Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/include/net/sock.h

```
 1 /*
 2  * INET         An implementation of the TCP/IP protocol suite for the LINUX
 3  *              operating system.  INET is implemented using the  BSD Socket
 4  *              interface as the means of communication with the user level.
 5  *
 6  *              Definitions for the AF_INET socket handler.
 7  *
 8  * Version:     @(#)sock.h      1.0.4   05/13/93
 9  *
10  * Authors:     Ross Biro
11  *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12  *              Corey Minyard <wf-rch!minyard@relay.EU.net>
13  *              Florian La Roche <flla@stud.uni-sb.de>
14  *
15  * Fixes:
16  *              Alan Cox        :       Volatiles in skbuff pointers. See
17  *                                      skbuff comments. May be overdone,
18  *                                      better to prove they can be removed
19  *                                      than the reverse.
20  *              Alan Cox        :       Added a zapped field for tcp to note
21  *                                      a socket is reset and must stay shut up
22  *              Alan Cox        :       New fields for options
23  *      Pauline Middelink       :       identd support
24  *              Alan Cox        :       Eliminate low level recv/recvfrom
25  *              David S. Miller :       New socket lookup architecture.
26  *              Steve Whitehouse:       Default routines for sock_ops
27  *              Arnaldo C. Melo :       removed net_pinfo, tp_pinfo and made
28  *                                      protinfo be just a void pointer, as the
29  *                                      protocol specific parts were moved to
30  *                                      respective headers and ipv4/v6, etc now
31  *                                      use private slabcaches for its socks
32  *              Pedro Hortas    :       New flags field for socket options
33  *
34  *
35  *              This program is free software; you can redistribute it and/or
36  *              modify it under the terms of the GNU General Public License
37  *              as published by the Free Software Foundation; either version
38  *              2 of the License, or (at your option) any later version.
39  */
40 #ifndef _SOCK_H
41 #define _SOCK_H
42
43 #include <linux/hardirq.h>
44 #include <linux/kernel.h>
45 #include <linux/list.h>
46 #include <linux/list_nulls.h>
47 #include <linux/timer.h>
```

```
 48 #include <linux/cache.h>
 49 #include <linux/bitops.h>
 50 #include <linux/lockdep.h>
 51 #include <linux/netdevice.h>
 52 #include <linux/skbuff.h>       /* struct sk_buff */
 53 #include <linux/mm.h>
 54 #include <linux/security.h>
 55 #include <linux/slab.h>
 56 #include <linux/uaccess.h>
 57 #include <linux/page_counter.h>
 58 #include <linux/memcontrol.h>
 59 #include <linux/static_key.h>
 60 #include <linux/sched.h>
 61
 62 #include <linux/filter.h>
 63 #include <linux/rculist_nulls.h>
 64 #include <linux/poll.h>
 65
 66 #include <linux/atomic.h>
 67 #include <net/dst.h>
 68 #include <net/checksum.h>
 69 #include <net/tcp_states.h>
 70 #include <linux/net_tstamp.h>
 71
 72 struct cgroup;
 73 struct cgroup_subsys;
 74 #ifdef CONFIG_NET
 75 int mem_cgroup_sockets_init(struct mem_cgroup *memcg, struct cgroup_subsys *ss);
 76 void mem_cgroup_sockets_destroy(struct mem_cgroup *memcg);
 77 #else
 78 static inline
 79 int mem_cgroup_sockets_init(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
 80 {
 81         return 0;
 82 }
 83 static inline
 84 void mem_cgroup_sockets_destroy(struct mem_cgroup *memcg)
 85 {
 86 }
 87 #endif
 88 /*
 89  * This structure really needs to be cleaned up.
 90  * Most of it is for TCP, and not used by any of
 91  * the other protocols.
 92  */
 93
 94 /* Define this to get the SOCK_DBG debugging facility. */
 95 #define SOCK_DEBUGGING
 96 #ifdef SOCK_DEBUGGING
 97 #define SOCK_DEBUG(sk, msg...) do { if ((sk) && sock_flag((sk), SOCK_DBG)) \
 98                                         printk(KERN_DEBUG msg); } while (0)
 99 #else
100 /* Validate arguments and do nothing */
101 static inline __printf(2, 3)
102 void SOCK_DEBUG(const struct sock *sk, const char *msg, ...)
103 {
104 }
105 #endif
106
107 /* This is the per-socket lock.  The spinlock provides a synchronization
108  * between user contexts and software interrupt processing, whereas the
109  * mini-semaphore synchronizes multiple users amongst themselves.
110  */
111 typedef struct {
112         spinlock_t              slock;
113         int                     owned;
114         wait_queue_head_t       wq;
115         /*
```

```
116          * We express the mutex-alike socket_lock semantics
117          * to the lock validator by explicitly managing
118          * the slock as a lock variant (in addition to
119          * the slock itself):
120          */
121 #ifdef CONFIG_DEBUG_LOCK_ALLOC
122         struct lockdep_map dep_map;
123 #endif
124 } socket_lock_t;
125
126 struct sock;
127 struct proto;
128 struct net;
129
130 typedef __u32 __bitwise __portpair;
131 typedef __u64 __bitwise __addrpair;
132
133 /**
134  *      struct sock_common - minimal network layer representation of sockets
135  *      @skc_daddr: Foreign IPv4 addr
136  *      @skc_rcv_saddr: Bound local IPv4 addr
137  *      @skc_hash: hash value used with various protocol lookup tables
138  *      @skc_u16hashes: two u16 hash values used by UDP lookup tables
139  *      @skc_dport: placeholder for inet_dport/tw_dport
140  *      @skc_num: placeholder for inet_num/tw_num
141  *      @skc_family: network address family
142  *      @skc_state: Connection state
143  *      @skc_reuse: %SO_REUSEADDR setting
144  *      @skc_reuseport: %SO_REUSEPORT setting
145  *      @skc_bound_dev_if: bound device index if != 0
146  *      @skc_bind_node: bind hash linkage for various protocol lookup tables
147  *      @skc_portaddr_node: second hash linkage for UDP/UDP-Lite protocol
148  *      @skc_prot: protocol handlers inside a network family
149  *      @skc_net: reference to the network namespace of this socket
150  *      @skc_node: main hash linkage for various protocol lookup tables
151  *      @skc_nulls_node: main hash linkage for TCP/UDP/UDP-Lite protocol
152  *      @skc_tx_queue_mapping: tx queue number for this connection
153  *      @skc_refcnt: reference count
154  *
155  *      This is the minimal network layer representation of sockets, the header
156  *      for struct sock and struct inet_timewait_sock.
