EINVAL;
2467
2468                 val = strncpy_from_user(name, optval,
2469                                         min_t(long, TCP_CA_NAME_MAX-1, optlen));
2470                 if (val < 0)
2471                         return -EFAULT;
2472                 name[val] = 0;
2473
2474                 lock_sock(sk);
2475                 err = tcp_set_congestion_control(sk, name);
2476                 release_sock(sk);
2477                 return err;
2478         }
2479         default:
2480                 /* fallthru */
2481                 break;
2482         }
2483
2484         if (optlen < sizeof(int))
2485                 return -EINVAL;
2486
2487         if (get_user(val, (int __user *)optval))
2488                 return -EFAULT;
2489
2490         lock_sock(sk);
2491
2492         switch (optname) {
2493         case TCP_MAXSEG:
2494                 /* Values greater than interface MTU won't take effect. However
2495                  * at the point when this call is done we typically don't yet
2496                  * know which interface is going to be used */
2497                 if (val < TCP_MIN_MSS || val > MAX_TCP_WINDOW) {
2498                         err = -EINVAL;
2499                         break;
2500                 }
2501                 tp->rx_opt.user_mss = val;
2502                 break;
2503
2504         case TCP_NODELAY:
2505                 if (val) {
2506                         /* TCP_NODELAY is weaker than TCP_CORK, so that
2507                          * this option on corked socket is remembered, but
2508                          * it is not activated until cork is cleared.
2509                          *
2510                          * However, when TCP_NODELAY is set we make
2511                          * an explicit push, which overrides even TCP_CORK
2512                          * for currently queued segments.
2513                          */
2514                         tp->nonagle |= TCP_NAGLE_OFF|TCP_NAGLE_PUSH;
2515                         tcp_push_pending_frames(sk);
2516                 } else {
2517                         tp->nonagle &= ~TCP_NAGLE_OFF;
2518                 }
2519                 break;
2520
2521         case TCP_THIN_LINEAR_TIMEOUTS:
2522                 if (val < 0 || val > 1)
2523                         err = -EINVAL;
2524                 else
2525                         tp->thin_lto = val;
2526                 break;
2527
2528         case TCP_THIN_DUPACK:
2529                 if (val < 0 || val > 1)
2530                         err = -EINVAL;
```

```
2531                    else {
2532                            tp->thin_dupack = val;
2533                            if (tp->thin_dupack)
2534                                    tcp_disable_early_retrans(tp);
2535                    }
2536                    break;
2537
2538            case TCP_REPAIR:
2539                    if (!tcp_can_repair_sock(sk))
2540                            err = -EPERM;
2541                    else if (val == 1) {
2542                            tp->repair = 1;
2543                            sk->sk_reuse = SK_FORCE_REUSE;
2544                            tp->repair_queue = TCP_NO_QUEUE;
2545                    } else if (val == 0) {
2546                            tp->repair = 0;
2547                            sk->sk_reuse = SK_NO_REUSE;
2548                            tcp_send_window_probe(sk);
2549                    } else
2550                            err = -EINVAL;
2551
2552                    break;
2553
2554            case TCP_REPAIR_QUEUE:
2555                    if (!tp->repair)
2556                            err = -EPERM;
2557                    else if (val < TCP_QUEUES_NR)
2558                            tp->repair_queue = val;
2559                    else
2560                            err = -EINVAL;
2561                    break;
2562
2563            case TCP_QUEUE_SEQ:
2564                    if (sk->sk_state != TCP_CLOSE)
2565                            err = -EPERM;
2566                    else if (tp->repair_queue == TCP_SEND_QUEUE)
2567                            tp->write_seq = val;
2568                    else if (tp->repair_queue == TCP_RECV_QUEUE)
2569                            tp->rcv_nxt = val;
2570                    else
2571                            err = -EINVAL;
2572                    break;
2573
2574            case TCP_REPAIR_OPTIONS:
2575                    if (!tp->repair)
2576                            err = -EINVAL;
2577                    else if (sk->sk_state == TCP_ESTABLISHED)
2578                            err = tcp_repair_options_est(tp,
2579                                            (struct tcp_repair_opt __user *)optval,
2580                                            optlen);
2581                    else
2582                            err = -EPERM;
2583                    break;
2584
2585            case TCP_CORK:
2586                    /* When set indicates to always queue non-full frames.
