Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version: 2.0.40 2.2.26 2.4.37 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 ***3.17***

Linux/net/ipv4/tcp.c

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              Implementation of the Transmission Control Protocol(TCP).
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Mark Evans, <evansmp@uhura.aston.ac.uk>
 *              Corey Minyard <wf-rch!minyard@relay.EU.net>
 *              Florian La Roche, <flla@stud.uni-sb.de>
 *              Charles Hedrick, <hedrick@klinzhai.rutgers.edu>
 *              Linus Torvalds, <torvalds@cs.helsinki.fi>
 *              Alan Cox, <gw4pts@gw4pts.ampr.org>
 *              Matthew Dillon, <dillon@apollo.west.oic.com>
 *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
 *              Jorge Cwik, <jorge@laser.satlink.net>
 *
 * Fixes:
 *              Alan Cox        :       Numerous verify_area() calls
 *              Alan Cox        :       Set the ACK bit on a reset
 *              Alan Cox        :       Stopped it crashing if it closed while
 *                                      sk->inuse=1 and was trying to connect
 *                                      (tcp_err()).
 *              Alan Cox        :       All icmp error handling was broken
 *                                      pointers passed where wrong and the
 *                                      socket was looked up backwards. Nobody
 *                                      tested any icmp error code obviously.
 *              Alan Cox        :       tcp_err() now handled properly. It
 *                                      wakes people on errors. poll
 *                                      behaves and the icmp error race
 *                                      has gone by moving it into sock.c
 *              Alan Cox        :       tcp_send_reset() fixed to work for
 *                                      everything not just packets for
 *                                      unknown sockets.
 *              Alan Cox        :       tcp option processing.
 *              Alan Cox        :       Reset tweaked (still not 100%) [Had
 *                                      syn rule wrong]
 *              Herp Rosmanith  :       More reset fixes
 *              Alan Cox        :       No longer acks invalid rst frames.
 *                                      Acking any kind of RST is right out.
 *              Alan Cox        :       Sets an ignore me flag on an rst
 *                                      receive otherwise odd bits of prattle
 *                                      escape still
 *              Alan Cox        :       Fixed another acking RST frame bug.
 *                                      Should stop LAN workplace lockups.
 *              Alan Cox        :       Some tidyups using the new skb list
 *                                      facilities
 *              Alan Cox        :       sk->keepopen now seems to work
 *              Alan Cox        :       Pulls options out correctly on accepts
 *              Alan Cox        :       Fixed assorted sk->rqueue->next errors
 *              Alan Cox        :       PSH doesn't end a TCP read. Switched a
 *                                      bit to skb ops.
 *              Alan Cox        :       Tidied tcp_data to avoid a potential
 *                                      nasty.
 *              Alan Cox        :       Added some better commenting, as the
 *                                      tcp is hard to follow
 *              Alan Cox        :       Removed incorrect check for 20 * psh
 *      Michael O'Reilly        :       ack < copied bug fix.
 *      Johannes Stille         :       Misc tcp fixes (not all in yet).
 *              Alan Cox        :       FIN with no memory -> CRASH
 *              Alan Cox        :       Added socket option proto entries.
 *                                      Also added awareness of them to accept.
 *              Alan Cox        :       Added TCP options (SOL_TCP)
 *             Alan Cox        :      Switched wakeup calls to callbacks,
 *                                    so the kernel can layer network
 *                                    sockets.
 *             Alan Cox        :      Use ip_tos/ip_ttl settings.
 *             Alan Cox        :      Handle FIN (more) properly (we hope).
 *             Alan Cox        :      RST frames sent on unsynchronised
 *                                    state ack error.
 *             Alan Cox        :      Put in missing check for SYN bit.
 *             Alan Cox        :      Added tcp_select_window() aka NET2E
 *                                    window non shrink trick.
 *             Alan Cox        :      Added a couple of small NET2E timer
 *                                    fixes
 *             Charles Hedrick :      TCP fixes
 *             Toomas Tamm     :      TCP window fixes
 *             Alan Cox        :      Small URG fix to rlogin ^C ack fight
 *             Charles Hedrick :      Rewrote most of it to actually work
 *             Linus           :      Rewrote tcp_read() and URG handling
 *                                    completely
 *             Gerhard Koerting:      Fixed some missing timer handling
 *             Matthew Dillon  :      Reworked TCP machine states as per RFC
 *             Gerhard Koerting:      PC/TCP workarounds
 *             Adam Caldwell   :      Assorted timer/timing errors
 *             Matthew Dillon  :      Fixed another RST bug
 *             Alan Cox        :      Move to kernel side addressing changes.
 *             Alan Cox        :      Beginning work on TCP fastpathing
 *                                    (not yet usable)
 *             Arnt Gulbrandsen:      Turbocharged tcp_check() routine.
 *             Alan Cox        :      TCP fast path debugging
 *             Alan Cox        :      Window clamping
 *             Michael Riepe   :      Bug in tcp_check()
 *             Matt Dillon     :      More TCP improvements and RST bug fixes
 *             Matt Dillon     :      Yet more small nasties remove from the
 *                                    TCP code (Be very nice to this man if
 *                                    tcp finally works 100%) 8)
 *             Alan Cox        :      BSD accept semantics.
 *             Alan Cox        :      Reset on closedown bug.
 *     Peter De Schrijver      :      ENOTCONN check missing in tcp_sendto().
 *             Michael Pall    :      Handle poll() after URG properly in
 *                                    all cases.
 *             Michael Pall    :      Undo the last fix in tcp_read_urg()
 *                                    (multi URG PUSH broke rlogin).
 *             Michael Pall    :      Fix the multi URG PUSH problem in
 *                                    tcp_readable(), poll() after URG
 *                                    works now.
 *             Michael Pall    :      recv(...,MSG_OOB) never blocks in the
 *                                    BSD api.
 *             Alan Cox        :      Changed the semantics of sk->socket to
 *                                    fix a race and a signal problem with
 *                                    accept() and async I/O.
 *             Alan Cox        :      Relaxed the rules on tcp_sendto().
 *             Yury Shevchuk   :      Really fixed accept() blocking problem.
 *             Craig I. Hagan  :      Allow for BSD compatible TIME_WAIT for
 *                                    clients/servers which listen in on
 *                                    fixed ports.
 *             Alan Cox        :      Cleaned the above up and shrank it to
 *                                    a sensible code size.
 *             Alan Cox        :      Self connect lockup fix.
 *             Alan Cox        :      No connect to multicast.
 *             Ross Biro       :      Close unaccepted children on master
 *                                    socket close.
 *             Alan Cox        :      Reset tracing code.
 *             Alan Cox        :      Spurious resets on shutdown.
 *             Alan Cox        :      Giant 15 minute/60 second timer error
 *             Alan Cox        :      Small whoops in polling before an
 *                                    accept.
 *             Alan Cox        :      Kept the state trace facility since
 *                                    it's handy for debugging.
 *             Alan Cox        :      More reset handler fixes.
 *             Alan Cox        :      Started rewriting the code based on
 *                                    the RFC's for other useful protocol
 *                                    references see: Comer, KA9Q NOS, and
 *                                    for a reference on the difference
 *                                    between specifications and how BSD
 *                                    works see the 4.4lite source.
 *             A.N.Kuznetsov   :      Don't time wait on completion of tidy
 *                                    close.
 *             Linus Torvalds  :      Fin/Shutdown & copied_seq changes.
 *             Linus Torvalds  :      Fixed BSD port reuse to work first syn
 *             Alan Cox        :      Reimplemented timers as per the RFC
 *                                    and using multiple timers for sanity.
 *             Alan Cox        :      Small bug fixes, and a lot of new
 *                                    comments.
 *             Alan Cox        :      Fixed dual reader crash by locking
 *                                    the buffers (much like datagram.c)
 *             Alan Cox        :      Fixed stuck sockets in probe. A probe
 *                                      now gets fed up of retrying without
 *                                      (even a no space) answer.
 *              Alan Cox        :       Extracted closing code better
 *              Alan Cox        :       Fixed the closing state machine to
 *                                      resemble the RFC.
 *              Alan Cox        :       More 'per spec' fixes.
 *              Jorge Cwik      :       Even faster checksumming.
 *              Alan Cox        :       tcp_data() doesn't ack illegal PSH
 *                                      only frames. At least one pc tcp stack
 *                                      generates them.
 *              Alan Cox        :       Cache last socket.
 *              Alan Cox        :       Per route irtt.
 *              Matt Day        :       poll()->select() match BSD precisely on error
 *              Alan Cox        :       New buffers
 *              Marc Tamsky     :       Various sk->prot->retransmits and
 *                                      sk->retransmits misupdating fixed.
 *                                      Fixed tcp_write_timeout: stuck close,
 *                                      and TCP syn retries gets used now.
 *              Mark Yarvis     :       In tcp_read_wakeup(), don't send an
 *                                      ack if state is TCP_CLOSED.
 *              Alan Cox        :       Look up device on a retransmit - routes may
 *                                      change. Doesn't yet cope with MSS shrink right
 *                                      but it's a start!
 *              Marc Tamsky     :       Closing in closing fixes.
 *              Mike Shaver     :       RFC1122 verifications.
 *              Alan Cox        :       rcv_saddr errors.
 *              Alan Cox        :       Block double connect().
 *              Alan Cox        :       Small hooks for enSKIP.
 *              Alexey Kuznetsov:       Path MTU discovery.
 *              Alan Cox        :       Support soft errors.
 *              Alan Cox        :       Fix MTU discovery pathological case
 *                                      when the remote claims no mtu!
 *              Marc Tamsky     :       TCP_CLOSE fix.
 *              Colin (G3TNE)   :       Send a reset on syn ack replies in
 *                                      window but wrong (fixes NT lpd problems)
 *              Pedro Roque     :       Better TCP window handling, delayed ack.
 *              Joerg Reuter    :       No modification of locked buffers in
 *                                      tcp_do_retransmit()
 *              Eric Schenk     :       Changed receiver side silly window
 *                                      avoidance algorithm to BSD style
 *                                      algorithm. This doubles throughput
 *                                      against machines running Solaris,
 *                                      and seems to result in general
 *                                      improvement.
 *      Stefan Magdalinski      :       adjusted tcp_readable() to fix FIONREAD
 *      Willy Konynenberg       :       Transparent proxying support.
 *      Mike McLagan            :       Routing by source
 *              Keith Owens     :       Do proper merging with partial SKB's in
 *                                      tcp_do_sendmsg to avoid burstiness.
 *              Eric Schenk     :       Fix fast close down bug with
 *                                      shutdown() followed by close().
 *              Andi Kleen      :       Make poll agree with SIGIO
 *      Salvatore Sanfilippo    :       Support SO_LINGER with linger == 1 and
 *                                      lingertime == 0 (RFC 793 ABORT Call)
 *      Hirokazu Takahashi      :       Use copy_from_user() instead of
 *                                      csum_and_copy_from_user() if possible.
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or(at your option) any later version.
 *
 * Description of States:
 *
 *      TCP_SYN_SENT            sent a connection request, waiting for ack
 *
 *      TCP_SYN_RECV            received a connection request, sent ack,
 *                              waiting for final ack in three-way handshake.
 *
 *      TCP_ESTABLISHED         connection established
 *
 *      TCP_FIN_WAIT1           our side has shutdown, waiting to complete
 *                              transmission of remaining buffered data
 *
 *      TCP_FIN_WAIT2           all buffered data sent, waiting for remote
 *                              to shutdown
 *
 *      TCP_CLOSING             both sides have shutdown but we still have
 *                              data we have to finish sending
 *
 *      TCP_TIME_WAIT           timeout to catch resent junk before entering
 *                              closed, can only be entered from FIN_WAIT2
 *                              or CLOSING.  Required because the other end
 *                              may not have gotten our last ACK causing it
 *                              to retransmit the data packet (which we ignore)
 *
 *      TCP_CLOSE_WAIT          remote side has shutdown and is waiting for
 *                              us to finish writing our data and to shutdown
 *                              (we have to close() to move on to LAST_ACK)
 *
 *      TCP_LAST_ACK            out side has shutdown after remote has
 *                              shutdown.  There may still be data in our
 *                              buffer that we have to finish sending
 *
 *      TCP_CLOSE               socket is finished
 */

#define pr_fmt(fmt) "TCP: " fmt

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/poll.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/skbuff.h>
#include <linux/scatterlist.h>
#include <linux/splice.h>
#include <linux/net.h>
#include <linux/socket.h>
#include <linux/random.h>
#include <linux/bootmem.h>
#include <linux/highmem.h>
#include <linux/swap.h>
#include <linux/cache.h>
#include <linux/err.h>
#include <linux/crypto.h>
#include <linux/time.h>
#include <linux/slab.h>

#include <net/icmp.h>
#include <net/inet_common.h>
#include <net/tcp.h>
#include <net/xfrm.h>
#include <net/ip.h>
#include <net/netdma.h>
#include <net/sock.h>

#include <asm/uaccess.h>
#include <asm/ioctls.h>
#include <net/busy_poll.h>

int sysctl_tcp_fin_timeout __read_mostly = TCP_FIN_TIMEOUT;

int sysctl_tcp_min_tso_segs __read_mostly = 2;

int sysctl_tcp_autocorking __read_mostly = 1;

struct percpu_counter tcp_orphan_count;
EXPORT_SYMBOL_GPL(tcp_orphan_count);

long sysctl_tcp_mem[3] __read_mostly;
int sysctl_tcp_wmem[3] __read_mostly;
int sysctl_tcp_rmem[3] __read_mostly;

EXPORT_SYMBOL(sysctl_tcp_mem);
EXPORT_SYMBOL(sysctl_tcp_rmem);
EXPORT_SYMBOL(sysctl_tcp_wmem);

atomic_long_t tcp_memory_allocated;     /* Current allocated memory. */
EXPORT_SYMBOL(tcp_memory_allocated);