157  */
158 struct sock_common {
159         /* skc_daddr and skc_rcv_saddr must be grouped on a 8 bytes aligned
160          * address on 64bit arches : cf INET_MATCH()
161          */
162         union {
163                 __addrpair      skc_addrpair;
164                 struct {
165                         __be32  skc_daddr;
166                         __be32  skc_rcv_saddr;
167                 };
168         };
169         union  {
170                 unsigned int    skc_hash;
171                 __u16           skc_u16hashes[2];
172         };
173         /* skc_dport && skc_num must be grouped as well */
174         union {
175                 __portpair      skc_portpair;
176                 struct {
177                         __be16  skc_dport;
178                         __u16   skc_num;
179                 };
180         };
181
182         unsigned short          skc_family;
183         volatile unsigned char  skc_state;
```

```
184         unsigned char           skc_reuse:4;
185         unsigned char           skc_reuseport:1;
186         unsigned char           skc_ipv6only:1;
187         unsigned char           skc_net_refcnt:1;
188         int                     skc_bound_dev_if;
189         union {
190                 struct hlist_node       skc_bind_node;
191                 struct hlist_nulls_node skc_portaddr_node;
192         };
193         struct proto            *skc_prot;
194         possible_net_t          skc_net;
195
196 #if IS_ENABLED(CONFIG_IPV6)
197         struct in6_addr         skc_v6_daddr;
198         struct in6_addr         skc_v6_rcv_saddr;
199 #endif
200
201         atomic64_t              skc_cookie;
202
203         /*
204          * fields between dontcopy_begin/dontcopy_end
205          * are not copied in sock_copy()
206          */
207         /* private: */
208         int                     skc_dontcopy_begin[0];
209         /* public: */
210         union {
211                 struct hlist_node       skc_node;
212                 struct hlist_nulls_node skc_nulls_node;
213         };
214         int                     skc_tx_queue_mapping;
215         atomic_t                skc_refcnt;
216         /* private: */
217         int                     skc_dontcopy_end[0];
218         /* public: */
219 };
220
221 struct cg_proto;
222 /**
223   *     struct sock - network layer representation of sockets
224   *     @__sk_common: shared layout with inet_timewait_sock
225   *     @sk_shutdown: mask of %SEND_SHUTDOWN and/or %RCV_SHUTDOWN
226   *     @sk_userlocks: %SO_SNDBUF and %SO_RCVBUF settings
227   *     @sk_lock:       synchronizer
228   *     @sk_rcvbuf: size of receive buffer in bytes
229   *     @sk_wq: sock wait queue and async head
230   *     @sk_rx_dst: receive input route used by early demux
231   *     @sk_dst_cache: destination cache
232   *     @sk_dst_lock: destination cache lock
233   *     @sk_policy: flow policy
234   *     @sk_receive_queue: incoming packets
235   *     @sk_wmem_alloc: transmit queue bytes committed
236   *     @sk_write_queue: Packet sending queue
237   *     @sk_omem_alloc: "o" is "option" or "other"
238   *     @sk_wmem_queued: persistent queue size
239   *     @sk_forward_alloc: space allocated forward
240   *     @sk_napi_id: id of the last napi context to receive data for sk
241   *     @sk_ll_usec: usecs to busypoll when there is no data
242   *     @sk_allocation: allocation mode
243   *     @sk_pacing_rate: Pacing rate (if supported by transport/packet scheduler)
244   *     @sk_max_pacing_rate: Maximum pacing rate (%SO_MAX_PACING_RATE)
245   *     @sk_sndbuf: size of send buffer in bytes
246   *     @sk_flags: %SO_LINGER (l_onoff), %SO_BROADCAST, %SO_KEEPALIVE,
247   *                %SO_OOBINLINE settings, %SO_TIMESTAMPING settings
248   *     @sk_no_check_tx: %SO_NO_CHECK setting, set checksum in TX packets
249   *     @sk_no_check_rx: allow zero checksum in RX packets
250   *     @sk_route_caps: route capabilities (e.g. %NETIF_F_TSO)
251   *     @sk_route_nocaps: forbidden route capabilities (e.g NETIF_F_GSO_MASK)
```

```
252  *     @sk_gso_type: GSO type (e.g. %SKB_GSO_TCPV4)
253  *     @sk_gso_max_size: Maximum GSO segment size to build
254  *     @sk_gso_max_segs: Maximum number of GSO segments
255  *     @sk_lingertime: %SO_LINGER l_linger setting
256  *     @sk_backlog: always used with the per-socket spinlock held
257  *     @sk_callback_lock: used with the callbacks in the end of this struct
258  *     @sk_error_queue: rarely used
259  *     @sk_prot_creator: sk_prot of original sock creator (see ipv6_setsockopt,
260  *                       IPV6_ADDRFORM for instance)
261  *     @sk_err: last error
262  *     @sk_err_soft: errors that don't cause failure but are the cause of a
263  *                   persistent failure not just 'timed out'
264  *     @sk_drops: raw/udp drops counter
265  *     @sk_ack_backlog: current listen backlog
266  *     @sk_max_ack_backlog: listen backlog set in listen()
267  *     @sk_priority: %SO_PRIORITY setting
268  *     @sk_cgrp_prioidx: socket group's priority map index
269  *     @sk_type: socket type (%SOCK_STREAM, etc)
270  *     @sk_protocol: which protocol this socket belongs in this network family
271  *     @sk_peer_pid: &struct pid for this socket's peer
272  *     @sk_peer_cred: %SO_PEERCRED setting
273  *     @sk_rcvlowat: %SO_RCVLOWAT setting
274  *     @sk_rcvtimeo: %SO_RCVTIMEO setting
275  *     @sk_sndtimeo: %SO_SNDTIMEO setting
276  *     @sk_rxhash: flow hash received from netif layer
277  *     @sk_incoming_cpu: record cpu processing incoming packets
278  *     @sk_txhash: computed flow hash for use on transmit
279  *     @sk_filter: socket filtering instructions
280  *     @sk_timer: sock cleanup timer
281  *     @sk_stamp: time stamp of last packet received
282  *     @sk_tsflags: SO_TIMESTAMPING socket options
283  *     @sk_tskey: counter to disambiguate concurrent tstamp requests
284  *     @sk_socket: Identd and reporting IO signals
285  *     @sk_user_data: RPC layer private data
286  *     @sk_frag: cached page frag
287  *     @sk_peek_off: current peek_offset value
288  *     @sk_send_head: front of stuff to transmit
289  *     @sk_security: used by security modules
290  *     @sk_mark: generic packet mark
291  *     @sk_classid: this socket's cgroup classid
292  *     @sk_cgrp: this socket's cgroup-specific proto data
293  *     @sk_write_pending: a write to stream socket waits to start
294  *     @sk_state_change: callback to indicate change in the state of the sock
295  *     @sk_data_ready: callback to indicate there is data to be processed
296  *     @sk_write_space: callback to indicate there is bf sending space available
297  *     @sk_error_report: callback to indicate errors (e.g. %MSG_ERRQUEUE)
298  *     @sk_backlog_rcv: callback to process the backlog
299  *     @sk_destruct: called at sock freeing time, i.e. when all refcnt == 0
300  */
301 struct sock {
302         /*
303          * Now struct inet_timewait_sock also uses sock_common, so please just
304          * don't add nothing before this first member (__sk_common) --acme
305          */
306         struct sock_common      __sk_common;
307 #define sk_node                 __sk_common.skc_node
308 #define sk_nulls_node           __sk_common.skc_nulls_node
309 #define sk_refcnt               __sk_common.skc_refcnt
310 #define sk_tx_queue_mapping     __sk_common.skc_tx_queue_mapping
311
312 #define sk_dontcopy_begin       __sk_common.skc_dontcopy_begin
313 #define sk_dontcopy_end         __sk_common.skc_dontcopy_end
314 #define sk_hash                 __sk_common.skc_hash
315 #define sk_portpair             __sk_common.skc_portpair
316 #define sk_num                  __sk_common.skc_num
317 #define sk_dport                __sk_common.skc_dport
318 #define sk_addrpair             __sk_common.skc_addrpair
319 #define sk_daddr                __sk_common.skc_daddr
```

```
320 #define sk_rcv_saddr            __sk_common.skc_rcv_saddr
321 #define sk_family               __sk_common.skc_family
322 #define sk_state                __sk_common.skc_state
323 #define sk_reuse                __sk_common.skc_reuse
324 #define sk_reuseport            __sk_common.skc_reuseport
325 #define sk_ipv6only             __sk_common.skc_ipv6only
326 #define sk_net_refcnt           __sk_common.skc_net_refcnt
327 #define sk_bound_dev_if         __sk_common.skc_bound_dev_if
328 #define sk_bind_node            __sk_common.skc_bind_node
329 #define sk_prot                 __sk_common.skc_prot
330 #define sk_net                  __sk_common.skc_net
331 #define sk_v6_daddr             __sk_common.skc_v6_daddr
332 #define sk_v6_rcv_saddr __sk_common.skc_v6_rcv_saddr
333 #define sk_cookie               __sk_common.skc_cookie
334
335         socket_lock_t           sk_lock;
336         struct sk_buff_head     sk_receive_queue;
337         /*
338          * The backlog queue is special, it is always used with
339          * the per-socket spinlock held and requires low latency
340          * access. Therefore we special case it's implementation.
341          * Note : rmem_alloc is in this structure to fill a hole
342          * on 64bit arches, not because its logically part of
343          * backlog.