2587                     * Later the user clears this option and we transmit
2588                     * any pending partial frames in the queue.  This is
2589                     * meant to be used alongside sendfile() to get properly
2590                     * filled frames when the user (for example) must write
2591                     * out headers with a write() call first and then use
2592                     * sendfile to send out the data parts.
2593                     *
2594                     * TCP_CORK can be set together with TCP_NODELAY and it is
2595                     * stronger than TCP_NODELAY.
2596                     */
2597                    if (val) {
2598                            tp->nonagle |= TCP_NAGLE_CORK;
2599                    } else {
2600                            tp->nonagle &= ~TCP_NAGLE_CORK;
2601                            if (tp->nonagle&TCP_NAGLE_OFF)
2602                                    tp->nonagle |= TCP_NAGLE_PUSH;
2603                            tcp_push_pending_frames(sk);
2604                    }
2605                    break;
2606
2607            case TCP_KEEPIDLE:
2608                    if (val < 1 || val > MAX_TCP_KEEPIDLE)
2609                            err = -EINVAL;
2610                    else {
2611                            tp->keepalive_time = val * HZ;
2612                            if (sock_flag(sk, SOCK_KEEPOPEN) &&
2613                                !((1 << sk->sk_state) &
2614                                  (TCPF_CLOSE | TCPF_LISTEN))) {
2615                                    u32 elapsed = keepalive_time_elapsed(tp);
```

```
2616                                if (tp->keepalive_time > elapsed)
2617                                        elapsed = tp->keepalive_time - elapsed;
2618                                else
2619                                        elapsed = 0;
2620                                inet_csk_reset_keepalive_timer(sk, elapsed);
2621                        }
2622                }
2623                break;
2624        case TCP_KEEPINTVL:
2625                if (val < 1 || val > MAX_TCP_KEEPINTVL)
2626                        err = -EINVAL;
2627                else
2628                        tp->keepalive_intvl = val * HZ;
2629                break;
2630        case TCP_KEEPCNT:
2631                if (val < 1 || val > MAX_TCP_KEEPCNT)
2632                        err = -EINVAL;
2633                else
2634                        tp->keepalive_probes = val;
2635                break;
2636        case TCP_SYNCNT:
2637                if (val < 1 || val > MAX_TCP_SYNCNT)
2638                        err = -EINVAL;
2639                else
2640                        icsk->icsk_syn_retries = val;
2641                break;
2642
2643        case TCP_LINGER2:
2644                if (val < 0)
2645                        tp->linger2 = -1;
2646                else if (val > sysctl_tcp_fin_timeout / HZ)
2647                        tp->linger2 = 0;
2648                else
2649                        tp->linger2 = val * HZ;
2650                break;
2651
2652        case TCP_DEFER_ACCEPT:
2653                /* Translate value in seconds to number of retransmits */
2654                icsk->icsk_accept_queue.rskq_defer_accept =
2655                        secs_to_retrans(val, TCP_TIMEOUT_INIT / HZ,
2656                                        TCP_RTO_MAX / HZ);
2657                break;
2658
2659        case TCP_WINDOW_CLAMP:
2660                if (!val) {
2661                        if (sk->sk_state != TCP_CLOSE) {
2662                                err = -EINVAL;
2663                                break;
2664                        }
2665                        tp->window_clamp = 0;
2666                } else
2667                        tp->window_clamp = val < SOCK_MIN_RCVBUF / 2 ?
2668                                                SOCK_MIN_RCVBUF / 2 : val;
2669                break;
2670
2671        case TCP_QUICKACK:
2672                if (!val) {
2673                        icsk->icsk_ack.pingpong = 1;
2674                } else {
2675                        icsk->icsk_ack.pingpong = 0;
2676                        if ((1 << sk->sk_state) &
2677                            (TCPF_ESTABLISHED | TCPF_CLOSE_WAIT) &&
2678                            inet_csk_ack_scheduled(sk)) {
2679                                icsk->icsk_ack.pending |= ICSK_ACK_PUSHED;
2680                                tcp_cleanup_rbuf(sk, 1);
2681                                if (!(val & 1))
2682                                        icsk->icsk_ack.pingpong = 1;
2683                        }
2684                }
2685                break;
2686
2687 #ifdef CONFIG_TCP_MD5SIG
2688        case TCP_MD5SIG:
2689                /* Read the IP->Key mappings from userspace */
2690                err = tp->af_specific->md5_parse(sk, optval, optlen);
2691                break;
2692 #endif
2693        case TCP_USER_TIMEOUT:
2694                /* Cap the max timeout in ms TCP will retry/retrans
2695                 * before giving up and aborting (ETIMEDOUT) a connection.