/*
 * Current number of TCP sockets.
 */
struct percpu_counter tcp_sockets_allocated;
EXPORT_SYMBOL(tcp_sockets_allocated);

/*
 * TCP splice context
 */
struct tcp_splice_state {
        struct pipe_inode_info *pipe;
        size_t len;
        unsigned int flags;
};

/*
 * Pressure flag: try to collapse.
 * Technical note: it is used by multiple contexts non atomically.
 * All the __sk_mem_schedule() is of this nature: accounting
 * is strict, actions are advisory and have some latency.
 */
int tcp_memory_pressure __read_mostly;
EXPORT_SYMBOL(tcp_memory_pressure);

void tcp_enter_memory_pressure(struct sock *sk)
{
        if (!tcp_memory_pressure) {
                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPMEMORYPRESSURES);
                tcp_memory_pressure = 1;
        }
}
EXPORT_SYMBOL(tcp_enter_memory_pressure);

/* Convert seconds to retransmits based on initial and max timeout */
static u8 secs_to_retrans(int seconds, int timeout, int rto_max)
{
        u8 res = 0;

        if (seconds > 0) {
                int period = timeout;

                res = 1;
                while (seconds > period && res < 255) {
                        res++;
                        timeout <<= 1;
                        if (timeout > rto_max)
                                timeout = rto_max;
                        period += timeout;
                }
        }
        return res;
}

/* Convert retransmits to seconds based on initial and max timeout */
static int retrans_to_secs(u8 retrans, int timeout, int rto_max)
{
        int period = 0;

        if (retrans > 0) {
                period = timeout;
                while (--retrans) {
                        timeout <<= 1;
                        if (timeout > rto_max)
                                timeout = rto_max;
                        period += timeout;
                }
        }
        return period;
}

/* Address-family independent initialization for a tcp_sock.
 *
 * NOTE: A lot of things set to zero explicitly by call to
 *       sk_alloc() so need not be done here.
 */
void tcp_init_sock(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);

        __skb_queue_head_init(&tp->out_of_order_queue);
        tcp_init_xmit_timers(sk);
        tcp_prequeue_init(tp);
        INIT_LIST_HEAD(&tp->tsq_node);

        icsk->icsk_rto = TCP_TIMEOUT_INIT;
        tp->mdev_us = jiffies_to_usecs(TCP_TIMEOUT_INIT);

        /* So many TCP implementations out there (incorrectly) count the
         * initial SYN frame in their delayed-ACK and congestion control
         * algorithms that we must have the following bandaid to talk
         * efficiently to them.  -DaveM
         */
        tp->snd_cwnd = TCP_INIT_CWND;

        /* See draft-stevens-tcpca-spec-01 for discussion of the
         * initialization of these values.
         */
        tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
        tp->snd_cwnd_clamp = ~0;
        tp->mss_cache = TCP_MSS_DEFAULT;

        tp->reordering = sysctl_tcp_reordering;
        tcp_enable_early_retrans(tp);
        icsk->icsk_ca_ops = &tcp_init_congestion_ops;

        tp->tsoffset = 0;

        sk->sk_state = TCP_CLOSE;

        sk->sk_write_space = sk_stream_write_space;
        sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);

        icsk->icsk_sync_mss = tcp_sync_mss;

        sk->sk_sndbuf = sysctl_tcp_wmem[1];
        sk->sk_rcvbuf = sysctl_tcp_rmem[1];

        local_bh_disable();
        sock_update_memcg(sk);
        sk_sockets_allocated_inc(sk);
        local_bh_enable();
}
EXPORT_SYMBOL(tcp_init_sock);

static void tcp_tx_timestamp(struct sock *sk, struct sk_buff *skb)
{
        if (sk->sk_tsflags) {
                struct skb_shared_info *shinfo = skb_shinfo(skb);

                sock_tx_timestamp(sk, &shinfo->tx_flags);
                if (shinfo->tx_flags & SKBTX_ANY_TSTAMP)
                        shinfo->tskey = TCP_SKB_CB(skb)->seq + skb->len - 1;
        }
}

/*
 *      Wait for a TCP event.
 *
 *      Note that we don't need to lock the socket, as the upper poll layers
 *      take care of normal races (between the test and the event) and we don't
 *      go look at any of the socket buffers directly.
 */
unsigned int tcp_poll(struct file *file, struct socket *sock, poll_table *wait)
{
        unsigned int mask;
        struct sock *sk = sock->sk;
        const struct tcp_sock *tp = tcp_sk(sk);

        sock_rps_record_flow(sk);

        sock_poll_wait(file, sk_sleep(sk), wait);
        if (sk->sk_state == TCP_LISTEN)
                return inet_csk_listen_poll(sk);

        /* Socket is not locked. We are protected from async events
         * by poll logic and correct handling of state changes
         * made by other threads is impossible in any case.
         */

        mask = 0;

        /*
         * POLLHUP is certainly not done right. But poll() doesn't
         * have a notion of HUP in just one direction, and for a
         * socket the read side is more interesting.
         *
         * Some poll() documentation says that POLLHUP is incompatible
         * with the POLLOUT/POLLWR flags, so somebody should check this
         * all. But careful, it tends to be safer to return too many
         * bits than too few, and you can easily break real applications
         * if you don't tell them that something has hung up!
         *
         * Check-me.
         *
         * Check number 1. POLLHUP is _UNMASKABLE_ event (see UNIX98 and
         * our fs/select.c). It means that after we received EOF,
         * poll always returns immediately, making impossible poll() on write()
         * in state CLOSE_WAIT. One solution is evident --- to set POLLHUP
         * if and only if shutdown has been made in both directions.
         * Actually, it is interesting to look how Solaris and DUX
         * solve this dilemma. I would prefer, if POLLHUP were maskable,
         * then we could set it on SND_SHUTDOWN. BTW examples given
         * in Stevens' books assume exactly this behaviour, it explains
         * why POLLHUP is incompatible with POLLOUT.    --ANK
         *
         * NOTE. Check for TCP_CLOSE is added. The goal is to prevent
         * blocking on fresh not-connected or disconnected socket. --ANK
         */
        if (sk->sk_shutdown == SHUTDOWN_MASK || sk->sk_state == TCP_CLOSE)
                mask |= POLLHUP;
        if (sk->sk_shutdown & RCV_SHUTDOWN)
                mask |= POLLIN | POLLRDNORM | POLLRDHUP;

        /* Connected or passive Fast Open socket? */
        if (sk->sk_state != TCP_SYN_SENT &&
            (sk->sk_state != TCP_SYN_RECV || tp->fastopen_rsk != NULL)) {
                int target = sock_rcvlowat(sk, 0, INT_MAX);

                if (tp->urg_seq == tp->copied_seq &&
                    !sock_flag(sk, SOCK_URGINLINE) &&
                    tp->urg_data)
                        target++;

                /* Potential race condition. If read of tp below will
                 * escape above sk->sk_state, we can be illegally awaken
                 * in SYN_* states. */
                if (tp->rcv_nxt - tp->copied_seq >= target)
                        mask |= POLLIN | POLLRDNORM;

                if (!(sk->sk_shutdown & SEND_SHUTDOWN)) {
                        if (sk_stream_is_writeable(sk)) {
                                mask |= POLLOUT | POLLWRNORM;
                        } else {  /* send SIGIO later */
                                set_bit(SOCK_ASYNC_NOSPACE,
                                        &sk->sk_socket->flags);
                                set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);

                                /* Race breaker. If space is freed after
                                 * wspace test but before the flags are set,
                                 * IO signal will be lost.
                                 */
                                if (sk_stream_is_writeable(sk))
                                        mask |= POLLOUT | POLLWRNORM;
                        }
                } else
                        mask |= POLLOUT | POLLWRNORM;

                if (tp->urg_data & TCP_URG_VALID)
                        mask |= POLLPRI;
        }
        /* This barrier is coupled with smp_wmb() in tcp_reset() */
        smp_rmb();
        if (sk->sk_err || !skb_queue_empty(&sk->sk_error_queue))
                mask |= POLLERR;

        return mask;
}
EXPORT_SYMBOL(tcp_poll);

int tcp_ioctl(struct sock *sk, int cmd, unsigned long arg)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int answ;
        bool slow;

        switch (cmd) {
        case SIOCINQ:
                if (sk->sk_state == TCP_LISTEN)
                        return -EINVAL;

                slow = lock_sock_fast(sk);
                if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))
                        answ = 0;
                else if (sock_flag(sk, SOCK_URGINLINE) ||
                         !tp->urg_data ||
                         before(tp->urg_seq, tp->copied_seq) ||
                         !before(tp->urg_seq, tp->rcv_nxt)) {

                        answ = tp->rcv_nxt - tp->copied_seq;

                        /* Subtract 1, if FIN was received */
                        if (answ && sock_flag(sk, SOCK_DONE))
                                answ--;
                } else
                        answ = tp->urg_seq - tp->copied_seq;
                unlock_sock_fast(sk, slow);
                break;
        case SIOCATMARK:
                answ = tp->urg_data && tp->urg_seq == tp->copied_seq;
                break;
        case SIOCOUTQ:
                if (sk->sk_state == TCP_LISTEN)
                        return -EINVAL;

                if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))
                        answ = 0;
                else
                        answ = tp->write_seq - tp->snd_una;
                break;
        case SIOCOUTQNSD:
                if (sk->sk_state == TCP_LISTEN)
                        return -EINVAL;

                if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))
                        answ = 0;
                else
                        answ = tp->write_seq - tp->snd_nxt;
                break;
        default:
                return -ENOIOCTLCMD;
        }

        return put_user(answ, (int __user *)arg);
}
EXPORT_SYMBOL(tcp_ioctl);

static inline void tcp_mark_push(struct tcp_sock *tp, struct sk_buff *skb)
{
        TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_PSH;
        tp->pushed_seq = tp->write_seq;
}

static inline bool forced_push(const struct tcp_sock *tp)
{
        return after(tp->write_seq, tp->pushed_seq + (tp->max_window >> 1));
}

static inline void skb_entail(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);

        skb->csum    = 0;
        tcb->seq     = tcb->end_seq = tp->write_seq;
        tcb->tcp_flags = TCPHDR_ACK;
        tcb->sacked  = 0;
        skb_header_release(skb);
        tcp_add_write_queue_tail(sk, skb);
        sk->sk_wmem_queued += skb->truesize;
        sk_mem_charge(sk, skb->truesize);
        if (tp->nonagle & TCP_NAGLE_PUSH)
                tp->nonagle &= ~TCP_NAGLE_PUSH;
}

static inline void tcp_mark_urg(struct tcp_sock *tp, int flags)
{
        if (flags & MSG_OOB)
                tp->snd_up = tp->write_seq;
}

/* If a not yet filled skb is pushed, do not send it if
 * we have data packets in Qdisc or NIC queues :
 * Because TX completion will happen shortly, it gives a chance
 * to coalesce future sendmsg() payload into this skb, without
 * need for a timer, and with no latency trade off.
 * As packets containing data payload have a bigger truesize
 * than pure acks (dataless) packets, the last checks prevent
 * autocorking if we only have an ACK in Qdisc/NIC queues,
 * or if TX completion was delayed after we processed ACK packet.
 */
static bool tcp_should_autocork(struct sock *sk, struct sk_buff *skb,
                                int size_goal)
{
        return skb->len < size_goal &&
               sysctl_tcp_autocorking &&
               skb != tcp_write_queue_head(sk) &&
               atomic_read(&sk->sk_wmem_alloc) > skb->truesize;
}

static void tcp_push(struct sock *sk, int flags, int mss_now,
                     int nonagle, int size_goal)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;

        if (!tcp_send_head(sk))
```

```
                return;

        skb = tcp_write_queue_tail(sk);
        if (!(flags & MSG_MORE) || forced_push(tp))
                tcp_mark_push(tp, skb);

        tcp_mark_urg(tp, flags);

        if (tcp_should_autocork(sk, skb, size_goal)) {

                /* avoid atomic op if TSQ_THROTTLED bit is already set */
                if (!test_bit(TSQ_THROTTLED, &tp->tsq_flags)) {
                        NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPAUTOCORKING);
                        set_bit(TSQ_THROTTLED, &tp->tsq_flags);
                }
                /* It is possible TX completion already happened
                 * before we set TSQ_THROTTLED.
                 */
                if (atomic_read(&sk->sk_wmem_alloc) > skb->truesize)
                        return;
        }

        if (flags & MSG_MORE)
                nonagle = TCP_NAGLE_CORK;