344          */
345         struct {
346                 atomic_t        rmem_alloc;
347                 int             len;
348                 struct sk_buff  *head;
349                 struct sk_buff  *tail;
350         } sk_backlog;
351 #define sk_rmem_alloc sk_backlog.rmem_alloc
352         int                     sk_forward_alloc;
353 #ifdef CONFIG_RPS
354         __u32                   sk_rxhash;
355 #endif
356         u16                     sk_incoming_cpu;
357         /* 16bit hole
358          * Warned : sk_incoming_cpu can be set from softirq,
359          * Do not use this hole without fully understanding possible issues.
360          */
361
362         __u32                   sk_txhash;
363 #ifdef CONFIG_NET_RX_BUSY_POLL
364         unsigned int            sk_napi_id;
365         unsigned int            sk_ll_usec;
366 #endif
367         atomic_t                sk_drops;
368         int                     sk_rcvbuf;
369
370         struct sk_filter __rcu  *sk_filter;
371         struct socket_wq __rcu  *sk_wq;
372
373 #ifdef CONFIG_XFRM
374         struct xfrm_policy      *sk_policy[2];
375 #endif
376         unsigned long           sk_flags;
377         struct dst_entry        *sk_rx_dst;
378         struct dst_entry __rcu  *sk_dst_cache;
379         spinlock_t              sk_dst_lock;
380         atomic_t                sk_wmem_alloc;
381         atomic_t                sk_omem_alloc;
382         int                     sk_sndbuf;
383         struct sk_buff_head     sk_write_queue;
384         kmemcheck_bitfield_begin(flags);
385         unsigned int            sk_shutdown  : 2,
386                                 sk_no_check_tx : 1,
387                                 sk_no_check_rx : 1,
```

```
388                                 sk_userlocks : 4,
389                                 sk_protocol  : 8,
390                                 sk_type      : 16;
391         kmemcheck_bitfield_end(flags);
392         int                     sk_wmem_queued;
393         gfp_t                   sk_allocation;
394         u32                     sk_pacing_rate; /* bytes per second */
395         u32                     sk_max_pacing_rate;
396         netdev_features_t       sk_route_caps;
397         netdev_features_t       sk_route_nocaps;
398         int                     sk_gso_type;
399         unsigned int            sk_gso_max_size;
400         u16                     sk_gso_max_segs;
401         int                     sk_rcvlowat;
402         unsigned long           sk_lingertime;
403         struct sk_buff_head     sk_error_queue;
404         struct proto            *sk_prot_creator;
405         rwlock_t                sk_callback_lock;
406         int                     sk_err,
407                                 sk_err_soft;
408         u32                     sk_ack_backlog;
409         u32                     sk_max_ack_backlog;
410         __u32                   sk_priority;
411 #if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
412         __u32                   sk_cgrp_prioidx;
413 #endif
414         struct pid              *sk_peer_pid;
415         const struct cred       *sk_peer_cred;
416         long                    sk_rcvtimeo;
417         long                    sk_sndtimeo;
418         struct timer_list       sk_timer;
419         ktime_t                 sk_stamp;
420         u16                     sk_tsflags;
421         u32                     sk_tskey;
422         struct socket           *sk_socket;
423         void                    *sk_user_data;
424         struct page_frag        sk_frag;
425         struct sk_buff          *sk_send_head;
426         __s32                   sk_peek_off;
427         int                     sk_write_pending;
428 #ifdef CONFIG_SECURITY
429         void                    *sk_security;
430 #endif
431         __u32                   sk_mark;
432         u32                     sk_classid;
433         struct cg_proto         *sk_cgrp;
434         void                    (*sk_state_change)(struct sock *sk);
435         void                    (*sk_data_ready)(struct sock *sk);
436         void                    (*sk_write_space)(struct sock *sk);
437         void                    (*sk_error_report)(struct sock *sk);
438         int                     (*sk_backlog_rcv)(struct sock *sk,
439                                                   struct sk_buff *skb);
440         void                    (*sk_destruct)(struct sock *sk);
441 };
442
443 #define __sk_user_data(sk) ((*((void __rcu **)&(sk)->sk_user_data)))
444
445 #define rcu_dereference_sk_user_data(sk)        rcu_dereference(__sk_user_data((sk)))
446 #define rcu_assign_sk_user_data(sk, ptr)        rcu_assign_pointer(__sk_user_data((sk)), ptr)
447
448 /*
449  * SK_CAN_REUSE and SK_NO_REUSE on a socket mean that the socket is OK
450  * or not whether his port will be reused by someone else. SK_FORCE_REUSE
451  * on a socket means that the socket will reuse everybody else's port
452  * without looking at the other's sk_reuse value.
453  */
454
455 #define SK_NO_REUSE     0
```

```
456 #define SK_CAN_REUSE    1
457 #define SK_FORCE_REUSE  2
458
459 static inline int sk_peek_offset(struct sock *sk, int flags)
460 {
461         if ((flags & MSG_PEEK) && (sk->sk_peek_off >= 0))
462                 return sk->sk_peek_off;
463         else
464                 return 0;
465 }
466
467 static inline void sk_peek_offset_bwd(struct sock *sk, int val)
468 {
469         if (sk->sk_peek_off >= 0) {
470                 if (sk->sk_peek_off >= val)
471                         sk->sk_peek_off -= val;
472                 else
473                         sk->sk_peek_off = 0;
474         }
475 }
476
477 static inline void sk_peek_offset_fwd(struct sock *sk, int val)
478 {
479         if (sk->sk_peek_off >= 0)
480                 sk->sk_peek_off += val;
481 }
482
483 /*
484  * Hashed lists helper routines
485  */
486 static inline struct sock *sk_entry(const struct hlist_node *node)
487 {
488         return hlist_entry(node, struct sock, sk_node);
489 }
490
491 static inline struct sock *__sk_head(const struct hlist_head *head)
492 {
493         return hlist_entry(head->first, struct sock, sk_node);
494 }
495
496 static inline struct sock *sk_head(const struct hlist_head *head)
497 {
498         return hlist_empty(head) ? NULL : __sk_head(head);
499 }
500
501 static inline struct sock *__sk_nulls_head(const struct hlist_nulls_head *head)
502 {
503         return hlist_nulls_entry(head->first, struct sock, sk_nulls_node);
504 }
505
506 static inline struct sock *sk_nulls_head(const struct hlist_nulls_head *head)
507 {
508         return hlist_nulls_empty(head) ? NULL : __sk_nulls_head(head);
509 }
510
511 static inline struct sock *sk_next(const struct sock *sk)
512 {
513         return sk->sk_node.next ?
514                 hlist_entry(sk->sk_node.next, struct sock, sk_node) : NULL;
515 }
516
517 static inline struct sock *sk_nulls_next(const struct sock *sk)
518 {
519         return (!is_a_nulls(sk->sk_nulls_node.next)) ?
520                 hlist_nulls_entry(sk->sk_nulls_node.next,
521                                   struct sock, sk_nulls_node) :
522                 NULL;
523 }
```

```
524
525 static inline bool sk_unhashed(const struct sock *sk)
526 {
527         return hlist_unhashed(&sk->sk_node);
528 }
529
530 static inline bool sk_hashed(const struct sock *sk)
531 {
532         return !sk_unhashed(sk);
533 }
534
535 static inline void sk_node_init(struct hlist_node *node)
536 {
537         node->pprev = NULL;
538 }
539
540 static inline void sk_nulls_node_init(struct hlist_nulls_node *node)
541 {
542         node->pprev = NULL;
543 }
544
545 static inline void __sk_del_node(struct sock *sk)
546 {
547         __hlist_del(&sk->sk_node);
548 }
549
550 /* NB: equivalent to hlist_del_init_rcu */
551 static inline bool __sk_del_node_init(struct sock *sk)
552 {
553         if (sk_hashed(sk)) {
554                 __sk_del_node(sk);
555                 sk_node_init(&sk->sk_node);
556                 return true;
557         }
558         return false;
559 }
560
561 /* Grab socket reference count. This operation is valid only
562    when sk is ALREADY grabbed f.e. it is found in hash table
563    or a list and the lookup is made under lock preventing hash table
564    modifications.