2696                 */
2697                if (val < 0)
2698                        err = -EINVAL;
2699                else
2700                        icsk->icsk_user_timeout = msecs_to_jiffies(val);
```

```
2701                    break;
2702
2703            case TCP_FASTOPEN:
2704                    if (val >= 0 && ((1 << sk->sk_state) & (TCPF_CLOSE |
2705                        TCPF_LISTEN)))
2706                            err = fastopen_init_queue(sk, val);
2707                    else
2708                            err = -EINVAL;
2709                    break;
2710            case TCP_TIMESTAMP:
2711                    if (!tp->repair)
2712                            err = -EPERM;
2713                    else
2714                            tp->tsoffset = val - tcp_time_stamp;
2715                    break;
2716            case TCP_NOTSENT_LOWAT:
2717                    tp->notsent_lowat = val;
2718                    sk->sk_write_space(sk);
2719                    break;
2720            default:
2721                    err = -ENOPROTOOPT;
2722                    break;
2723            }
2724
2725            release_sock(sk);
2726            return err;
2727    }
2728
2729    int tcp_setsockopt(struct sock *sk, int level, int optname, char __user *optval,
2730                       unsigned int optlen)
2731    {
2732            const struct inet_connection_sock *icsk = inet_csk(sk);
2733
2734            if (level != SOL_TCP)
2735                    return icsk->icsk_af_ops->setsockopt(sk, level, optname,
2736                                                         optval, optlen);
2737            return do_tcp_setsockopt(sk, level, optname, optval, optlen);
2738    }
2739    EXPORT_SYMBOL(tcp_setsockopt);
2740
2741    #ifdef CONFIG_COMPAT
2742    int compat_tcp_setsockopt(struct sock *sk, int level, int optname,
2743                              char __user *optval, unsigned int optlen)
2744    {
2745            if (level != SOL_TCP)
2746                    return inet_csk_compat_setsockopt(sk, level, optname,
2747                                                      optval, optlen);
2748            return do_tcp_setsockopt(sk, level, optname, optval, optlen);
2749    }
2750    EXPORT_SYMBOL(compat_tcp_setsockopt);
2751    #endif
2752
2753    /* Return information about state of tcp endpoint in API format. */
2754    void tcp_get_info(const struct sock *sk, struct tcp_info *info)
2755    {
2756            const struct tcp_sock *tp = tcp_sk(sk);
2757            const struct inet_connection_sock *icsk = inet_csk(sk);
2758            u32 now = tcp_time_stamp;
2759
2760            memset(info, 0, sizeof(*info));
2761
2762            info->tcpi_state = sk->sk_state;
2763            info->tcpi_ca_state = icsk->icsk_ca_state;
2764            info->tcpi_retransmits = icsk->icsk_retransmits;
2765            info->tcpi_probes = icsk->icsk_probes_out;
2766            info->tcpi_backoff = icsk->icsk_backoff;
2767
2768            if (tp->rx_opt.tstamp_ok)
2769                    info->tcpi_options |= TCPI_OPT_TIMESTAMPS;
2770            if (tcp_is_sack(tp))
2771                    info->tcpi_options |= TCPI_OPT_SACK;
2772            if (tp->rx_opt.wscale_ok) {
2773                    info->tcpi_options |= TCPI_OPT_WSCALE;
2774                    info->tcpi_snd_wscale = tp->rx_opt.snd_wscale;
2775                    info->tcpi_rcv_wscale = tp->rx_opt.rcv_wscale;
2776            }
2777
2778            if (tp->ecn_flags & TCP_ECN_OK)
2779                    info->tcpi_options |= TCPI_OPT_ECN;