        __tcp_push_pending_frames(sk, mss_now, nonagle);
}

static int tcp_splice_data_recv(read_descriptor_t *rd_desc, struct sk_buff *skb,
                                unsigned int offset, size_t len)
{
        struct tcp_splice_state *tss = rd_desc->arg.data;
        int ret;

        ret = skb_splice_bits(skb, offset, tss->pipe, min(rd_desc->count, len),
                              tss->flags);
        if (ret > 0)
                rd_desc->count -= ret;
        return ret;
}

static int __tcp_splice_read(struct sock *sk, struct tcp_splice_state *tss)
{
        /* Store TCP splice context information in read_descriptor_t. */
        read_descriptor_t rd_desc = {
                .arg.data = tss,
                .count    = tss->len,
        };

        return tcp_read_sock(sk, &rd_desc, tcp_splice_data_recv);
}

/**
 *  tcp_splice_read - splice data from TCP socket to a pipe
 * @sock:       socket to splice from
 * @ppos:       position (not valid)
 * @pipe:       pipe to splice to
 * @len:        number of bytes to splice
 * @flags:      splice modifier flags
 *
 * Description:
 *    Will read pages from given socket and fill them into a pipe.
 *
 **/
ssize_t tcp_splice_read(struct socket *sock, loff_t *ppos,
                        struct pipe_inode_info *pipe, size_t len,
                        unsigned int flags)
{
        struct sock *sk = sock->sk;
        struct tcp_splice_state tss = {
                .pipe = pipe,
                .len = len,
                .flags = flags,
        };
        long timeo;
        ssize_t spliced;
        int ret;

        sock_rps_record_flow(sk);
        /*
         * We can't seek on a socket input
         */
        if (unlikely(*ppos))
                return -ESPIPE;

        ret = spliced = 0;

        lock_sock(sk);

        timeo = sock_rcvtimeo(sk, sock->file->f_flags & O_NONBLOCK);
        while (tss.len) {
                ret = __tcp_splice_read(sk, &tss);
                if (ret < 0)
                        break;
                else if (!ret) {
                        if (spliced)
                                break;
                        if (sock_flag(sk, SOCK_DONE))
                                break;
                        if (sk->sk_err) {
                                ret = sock_error(sk);
                                break;
                        }
                        if (sk->sk_shutdown & RCV_SHUTDOWN)
                                break;
                        if (sk->sk_state == TCP_CLOSE) {
                                /*
                                 * This occurs when user tries to read
                                 * from never connected socket.
                                 */
                                if (!sock_flag(sk, SOCK_DONE))
                                        ret = -ENOTCONN;
                                break;
                        }
                        if (!timeo) {
                                ret = -EAGAIN;
                                break;
                        }
                        sk_wait_data(sk, &timeo);
                        if (signal_pending(current)) {
                                ret = sock_intr_errno(timeo);
                                break;
                        }
                        continue;
                }
                tss.len -= ret;
                spliced += ret;

                if (!timeo)
                        break;
                release_sock(sk);
                lock_sock(sk);

                if (sk->sk_err || sk->sk_state == TCP_CLOSE ||
                    (sk->sk_shutdown & RCV_SHUTDOWN) ||
                    signal_pending(current))
                        break;
        }

        release_sock(sk);

        if (spliced)
                return spliced;

        return ret;
}
EXPORT_SYMBOL(tcp_splice_read);

struct sk_buff *sk_stream_alloc_skb(struct sock *sk, int size, gfp_t gfp)
{
        struct sk_buff *skb;

        /* The TCP header must be at least 32-bit aligned.  */
        size = ALIGN(size, 4);

        skb = alloc_skb_fclone(size + sk->sk_prot->max_header, gfp);
        if (skb) {
                if (sk_wmem_schedule(sk, skb->truesize)) {
                        skb_reserve(skb, sk->sk_prot->max_header);
                        /*
                         * Make sure that we have exactly size bytes
                         * available to the caller, no more, no less.
                         */
                        skb->reserved_tailroom = skb->end - skb->tail - size;
                        return skb;
                }
                __kfree_skb(skb);
        } else {
                sk->sk_prot->enter_memory_pressure(sk);
                sk_stream_moderate_sndbuf(sk);
        }
        return NULL;
}

static unsigned int tcp_xmit_size_goal(struct sock *sk, u32 mss_now,
                                       int large_allowed)
{
        struct tcp_sock *tp = tcp_sk(sk);
        u32 xmit_size_goal, old_size_goal;

        xmit_size_goal = mss_now;

        if (large_allowed && sk_can_gso(sk)) {
                u32 gso_size, hlen;

                /* Maybe we should/could use sk->sk_prot->max_header here ? */
                hlen = inet_csk(sk)->icsk_af_ops->net_header_len +
                       inet_csk(sk)->icsk_ext_hdr_len +
                       tp->tcp_header_len;

                /* Goal is to send at least one packet per ms,
                 * not one big TSO packet every 100 ms.
                 * This preserves ACK clocking and is consistent
                 * with tcp_tso_should_defer() heuristic.
                 */
                gso_size = sk->sk_pacing_rate / (2 * MSEC_PER_SEC);
                gso_size = max_t(u32, gso_size,
                                 sysctl_tcp_min_tso_segs * mss_now);

                xmit_size_goal = min_t(u32, gso_size,
                                       sk->sk_gso_max_size - 1 - hlen);

                xmit_size_goal = tcp_bound_to_half_wnd(tp, xmit_size_goal);

                /* We try hard to avoid divides here */
                old_size_goal = tp->xmit_size_goal_segs * mss_now;

                if (likely(old_size_goal <= xmit_size_goal &&
                           old_size_goal + mss_now > xmit_size_goal)) {
                        xmit_size_goal = old_size_goal;
                } else {
                        tp->xmit_size_goal_segs =
                                min_t(u16, xmit_size_goal / mss_now,
                                      sk->sk_gso_max_segs);
                        xmit_size_goal = tp->xmit_size_goal_segs * mss_now;
                }
        }

        return max(xmit_size_goal, mss_now);
}

static int tcp_send_mss(struct sock *sk, int *size_goal, int flags)
{
        int mss_now;

        mss_now = tcp_current_mss(sk);
        *size_goal = tcp_xmit_size_goal(sk, mss_now, !(flags & MSG_OOB));

        return mss_now;
}

static ssize_t do_tcp_sendpages(struct sock *sk, struct page *page, int offset,
                                size_t size, int flags)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int mss_now, size_goal;
        int err;
        ssize_t copied;
        long timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);

        /* Wait for a connection to finish. One exception is TCP Fast Open
         * (passive side) where data is allowed to be sent before a connection
         * is fully established.
         */
        if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
            !tcp_passive_fastopen(sk)) {
                if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
                        goto out_err;
        }

        clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);

        mss_now = tcp_send_mss(sk, &size_goal, flags);
        copied = 0;

        err = -EPIPE;
        if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
                goto out_err;

        while (size > 0) {
                struct sk_buff *skb = tcp_write_queue_tail(sk);
                int copy, i;
                bool can_coalesce;

                if (!tcp_send_head(sk) || (copy = size_goal - skb->len) <= 0) {
new_segment:
                        if (!sk_stream_memory_free(sk))
                                goto wait_for_sndbuf;

                        skb = sk_stream_alloc_skb(sk, 0, sk->sk_allocation);
                        if (!skb)
                                goto wait_for_memory;

                        skb_entail(sk, skb);
                        copy = size_goal;
                }

                if (copy > size)
                        copy = size;

                i = skb_shinfo(skb)->nr_frags;
                can_coalesce = skb_can_coalesce(skb, i, page, offset);
                if (!can_coalesce && i >= MAX_SKB_FRAGS) {
                        tcp_mark_push(tp, skb);
                        goto new_segment;
                }
                if (!sk_wmem_schedule(sk, copy))
                        goto wait_for_memory;

                if (can_coalesce) {
                        skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
                } else {
                        get_page(page);
                        skb_fill_page_desc(skb, i, page, offset, copy);
                }
                skb_shinfo(skb)->tx_flags |= SKBTX_SHARED_FRAG;

                skb->len += copy;
                skb->data_len += copy;
                skb->truesize += copy;
                sk->sk_wmem_queued += copy;
                sk_mem_charge(sk, copy);
                skb->ip_summed = CHECKSUM_PARTIAL;
                tp->write_seq += copy;
                TCP_SKB_CB(skb)->end_seq += copy;
                skb_shinfo(skb)->gso_segs = 0;

                if (!copied)
                        TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;

                copied += copy;
                offset += copy;
                if (!(size -= copy)) {
                        tcp_tx_timestamp(sk, skb);
                        goto out;
                }

                if (skb->len < size_goal || (flags & MSG_OOB))
                        continue;

                if (forced_push(tp)) {
                        tcp_mark_push(tp, skb);
                        __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
                } else if (skb == tcp_send_head(sk))
                        tcp_push_one(sk, mss_now);
                continue;

wait_for_sndbuf:
                set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
wait_for_memory:
                tcp_push(sk, flags & ~MSG_MORE, mss_now,
                         TCP_NAGLE_PUSH, size_goal);

                if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
                        goto do_error;

                mss_now = tcp_send_mss(sk, &size_goal, flags);
        }

out:
         if (copied && !(flags & MSG_SENDPAGE_NOTLAST))
                 tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
         return copied;

 do_error:
         if (copied)
                 goto out;
 out_err:
         return sk_stream_error(sk, flags, err);
 }

 int tcp_sendpage(struct sock *sk, struct page *page, int offset,
                  size_t size, int flags)
 {
         ssize_t res;

         if (!(sk->sk_route_caps & NETIF_F_SG) ||
             !(sk->sk_route_caps & NETIF_F_ALL_CSUM))
                 return sock_no_sendpage(sk->sk_socket, page, offset, size,
                                         flags);

         lock_sock(sk);
         res = do_tcp_sendpages(sk, page, offset, size, flags);
         release_sock(sk);
         return res;
 }
 EXPORT_SYMBOL(tcp_sendpage);

 static inline int select_size(const struct sock *sk, bool sg)
 {
         const struct tcp_sock *tp = tcp_sk(sk);
         int tmp = tp->mss_cache;

         if (sg) {
                 if (sk_can_gso(sk)) {
                         /* Small frames wont use a full page:
                          * Payload will immediately follow tcp header.
                          */
                         tmp = SKB_WITH_OVERHEAD(2048 - MAX_TCP_HEADER);
                 } else {
                         int pgbreak = SKB_MAX_HEAD(MAX_TCP_HEADER);

                         if (tmp >= pgbreak &&
                             tmp <= pgbreak + (MAX_SKB_FRAGS - 1) * PAGE_SIZE)
                                 tmp = pgbreak;
                 }
         }

         return tmp;
 }

 void tcp_free_fastopen_req(struct tcp_sock *tp)
 {
         if (tp->fastopen_req != NULL) {
                 kfree(tp->fastopen_req);
                 tp->fastopen_req = NULL;
         }
 }

 static int tcp_sendmsg_fastopen(struct sock *sk, struct msghdr *msg,
                                 int *copied, size_t size)
 {
         struct tcp_sock *tp = tcp_sk(sk);
         int err, flags;

         if (!(sysctl_tcp_fastopen & TFO_CLIENT_ENABLE))
                 return -EOPNOTSUPP;
         if (tp->fastopen_req != NULL)
                 return -EALREADY; /* Another Fast Open is in progress */

         tp->fastopen_req = kzalloc(sizeof(struct tcp_fastopen_request),
                                    sk->sk_allocation);
         if (unlikely(tp->fastopen_req == NULL))
                 return -ENOBUFS;
         tp->fastopen_req->data = msg;
         tp->fastopen_req->size = size;

         flags = (msg->msg_flags & MSG_DONTWAIT) ? O_NONBLOCK : 0;
         err = __inet_stream_connect(sk->sk_socket, msg->msg_name,
                                     msg->msg_namelen, flags);
         *copied = tp->fastopen_req->copied;
         tcp_free_fastopen_req(tp);
         return err;
 }
```

```
int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t size)
{
        struct iovec *iov;
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;
        int iovlen, flags, err, copied = 0;
        int mss_now = 0, size_goal, copied_syn = 0, offset = 0;
        bool sg;
        long timeo;

        lock_sock(sk);

        flags = msg->msg_flags;
        if (flags & MSG_FASTOPEN) {
                err = tcp_sendmsg_fastopen(sk, msg, &copied_syn, size);
                if (err == -EINPROGRESS && copied_syn > 0)
                        goto out;
                else if (err)
                        goto out_err;
                offset = copied_syn;
        }

        timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);

        /* Wait for a connection to finish. One exception is TCP Fast Open
         * (passive side) where data is allowed to be sent before a connection
         * is fully established.
         */
        if (((1 << sk->sk_state) & ~(TCPF_ESTABLISHED | TCPF_CLOSE_WAIT)) &&
            !tcp_passive_fastopen(sk)) {
                if ((err = sk_stream_wait_connect(sk, &timeo)) != 0)
                        goto do_error;
        }

        if (unlikely(tp->repair)) {
                if (tp->repair_queue == TCP_RECV_QUEUE) {
                        copied = tcp_send_rcvq(sk, msg, size);
                        goto out_nopush;
                }

                err = -EINVAL;
                if (tp->repair_queue == TCP_NO_QUEUE)
                        goto out_err;

                /* 'common' sending to sendq */
        }

        /* This should be in poll */
        clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);

        mss_now = tcp_send_mss(sk, &size_goal, flags);

        /* Ok commence sending. */
        iovlen = msg->msg_iovlen;
        iov = msg->msg_iov;
        copied = 0;

        err = -EPIPE;
        if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
                goto out_err;

        sg = !!(sk->sk_route_caps & NETIF_F_SG);

        while (--iovlen >= 0) {
                size_t seglen = iov->iov_len;
                unsigned char __user *from = iov->iov_base;

                iov++;
                if (unlikely(offset > 0)) {  /* Skip bytes copied in SYN */
                        if (offset >= seglen) {
                                offset -= seglen;
                                continue;
                        }
                        seglen -= offset;
                        from += offset;
                        offset = 0;
                }

                while (seglen > 0) {
                        int copy = 0;
                        int max = size_goal;

                        skb = tcp_write_queue_tail(sk);
                        if (tcp_send_head(sk)) {
                                if (skb->ip_summed == CHECKSUM_NONE)
                                        max = mss_now;
                                copy = max - skb->len;
                        }

                        if (copy <= 0) {
new_segment:
                                /* Allocate new segment. If the interface is SG,
                                 * allocate skb fitting to single page.
                                 */
                                if (!sk_stream_memory_free(sk))
                                        goto wait_for_sndbuf;

                                skb = sk_stream_alloc_skb(sk,
                                                          select_size(sk, sg),
                                                          sk->sk_allocation);
                                if (!skb)
                                        goto wait_for_memory;