565  */
566
567 static inline void sock_hold(struct sock *sk)
568 {
569         atomic_inc(&sk->sk_refcnt);
570 }
571
572 /* Ungrab socket in the context, which assumes that socket refcnt
573    cannot hit zero, f.e. it is true in context of any socketcall.
574  */
575 static inline void __sock_put(struct sock *sk)
576 {
577         atomic_dec(&sk->sk_refcnt);
578 }
579
580 static inline bool sk_del_node_init(struct sock *sk)
581 {
582         bool rc = __sk_del_node_init(sk);
583
584         if (rc) {
585                 /* paranoid for a while -acme */
586                 WARN_ON(atomic_read(&sk->sk_refcnt) == 1);
587                 __sock_put(sk);
588         }
589         return rc;
590 }
591 #define sk_del_node_init_rcu(sk)        sk_del_node_init(sk)
```

```
592
593 static inline bool __sk_nulls_del_node_init_rcu(struct sock *sk)
594 {
595         if (sk_hashed(sk)) {
596                 hlist_nulls_del_init_rcu(&sk->sk_nulls_node);
597                 return true;
598         }
599         return false;
600 }
601
602 static inline bool sk_nulls_del_node_init_rcu(struct sock *sk)
603 {
604         bool rc = __sk_nulls_del_node_init_rcu(sk);
605
606         if (rc) {
607                 /* paranoid for a while -acme */
608                 WARN_ON(atomic_read(&sk->sk_refcnt) == 1);
609                 __sock_put(sk);
610         }
611         return rc;
612 }
613
614 static inline void __sk_add_node(struct sock *sk, struct hlist_head *list)
615 {
616         hlist_add_head(&sk->sk_node, list);
617 }
618
619 static inline void sk_add_node(struct sock *sk, struct hlist_head *list)
620 {
621         sock_hold(sk);
622         __sk_add_node(sk, list);
623 }
624
625 static inline void sk_add_node_rcu(struct sock *sk, struct hlist_head *list)
626 {
627         sock_hold(sk);
628         hlist_add_head_rcu(&sk->sk_node, list);
629 }
630
631 static inline void __sk_nulls_add_node_rcu(struct sock *sk, struct hlist_nulls_head *list)
632 {
633         hlist_nulls_add_head_rcu(&sk->sk_nulls_node, list);
634 }
635
636 static inline void sk_nulls_add_node_rcu(struct sock *sk, struct hlist_nulls_head *list)
637 {
638         sock_hold(sk);
639         __sk_nulls_add_node_rcu(sk, list);
640 }
641
642 static inline void __sk_del_bind_node(struct sock *sk)
643 {
644         __hlist_del(&sk->sk_bind_node);
645 }
646
647 static inline void sk_add_bind_node(struct sock *sk,
648                                     struct hlist_head *list)
649 {
650         hlist_add_head(&sk->sk_bind_node, list);
651 }
652
653 #define sk_for_each(__sk, list) \
654         hlist_for_each_entry(__sk, list, sk_node)
655 #define sk_for_each_rcu(__sk, list) \
656         hlist_for_each_entry_rcu(__sk, list, sk_node)
657 #define sk_nulls_for_each(__sk, node, list) \
658         hlist_nulls_for_each_entry(__sk, node, list, sk_nulls_node)
659 #define sk_nulls_for_each_rcu(__sk, node, list) \
```

```
660         hlist_nulls_for_each_entry_rcu(__sk, node, list, sk_nulls_node)
661 #define sk_for_each_from(__sk) \
662         hlist_for_each_entry_from(__sk, sk_node)
663 #define sk_nulls_for_each_from(__sk, node) \
664         if (__sk && ({ node = &(__sk)->sk_nulls_node; 1; })) \
665                 hlist_nulls_for_each_entry_from(__sk, node, sk_nulls_node)
666 #define sk_for_each_safe(__sk, tmp, list) \
667         hlist_for_each_entry_safe(__sk, tmp, list, sk_node)
668 #define sk_for_each_bound(__sk, list) \
669         hlist_for_each_entry(__sk, list, sk_bind_node)
670
671 /**
672  * sk_nulls_for_each_entry_offset - iterate over a list at a given struct offset
673  * @tpos:       the type * to use as a loop cursor.
674  * @pos:        the &struct hlist_node to use as a loop cursor.
675  * @head:       the head for your list.
676  * @offset:     offset of hlist_node within the struct.
677  *
678  */
679 #define sk_nulls_for_each_entry_offset(tpos, pos, head, offset)                 \
680         for (pos = (head)->first;                                               \
681              (!is_a_nulls(pos)) &&                                              \
682                 ({ tpos = (typeof(*tpos) *)((void *)pos - offset); 1;});        \
683              pos = pos->next)
684
685 static inline struct user_namespace *sk_user_ns(struct sock *sk)
686 {
687         /* Careful only use this in a context where these parameters
688          * can not change and must all be valid, such as recvmsg from
689          * userspace.
690          */
691         return sk->sk_socket->file->f_cred->user_ns;
692 }
693
694 /* Sock flags */
695 enum sock_flags {
696         SOCK_DEAD,
697         SOCK_DONE,
698         SOCK_URGINLINE,
699         SOCK_KEEPOPEN,
700         SOCK_LINGER,
701         SOCK_DESTROY,
702         SOCK_BROADCAST,
703         SOCK_TIMESTAMP,
704         SOCK_ZAPPED,
705         SOCK_USE_WRITE_QUEUE, /* whether to call sk->sk_write_space in sock_wfree */
706         SOCK_DBG, /* %SO_DEBUG setting */
707         SOCK_RCVTSTAMP, /* %SO_TIMESTAMP setting */
708         SOCK_RCVTSTAMPNS, /* %SO_TIMESTAMPNS setting */
709         SOCK_LOCALROUTE, /* route locally only, %SO_DONTROUTE setting */
710         SOCK_QUEUE_SHRUNK, /* write queue has been shrunk recently */
711         SOCK_MEMALLOC, /* VM depends on this socket for swapping */
712         SOCK_TIMESTAMPING_RX_SOFTWARE,  /* %SOF_TIMESTAMPING_RX_SOFTWARE */
713         SOCK_FASYNC, /* fasync() active */
714         SOCK_RXQ_OVFL,
715         SOCK_ZEROCOPY, /* buffers from userspace */
716         SOCK_WIFI_STATUS, /* push wifi status to userspace */
717         SOCK_NOFCS, /* Tell NIC not to do the Ethernet FCS.
718                      * Will use last 4 bytes of packet sent from
719                      * user-space instead.
720                      */
721         SOCK_FILTER_LOCKED, /* Filter cannot be changed anymore */
722         SOCK_SELECT_ERR_QUEUE, /* Wake select on error queue */
723 };
724
725 static inline void sock_copy_flags(struct sock *nsk, struct sock *osk)
726 {
727         nsk->sk_flags = osk->sk_flags;
```

```
728 }
729
730 static inline void sock_set_flag(struct sock *sk, enum sock_flags flag)
731 {
732         __set_bit(flag, &sk->sk_flags);
733 }
734
735 static inline void sock_reset_flag(struct sock *sk, enum sock_flags flag)
736 {
737         __clear_bit(flag, &sk->sk_flags);
738 }
739
740 static inline bool sock_flag(const struct sock *sk, enum sock_flags flag)
741 {
742         return test_bit(flag, &sk->sk_flags);
743 }
744
745 #ifdef CONFIG_NET
746 extern struct static_key memalloc_socks;
747 static inline int sk_memalloc_socks(void)
748 {
749         return static_key_false(&memalloc_socks);
750 }
751 #else
752
753 static inline int sk_memalloc_socks(void)
754 {
755         return 0;
756 }
757
758 #endif
759
760 static inline gfp_t sk_gfp_atomic(struct sock *sk, gfp_t gfp_mask)
761 {
762         return GFP_ATOMIC | (sk->sk_allocation & __GFP_MEMALLOC);
763 }
764
765 static inline void sk_acceptq_removed(struct sock *sk)
766 {
767         sk->sk_ack_backlog--;
768 }
769
770 static inline void sk_acceptq_added(struct sock *sk)
771 {
772         sk->sk_ack_backlog++;
773 }
774
775 static inline bool sk_acceptq_is_full(const struct sock *sk)
776 {
777         return sk->sk_ack_backlog > sk->sk_max_ack_backlog;
778 }
779
780 /*
781  * Compute minimal free write space needed to queue new packets.