2780            if (tp->ecn_flags & TCP_ECN_SEEN)
2781                    info->tcpi_options |= TCPI_OPT_ECN_SEEN;
2782            if (tp->syn_data_acked)
2783                    info->tcpi_options |= TCPI_OPT_SYN_DATA;
2784
2785            info->tcpi_rto = jiffies_to_usecs(icsk->icsk_rto);
```

```
2786         info->tcpi_ato = jiffies_to_usecs(icsk->icsk_ack.ato);
2787         info->tcpi_snd_mss = tp->mss_cache;
2788         info->tcpi_rcv_mss = icsk->icsk_ack.rcv_mss;
2789
2790         if (sk->sk_state == TCP_LISTEN) {
2791                 info->tcpi_unacked = sk->sk_ack_backlog;
2792                 info->tcpi_sacked = sk->sk_max_ack_backlog;
2793         } else {
2794                 info->tcpi_unacked = tp->packets_out;
2795                 info->tcpi_sacked = tp->sacked_out;
2796         }
2797         info->tcpi_lost = tp->lost_out;
2798         info->tcpi_retrans = tp->retrans_out;
2799         info->tcpi_fackets = tp->fackets_out;
2800
2801         info->tcpi_last_data_sent = jiffies_to_msecs(now - tp->lsndtime);
2802         info->tcpi_last_data_recv = jiffies_to_msecs(now - icsk->icsk_ack.lrcvtime);
2803         info->tcpi_last_ack_recv = jiffies_to_msecs(now - tp->rcv_tstamp);
2804
2805         info->tcpi_pmtu = icsk->icsk_pmtu_cookie;
2806         info->tcpi_rcv_ssthresh = tp->rcv_ssthresh;
2807         info->tcpi_rtt = tp->srtt_us >> 3;
2808         info->tcpi_rttvar = tp->mdev_us >> 2;
2809         info->tcpi_snd_ssthresh = tp->snd_ssthresh;
2810         info->tcpi_snd_cwnd = tp->snd_cwnd;
2811         info->tcpi_advmss = tp->advmss;
2812         info->tcpi_reordering = tp->reordering;
2813
2814         info->tcpi_rcv_rtt = jiffies_to_usecs(tp->rcv_rtt_est.rtt)>>3;
2815         info->tcpi_rcv_space = tp->rcvq_space.space;
2816
2817         info->tcpi_total_retrans = tp->total_retrans;
2818
2819         info->tcpi_pacing_rate = sk->sk_pacing_rate != ~0U ?
2820                                         sk->sk_pacing_rate : ~0ULL;
2821         info->tcpi_max_pacing_rate = sk->sk_max_pacing_rate != ~0U ?
2822                                         sk->sk_max_pacing_rate : ~0ULL;
2823 }
2824 EXPORT_SYMBOL_GPL(tcp_get_info);
2825
2826 static int do_tcp_getsockopt(struct sock *sk, int level,
2827                 int optname, char __user *optval, int __user *optlen)
2828 {
2829         struct inet_connection_sock *icsk = inet_csk(sk);
2830         struct tcp_sock *tp = tcp_sk(sk);
2831         int val, len;
2832
2833         if (get_user(len, optlen))
2834                 return -EFAULT;
2835
2836         len = min_t(unsigned int, len, sizeof(int));
2837
2838         if (len < 0)
2839                 return -EINVAL;
2840
2841         switch (optname) {
2842         case TCP_MAXSEG:
2843                 val = tp->mss_cache;
2844                 if (!val && ((1 << sk->sk_state) & (TCPF_CLOSE | TCPF_LISTEN)))
2845                         val = tp->rx_opt.user_mss;
2846                 if (tp->repair)
2847                         val = tp->rx_opt.mss_clamp;
2848                 break;
2849         case TCP_NODELAY:
2850                 val = !!(tp->nonagle&TCP_NAGLE_OFF);
2851                 break;
2852         case TCP_CORK:
2853                 val = !!(tp->nonagle&TCP_NAGLE_CORK);
2854                 break;
2855         case TCP_KEEPIDLE:
2856                 val = keepalive_time_when(tp) / HZ;
2857                 break;
2858         case TCP_KEEPINTVL:
2859                 val = keepalive_intvl_when(tp) / HZ;
2860                 break;
2861         case TCP_KEEPCNT:
2862                 val = keepalive_probes(tp);