                                /*
                                 * Check whether we can use HW checksum.
                                 */
                                if (sk->sk_route_caps & NETIF_F_ALL_CSUM)
                                        skb->ip_summed = CHECKSUM_PARTIAL;

                                skb_entail(sk, skb);
                                copy = size_goal;
                                max = size_goal;

                                /* All packets are restored as if they have
                                 * already been sent. skb_mstamp isn't set to
                                 * avoid wrong rtt estimation.
                                 */
                                if (tp->repair)
                                        TCP_SKB_CB(skb)->sacked |= TCPCB_REPAIRED;
                        }

                        /* Try to append data to the end of skb. */
                        if (copy > seglen)
                                copy = seglen;

                        /* Where to copy to? */
                        if (skb_availroom(skb) > 0) {
                                /* We have some space in skb head. Superb! */
                                copy = min_t(int, copy, skb_availroom(skb));
                                err = skb_add_data_nocache(sk, skb, from, copy);
                                if (err)
                                        goto do_fault;
                        } else {
                                bool merge = true;
                                int i = skb_shinfo(skb)->nr_frags;
                                struct page_frag *pfrag = sk_page_frag(sk);

                                if (!sk_page_frag_refill(sk, pfrag))
                                        goto wait_for_memory;

                                if (!skb_can_coalesce(skb, i, pfrag->page,
                                                      pfrag->offset)) {
                                        if (i == MAX_SKB_FRAGS || !sg) {
                                                tcp_mark_push(tp, skb);
                                                goto new_segment;
                                        }
                                        merge = false;
                                }

                                copy = min_t(int, copy, pfrag->size - pfrag->offset);

                                if (!sk_wmem_schedule(sk, copy))
                                        goto wait_for_memory;

                                err = skb_copy_to_page_nocache(sk, from, skb,
                                                               pfrag->page,
                                                               pfrag->offset,
                                                               copy);
                                if (err)
                                        goto do_error;

                                /* Update the skb. */
                                if (merge) {
                                        skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
                                } else {
                                        skb_fill_page_desc(skb, i, pfrag->page,
                                                           pfrag->offset, copy);
                                        get_page(pfrag->page);
                                }
                                pfrag->offset += copy;
                        }

                        if (!copied)
                                TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH;

                        tp->write_seq += copy;
                        TCP_SKB_CB(skb)->end_seq += copy;
                        skb_shinfo(skb)->gso_segs = 0;

                        from += copy;
                        copied += copy;
                        if ((seglen -= copy) == 0 && iovlen == 0) {
                                tcp_tx_timestamp(sk, skb);
                                goto out;
                        }

                        if (skb->len < max || (flags & MSG_OOB) || unlikely(tp->repair))
                                continue;

                        if (forced_push(tp)) {
                                tcp_mark_push(tp, skb);
                                __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH);
                        } else if (skb == tcp_send_head(sk))
                                tcp_push_one(sk, mss_now);
                        continue;

wait_for_sndbuf:
                        set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
wait_for_memory:
                        if (copied)
                                tcp_push(sk, flags & ~MSG_MORE, mss_now,
                                         TCP_NAGLE_PUSH, size_goal);

                        if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
                                goto do_error;

                        mss_now = tcp_send_mss(sk, &size_goal, flags);
                }
        }

out:
        if (copied)
                tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
out_nopush:
        release_sock(sk);
        return copied + copied_syn;

do_fault:
        if (!skb->len) {
                tcp_unlink_write_queue(skb, sk);
                /* It is the one place in all of TCP, except connection
                 * reset, where we can be unlinking the send_head.
                 */
                tcp_check_send_head(sk, skb);
                sk_wmem_free_skb(sk, skb);
        }

do_error:
        if (copied + copied_syn)
                goto out;
out_err:
        err = sk_stream_error(sk, flags, err);
        release_sock(sk);
        return err;
}
EXPORT_SYMBOL(tcp_sendmsg);

/*
 *      Handle reading urgent data. BSD has very simple semantics for
 *      this, no blocking and very strange errors 8)
 */

static int tcp_recv_urg(struct sock *sk, struct msghdr *msg, int len, int flags)
{
        struct tcp_sock *tp = tcp_sk(sk);

        /* No URG data to read. */
        if (sock_flag(sk, SOCK_URGINLINE) || !tp->urg_data ||
            tp->urg_data == TCP_URG_READ)
                return -EINVAL; /* Yes this is right ! */

        if (sk->sk_state == TCP_CLOSE && !sock_flag(sk, SOCK_DONE))
                return -ENOTCONN;

        if (tp->urg_data & TCP_URG_VALID) {
                int err = 0;
                char c = tp->urg_data;

                if (!(flags & MSG_PEEK))
                        tp->urg_data = TCP_URG_READ;

                /* Read urgent data. */
                msg->msg_flags |= MSG_OOB;

                if (len > 0) {
                        if (!(flags & MSG_TRUNC))
                                err = memcpy_toiovec(msg->msg_iov, &c, 1);
                        len = 1;
                } else
                        msg->msg_flags |= MSG_TRUNC;

                return err ? -EFAULT : len;
        }

        if (sk->sk_state == TCP_CLOSE || (sk->sk_shutdown & RCV_SHUTDOWN))
                return 0;

        /* Fixed the recv(..., MSG_OOB) behaviour.  BSD docs and
         * the available implementations agree in this case:
         * this call should never block, independent of the
         * blocking state of the socket.
         * Mike <pall@rz.uni-karlsruhe.de>
         */
        return -EAGAIN;
}

static int tcp_peek_sndq(struct sock *sk, struct msghdr *msg, int len)
{
        struct sk_buff *skb;
        int copied = 0, err = 0;

        /* XXX -- need to support SO_PEEK_OFF */

        skb_queue_walk(&sk->sk_write_queue, skb) {
                err = skb_copy_datagram_iovec(skb, 0, msg->msg_iov, skb->len);
                if (err)
                        break;

                copied += skb->len;
        }

        return err ?: copied;
}

/* Clean up the receive buffer for full frames taken by the user,
 * then send an ACK if necessary.  COPIED is the number of bytes
 * tcp_recvmsg has given to the user so far, it speeds up the
 * calculation of whether or not we must ACK for the sake of
 * a window update.
 */
void tcp_cleanup_rbuf(struct sock *sk, int copied)
{
        struct tcp_sock *tp = tcp_sk(sk);
        bool time_to_ack = false;

        struct sk_buff *skb = skb_peek(&sk->sk_receive_queue);

        WARN(skb && !before(tp->copied_seq, TCP_SKB_CB(skb)->end_seq),
             "cleanup rbuf bug: copied %X seq %X rcvnxt %X\n",
             tp->copied_seq, TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt);

        if (inet_csk_ack_scheduled(sk)) {
                const struct inet_connection_sock *icsk = inet_csk(sk);
                   /* Delayed ACKs frequently hit locked sockets during bulk
                    * receive. */
                if (icsk->icsk_ack.blocked ||
                    /* Once-per-two-segments ACK was not sent by tcp_input.c */
                    tp->rcv_nxt - tp->rcv_wup > icsk->icsk_ack.rcv_mss ||
                    /*
                     * If this read emptied read buffer, we send ACK, if
                     * connection is not bidirectional, user drained
                     * receive buffer and there was a small segment
                     * in queue.
                     */
                    (copied > 0 &&
                     ((icsk->icsk_ack.pending & ICSK_ACK_PUSHED2) ||
                      ((icsk->icsk_ack.pending & ICSK_ACK_PUSHED) &&
                       !icsk->icsk_ack.pingpong)) &&
                      !atomic_read(&sk->sk_rmem_alloc)))
```

```
                        time_to_ack = true;
        }

        /* We send an ACK if we can now advertise a non-zero window
         * which has been raised "significantly".
         *
         * Even if window raised up to infinity, do not send window open ACK
         * in states, where we will not receive more. It is useless.
         */
        if (copied > 0 && !time_to_ack && !(sk->sk_shutdown & RCV_SHUTDOWN)) {
                __u32 rcv_window_now = tcp_receive_window(tp);

                /* Optimize, __tcp_select_window() is not cheap. */
                if (2*rcv_window_now <= tp->window_clamp) {
                        __u32 new_window = __tcp_select_window(sk);

                        /* Send ACK now, if this read freed lots of space
                         * in our buffer. Certainly, new_window is new window.
                         * We can advertise it now, if it is not less than current one.
                         * "Lots" means "at least twice" here.
                         */
                        if (new_window && new_window >= 2 * rcv_window_now)
                                time_to_ack = true;
                }
        }
        if (time_to_ack)
                tcp_send_ack(sk);
}

static void tcp_prequeue_process(struct sock *sk)
{
        struct sk_buff *skb;
        struct tcp_sock *tp = tcp_sk(sk);

        NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPPREQUEUED);

        /* RX process wants to run with disabled BHs, though it is not
         * necessary */
        local_bh_disable();
        while ((skb = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)
                sk_backlog_rcv(sk, skb);
        local_bh_enable();

        /* Clear memory counter. */
        tp->ucopy.memory = 0;
}

#ifdef CONFIG_NET_DMA
static void tcp_service_net_dma(struct sock *sk, bool wait)
{
        dma_cookie_t done, used;
        dma_cookie_t last_issued;
        struct tcp_sock *tp = tcp_sk(sk);

        if (!tp->ucopy.dma_chan)
                return;

        last_issued = tp->ucopy.dma_cookie;
        dma_async_issue_pending(tp->ucopy.dma_chan);

        do {
                if (dma_async_is_tx_complete(tp->ucopy.dma_chan,
                                              last_issued, &done,
                                              &used) == DMA_COMPLETE) {
                        /* Safe to free early-copied skbs now */
                        __skb_queue_purge(&sk->sk_async_wait_queue);
                        break;
                } else {
                        struct sk_buff *skb;
                        while ((skb = skb_peek(&sk->sk_async_wait_queue)) &&
                               (dma_async_is_complete(skb->dma_cookie, done,
                                                      used) == DMA_COMPLETE)) {
                                __skb_dequeue(&sk->sk_async_wait_queue);
                                kfree_skb(skb);
                        }
                }
        } while (wait);
}
#endif

static struct sk_buff *tcp_recv_skb(struct sock *sk, u32 seq, u32 *off)
{
        struct sk_buff *skb;
        u32 offset;

        while ((skb = skb_peek(&sk->sk_receive_queue)) != NULL) {
                offset = seq - TCP_SKB_CB(skb)->seq;
                if (tcp_hdr(skb)->syn)
                        offset--;
                if (offset < skb->len || tcp_hdr(skb)->fin) {
                        *off = offset;
                        return skb;
                }
                /* This looks weird, but this can happen if TCP collapsing
                 * splitted a fat GRO packet, while we released socket lock
                 * in skb_splice_bits()
                 */
                sk_eat_skb(sk, skb, false);
        }
        return NULL;
}

/*
 * This routine provides an alternative to tcp_recvmsg() for routines
 * that would like to handle copying from skbuffs directly in 'sendfile'
 * fashion.
 * Note:
 *      - It is assumed that the socket was locked by the caller.
 *      - The routine does not block.
 *      - At present, there is no support for reading OOB data
 *        or for 'peeking' the socket using this routine
 *        (although both would be easy to implement).
 */
int tcp_read_sock(struct sock *sk, read_descriptor_t *desc,
                  sk_read_actor_t recv_actor)
{
        struct sk_buff *skb;
        struct tcp_sock *tp = tcp_sk(sk);
        u32 seq = tp->copied_seq;
        u32 offset;
        int copied = 0;

        if (sk->sk_state == TCP_LISTEN)
                return -ENOTCONN;
        while ((skb = tcp_recv_skb(sk, seq, &offset)) != NULL) {
                if (offset < skb->len) {
                        int used;
                        size_t len;

                        len = skb->len - offset;
                        /* Stop reading if we hit a patch of urgent data */
                        if (tp->urg_data) {
                                u32 urg_offset = tp->urg_seq - seq;
                                if (urg_offset < len)
                                        len = urg_offset;
                                if (!len)
                                        break;
                        }
                        used = recv_actor(desc, skb, offset, len);
                        if (used <= 0) {
                                if (!copied)
                                        copied = used;
                                break;
                        } else if (used <= len) {
                                seq += used;
                                copied += used;
                                offset += used;
                        }
                        /* If recv_actor drops the lock (e.g. TCP splice
                         * receive) the skb pointer might be invalid when
                         * getting here: tcp_collapse might have deleted it
                         * while aggregating skbs from the socket queue.
                         */
                        skb = tcp_recv_skb(sk, seq - 1, &offset);
                        if (!skb)
                                break;
                        /* TCP coalescing might have appended data to the skb.
                         * Try to splice more frags
                         */
                        if (offset + 1 != skb->len)
                                continue;
                }
                if (tcp_hdr(skb)->fin) {
                        sk_eat_skb(sk, skb, false);
                        ++seq;
                        break;
                }
                sk_eat_skb(sk, skb, false);
                if (!desc->count)
                        break;
                tp->copied_seq = seq;
        }
        tp->copied_seq = seq;

        tcp_rcv_space_adjust(sk);

        /* Clean up data we have read: This will do ACK frames. */
        if (copied > 0) {
                tcp_recv_skb(sk, seq, &offset);
                tcp_cleanup_rbuf(sk, copied);
        }
        return copied;
}
EXPORT_SYMBOL(tcp_read_sock);