782  */
783 static inline int sk_stream_min_wspace(const struct sock *sk)
784 {
785         return sk->sk_wmem_queued >> 1;
786 }
787
788 static inline int sk_stream_wspace(const struct sock *sk)
789 {
790         return sk->sk_sndbuf - sk->sk_wmem_queued;
791 }
792
793 void sk_stream_write_space(struct sock *sk);
794
795 /* OOB backlog add */
```

```
796 static inline void __sk_add_backlog(struct sock *sk, struct sk_buff *skb)
797 {
798         /* dont let skb dst not refcounted, we are going to leave rcu lock */
799         skb_dst_force(skb);
800
801         if (!sk->sk_backlog.tail)
802                 sk->sk_backlog.head = skb;
803         else
804                 sk->sk_backlog.tail->next = skb;
805
806         sk->sk_backlog.tail = skb;
807         skb->next = NULL;
808 }
809
810 /*
811  * Take into account size of receive queue and backlog queue
812  * Do not take into account this skb truesize,
813  * to allow even a single big packet to come.
814  */
815 static inline bool sk_rcvqueues_full(const struct sock *sk, unsigned int limit)
816 {
817         unsigned int qsize = sk->sk_backlog.len + atomic_read(&sk->sk_rmem_alloc);
818
819         return qsize > limit;
820 }
821
822 /* The per-socket spinlock must be held here. */
823 static inline __must_check int sk_add_backlog(struct sock *sk, struct sk_buff *skb,
824                                               unsigned int limit)
825 {
826         if (sk_rcvqueues_full(sk, limit))
827                 return -ENOBUFS;
828
829         __sk_add_backlog(sk, skb);
830         sk->sk_backlog.len += skb->truesize;
831         return 0;
832 }
833
834 int __sk_backlog_rcv(struct sock *sk, struct sk_buff *skb);
835
836 static inline int sk_backlog_rcv(struct sock *sk, struct sk_buff *skb)
837 {
838         if (sk_memalloc_socks() && skb_pfmemalloc(skb))
839                 return __sk_backlog_rcv(sk, skb);
840
841         return sk->sk_backlog_rcv(sk, skb);
842 }
843
844 static inline void sk_incoming_cpu_update(struct sock *sk)
845 {
846         sk->sk_incoming_cpu = raw_smp_processor_id();
847 }
848
849 static inline void sock_rps_record_flow_hash(__u32 hash)
850 {
851 #ifdef CONFIG_RPS
852         struct rps_sock_flow_table *sock_flow_table;
853
854         rcu_read_lock();
855         sock_flow_table = rcu_dereference(rps_sock_flow_table);
856         rps_record_sock_flow(sock_flow_table, hash);
857         rcu_read_unlock();
858 #endif
859 }
860
861 static inline void sock_rps_record_flow(const struct sock *sk)
862 {
863 #ifdef CONFIG_RPS
```

```
864         sock_rps_record_flow_hash(sk->sk_rxhash);
865 #endif
866 }
867
868 static inline void sock_rps_save_rxhash(struct sock *sk,
869                                         const struct sk_buff *skb)
870 {
871 #ifdef CONFIG_RPS
872         if (unlikely(sk->sk_rxhash != skb->hash))
873                 sk->sk_rxhash = skb->hash;
874 #endif
875 }
876
877 static inline void sock_rps_reset_rxhash(struct sock *sk)
878 {
879 #ifdef CONFIG_RPS
880         sk->sk_rxhash = 0;
881 #endif
882 }
883
884 #define sk_wait_event(__sk, __timeo, __condition)                       \
885         ({      int __rc;                                               \
886                 release_sock(__sk);                                     \
887                 __rc = __condition;                                     \
888                 if (!__rc) {                                            \
889                         *(__timeo) = schedule_timeout(*(__timeo));      \
890                 }                                                       \
891                 sched_annotate_sleep();                                         \
892                 lock_sock(__sk);                                        \
893                 __rc = __condition;                                     \
894                 __rc;                                                   \
895         })
896
897 int sk_stream_wait_connect(struct sock *sk, long *timeo_p);
898 int sk_stream_wait_memory(struct sock *sk, long *timeo_p);
899 void sk_stream_wait_close(struct sock *sk, long timeo_p);
900 int sk_stream_error(struct sock *sk, int flags, int err);
901 void sk_stream_kill_queues(struct sock *sk);
902 void sk_set_memalloc(struct sock *sk);
903 void sk_clear_memalloc(struct sock *sk);
904
905 int sk_wait_data(struct sock *sk, long *timeo, const struct sk_buff *skb);
906
907 struct request_sock_ops;
908 struct timewait_sock_ops;
909 struct inet_hashinfo;
910 struct raw_hashinfo;
911 struct module;
912
913 /*
914  * caches using SLAB_DESTROY_BY_RCU should let .next pointer from nulls nodes
915  * un-modified. Special care is taken when initializing object to zero.
916  */
917 static inline void sk_prot_clear_nulls(struct sock *sk, int size)
918 {
919         if (offsetof(struct sock, sk_node.next) != 0)
920                 memset(sk, 0, offsetof(struct sock, sk_node.next));
921         memset(&sk->sk_node.pprev, 0,
922                size - offsetof(struct sock, sk_node.pprev));
923 }
924
925 /* Networking protocol blocks we attach to sockets.
926  * socket layer -> transport layer interface
927  */
928 struct proto {
929         void                    (*close)(struct sock *sk,
930                                         long timeout);
931         int                     (*connect)(struct sock *sk,
```

```
932                                     struct sockaddr *uaddr,
933                                     int addr_len);
934             int                     (*disconnect)(struct sock *sk, int flags);
935
936             struct sock *           (*accept)(struct sock *sk, int flags, int *err);
937
938             int                     (*ioctl)(struct sock *sk, int cmd,
939                                              unsigned long arg);
940             int                     (*init)(struct sock *sk);
941             void                    (*destroy)(struct sock *sk);
942             void                    (*shutdown)(struct sock *sk, int how);
943             int                     (*setsockopt)(struct sock *sk, int level,
944                                     int optname, char __user *optval,
945                                     unsigned int optlen);
946             int                     (*getsockopt)(struct sock *sk, int level,
947                                     int optname, char __user *optval,
948                                     int __user *option);
949     #ifdef CONFIG_COMPAT
950             int                     (*compat_setsockopt)(struct sock *sk,
951                                     int level,
952                                     int optname, char __user *optval,
953                                     unsigned int optlen);
954             int                     (*compat_getsockopt)(struct sock *sk,
955                                     int level,
956                                     int optname, char __user *optval,
957                                     int __user *option);
958             int                     (*compat_ioctl)(struct sock *sk,
959                                     unsigned int cmd, unsigned long arg);
960     #endif
961             int                     (*sendmsg)(struct sock *sk, struct msghdr *msg,
962                                                size_t len);
963             int                     (*recvmsg)(struct sock *sk, struct msghdr *msg,
964                                                size_t len, int noblock, int flags,
965                                                int *addr_len);
966             int                     (*sendpage)(struct sock *sk, struct page *page,
967                                     int offset, size_t size, int flags);
968             int                     (*bind)(struct sock *sk,
969                                     struct sockaddr *uaddr, int addr_len);
970
971             int                     (*backlog_rcv) (struct sock *sk,
972                                                     struct sk_buff *skb);
973
974             void            (*release_cb)(struct sock *sk);
975
976             /* Keeping track of sk's, looking them up, and port selection methods. */
977             void                    (*hash)(struct sock *sk);
978             void                    (*unhash)(struct sock *sk);
979             void                    (*rehash)(struct sock *sk);
980             int                     (*get_port)(struct sock *sk, unsigned short snum);
981             void                    (*clear_sk)(struct sock *sk, int size);
982
983             /* Keeping track of sockets in use */
984     #ifdef CONFIG_PROC_FS
985             unsigned int            inuse_idx;
986     #endif
987
988             bool                    (*stream_memory_free)(const struct sock *sk);
989             /* Memory pressure */
990             void                    (*enter_memory_pressure)(struct sock *sk);
991             atomic_long_t           *memory_allocated;      /* Current allocated memory. */
992             struct percpu_counter   *sockets_allocated;     /* Current number of sockets. */
993             /*
994              * Pressure flag: try to collapse.