2863                 break;
2864         case TCP_SYNCNT:
2865                 val = icsk->icsk_syn_retries ? : sysctl_tcp_syn_retries;
2866                 break;
2867         case TCP_LINGER2:
2868                 val = tp->linger2;
2869                 if (val >= 0)
2870                         val = (val ? : sysctl_tcp_fin_timeout) / HZ;
```

```
2871                    break;
2872            case TCP_DEFER_ACCEPT:
2873                    val = retrans_to_secs(icsk->icsk_accept_queue.rskq_defer_accept,
2874                                          TCP_TIMEOUT_INIT / HZ, TCP_RTO_MAX / HZ);
2875                    break;
2876            case TCP_WINDOW_CLAMP:
2877                    val = tp->window_clamp;
2878                    break;
2879            case TCP_INFO: {
2880                    struct tcp_info info;
2881
2882                    if (get_user(len, optlen))
2883                            return -EFAULT;
2884
2885                    tcp_get_info(sk, &info);
2886
2887                    len = min_t(unsigned int, len, sizeof(info));
2888                    if (put_user(len, optlen))
2889                            return -EFAULT;
2890                    if (copy_to_user(optval, &info, len))
2891                            return -EFAULT;
2892                    return 0;
2893            }
2894            case TCP_QUICKACK:
2895                    val = !icsk->icsk_ack.pingpong;
2896                    break;
2897
2898            case TCP_CONGESTION:
2899                    if (get_user(len, optlen))
2900                            return -EFAULT;
2901                    len = min_t(unsigned int, len, TCP_CA_NAME_MAX);
2902                    if (put_user(len, optlen))
2903                            return -EFAULT;
2904                    if (copy_to_user(optval, icsk->icsk_ca_ops->name, len))
2905                            return -EFAULT;
2906                    return 0;
2907
2908            case TCP_THIN_LINEAR_TIMEOUTS:
2909                    val = tp->thin_lto;
2910                    break;
2911            case TCP_THIN_DUPACK:
2912                    val = tp->thin_dupack;
2913                    break;
2914
2915            case TCP_REPAIR:
2916                    val = tp->repair;
2917                    break;
2918
2919            case TCP_REPAIR_QUEUE:
2920                    if (tp->repair)
2921                            val = tp->repair_queue;
2922                    else
2923                            return -EINVAL;
2924                    break;
2925
2926            case TCP_QUEUE_SEQ:
2927                    if (tp->repair_queue == TCP_SEND_QUEUE)
2928                            val = tp->write_seq;
2929                    else if (tp->repair_queue == TCP_RECV_QUEUE)
2930                            val = tp->rcv_nxt;
2931                    else
2932                            return -EINVAL;
2933                    break;
2934
2935            case TCP_USER_TIMEOUT:
2936                    val = jiffies_to_msecs(icsk->icsk_user_timeout);
2937                    break;
2938
2939            case TCP_FASTOPEN:
2940                    if (icsk->icsk_accept_queue.fastopenq != NULL)
2941                            val = icsk->icsk_accept_queue.fastopenq->max_qlen;
2942                    else
2943                            val = 0;
2944                    break;
2945
2946            case TCP_TIMESTAMP:
2947                    val = tcp_time_stamp + tp->tsoffset;
2948                    break;
2949            case TCP_NOTSENT_LOWAT:
2950                    val = tp->notsent_lowat;
2951                    break;
2952            default:
2953                    return -ENOPROTOOPT;
2954            }
2955
```

```
2956         if (put_user(len, optlen))
2957                 return -EFAULT;
2958         if (copy_to_user(optval, &val, len))
2959                 return -EFAULT;
2960         return 0;
2961 }
2962
2963 int tcp_getsockopt(struct sock *sk, int level, int optname, char __user *optval,
2964                    int __user *optlen)
2965 {
2966         struct inet_connection_sock *icsk = inet_csk(sk);
2967
2968         if (level != SOL_TCP)
2969                 return icsk->icsk_af_ops->getsockopt(sk, level, optname,
2970                                                      optval, optlen);
2971         return do_tcp_getsockopt(sk, level, optname, optval, optlen);
2972 }
2973 EXPORT_SYMBOL(tcp_getsockopt);
2974
2975 #ifdef CONFIG_COMPAT
2976 int compat_tcp_getsockopt(struct sock *sk, int level, int optname,
2977                           char __user *optval, int __user *optlen)
2978 {
2979         if (level != SOL_TCP)
2980                 return inet_csk_compat_getsockopt(sk, level, optname,
2981                                                   optval, optlen);
2982         return do_tcp_getsockopt(sk, level, optname, optval, optlen);
2983 }
2984 EXPORT_SYMBOL(compat_tcp_getsockopt);
2985 #endif
2986
2987 #ifdef CONFIG_TCP_MD5SIG
2988 static struct tcp_md5sig_pool __percpu *tcp_md5sig_pool __read_mostly;
2989 static DEFINE_MUTEX(tcp_md5sig_mutex);
2990
2991 static void __tcp_free_md5sig_pool(struct tcp_md5sig_pool __percpu *pool)
2992 {
2993         int cpu;
2994
2995         for_each_possible_cpu(cpu) {
2996                 struct tcp_md5sig_pool *p = per_cpu_ptr(pool, cpu);
2997
2998                 if (p->md5_desc.tfm)
2999                         crypto_free_hash(p->md5_desc.tfm);
3000         }
3001         free_percpu(pool);
3002 }
3003
3004 static void __tcp_alloc_md5sig_pool(void)
3005 {
3006         int cpu;
3007         struct tcp_md5sig_pool __percpu *pool;
3008
3009         pool = alloc_percpu(struct tcp_md5sig_pool);
3010         if (!pool)
3011                 return;
3012
3013         for_each_possible_cpu(cpu) {
3014                 struct crypto_hash *hash;
3015
3016                 hash = crypto_alloc_hash("md5", 0, CRYPTO_ALG_ASYNC);
3017                 if (IS_ERR_OR_NULL(hash))
3018                         goto out_free;
3019
3020                 per_cpu_ptr(pool, cpu)->md5_desc.tfm = hash;
3021         }
3022         /* before setting tcp_md5sig_pool, we must commit all writes
3023          * to memory. See ACCESS_ONCE() in tcp_get_md5sig_pool()
3024          */
3025         smp_wmb();
3026         tcp_md5sig_pool = pool;
3027         return;
3028 out_free:
3029         __tcp_free_md5sig_pool(pool);
3030 }
3031
3032 bool tcp_alloc_md5sig_pool(void)
3033 {
3034         if (unlikely(!tcp_md5sig_pool)) {
3035                 mutex_lock(&tcp_md5sig_mutex);
3036
3037                 if (!tcp_md5sig_pool)
3038                         __tcp_alloc_md5sig_pool();
3039
3040                 mutex_unlock(&tcp_md5sig_mutex);
```

```
3041         }
3042         return tcp_md5sig_pool != NULL;
3043 }
3044 EXPORT_SYMBOL(tcp_alloc_md5sig_pool);
3045
3046
3047 /**
3048  *      tcp_get_md5sig_pool - get md5sig_pool for this user
3049  *
3050  *      We use percpu structure, so if we succeed, we exit with preemption
3051  *      and BH disabled, to make sure another thread or softirq handling
3052  *      wont try to get same context.