/*
 *      This routine copies from a sock struct into the user buffer.
 *
 *      Technical note: in 2.3 we work on _locked_ socket, so that
 *      tricks with *seq access order and skb->users are not required.
 *      Probably, code can be easily improved even more.
 */

int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t len, int nonblock, int flags, int *addr_len)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int copied = 0;
        u32 peek_seq;
        u32 *seq;
        unsigned long used;
        int err;
        int target;             /* Read at least this many bytes */
        long timeo;
        struct task_struct *user_recv = NULL;
        bool copied_early = false;
        struct sk_buff *skb;
        u32 urg_hole = 0;

        if (unlikely(flags & MSG_ERRQUEUE))
                return ip_recv_error(sk, msg, len, addr_len);

        if (sk_can_busy_loop(sk) && skb_queue_empty(&sk->sk_receive_queue) &&
            (sk->sk_state == TCP_ESTABLISHED))
                sk_busy_loop(sk, nonblock);

        lock_sock(sk);

        err = -ENOTCONN;
        if (sk->sk_state == TCP_LISTEN)
                goto out;

        timeo = sock_rcvtimeo(sk, nonblock);

        /* Urgent data needs to be handled specially. */
        if (flags & MSG_OOB)
                goto recv_urg;

        if (unlikely(tp->repair)) {
                err = -EPERM;
                if (!(flags & MSG_PEEK))
                        goto out;

                if (tp->repair_queue == TCP_SEND_QUEUE)
                        goto recv_sndq;

                err = -EINVAL;
                if (tp->repair_queue == TCP_NO_QUEUE)
                        goto out;

                /* 'common' recv queue MSG_PEEK-ing */
        }

        seq = &tp->copied_seq;
        if (flags & MSG_PEEK) {
                peek_seq = tp->copied_seq;
                seq = &peek_seq;
        }

        target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);

#ifdef CONFIG_NET_DMA
        tp->ucopy.dma_chan = NULL;
        preempt_disable();
        skb = skb_peek_tail(&sk->sk_receive_queue);
        {
                int available = 0;

                if (skb)
                        available = TCP_SKB_CB(skb)->seq + skb->len - (*seq);
                if ((available < target) &&
                    (len > sysctl_tcp_dma_copybreak) && !(flags & MSG_PEEK) &&
                    !sysctl_tcp_low_latency &&
                    net_dma_find_channel()) {
                        preempt_enable();
                        tp->ucopy.pinned_list =
                                        dma_pin_iovec_pages(msg->msg_iov, len);
                } else {
                        preempt_enable();
                }
        }
#endif

        do {
                u32 offset;

                /* Are we at urgent data? Stop if we have read anything or have SIGURG pending. */
                if (tp->urg_data && tp->urg_seq == *seq) {
                        if (copied)
                                break;
                        if (signal_pending(current)) {
                                copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
                                break;
                        }
                }

                /* Next get a buffer. */

                skb_queue_walk(&sk->sk_receive_queue, skb) {
                        /* Now that we have two receive queues this
                         * shouldn't happen.
                         */
                        if (WARN(before(*seq, TCP_SKB_CB(skb)->seq),
                                 "recvmsg bug: copied %X seq %X rcvnxt %X fl %X\n",
                                 *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt,
                                 flags))
                                break;

                        offset = *seq - TCP_SKB_CB(skb)->seq;
                        if (tcp_hdr(skb)->syn)
                                offset--;
                        if (offset < skb->len)
                                goto found_ok_skb;
                        if (tcp_hdr(skb)->fin)
                                goto found_fin_ok;
                        WARN(!(flags & MSG_PEEK),
                             "recvmsg bug 2: copied %X seq %X rcvnxt %X fl %X\n",
                             *seq, TCP_SKB_CB(skb)->seq, tp->rcv_nxt, flags);
                }

                /* Well, if we have backlog, try to process it now yet. */

                if (copied >= target && !sk->sk_backlog.tail)
                        break;

                if (copied) {
                        if (sk->sk_err ||
                            sk->sk_state == TCP_CLOSE ||
                            (sk->sk_shutdown & RCV_SHUTDOWN) ||
                            !timeo ||
                            signal_pending(current))
                                break;
                } else {
                        if (sock_flag(sk, SOCK_DONE))
                                break;

                        if (sk->sk_err) {
                                copied = sock_error(sk);
                                break;
                        }

                        if (sk->sk_shutdown & RCV_SHUTDOWN)
                                break;

                        if (sk->sk_state == TCP_CLOSE) {
                                if (!sock_flag(sk, SOCK_DONE)) {
                                        /* This occurs when user tries to read
                                         * from never connected socket.
                                         */
                                        copied = -ENOTCONN;
                                        break;
                                }
                                break;
                        }

                        if (!timeo) {
                                copied = -EAGAIN;
                                break;
                        }

                        if (signal_pending(current)) {
                                copied = sock_intr_errno(timeo);
                                break;
                        }
                }

                tcp_cleanup_rbuf(sk, copied);

                if (!sysctl_tcp_low_latency && tp->ucopy.task == user_recv) {
                        /* Install new reader */
                        if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
                                user_recv = current;
                                tp->ucopy.task = user_recv;
                                tp->ucopy.iov = msg->msg_iov;
                        }

                        tp->ucopy.len = len;

                        WARN_ON(tp->copied_seq != tp->rcv_nxt &&
                                !(flags & (MSG_PEEK | MSG_TRUNC)));

                        /* Ugly... If prequeue is not empty, we have to
                         * process it before releasing socket, otherwise
                         * order will be broken at second iteration.
                         * More elegant solution is required!!!
                         *
                         * Look: we have the following (pseudo)queues:
                         *
                         * 1. packets in flight
                         * 2. backlog
                         * 3. prequeue
                         * 4. receive_queue
                         *
                         * Each queue can be processed only if the next ones
                         * are empty. At this point we have empty receive_queue.
                         * But prequeue _can_ be not empty after 2nd iteration,
                         * when we jumped to start of loop because backlog
                         * processing added something to receive_queue.
                         * We cannot release_sock(), because backlog contains
                         * packets arrived _after_ prequeued ones.
                         *
                         * Shortly, algorithm is clear --- to process all
                         * the queues in order. We could make it more directly,
                         * requeueing packets from backlog to prequeue, if
                         * is not empty. It is more elegant, but eats cycles,
                         * unfortunately.
                         */
                        if (!skb_queue_empty(&tp->ucopy.prequeue))
                                goto do_prequeue;

                        /* __ Set realtime policy in scheduler __ */
                }

#ifdef CONFIG_NET_DMA
                if (tp->ucopy.dma_chan) {
                        if (tp->rcv_wnd == 0 &&
                            !skb_queue_empty(&sk->sk_async_wait_queue)) {
                                tcp_service_net_dma(sk, true);
                                tcp_cleanup_rbuf(sk, copied);
                        } else
                                dma_async_issue_pending(tp->ucopy.dma_chan);
                }
#endif
                if (copied >= target) {
                        /* Do not sleep, just process backlog. */
                        release_sock(sk);
                        lock_sock(sk);
                } else
                        sk_wait_data(sk, &timeo);

#ifdef CONFIG_NET_DMA
                tcp_service_net_dma(sk, false);  /* Don't block */
                tp->ucopy.wakeup = 0;
#endif

```

```
                if (user_recv) {
                        int chunk;

                        /* __ Restore normal policy in scheduler __ */

                        if ((chunk = len - tp->ucopy.len) != 0) {
                                NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, chunk);
                                len -= chunk;
                                copied += chunk;
                        }

                        if (tp->rcv_nxt == tp->copied_seq &&
                            !skb_queue_empty(&tp->ucopy.prequeue)) {
do_prequeue:
                                tcp_prequeue_process(sk);

                                if ((chunk = len - tp->ucopy.len) != 0) {
                                        NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
                                        len -= chunk;
                                        copied += chunk;
                                }
                        }
                }
                if ((flags & MSG_PEEK) &&
                    (peek_seq - copied - urg_hole != tp->copied_seq)) {
                        net_dbg_ratelimited("TCP(%s:%d): Application bug, race in MSG_PEEK\n",
                                            current->comm,
                                            task_pid_nr(current));
                        peek_seq = tp->copied_seq;
                }
                continue;

        found_ok_skb:
                /* Ok so how much can we use? */
                used = skb->len - offset;
                if (len < used)
                        used = len;

                /* Do we have urgent data here? */
                if (tp->urg_data) {
                        u32 urg_offset = tp->urg_seq - *seq;
                        if (urg_offset < used) {
                                if (!urg_offset) {
                                        if (!sock_flag(sk, SOCK_URGINLINE)) {
                                                ++*seq;
                                                urg_hole++;
                                                offset++;
                                                used--;
                                                if (!used)
                                                        goto skip_copy;
                                        }
                                } else
                                        used = urg_offset;
                        }
                }

                if (!(flags & MSG_TRUNC)) {
#ifdef CONFIG_NET_DMA
                        if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
                                tp->ucopy.dma_chan = net_dma_find_channel();

                        if (tp->ucopy.dma_chan) {
                                tp->ucopy.dma_cookie = dma_skb_copy_datagram_iovec(
                                        tp->ucopy.dma_chan, skb, offset,
                                        msg->msg_iov, used,
                                        tp->ucopy.pinned_list);

                                if (tp->ucopy.dma_cookie < 0) {

                                        pr_alert("%s: dma_cookie < 0\n",
                                                 __func__);

                                        /* Exception. Bailout! */
                                        if (!copied)
                                                copied = -EFAULT;
                                        break;
                                }

                                dma_async_issue_pending(tp->ucopy.dma_chan);

                                if ((offset + used) == skb->len)
                                        copied_early = true;

                        } else
#endif
```

```
                        {
                                err = skb_copy_datagram_iovec(skb, offset,
                                                msg->msg_iov, used);
                                if (err) {
                                        /* Exception. Bailout! */
                                        if (!copied)
                                                copied = -EFAULT;
                                        break;
                                }
                        }
                }

                *seq += used;
                copied += used;
                len -= used;

                tcp_rcv_space_adjust(sk);

skip_copy:
                if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
                        tp->urg_data = 0;
                        tcp_fast_path_check(sk);
                }
                if (used + offset < skb->len)
                        continue;

                if (tcp_hdr(skb)->fin)
                        goto found_fin_ok;
                if (!(flags & MSG_PEEK)) {
                        sk_eat_skb(sk, skb, copied_early);
                        copied_early = false;
                }
                continue;

        found_fin_ok:
                /* Process the FIN. */
                ++*seq;
                if (!(flags & MSG_PEEK)) {
                        sk_eat_skb(sk, skb, copied_early);
                        copied_early = false;
                }
                break;
        } while (len > 0);

        if (user_recv) {
                if (!skb_queue_empty(&tp->ucopy.prequeue)) {
                        int chunk;

                        tp->ucopy.len = copied > 0 ? len : 0;

                        tcp_prequeue_process(sk);

                        if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
                                NET_ADD_STATS_USER(sock_net(sk), LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
                                len -= chunk;
                                copied += chunk;
                        }
                }

                tp->ucopy.task = NULL;
                tp->ucopy.len = 0;
        }

#ifdef CONFIG_NET_DMA
        tcp_service_net_dma(sk, true);  /* Wait for queue to drain */
        tp->ucopy.dma_chan = NULL;

        if (tp->ucopy.pinned_list) {
                dma_unpin_iovec_pages(tp->ucopy.pinned_list);
                tp->ucopy.pinned_list = NULL;
        }
#endif

        /* According to UNIX98, msg_name/msg_namelen are ignored
         * on connected socket. I was just happy when found this 8) --ANK
         */

        /* Clean up data we have read: This will do ACK frames. */
        tcp_cleanup_rbuf(sk, copied);

        release_sock(sk);
        return copied;

out:
        release_sock(sk);
        return err;

recv_urg:
        err = tcp_recv_urg(sk, msg, len, flags);
        goto out;

recv_sndq:
        err = tcp_peek_sndq(sk, msg, len);
        goto out;
}
EXPORT_SYMBOL(tcp_recvmsg);

void tcp_set_state(struct sock *sk, int state)
{
        int oldstate = sk->sk_state;

        switch (state) {
        case TCP_ESTABLISHED:
                if (oldstate != TCP_ESTABLISHED)
                        TCP_INC_STATS(sock_net(sk), TCP_MIB_CURRESTAB);
                break;

        case TCP_CLOSE:
                if (oldstate == TCP_CLOSE_WAIT || oldstate == TCP_ESTABLISHED)
                        TCP_INC_STATS(sock_net(sk), TCP_MIB_ESTABRESETS);

                sk->sk_prot->unhash(sk);
                if (inet_csk(sk)->icsk_bind_hash &&
                    !(sk->sk_userlocks & SOCK_BINDPORT_LOCK))
                        inet_put_port(sk);
                /* fall through */
        default:
                if (oldstate == TCP_ESTABLISHED)
                        TCP_DEC_STATS(sock_net(sk), TCP_MIB_CURRESTAB);
        }

        /* Change state AFTER socket is unhashed to avoid closed
         * socket sitting in hash tables.
         */
        sk->sk_state = state;

#ifdef STATE_TRACE
        SOCK_DEBUG(sk, "TCP sk=%p, State %s -> %s\n", sk, statename[oldstate], statename[state]);
#endif
}
EXPORT_SYMBOL_GPL(tcp_set_state);