995              * Technical note: it is used by multiple contexts non atomically.
996              * All the __sk_mem_schedule() is of this nature: accounting
997              * is strict, actions are advisory and have some latency.
998              */
999             int                     *memory_pressure;
```

```
1000         long                    *sysctl_mem;
1001         int                     *sysctl_wmem;
1002         int                     *sysctl_rmem;
1003         int                     max_header;
1004         bool                    no_autobind;
1005
1006         struct kmem_cache       *slab;
1007         unsigned int            obj_size;
1008         int                     slab_flags;
1009
1010         struct percpu_counter   *orphan_count;
1011
1012         struct request_sock_ops *rsk_prot;
1013         struct timewait_sock_ops *twsk_prot;
1014
1015         union {
1016                 struct inet_hashinfo    *hashinfo;
1017                 struct udp_table        *udp_table;
1018                 struct raw_hashinfo     *raw_hash;
1019         } h;
1020
1021         struct module           *owner;
1022
1023         char                    name[32];
1024
1025         struct list_head        node;
1026 #ifdef SOCK_REFCNT_DEBUG
1027         atomic_t                socks;
1028 #endif
1029 #ifdef CONFIG_MEMCG_KMEM
1030         /*
1031          * cgroup specific init/deinit functions. Called once for all
1032          * protocols that implement it, from cgroups populate function.
1033          * This function has to setup any files the protocol want to
1034          * appear in the kmem cgroup filesystem.
1035          */
1036         int                     (*init_cgroup)(struct mem_cgroup *memcg,
1037                                                struct cgroup_subsys *ss);
1038         void                    (*destroy_cgroup)(struct mem_cgroup *memcg);
1039         struct cg_proto         *(*proto_cgroup)(struct mem_cgroup *memcg);
1040 #endif
1041 };
1042
1043 /*
1044  * Bits in struct cg_proto.flags
1045  */
1046 enum cg_proto_flags {
1047         /* Currently active and new sockets should be assigned to cgroups */
1048         MEMCG_SOCK_ACTIVE,
1049         /* It was ever activated; we must disarm static keys on destruction */
1050         MEMCG_SOCK_ACTIVATED,
1051 };
1052
1053 struct cg_proto {
1054         struct page_counter     memory_allocated;       /* Current allocated memory. */
1055         struct percpu_counter   sockets_allocated;      /* Current number of sockets. */
1056         int                     memory_pressure;
1057         long                    sysctl_mem[3];
1058         unsigned long           flags;
1059         /*
1060          * memcg field is used to find which memcg we belong directly
1061          * Each memcg struct can hold more than one cg_proto, so container_of
1062          * won't really cut.
1063          *
1064          * The elegant solution would be having an inverse function to
1065          * proto_cgroup in struct proto, but that means polluting the structure
1066          * for everybody, instead of just for memcg users.
1067          */
```

```
1068         struct mem_cgroup       *memcg;
1069 };
1070
1071 int proto_register(struct proto *prot, int alloc_slab);
1072 void proto_unregister(struct proto *prot);
1073
1074 static inline bool memcg_proto_active(struct cg_proto *cg_proto)
1075 {
1076         return test_bit(MEMCG_SOCK_ACTIVE, &cg_proto->flags);
1077 }
1078
1079 #ifdef SOCK_REFCNT_DEBUG
1080 static inline void sk_refcnt_debug_inc(struct sock *sk)
1081 {
1082         atomic_inc(&sk->sk_prot->socks);
1083 }
1084
1085 static inline void sk_refcnt_debug_dec(struct sock *sk)
1086 {
1087         atomic_dec(&sk->sk_prot->socks);
1088         printk(KERN_DEBUG "%s socket %p released, %d are still alive\n",
1089                sk->sk_prot->name, sk, atomic_read(&sk->sk_prot->socks));
1090 }
1091
1092 static inline void sk_refcnt_debug_release(const struct sock *sk)
1093 {
1094         if (atomic_read(&sk->sk_refcnt) != 1)
1095                 printk(KERN_DEBUG "Destruction of the %s socket %p delayed, refcnt=%d\n",
1096                        sk->sk_prot->name, sk, atomic_read(&sk->sk_refcnt));
1097 }
1098 #else /* SOCK_REFCNT_DEBUG */
1099 #define sk_refcnt_debug_inc(sk) do { } while (0)
1100 #define sk_refcnt_debug_dec(sk) do { } while (0)
1101 #define sk_refcnt_debug_release(sk) do { } while (0)
1102 #endif /* SOCK_REFCNT_DEBUG */
1103
1104 #if defined(CONFIG_MEMCG_KMEM) && defined(CONFIG_NET)
1105 extern struct static_key memcg_socket_limit_enabled;
1106 static inline struct cg_proto *parent_cg_proto(struct proto *proto,
1107                                                struct cg_proto *cg_proto)
1108 {
1109         return proto->proto_cgroup(parent_mem_cgroup(cg_proto->memcg));
1110 }
1111 #define mem_cgroup_sockets_enabled static_key_false(&memcg_socket_limit_enabled)
1112 #else
1113 #define mem_cgroup_sockets_enabled 0
1114 static inline struct cg_proto *parent_cg_proto(struct proto *proto,
1115                                                struct cg_proto *cg_proto)
1116 {
1117         return NULL;
1118 }
1119 #endif
1120
1121 static inline bool sk_stream_memory_free(const struct sock *sk)
1122 {
1123         if (sk->sk_wmem_queued >= sk->sk_sndbuf)
1124                 return false;
1125
1126         return sk->sk_prot->stream_memory_free ?