3053  */
3054 struct tcp_md5sig_pool *tcp_get_md5sig_pool(void)
3055 {
3056         struct tcp_md5sig_pool __percpu *p;
3057
3058         local_bh_disable();
3059         p = ACCESS_ONCE(tcp_md5sig_pool);
3060         if (p)
3061                 return __this_cpu_ptr(p);
3062
3063         local_bh_enable();
3064         return NULL;
3065 }
3066 EXPORT_SYMBOL(tcp_get_md5sig_pool);
3067
3068 int tcp_md5_hash_header(struct tcp_md5sig_pool *hp,
3069                         const struct tcphdr *th)
3070 {
3071         struct scatterlist sg;
3072         struct tcphdr hdr;
3073         int err;
3074
3075         /* We are not allowed to change tcphdr, make a local copy */
3076         memcpy(&hdr, th, sizeof(hdr));
3077         hdr.check = 0;
3078
3079         /* options aren't included in the hash */
3080         sg_init_one(&sg, &hdr, sizeof(hdr));
3081         err = crypto_hash_update(&hp->md5_desc, &sg, sizeof(hdr));
3082         return err;
3083 }
3084 EXPORT_SYMBOL(tcp_md5_hash_header);
3085
3086 int tcp_md5_hash_skb_data(struct tcp_md5sig_pool *hp,
3087                           const struct sk_buff *skb, unsigned int header_len)
3088 {
3089         struct scatterlist sg;
3090         const struct tcphdr *tp = tcp_hdr(skb);
3091         struct hash_desc *desc = &hp->md5_desc;
3092         unsigned int i;
3093         const unsigned int head_data_len = skb_headlen(skb) > header_len ?
3094                                            skb_headlen(skb) - header_len : 0;
3095         const struct skb_shared_info *shi = skb_shinfo(skb);
3096         struct sk_buff *frag_iter;
3097
3098         sg_init_table(&sg, 1);
3099
3100         sg_set_buf(&sg, ((u8 *) tp) + header_len, head_data_len);
3101         if (crypto_hash_update(desc, &sg, head_data_len))
3102                 return 1;
3103
3104         for (i = 0; i < shi->nr_frags; ++i) {
3105                 const struct skb_frag_struct *f = &shi->frags[i];
3106                 unsigned int offset = f->page_offset;
3107                 struct page *page = skb_frag_page(f) + (offset >> PAGE_SHIFT);
3108
3109                 sg_set_page(&sg, page, skb_frag_size(f),
3110                             offset_in_page(offset));
3111                 if (crypto_hash_update(desc, &sg, skb_frag_size(f)))
3112                         return 1;
3113         }
3114
3115         skb_walk_frags(skb, frag_iter)
3116                 if (tcp_md5_hash_skb_data(hp, frag_iter, 0))
3117                         return 1;
3118
3119         return 0;
3120 }
3121 EXPORT_SYMBOL(tcp_md5_hash_skb_data);
3122
3123 int tcp_md5_hash_key(struct tcp_md5sig_pool *hp, const struct tcp_md5sig_key *key)
3124 {
3125         struct scatterlist sg;
```

```
3126
3127            sg_init_one(&sg, key->key, key->keylen);
3128            return crypto_hash_update(&hp->md5_desc, &sg, key->keylen);
3129    }
3130    EXPORT_SYMBOL(tcp_md5_hash_key);
3131
3132    #endif
3133
3134    void tcp_done(struct sock *sk)
3135    {
3136            struct request_sock *req = tcp_sk(sk)->fastopen_rsk;
3137
3138            if (sk->sk_state == TCP_SYN_SENT || sk->sk_state == TCP_SYN_RECV)
3139                    TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_ATTEMPTFAILS);
3140
3141            tcp_set_state(sk, TCP_CLOSE);
3142            tcp_clear_xmit_timers(sk);
3143            if (req != NULL)
3144                    reqsk_fastopen_remove(sk, req, false);
3145
3146            sk->sk_shutdown = SHUTDOWN_MASK;
3147
3148            if (!sock_flag(sk, SOCK_DEAD))
3149                    sk->sk_state_change(sk);
3150            else
3151                    inet_csk_destroy_sock(sk);
3152    }
3153    EXPORT_SYMBOL_GPL(tcp_done);
3154
3155    extern struct tcp_congestion_ops tcp_reno;
3156
3157    static __initdata unsigned long thash_entries;
3158    static int __init set_thash_entries(char *str)
3159    {
3160            ssize_t ret;
3161
3162            if (!str)
3163                    return 0;
3164
3165            ret = kstrtoul(str, 0, &thash_entries);
3166            if (ret)
3167                    return 0;
3168
3169            return 1;
3170    }
3171    __setup("thash_entries=", set_thash_entries);
3172
3173    static void tcp_init_mem(void)
3174    {
3175            unsigned long limit = nr_free_buffer_pages() / 8;
3176            limit = max(limit, 128UL);
3177            sysctl_tcp_mem[0] = limit / 4 * 3;
3178            sysctl_tcp_mem[1] = limit;
3179            sysctl_tcp_mem[2] = sysctl_tcp_mem[0] * 2;
3180    }
3181
3182    void __init tcp_init(void)
3183    {
3184            struct sk_buff *skb = NULL;
3185            unsigned long limit;
3186            int max_rshare, max_wshare, cnt;
3187            unsigned int i;
3188
3189            BUILD_BUG_ON(sizeof(struct tcp_skb_cb) > sizeof(skb->cb));
3190
3191            percpu_counter_init(&tcp_sockets_allocated, 0);
3192            percpu_counter_init(&tcp_orphan_count, 0);
3193            tcp_hashinfo.bind_bucket_cachep =
3194                    kmem_cache_create("tcp_bind_bucket",
3195                                      sizeof(struct inet_bind_bucket), 0,
3196                                      SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
3197
3198            /* Size and allocate the main established and bind bucket
3199             * hash tables.