/*
 *      State processing on a close. This implements the state shift for
 *      sending our FIN frame. Note that we only send a FIN for some
 *      states. A shutdown() may have already sent the FIN, or we may be
 *      closed.
 */

static const unsigned char new_state[16] = {
  /* current state:        new state:      action:      */
  /* (Invalid)          */ TCP_CLOSE,
  /* TCP_ESTABLISHED    */ TCP_FIN_WAIT1 | TCP_ACTION_FIN,
  /* TCP_SYN_SENT       */ TCP_CLOSE,
  /* TCP_SYN_RECV       */ TCP_FIN_WAIT1 | TCP_ACTION_FIN,
  /* TCP_FIN_WAIT1      */ TCP_FIN_WAIT1,
  /* TCP_FIN_WAIT2      */ TCP_FIN_WAIT2,
  /* TCP_TIME_WAIT      */ TCP_CLOSE,
  /* TCP_CLOSE          */ TCP_CLOSE,
  /* TCP_CLOSE_WAIT     */ TCP_LAST_ACK  | TCP_ACTION_FIN,
  /* TCP_LAST_ACK       */ TCP_LAST_ACK,
  /* TCP_LISTEN         */ TCP_CLOSE,
  /* TCP_CLOSING        */ TCP_CLOSING,
};

static int tcp_close_state(struct sock *sk)
{
        int next = (int)new_state[sk->sk_state];
        int ns = next & TCP_STATE_MASK;

        tcp_set_state(sk, ns);

        return next & TCP_ACTION_FIN;
}

/*
 *      Shutdown the sending side of a connection. Much like close except
 *      that we don't receive shut down or sock_set_flag(sk, SOCK_DEAD).
 */

```

```
void tcp_shutdown(struct sock *sk, int how)
{
        /*      We need to grab some memory, and put together a FIN,
         *      and then put it into the queue to be sent.
         *              Tim MacKenzie(tym@dibbler.cs.monash.edu.au) 4 Dec '92.
         */
        if (!(how & SEND_SHUTDOWN))
                return;

        /* If we've already sent a FIN, or it's a closed state, skip this. */
        if ((1 << sk->sk_state) &
            (TCPF_ESTABLISHED | TCPF_SYN_SENT |
             TCPF_SYN_RECV | TCPF_CLOSE_WAIT)) {
                /* Clear out any half completed packets.  FIN if needed. */
                if (tcp_close_state(sk))
                        tcp_send_fin(sk);
        }
}
EXPORT_SYMBOL(tcp_shutdown);

bool tcp_check_oom(struct sock *sk, int shift)
{
        bool too_many_orphans, out_of_socket_memory;

        too_many_orphans = tcp_too_many_orphans(sk, shift);
        out_of_socket_memory = tcp_out_of_memory(sk);

        if (too_many_orphans)
                net_info_ratelimited("too many orphaned sockets\n");
        if (out_of_socket_memory)
                net_info_ratelimited("out of memory -- consider tuning tcp_mem\n");
        return too_many_orphans || out_of_socket_memory;
}

void tcp_close(struct sock *sk, long timeout)
{
        struct sk_buff *skb;
        int data_was_unread = 0;
        int state;

        lock_sock(sk);
        sk->sk_shutdown = SHUTDOWN_MASK;

        if (sk->sk_state == TCP_LISTEN) {
                tcp_set_state(sk, TCP_CLOSE);

                /* Special case. */
                inet_csk_listen_stop(sk);

                goto adjudge_to_death;
        }

        /*  We need to flush the recv. buffs.  We do this only on the
         *  descriptor close, not protocol-sourced closes, because the
         *  reader process may not have drained the data yet!
         */
        while ((skb = __skb_dequeue(&sk->sk_receive_queue)) != NULL) {
                u32 len = TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq -
                          tcp_hdr(skb)->fin;
                data_was_unread += len;
                __kfree_skb(skb);
        }

        sk_mem_reclaim(sk);

        /* If socket has been already reset (e.g. in tcp_reset()) - kill it. */
        if (sk->sk_state == TCP_CLOSE)
                goto adjudge_to_death;

        /* As outlined in RFC 2525, section 2.17, we send a RST here because
         * data was lost. To witness the awful effects of the old behavior of
         * always doing a FIN, run an older 2.1.x kernel or 2.0.x, start a bulk
         * GET in an FTP client, suspend the process, wait for the client to
         * advertise a zero window, then kill -9 the FTP client, wheee...
         * Note: timeout is always zero in such a case.
         */
        if (unlikely(tcp_sk(sk)->repair)) {
                sk->sk_prot->disconnect(sk, 0);
        } else if (data_was_unread) {
                /* Unread data was tossed, zap the connection. */
                NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONCLOSE);
                tcp_set_state(sk, TCP_CLOSE);
                tcp_send_active_reset(sk, sk->sk_allocation);
        } else if (sock_flag(sk, SOCK_LINGER) && !sk->sk_lingertime) {
                /* Check zero linger _after_ checking for unread data. */
                sk->sk_prot->disconnect(sk, 0);
                NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
        } else if (tcp_close_state(sk)) {
                /* We FIN if the application ate all the data before
                 * zapping the connection.
                 */

                /* RED-PEN. Formally speaking, we have broken TCP state
                 * machine. State transitions:
                 *
                 * TCP_ESTABLISHED -> TCP_FIN_WAIT1
                 * TCP_SYN_RECV -> TCP_FIN_WAIT1 (forget it, it's impossible)
                 * TCP_CLOSE_WAIT -> TCP_LAST_ACK
                 *
                 * are legal only when FIN has been sent (i.e. in window),
                 * rather than queued out of window. Purists blame.
                 *
                 * F.e. "RFC state" is ESTABLISHED,
                 * if Linux state is FIN-WAIT-1, but FIN is still not sent.
                 *
                 * The visible declinations are that sometimes
                 * we enter time-wait state, when it is not required really
                 * (harmless), do not send active resets, when they are
                 * required by specs (TCP_ESTABLISHED, TCP_CLOSE_WAIT, when
                 * they look as CLOSING or LAST_ACK for Linux)
                 * Probably, I missed some more holelets.
                 *                                              --ANK
                 * XXX (TFO) - To start off we don't support SYN+ACK+FIN
                 * in a single packet! (May consider it later but will
                 * probably need API support or TCP_CORK SYN-ACK until
                 * data is written and socket is closed.)
                 */
                tcp_send_fin(sk);
        }

        sk_stream_wait_close(sk, timeout);

adjudge_to_death:
        state = sk->sk_state;
        sock_hold(sk);
        sock_orphan(sk);

        /* It is the last release_sock in its life. It will remove backlog. */
        release_sock(sk);


        /* Now socket is owned by kernel and we acquire BH lock
           to finish close. No need to check for user refs.
         */
        local_bh_disable();
        bh_lock_sock(sk);
        WARN_ON(sock_owned_by_user(sk));

        percpu_counter_inc(sk->sk_prot->orphan_count);

        /* Have we already been destroyed by a softirq or backlog? */
        if (state != TCP_CLOSE && sk->sk_state == TCP_CLOSE)
                goto out;

        /*      This is a (useful) BSD violating of the RFC. There is a
         *      problem with TCP as specified in that the other end could
         *      keep a socket open forever with no application left this end.
         *      We use a 1 minute timeout (about the same as BSD) then kill
         *      our end. If they send after that then tough - BUT: long enough
         *      that we won't make the old 4*rto = almost no time - whoops
         *      reset mistake.
         *
         *      Nope, it was not mistake. It is really desired behaviour
         *      f.e. on http servers, when such sockets are useless, but
         *      consume significant resources. Let's do it with special
         *      linger2 option.                                 --ANK
         */

        if (sk->sk_state == TCP_FIN_WAIT2) {
                struct tcp_sock *tp = tcp_sk(sk);
                if (tp->linger2 < 0) {
                        tcp_set_state(sk, TCP_CLOSE);
                        tcp_send_active_reset(sk, GFP_ATOMIC);
                        NET_INC_STATS_BH(sock_net(sk),
                                        LINUX_MIB_TCPABORTONLINGER);
                } else {
                        const int tmo = tcp_fin_time(sk);

                        if (tmo > TCP_TIMEWAIT_LEN) {
                                inet_csk_reset_keepalive_timer(sk,
                                                tmo - TCP_TIMEWAIT_LEN);
                        } else {
                                tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
                                goto out;
                        }
                }
        }
        if (sk->sk_state != TCP_CLOSE) {
                sk_mem_reclaim(sk);
                if (tcp_check_oom(sk, 0)) {
                        tcp_set_state(sk, TCP_CLOSE);
                        tcp_send_active_reset(sk, GFP_ATOMIC);
                        NET_INC_STATS_BH(sock_net(sk),
                                        LINUX_MIB_TCPABORTONMEMORY);
                }
        }

        if (sk->sk_state == TCP_CLOSE) {
                struct request_sock *req = tcp_sk(sk)->fastopen_rsk;
                /* We could get here with a non-NULL req if the socket is
                 * aborted (e.g., closed with unread data) before 3WHS
                 * finishes.
                 */
                if (req != NULL)
                        reqsk_fastopen_remove(sk, req, false);
                inet_csk_destroy_sock(sk);
        }
        /* Otherwise, socket is reprieved until protocol close. */

out:
        bh_unlock_sock(sk);
        local_bh_enable();
        sock_put(sk);
}
EXPORT_SYMBOL(tcp_close);

/* These states need RST on ABORT according to RFC793 */

static inline bool tcp_need_reset(int state)
{
        return (1 << state) &
               (TCPF_ESTABLISHED | TCPF_CLOSE_WAIT | TCPF_FIN_WAIT1 |
                TCPF_FIN_WAIT2 | TCPF_SYN_RECV);
}

int tcp_disconnect(struct sock *sk, int flags)
{
        struct inet_sock *inet = inet_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        int err = 0;
        int old_state = sk->sk_state;

        if (old_state != TCP_CLOSE)
                tcp_set_state(sk, TCP_CLOSE);

        /* ABORT function of RFC793 */
        if (old_state == TCP_LISTEN) {
                inet_csk_listen_stop(sk);
        } else if (unlikely(tp->repair)) {
                sk->sk_err = ECONNABORTED;
        } else if (tcp_need_reset(old_state) ||
                   (tp->snd_nxt != tp->write_seq &&
                    (1 << old_state) & (TCPF_CLOSING | TCPF_LAST_ACK))) {
                /* The last check adjusts for discrepancy of Linux wrt. RFC
                 * states
                 */
                tcp_send_active_reset(sk, gfp_any());
                sk->sk_err = ECONNRESET;
        } else if (old_state == TCP_SYN_SENT)
                sk->sk_err = ECONNRESET;

        tcp_clear_xmit_timers(sk);
        __skb_queue_purge(&sk->sk_receive_queue);
        tcp_write_queue_purge(sk);
        __skb_queue_purge(&tp->out_of_order_queue);
#ifdef CONFIG_NET_DMA
        __skb_queue_purge(&sk->sk_async_wait_queue);
#endif

        inet->inet_dport = 0;

        if (!(sk->sk_userlocks & SOCK_BINDADDR_LOCK))
                inet_reset_saddr(sk);

        sk->sk_shutdown = 0;
        sock_reset_flag(sk, SOCK_DONE);
        tp->srtt_us = 0;
        if ((tp->write_seq += tp->max_window + 2) == 0)
                tp->write_seq = 1;
        icsk->icsk_backoff = 0;
        tp->snd_cwnd = 2;
        icsk->icsk_probes_out = 0;
        tp->packets_out = 0;
        tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
        tp->snd_cwnd_cnt = 0;
        tp->window_clamp = 0;
        tcp_set_ca_state(sk, TCP_CA_Open);
        tcp_clear_retrans(tp);
        inet_csk_delack_init(sk);
        tcp_init_send_head(sk);
        memset(&tp->rx_opt, 0, sizeof(tp->rx_opt));
        __sk_dst_reset(sk);

        WARN_ON(inet->inet_num && !icsk->icsk_bind_hash);

        sk->sk_error_report(sk);
        return err;
}
EXPORT_SYMBOL(tcp_disconnect);

void tcp_sock_destruct(struct sock *sk)
{
        inet_sock_destruct(sk);

        kfree(inet_csk(sk)->icsk_accept_queue.fastopenq);
}

static inline bool tcp_can_repair_sock(const struct sock *sk)
{
        return ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN) &&
                ((1 << sk->sk_state) & (TCPF_CLOSE | TCPF_ESTABLISHED));
}

static int tcp_repair_options_est(struct tcp_sock *tp,
                struct tcp_repair_opt __user *optbuf, unsigned int len)
{
        struct tcp_repair_opt opt;

        while (len >= sizeof(opt)) {
                if (copy_from_user(&opt, optbuf, sizeof(opt)))
                        return -EFAULT;

                optbuf++;
                len -= sizeof(opt);

                switch (opt.opt_code) {
                case TCPOPT_MSS:
                        tp->rx_opt.mss_clamp = opt.opt_val;
                        break;
                case TCPOPT_WINDOW:
                        {
                                u16 snd_wscale = opt.opt_val & 0xFFFF;
                                u16 rcv_wscale = opt.opt_val >> 16;

                                if (snd_wscale > 14 || rcv_wscale > 14)
                                        return -EFBIG;

                                tp->rx_opt.snd_wscale = snd_wscale;
                                tp->rx_opt.rcv_wscale = rcv_wscale;
                                tp->rx_opt.wscale_ok = 1;
                        }
                        break;
                case TCPOPT_SACK_PERM:
                        if (opt.opt_val != 0)
                                return -EINVAL;

                        tp->rx_opt.sack_ok |= TCP_SACK_SEEN;
                        if (sysctl_tcp_fack)
                                tcp_enable_fack(tp);
                        break;
                case TCPOPT_TIMESTAMP:
                        if (opt.opt_val != 0)
                                return -EINVAL;

                        tp->rx_opt.tstamp_ok = 1;
                        break;
                }
        }

        return 0;
}

/*
 *      Socket option code for TCP.
 */
static int do_tcp_setsockopt(struct sock *sk, int level,
                int optname, char __user *optval, unsigned int optlen)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        int val;
        int err = 0;