1127                 sk->sk_prot->stream_memory_free(sk) : true;
1128 }
1129
1130 static inline bool sk_stream_is_writeable(const struct sock *sk)
1131 {
1132         return sk_stream_wspace(sk) >= sk_stream_min_wspace(sk) &&
1133                sk_stream_memory_free(sk);
1134 }
1135
```

```
1136
1137 static inline bool sk_has_memory_pressure(const struct sock *sk)
1138 {
1139         return sk->sk_prot->memory_pressure != NULL;
1140 }
1141
1142 static inline bool sk_under_memory_pressure(const struct sock *sk)
1143 {
1144         if (!sk->sk_prot->memory_pressure)
1145                 return false;
1146
1147         if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
1148                 return !!sk->sk_cgrp->memory_pressure;
1149
1150         return !!*sk->sk_prot->memory_pressure;
1151 }
1152
1153 static inline void sk_leave_memory_pressure(struct sock *sk)
1154 {
1155         int *memory_pressure = sk->sk_prot->memory_pressure;
1156
1157         if (!memory_pressure)
1158                 return;
1159
1160         if (*memory_pressure)
1161                 *memory_pressure = 0;
1162
1163         if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
1164                 struct cg_proto *cg_proto = sk->sk_cgrp;
1165                 struct proto *prot = sk->sk_prot;
1166
1167                 for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
1168                         cg_proto->memory_pressure = 0;
1169         }
1170
1171 }
1172
1173 static inline void sk_enter_memory_pressure(struct sock *sk)
1174 {
1175         if (!sk->sk_prot->enter_memory_pressure)
1176                 return;
1177
1178         if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
1179                 struct cg_proto *cg_proto = sk->sk_cgrp;
1180                 struct proto *prot = sk->sk_prot;
1181
1182                 for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
1183                         cg_proto->memory_pressure = 1;
1184         }
1185
1186         sk->sk_prot->enter_memory_pressure(sk);
1187 }
1188
1189 static inline long sk_prot_mem_limits(const struct sock *sk, int index)
1190 {
1191         long *prot = sk->sk_prot->sysctl_mem;
1192         if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
1193                 prot = sk->sk_cgrp->sysctl_mem;
1194         return prot[index];
1195 }
1196
1197 static inline void memcg_memory_allocated_add(struct cg_proto *prot,
1198                                               unsigned long amt,
1199                                               int *parent_status)
1200 {
1201         page_counter_charge(&prot->memory_allocated, amt);
1202
1203         if (page_counter_read(&prot->memory_allocated) >
```

```
1204             prot->memory_allocated.limit)
1205                 *parent_status = OVER_LIMIT;
1206 }
1207
1208 static inline void memcg_memory_allocated_sub(struct cg_proto *prot,
1209                                               unsigned long amt)
1210 {
1211         page_counter_uncharge(&prot->memory_allocated, amt);
1212 }
1213
1214 static inline long
1215 sk_memory_allocated(const struct sock *sk)
1216 {
1217         struct proto *prot = sk->sk_prot;
1218
1219         if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
1220                 return page_counter_read(&sk->sk_cgrp->memory_allocated);
1221
1222         return atomic_long_read(prot->memory_allocated);
1223 }
1224
1225 static inline long
1226 sk_memory_allocated_add(struct sock *sk, int amt, int *parent_status)
1227 {
1228         struct proto *prot = sk->sk_prot;
1229
1230         if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
1231                 memcg_memory_allocated_add(sk->sk_cgrp, amt, parent_status);
1232                 /* update the root cgroup regardless */
1233                 atomic_long_add_return(amt, prot->memory_allocated);
1234                 return page_counter_read(&sk->sk_cgrp->memory_allocated);
1235         }
1236
1237         return atomic_long_add_return(amt, prot->memory_allocated);
1238 }
1239
1240 static inline void
1241 sk_memory_allocated_sub(struct sock *sk, int amt)
1242 {
1243         struct proto *prot = sk->sk_prot;
1244
1245         if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
1246                 memcg_memory_allocated_sub(sk->sk_cgrp, amt);
1247
1248         atomic_long_sub(amt, prot->memory_allocated);
1249 }
1250
1251 static inline void sk_sockets_allocated_dec(struct sock *sk)
1252 {
1253         struct proto *prot = sk->sk_prot;
1254
1255         if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
1256                 struct cg_proto *cg_proto = sk->sk_cgrp;
1257
1258                 for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
1259                         percpu_counter_dec(&cg_proto->sockets_allocated);
1260         }
1261
1262         percpu_counter_dec(prot->sockets_allocated);
1263 }
1264
1265 static inline void sk_sockets_allocated_inc(struct sock *sk)
1266 {
1267         struct proto *prot = sk->sk_prot;
1268
1269         if (mem_cgroup_sockets_enabled && sk->sk_cgrp) {
1270                 struct cg_proto *cg_proto = sk->sk_cgrp;
1271
```

```
1272                 for (; cg_proto; cg_proto = parent_cg_proto(prot, cg_proto))
1273                         percpu_counter_inc(&cg_proto->sockets_allocated);
1274         }
1275
1276         percpu_counter_inc(prot->sockets_allocated);
1277 }
1278
1279 static inline int
1280 sk_sockets_allocated_read_positive(struct sock *sk)
1281 {
1282         struct proto *prot = sk->sk_prot;
1283
1284         if (mem_cgroup_sockets_enabled && sk->sk_cgrp)
1285                 return percpu_counter_read_positive(&sk->sk_cgrp->sockets_allocated);
1286
1287         return percpu_counter_read_positive(prot->sockets_allocated);
1288 }
1289
1290 static inline int
1291 proto_sockets_allocated_sum_positive(struct proto *prot)
1292 {
1293         return percpu_counter_sum_positive(prot->sockets_allocated);
1294 }
1295
1296 static inline long
1297 proto_memory_allocated(struct proto *prot)
1298 {
1299         return atomic_long_read(prot->memory_allocated);
1300 }
1301
1302 static inline bool
1303 proto_memory_pressure(struct proto *prot)
1304 {
1305         if (!prot->memory_pressure)
1306                 return false;
1307         return !!*prot->memory_pressure;
1308 }
1309
1310
1311 #ifdef CONFIG_PROC_FS
1312 /* Called with local bh disabled */
1313 void sock_prot_inuse_add(struct net *net, struct proto *prot, int inc);
1314 int sock_prot_inuse_get(struct net *net, struct proto *proto);
1315 #else
1316 static inline void sock_prot_inuse_add(struct net *net, struct proto *prot,
1317                 int inc)
1318 {
1319 }
1320 #endif
1321
1322
1323 /* With per-bucket locks this operation is not-atomic, so that
1324  * this version is not worse.
1325  */
1326 static inline void __sk_prot_rehash(struct sock *sk)
1327 {
1328         sk->sk_prot->unhash(sk);
1329         sk->sk_prot->hash(sk);
1330 }
1331
1332 void sk_prot_clear_portaddr_nulls(struct sock *sk, int size);
1333
1334 /* About 10 seconds */
1335 #define SOCK_DESTROY_TIME (10*HZ)
1336
1337 /* Sockets 0-1023 can't be bound to unless you are superuser */
1338 #define PROT_SOCK       1024
1339
```

```
1340 #define SHUTDOWN_MASK   3
1341 #define RCV_SHUTDOWN    1
1342 #define SEND_SHUTDOWN   2
1343
1344 #define SOCK_SNDBUF_LOCK        1
1345 #define SOCK_RCVBUF_LOCK        2
1346 #define SOCK_BINDADDR_LOCK      4
1347 #define SOCK_BINDPORT_LOCK      8
1348
1349 struct socket_alloc {
1350         struct socket socket;
1351         struct inode vfs_inode;
1352 };
1353
1354 static inline struct socket *SOCKET_I(struct inode *inode)
1355 {
1356         return &container_of(inode, struct socket_alloc, vfs_inode)->socket;
1357 }
1358
1359 static inline struct inode *SOCK_INODE(struct socket *socket)
1360 {
1361         return &container_of(socket, struct socket_alloc, socket)->vfs_inode;
1362 }
1363
1364 /*
1365  * Functions for memory accounting
1366  */
1367 int __sk_mem_schedule(struct sock *sk, int size, int kind);
1368 void __sk_mem_reclaim(struct sock *sk, int amount);
1369
1370 #define SK_MEM_QUANTUM ((int)PAGE_SIZE)
1371 #define SK_MEM_QUANTUM_SHIFT ilog2(SK_MEM_QUANTUM)
1372 #define SK_MEM_SEND     0
1373 #define SK_MEM_RECV     1
1374
1375 static inline int sk_mem_pages(int amt)
1376 {
1377         return (amt + SK_MEM_QUANTUM - 1) >> SK_MEM_QUANTUM_SHIFT;
1378 }
1379
1380 static inline bool sk_has_account(struct sock *sk)
1381 {
1382         /* return true if protocol supports memory accounting */
1383         return !!sk->sk_prot->memory_allocated;
1384 }
1385
1386 static inline bool sk_wmem_schedule(struct sock *sk, int size)
1387 {
1388         if (!sk_has_account(sk))
1389                 return true;
1390         return size <= sk->sk_forward_alloc ||
1391                 __sk_mem_schedule(sk, size, SK_MEM_SEND);
1392 }
1393
1394 static inline bool
1395 sk_rmem_schedule(struct sock *sk, struct sk_buff *skb, int size)
1396 {
1397         if (!sk_has_account(sk))
1398                 return true;
1399         return size<= sk->sk_forward_alloc ||
1400                 __sk_mem_schedule(sk, size, SK_MEM_RECV) ||
1401                 skb_pfmemalloc(skb);
1402 }
1403
1404 static inline void sk_mem_reclaim(struct sock *sk)
1405 {
1406         if (!sk_has_account(sk))
1407                 return;
```

```
1408         if (sk->sk_forward_alloc >= SK_MEM_QUANTUM)
1409                 __sk_mem_reclaim(sk, sk->sk_forward_alloc);
1410 }
1411
1412 static inline void sk_mem_reclaim_partial(struct sock *sk)
1413 {
1414         if (!sk_has_account(sk))
1415                 return;
1416         if (sk->sk_forward_alloc > SK_MEM_QUANTUM)
1417                 __sk_mem_reclaim(sk, sk->sk_forward_alloc - 1);
1418 }
1419
1420 static inline void sk_mem_charge(struct sock *sk, int size)
1421 {
1422         if (!sk_has_account(sk))
1423                 return;
1424         sk->sk_forward_alloc -= size;
1425 }
1426
1427 static inline void sk_mem_uncharge(struct sock *sk, int size)
1428 {
1429         if (!sk_has_account(sk))
1430                 return;
1431         sk->sk_forward_alloc += size;
1432 }
1433
1434 static inline void sk_wmem_free_skb(struct sock *sk, struct sk_buff *skb)
1435 {
1436         sock_set_flag(sk, SOCK_QUEUE_SHRUNK);
1437         sk->sk_wmem_queued -= skb->truesize;
1438         sk_mem_uncharge(sk, skb->truesize);
1439         __kfree_skb(skb);
1440 }
1441
1442 /* Used by processes to "lock" a socket state, so that
1443  * interrupts and bottom half handlers won't change it
1444  * from under us. It essentially blocks any incoming
1445  * packets, so that we won't get any new data or any
1446  * packets that change the state of the socket.