3200             *
3201             * The methodology is similar to that of the buffer cache.
3202             */
3203            tcp_hashinfo.ehash =
3204                    alloc_large_system_hash("TCP established",
3205                                            sizeof(struct inet_ehash_bucket),
3206                                            thash_entries,
3207                                            17, /* one slot per 128 KB of memory */
3208                                            0,
3209                                            NULL,
3210                                            &tcp_hashinfo.ehash_mask,
```

```
3211                                    0,
3212                                    thash_entries ? 0 : 512 * 1024);
3213            for (i = 0; i <= tcp_hashinfo.ehash_mask; i++)
3214                    INIT_HLIST_NULLS_HEAD(&tcp_hashinfo.ehash[i].chain, i);
3215
3216            if (inet_ehash_locks_alloc(&tcp_hashinfo))
3217                    panic("TCP: failed to alloc ehash_locks");
3218            tcp_hashinfo.bhash =
3219                    alloc_large_system_hash("TCP bind",
3220                                    sizeof(struct inet_bind_hashbucket),
3221                                    tcp_hashinfo.ehash_mask + 1,
3222                                    17, /* one slot per 128 KB of memory */
3223                                    0,
3224                                    &tcp_hashinfo.bhash_size,
3225                                    NULL,
3226                                    0,
3227                                    64 * 1024);
3228            tcp_hashinfo.bhash_size = 1U << tcp_hashinfo.bhash_size;
3229            for (i = 0; i < tcp_hashinfo.bhash_size; i++) {
3230                    spin_lock_init(&tcp_hashinfo.bhash[i].lock);
3231                    INIT_HLIST_HEAD(&tcp_hashinfo.bhash[i].chain);
3232            }
3233
3234
3235            cnt = tcp_hashinfo.ehash_mask + 1;
3236
3237            tcp_death_row.sysctl_max_tw_buckets = cnt / 2;
3238            sysctl_tcp_max_orphans = cnt / 2;
3239            sysctl_max_syn_backlog = max(128, cnt / 256);
3240
3241            tcp_init_mem();
3242            /* Set per-socket limits to no more than 1/128 the pressure threshold */
3243            limit = nr_free_buffer_pages() << (PAGE_SHIFT - 7);
3244            max_wshare = min(4UL*1024*1024, limit);
3245            max_rshare = min(6UL*1024*1024, limit);
3246
3247            sysctl_tcp_wmem[0] = SK_MEM_QUANTUM;
3248            sysctl_tcp_wmem[1] = 16*1024;
3249            sysctl_tcp_wmem[2] = max(64*1024, max_wshare);
3250
3251            sysctl_tcp_rmem[0] = SK_MEM_QUANTUM;
3252            sysctl_tcp_rmem[1] = 87380;
3253            sysctl_tcp_rmem[2] = max(87380, max_rshare);
3254
3255            pr_info("Hash tables configured (established %u bind %u)\n",
3256                    tcp_hashinfo.ehash_mask + 1, tcp_hashinfo.bhash_size);
3257
3258            tcp_metrics_init();
3259
3260            tcp_register_congestion_control(&tcp_reno);
3261
3262            tcp_tasklet_init();
3263 }
3264
```