        /* These are data/string values, all the others are ints */
        switch (optname) {
        case TCP_CONGESTION: {
                char name[TCP_CA_NAME_MAX];

                if (optlen < 1)
                        return -EINVAL;

                val = strncpy_from_user(name, optval,
                                        min_t(long, TCP_CA_NAME_MAX-1, optlen));
                if (val < 0)
                        return -EFAULT;
                name[val] = 0;

                lock_sock(sk);
                err = tcp_set_congestion_control(sk, name);
                release_sock(sk);
                return err;
        }
        default:
                /* fallthru */
                break;
        }

        if (optlen < sizeof(int))
                return -EINVAL;

        if (get_user(val, (int __user *)optval))
                return -EFAULT;

        lock_sock(sk);

        switch (optname) {
        case TCP_MAXSEG:
                /* Values greater than interface MTU won't take effect. However
                 * at the point when this call is done we typically don't yet
                 * know which interface is going to be used */
                if (val < TCP_MIN_MSS || val > MAX_TCP_WINDOW) {
                        err = -EINVAL;
                        break;
                }
                tp->rx_opt.user_mss = val;
                break;

        case TCP_NODELAY:
                if (val) {
                        /* TCP_NODELAY is weaker than TCP_CORK, so that
                         * this option on corked socket is remembered, but
                         * it is not activated until cork is cleared.
                         *
                         * However, when TCP_NODELAY is set we make
                         * an explicit push, which overrides even TCP_CORK
                         * for currently queued segments.
                         */
                        tp->nonagle |= TCP_NAGLE_OFF|TCP_NAGLE_PUSH;
                        tcp_push_pending_frames(sk);
                } else {
                        tp->nonagle &= ~TCP_NAGLE_OFF;
                }
                break;

        case TCP_THIN_LINEAR_TIMEOUTS:
                if (val < 0 || val > 1)
                        err = -EINVAL;
                else
                        tp->thin_lto = val;
                break;

        case TCP_THIN_DUPACK:
                if (val < 0 || val > 1)
                        err = -EINVAL;
                else {
                        tp->thin_dupack = val;
                        if (tp->thin_dupack)
                                tcp_disable_early_retrans(tp);
                }
                break;

        case TCP_REPAIR:
                if (!tcp_can_repair_sock(sk))
                        err = -EPERM;
                else if (val == 1) {
                        tp->repair = 1;
                        sk->sk_reuse = SK_FORCE_REUSE;
                        tp->repair_queue = TCP_NO_QUEUE;
                } else if (val == 0) {
                        tp->repair = 0;
                        sk->sk_reuse = SK_NO_REUSE;
                        tcp_send_window_probe(sk);
                } else
                        err = -EINVAL;

                break;

        case TCP_REPAIR_QUEUE:
                if (!tp->repair)
                        err = -EPERM;
                else if (val < TCP_QUEUES_NR)
                        tp->repair_queue = val;
                else
                        err = -EINVAL;
                break;

        case TCP_QUEUE_SEQ:
                if (sk->sk_state != TCP_CLOSE)
                        err = -EPERM;
                else if (tp->repair_queue == TCP_SEND_QUEUE)
                        tp->write_seq = val;
                else if (tp->repair_queue == TCP_RECV_QUEUE)
                        tp->rcv_nxt = val;
                else
                        err = -EINVAL;
                break;

        case TCP_REPAIR_OPTIONS:
                if (!tp->repair)
                        err = -EINVAL;
                else if (sk->sk_state == TCP_ESTABLISHED)
                        err = tcp_repair_options_est(tp,
                                        (struct tcp_repair_opt __user *)optval,
                                        optlen);
                else
                        err = -EPERM;
                break;

        case TCP_CORK:
                /* When set indicates to always queue non-full frames.
                 * Later the user clears this option and we transmit
                 * any pending partial frames in the queue.  This is
                 * meant to be used alongside sendfile() to get properly
                 * filled frames when the user (for example) must write
                 * out headers with a write() call first and then use
                 * sendfile to send out the data parts.
                 *
                 * TCP_CORK can be set together with TCP_NODELAY and it is
                 * stronger than TCP_NODELAY.
                 */
                if (val) {
                        tp->nonagle |= TCP_NAGLE_CORK;
                } else {
                        tp->nonagle &= ~TCP_NAGLE_CORK;
                        if (tp->nonagle&TCP_NAGLE_OFF)
                                tp->nonagle |= TCP_NAGLE_PUSH;
                        tcp_push_pending_frames(sk);
                }
                break;

        case TCP_KEEPIDLE:
                if (val < 1 || val > MAX_TCP_KEEPIDLE)
                        err = -EINVAL;
                else {
                        tp->keepalive_time = val * HZ;
                        if (sock_flag(sk, SOCK_KEEPOPEN) &&
                            !((1 << sk->sk_state) &
                              (TCPF_CLOSE | TCPF_LISTEN))) {
                                u32 elapsed = keepalive_time_elapsed(tp);
                                if (tp->keepalive_time > elapsed)
                                        elapsed = tp->keepalive_time - elapsed;
                                else
                                        elapsed = 0;
                                inet_csk_reset_keepalive_timer(sk, elapsed);
                        }
                }
                break;
        case TCP_KEEPINTVL:
                if (val < 1 || val > MAX_TCP_KEEPINTVL)
                        err = -EINVAL;
                else
                        tp->keepalive_intvl = val * HZ;
                break;
        case TCP_KEEPCNT:
                if (val < 1 || val > MAX_TCP_KEEPCNT)
                        err = -EINVAL;
                else
                        tp->keepalive_probes = val;
                break;
        case TCP_SYNCNT:
                if (val < 1 || val > MAX_TCP_SYNCNT)
                        err = -EINVAL;
                else
                        icsk->icsk_syn_retries = val;
                break;

        case TCP_LINGER2:
                if (val < 0)
                        tp->linger2 = -1;
                else if (val > sysctl_tcp_fin_timeout / HZ)
                        tp->linger2 = 0;
                else
                        tp->linger2 = val * HZ;
                break;

        case TCP_DEFER_ACCEPT:
                /* Translate value in seconds to number of retransmits */
                icsk->icsk_accept_queue.rskq_defer_accept =
                        secs_to_retrans(val, TCP_TIMEOUT_INIT / HZ,
                                        TCP_RTO_MAX / HZ);
                break;

        case TCP_WINDOW_CLAMP:
                if (!val) {
                        if (sk->sk_state != TCP_CLOSE) {
                                err = -EINVAL;
                                break;
                        }
                        tp->window_clamp = 0;
                } else
                        tp->window_clamp = val < SOCK_MIN_RCVBUF / 2 ?
                                                SOCK_MIN_RCVBUF / 2 : val;
                break;

        case TCP_QUICKACK:
                if (!val) {
                        icsk->icsk_ack.pingpong = 1;
                } else {
                        icsk->icsk_ack.pingpong = 0;
                        if ((1 << sk->sk_state) &
                            (TCPF_ESTABLISHED | TCPF_CLOSE_WAIT) &&
                            inet_csk_ack_scheduled(sk)) {
                                icsk->icsk_ack.pending |= ICSK_ACK_PUSHED;
                                tcp_cleanup_rbuf(sk, 1);
                                if (!(val & 1))
                                        icsk->icsk_ack.pingpong = 1;
                        }
                }
                break;

#ifdef CONFIG_TCP_MD5SIG
        case TCP_MD5SIG:
                /* Read the IP->Key mappings from userspace */
                err = tp->af_specific->md5_parse(sk, optval, optlen);
                break;
#endif
        case TCP_USER_TIMEOUT:
                /* Cap the max timeout in ms TCP will retry/retrans
                 * before giving up and aborting (ETIMEDOUT) a connection.
                 */
                if (val < 0)
                        err = -EINVAL;
                else
                        icsk->icsk_user_timeout = msecs_to_jiffies(val);
                break;

        case TCP_FASTOPEN:
                if (val >= 0 && ((1 << sk->sk_state) & (TCPF_CLOSE |
                    TCPF_LISTEN)))
                        err = fastopen_init_queue(sk, val);
                else
                        err = -EINVAL;
                break;
        case TCP_TIMESTAMP:
                if (!tp->repair)
                        err = -EPERM;
                else
                        tp->tsoffset = val - tcp_time_stamp;
                break;
        case TCP_NOTSENT_LOWAT:
                tp->notsent_lowat = val;
                sk->sk_write_space(sk);
                break;
        default:
                err = -ENOPROTOOPT;
                break;
        }

        release_sock(sk);
        return err;
}

int tcp_setsockopt(struct sock *sk, int level, int optname, char __user *optval,
                   unsigned int optlen)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (level != SOL_TCP)
                return icsk->icsk_af_ops->setsockopt(sk, level, optname,
                                                     optval, optlen);
        return do_tcp_setsockopt(sk, level, optname, optval, optlen);
}
EXPORT_SYMBOL(tcp_setsockopt);

#ifdef CONFIG_COMPAT
int compat_tcp_setsockopt(struct sock *sk, int level, int optname,
                          char __user *optval, unsigned int optlen)
{
        if (level != SOL_TCP)
                return inet_csk_compat_setsockopt(sk, level, optname,
                                                  optval, optlen);
        return do_tcp_setsockopt(sk, level, optname, optval, optlen);
}
EXPORT_SYMBOL(compat_tcp_setsockopt);
#endif

/* Return information about state of tcp endpoint in API format. */
void tcp_get_info(const struct sock *sk, struct tcp_info *info)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        const struct inet_connection_sock *icsk = inet_csk(sk);
        u32 now = tcp_time_stamp;

        memset(info, 0, sizeof(*info));

        info->tcpi_state = sk->sk_state;
        info->tcpi_ca_state = icsk->icsk_ca_state;
        info->tcpi_retransmits = icsk->icsk_retransmits;
        info->tcpi_probes = icsk->icsk_probes_out;
        info->tcpi_backoff = icsk->icsk_backoff;

        if (tp->rx_opt.tstamp_ok)
                info->tcpi_options |= TCPI_OPT_TIMESTAMPS;
        if (tcp_is_sack(tp))
                info->tcpi_options |= TCPI_OPT_SACK;
        if (tp->rx_opt.wscale_ok) {
                info->tcpi_options |= TCPI_OPT_WSCALE;
                info->tcpi_snd_wscale = tp->rx_opt.snd_wscale;
                info->tcpi_rcv_wscale = tp->rx_opt.rcv_wscale;
        }

        if (tp->ecn_flags & TCP_ECN_OK)
                info->tcpi_options |= TCPI_OPT_ECN;
        if (tp->ecn_flags & TCP_ECN_SEEN)
                info->tcpi_options |= TCPI_OPT_ECN_SEEN;
        if (tp->syn_data_acked)
                info->tcpi_options |= TCPI_OPT_SYN_DATA;

        info->tcpi_rto = jiffies_to_usecs(icsk->icsk_rto);
        info->tcpi_ato = jiffies_to_usecs(icsk->icsk_ack.ato);
        info->tcpi_snd_mss = tp->mss_cache;
        info->tcpi_rcv_mss = icsk->icsk_ack.rcv_mss;

        if (sk->sk_state == TCP_LISTEN) {
                info->tcpi_unacked = sk->sk_ack_backlog;
                info->tcpi_sacked = sk->sk_max_ack_backlog;
        } else {
                info->tcpi_unacked = tp->packets_out;
                info->tcpi_sacked = tp->sacked_out;
        }
        info->tcpi_lost = tp->lost_out;
        info->tcpi_retrans = tp->retrans_out;
        info->tcpi_fackets = tp->fackets_out;

        info->tcpi_last_data_sent = jiffies_to_msecs(now - tp->lsndtime);
        info->tcpi_last_data_recv = jiffies_to_msecs(now - icsk->icsk_ack.lrcvtime);
        info->tcpi_last_ack_recv = jiffies_to_msecs(now - tp->rcv_tstamp);

        info->tcpi_pmtu = icsk->icsk_pmtu_cookie;
        info->tcpi_rcv_ssthresh = tp->rcv_ssthresh;
        info->tcpi_rtt = tp->srtt_us >> 3;
        info->tcpi_rttvar = tp->mdev_us >> 2;
        info->tcpi_snd_ssthresh = tp->snd_ssthresh;
        info->tcpi_snd_cwnd = tp->snd_cwnd;
        info->tcpi_advmss = tp->advmss;
        info->tcpi_reordering = tp->reordering;

        info->tcpi_rcv_rtt = jiffies_to_usecs(tp->rcv_rtt_est.rtt)>>3;
        info->tcpi_rcv_space = tp->rcvq_space.space;

        info->tcpi_total_retrans = tp->total_retrans;

        info->tcpi_pacing_rate = sk->sk_pacing_rate != ~0U ?
                                        sk->sk_pacing_rate : ~0ULL;
        info->tcpi_max_pacing_rate = sk->sk_max_pacing_rate != ~0U ?
                                        sk->sk_max_pacing_rate : ~0ULL;
}
EXPORT_SYMBOL_GPL(tcp_get_info);

static int do_tcp_getsockopt(struct sock *sk, int level,
                int optname, char __user *optval, int __user *optlen)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        int val, len;

        if (get_user(len, optlen))
                return -EFAULT;

        len = min_t(unsigned int, len, sizeof(int));

        if (len < 0)
                return -EINVAL;

        switch (optname) {
        case TCP_MAXSEG:
                val = tp->mss_cache;
                if (!val && ((1 << sk->sk_state) & (TCPF_CLOSE | TCPF_LISTEN)))
                        val = tp->rx_opt.user_mss;
                if (tp->repair)
                        val = tp->rx_opt.mss_clamp;
                break;
        case TCP_NODELAY:
                val = !!(tp->nonagle&TCP_NAGLE_OFF);
                break;
        case TCP_CORK:
                val = !!(tp->nonagle&TCP_NAGLE_CORK);
                break;
        case TCP_KEEPIDLE:
                val = keepalive_time_when(tp) / HZ;
                break;
        case TCP_KEEPINTVL:
                val = keepalive_intvl_when(tp) / HZ;
                break;
        case TCP_KEEPCNT:
                val = keepalive_probes(tp);
                break;
        case TCP_SYNCNT:
                val = icsk->icsk_syn_retries ? : sysctl_tcp_syn_retries;
                break;
        case TCP_LINGER2:
                val = tp->linger2;
                if (val >= 0)
                        val = (val ? : sysctl_tcp_fin_timeout) / HZ;
                break;
        case TCP_DEFER_ACCEPT:
                val = retrans_to_secs(icsk->icsk_accept_queue.rskq_defer_accept,
                                      TCP_TIMEOUT_INIT / HZ, TCP_RTO_MAX / HZ);
                break;
        case TCP_WINDOW_CLAMP:
                val = tp->window_clamp;
                break;
        case TCP_INFO: {
                struct tcp_info info;

                if (get_user(len, optlen))
                        return -EFAULT;

                tcp_get_info(sk, &info);

                len = min_t(unsigned int, len, sizeof(info));
                if (put_user(len, optlen))
                        return -EFAULT;
                if (copy_to_user(optval, &info, len))
                        return -EFAULT;
                return 0;
        }
        case TCP_QUICKACK:
                val = !icsk->icsk_ack.pingpong;
                break;

        case TCP_CONGESTION:
                if (get_user(len, optlen))
                        return -EFAULT;
                len = min_t(unsigned int, len, TCP_CA_NAME_MAX);
                if (put_user(len, optlen))
                        return -EFAULT;
                if (copy_to_user(optval, icsk->icsk_ca_ops->name, len))
                        return -EFAULT;
                return 0;

        case TCP_THIN_LINEAR_TIMEOUTS:
                val = tp->thin_lto;
                break;
        case TCP_THIN_DUPACK:
                val = tp->thin_dupack;
                break;

        case TCP_REPAIR:
                val = tp->repair;
                break;

        case TCP_REPAIR_QUEUE:
                if (tp->repair)
                        val = tp->repair_queue;
                else
                        return -EINVAL;
                break;

        case TCP_QUEUE_SEQ:
                if (tp->repair_queue == TCP_SEND_QUEUE)
                        val = tp->write_seq;
                else if (tp->repair_queue == TCP_RECV_QUEUE)
                        val = tp->rcv_nxt;
                else
                        return -EINVAL;
                break;

        case TCP_USER_TIMEOUT:
                val = jiffies_to_msecs(icsk->icsk_user_timeout);
                break;

        case TCP_FASTOPEN:
                if (icsk->icsk_accept_queue.fastopenq != NULL)
                        val = icsk->icsk_accept_queue.fastopenq->max_qlen;
                else
                        val = 0;
                break;

        case TCP_TIMESTAMP:
                val = tcp_time_stamp + tp->tsoffset;
                break;
        case TCP_NOTSENT_LOWAT:
                val = tp->notsent_lowat;
                break;
        default:
                return -ENOPROTOOPT;
        }

        if (put_user(len, optlen))
                return -EFAULT;
        if (copy_to_user(optval, &val, len))
                return -EFAULT;
        return 0;
}

int tcp_getsockopt(struct sock *sk, int level, int optname, char __user *optval,
                   int __user *optlen)
{
        struct inet_connection_sock *icsk = inet_csk(sk);

        if (level != SOL_TCP)
                return icsk->icsk_af_ops->getsockopt(sk, level, optname,
                                                     optval, optlen);
        return do_tcp_getsockopt(sk, level, optname, optval, optlen);
}
EXPORT_SYMBOL(tcp_getsockopt);

#ifdef CONFIG_COMPAT
int compat_tcp_getsockopt(struct sock *sk, int level, int optname,
                          char __user *optval, int __user *optlen)
{
        if (level != SOL_TCP)
                return inet_csk_compat_getsockopt(sk, level, optname,
                                                  optval, optlen);
        return do_tcp_getsockopt(sk, level, optname, optval, optlen);
}
EXPORT_SYMBOL(compat_tcp_getsockopt);
#endif

#ifdef CONFIG_TCP_MD5SIG
static struct tcp_md5sig_pool __percpu *tcp_md5sig_pool __read_mostly;
static DEFINE_MUTEX(tcp_md5sig_mutex);

static void __tcp_free_md5sig_pool(struct tcp_md5sig_pool __percpu *pool)
{
        int cpu;

        for_each_possible_cpu(cpu) {
                struct tcp_md5sig_pool *p = per_cpu_ptr(pool, cpu);

                if (p->md5_desc.tfm)
                        crypto_free_hash(p->md5_desc.tfm);
        }
        free_percpu(pool);
}

static void __tcp_alloc_md5sig_pool(void)
{
        int cpu;
        struct tcp_md5sig_pool __percpu *pool;

        pool = alloc_percpu(struct tcp_md5sig_pool);
        if (!pool)
                return;

        for_each_possible_cpu(cpu) {
                struct crypto_hash *hash;

                hash = crypto_alloc_hash("md5", 0, CRYPTO_ALG_ASYNC);
                if (IS_ERR_OR_NULL(hash))
                        goto out_free;

                per_cpu_ptr(pool, cpu)->md5_desc.tfm = hash;
        }
        /* before setting tcp_md5sig_pool, we must commit all writes
         * to memory. See ACCESS_ONCE() in tcp_get_md5sig_pool()
         */
        smp_wmb();
        tcp_md5sig_pool = pool;
        return;
out_free:
        __tcp_free_md5sig_pool(pool);
}

bool tcp_alloc_md5sig_pool(void)
{
        if (unlikely(!tcp_md5sig_pool)) {
                mutex_lock(&tcp_md5sig_mutex);

                if (!tcp_md5sig_pool)
                        __tcp_alloc_md5sig_pool();

                mutex_unlock(&tcp_md5sig_mutex);
        }
        return tcp_md5sig_pool != NULL;
}
EXPORT_SYMBOL(tcp_alloc_md5sig_pool);


/**
 *      tcp_get_md5sig_pool - get md5sig_pool for this user
 *
 *      We use percpu structure, so if we succeed, we exit with preemption
 *      and BH disabled, to make sure another thread or softirq handling
 *      wont try to get same context.
 */
struct tcp_md5sig_pool *tcp_get_md5sig_pool(void)
{
        struct tcp_md5sig_pool __percpu *p;

        local_bh_disable();
        p = ACCESS_ONCE(tcp_md5sig_pool);
        if (p)
                return __this_cpu_ptr(p);

        local_bh_enable();
        return NULL;
}
EXPORT_SYMBOL(tcp_get_md5sig_pool);

int tcp_md5_hash_header(struct tcp_md5sig_pool *hp,
                        const struct tcphdr *th)
{
        struct scatterlist sg;
        struct tcphdr hdr;
        int err;

        /* We are not allowed to change tcphdr, make a local copy */
        memcpy(&hdr, th, sizeof(hdr));
        hdr.check = 0;

        /* options aren't included in the hash */
        sg_init_one(&sg, &hdr, sizeof(hdr));
        err = crypto_hash_update(&hp->md5_desc, &sg, sizeof(hdr));
        return err;
}
EXPORT_SYMBOL(tcp_md5_hash_header);

int tcp_md5_hash_skb_data(struct tcp_md5sig_pool *hp,
                          const struct sk_buff *skb, unsigned int header_len)
{
        struct scatterlist sg;
        const struct tcphdr *tp = tcp_hdr(skb);
        struct hash_desc *desc = &hp->md5_desc;
        unsigned int i;
        const unsigned int head_data_len = skb_headlen(skb) > header_len ?
                                           skb_headlen(skb) - header_len : 0;
        const struct skb_shared_info *shi = skb_shinfo(skb);
        struct sk_buff *frag_iter;

        sg_init_table(&sg, 1);

        sg_set_buf(&sg, ((u8 *) tp) + header_len, head_data_len);
        if (crypto_hash_update(desc, &sg, head_data_len))
                return 1;

        for (i = 0; i < shi->nr_frags; ++i) {
                const struct skb_frag_struct *f = &shi->frags[i];
                unsigned int offset = f->page_offset;
                struct page *page = skb_frag_page(f) + (offset >> PAGE_SHIFT);

                sg_set_page(&sg, page, skb_frag_size(f),
                            offset_in_page(offset));
                if (crypto_hash_update(desc, &sg, skb_frag_size(f)))
                        return 1;
        }

        skb_walk_frags(skb, frag_iter)
                if (tcp_md5_hash_skb_data(hp, frag_iter, 0))
                        return 1;

        return 0;
}
EXPORT_SYMBOL(tcp_md5_hash_skb_data);

int tcp_md5_hash_key(struct tcp_md5sig_pool *hp, const struct tcp_md5sig_key *key)
{
        struct scatterlist sg;

        sg_init_one(&sg, key->key, key->keylen);
        return crypto_hash_update(&hp->md5_desc, &sg, key->keylen);
}
EXPORT_SYMBOL(tcp_md5_hash_key);

#endif

void tcp_done(struct sock *sk)
{
        struct request_sock *req = tcp_sk(sk)->fastopen_rsk;

        if (sk->sk_state == TCP_SYN_SENT || sk->sk_state == TCP_SYN_RECV)
                TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_ATTEMPTFAILS);

        tcp_set_state(sk, TCP_CLOSE);
        tcp_clear_xmit_timers(sk);
        if (req != NULL)
                reqsk_fastopen_remove(sk, req, false);

        sk->sk_shutdown = SHUTDOWN_MASK;

        if (!sock_flag(sk, SOCK_DEAD))
                sk->sk_state_change(sk);
        else
                inet_csk_destroy_sock(sk);
}
EXPORT_SYMBOL_GPL(tcp_done);

extern struct tcp_congestion_ops tcp_reno;

static __initdata unsigned long thash_entries;
static int __init set_thash_entries(char *str)
{
        ssize_t ret;

        if (!str)
                return 0;

        ret = kstrtoul(str, 0, &thash_entries);
        if (ret)
                return 0;

        return 1;
}
__setup("thash_entries=", set_thash_entries);

static void tcp_init_mem(void)
{
        unsigned long limit = nr_free_buffer_pages() / 8;
        limit = max(limit, 128UL);
        sysctl_tcp_mem[0] = limit / 4 * 3;
        sysctl_tcp_mem[1] = limit;
        sysctl_tcp_mem[2] = sysctl_tcp_mem[0] * 2;
}

void __init tcp_init(void)
{
        struct sk_buff *skb = NULL;
        unsigned long limit;
        int max_rshare, max_wshare, cnt;
        unsigned int i;

        BUILD_BUG_ON(sizeof(struct tcp_skb_cb) > sizeof(skb->cb));

        percpu_counter_init(&tcp_sockets_allocated, 0);
        percpu_counter_init(&tcp_orphan_count, 0);
        tcp_hashinfo.bind_bucket_cachep =
                kmem_cache_create("tcp_bind_bucket",
                                  sizeof(struct inet_bind_bucket), 0,
                                  SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);

        /* Size and allocate the main established and bind bucket
         * hash tables.
         *
         * The methodology is similar to that of the buffer cache.
         */
        tcp_hashinfo.ehash =
                alloc_large_system_hash("TCP established",
                                        sizeof(struct inet_ehash_bucket),
                                        thash_entries,
                                        17, /* one slot per 128 KB of memory */
                                        0,
                                        NULL,
                                        &tcp_hashinfo.ehash_mask,
                                        0,
                                        thash_entries ? 0 : 512 * 1024);
        for (i = 0; i <= tcp_hashinfo.ehash_mask; i++)
                INIT_HLIST_NULLS_HEAD(&tcp_hashinfo.ehash[i].chain, i);

        if (inet_ehash_locks_alloc(&tcp_hashinfo))
                panic("TCP: failed to alloc ehash_locks");
        tcp_hashinfo.bhash =
                alloc_large_system_hash("TCP bind",
                                        sizeof(struct inet_bind_hashbucket),
                                        tcp_hashinfo.ehash_mask + 1,
                                        17, /* one slot per 128 KB of memory */
                                        0,
                                        &tcp_hashinfo.bhash_size,
                                        NULL,
                                        0,
                                        64 * 1024);
        tcp_hashinfo.bhash_size = 1U << tcp_hashinfo.bhash_size;
        for (i = 0; i < tcp_hashinfo.bhash_size; i++) {
                spin_lock_init(&tcp_hashinfo.bhash[i].lock);
                INIT_HLIST_HEAD(&tcp_hashinfo.bhash[i].chain);
        }


        cnt = tcp_hashinfo.ehash_mask + 1;

        tcp_death_row.sysctl_max_tw_buckets = cnt / 2;
        sysctl_tcp_max_orphans = cnt / 2;
        sysctl_max_syn_backlog = max(128, cnt / 256);

        tcp_init_mem();
        /* Set per-socket limits to no more than 1/128 the pressure threshold */
        limit = nr_free_buffer_pages() << (PAGE_SHIFT - 7);
        max_wshare = min(4UL*1024*1024, limit);
        max_rshare = min(6UL*1024*1024, limit);

        sysctl_tcp_wmem[0] = SK_MEM_QUANTUM;
        sysctl_tcp_wmem[1] = 16*1024;
        sysctl_tcp_wmem[2] = max(64*1024, max_wshare);

        sysctl_tcp_rmem[0] = SK_MEM_QUANTUM;
        sysctl_tcp_rmem[1] = 87380;
        sysctl_tcp_rmem[2] = max(87380, max_rshare);

        pr_info("Hash tables configured (established %u bind %u)\n",
                tcp_hashinfo.ehash_mask + 1, tcp_hashinfo.bhash_size);

        tcp_metrics_init();

        tcp_register_congestion_control(&tcp_reno);

        tcp_tasklet_init();
}

```

- Home
- Development
- Services
- Training
- Docs
- Community
- Company
- Blog