1447  *
1448  * While locked, BH processing will add new packets to
1449  * the backlog queue.  This queue is processed by the
1450  * owner of the socket lock right before it is released.
1451  *
1452  * Since ~2.3.5 it is also exclusive sleep lock serializing
1453  * accesses from user process context.
1454  */
1455 #define sock_owned_by_user(sk)  ((sk)->sk_lock.owned)
1456
1457 static inline void sock_release_ownership(struct sock *sk)
1458 {
1459         sk->sk_lock.owned = 0;
1460 }
1461
1462 /*
1463  * Macro so as to not evaluate some arguments when
1464  * lockdep is not enabled.
1465  *
1466  * Mark both the sk_lock and the sk_lock.slock as a
1467  * per-address-family lock class.
1468  */
1469 #define sock_lock_init_class_and_name(sk, sname, skey, name, key)       \
1470 do {                                                                    \
1471         sk->sk_lock.owned = 0;                                          \
1472         init_waitqueue_head(&sk->sk_lock.wq);                           \
1473         spin_lock_init(&(sk)->sk_lock.slock);                           \
1474         debug_check_no_locks_freed((void *)&(sk)->sk_lock,              \
1475                         sizeof((sk)->sk_lock));                         \
```

```
1476         lockdep_set_class_and_name(&(sk)->sk_lock.slock,                \
1477                                    (skey), (sname));                    \
1478         lockdep_init_map(&(sk)->sk_lock.dep_map, (name), (key), 0);     \
1479 } while (0)
1480
1481 void lock_sock_nested(struct sock *sk, int subclass);
1482
1483 static inline void lock_sock(struct sock *sk)
1484 {
1485         lock_sock_nested(sk, 0);
1486 }
1487
1488 void release_sock(struct sock *sk);
1489
1490 /* BH context may only use the following locking interface. */
1491 #define bh_lock_sock(__sk)      spin_lock(&((__sk)->sk_lock.slock))
1492 #define bh_lock_sock_nested(__sk) \
1493                                 spin_lock_nested(&((__sk)->sk_lock.slock), \
1494                                 SINGLE_DEPTH_NESTING)
1495 #define bh_unlock_sock(__sk)    spin_unlock(&((__sk)->sk_lock.slock))
1496
1497 bool lock_sock_fast(struct sock *sk);
1498 /**
1499  * unlock_sock_fast - complement of lock_sock_fast
1500  * @sk: socket
1501  * @slow: slow mode
1502  *
1503  * fast unlock socket for user context.
1504  * If slow mode is on, we call regular release_sock()
1505  */
1506 static inline void unlock_sock_fast(struct sock *sk, bool slow)
1507 {
1508         if (slow)
1509                 release_sock(sk);
1510         else
1511                 spin_unlock_bh(&sk->sk_lock.slock);
1512 }
1513
1514
1515 struct sock *sk_alloc(struct net *net, int family, gfp_t priority,
1516                       struct proto *prot, int kern);
1517 void sk_free(struct sock *sk);
1518 void sk_destruct(struct sock *sk);
1519 struct sock *sk_clone_lock(const struct sock *sk, const gfp_t priority);
1520
1521 struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force,
1522                              gfp_t priority);
1523 void sock_wfree(struct sk_buff *skb);
1524 void skb_orphan_partial(struct sk_buff *skb);
1525 void sock_rfree(struct sk_buff *skb);
1526 void sock_efree(struct sk_buff *skb);
1527 #ifdef CONFIG_INET
1528 void sock_edemux(struct sk_buff *skb);
1529 #else
1530 #define sock_edemux(skb) sock_efree(skb)
1531 #endif
1532
1533 int sock_setsockopt(struct socket *sock, int level, int op,
1534                     char __user *optval, unsigned int optlen);
1535
1536 int sock_getsockopt(struct socket *sock, int level, int op,
1537                     char __user *optval, int __user *optlen);
1538 struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size,
1539                                     int noblock, int *errcode);
1540 struct sk_buff *sock_alloc_send_pskb(struct sock *sk, unsigned long header_len,
1541                                      unsigned long data_len, int noblock,
1542                                      int *errcode, int max_page_order);
1543 void *sock_kmalloc(struct sock *sk, int size, gfp_t priority);
```

```
1544 void sock_kfree_s(struct sock *sk, void *mem, int size);
1545 void sock_kzfree_s(struct sock *sk, void *mem, int size);
1546 void sk_send_sigurg(struct sock *sk);
1547
1548 /*
1549  * Functions to fill in entries in struct proto_ops when a protocol
1550  * does not implement a particular function.
1551  */
1552 int sock_no_bind(struct socket *, struct sockaddr *, int);
1553 int sock_no_connect(struct socket *, struct sockaddr *, int, int);
1554 int sock_no_socketpair(struct socket *, struct socket *);
1555 int sock_no_accept(struct socket *, struct socket *, int);
1556 int sock_no_getname(struct socket *, struct sockaddr *, int *, int);
1557 unsigned int sock_no_poll(struct file *, struct socket *,
1558                           struct poll_table_struct *);
1559 int sock_no_ioctl(struct socket *, unsigned int, unsigned long);
1560 int sock_no_listen(struct socket *, int);
1561 int sock_no_shutdown(struct socket *, int);
1562 int sock_no_getsockopt(struct socket *, int , int, char __user *, int __user *);
1563 int sock_no_setsockopt(struct socket *, int, int, char __user *, unsigned int);
1564 int sock_no_sendmsg(struct socket *, struct msghdr *, size_t);
1565 int sock_no_recvmsg(struct socket *, struct msghdr *, size_t, int);
1566 int sock_no_mmap(struct file *file, struct socket *sock,
1567                  struct vm_area_struct *vma);
1568 ssize_t sock_no_sendpage(struct socket *sock, struct page *page, int offset,
1569                          size_t size, int flags);
1570
1571 /*
1572  * Functions to fill in entries in struct proto_ops when a protocol
1573  * uses the inet style.
1574  */
1575 int sock_common_getsockopt(struct socket *sock, int level, int optname,
1576                                  char __user *optval, int __user *optlen);
1577 int sock_common_recvmsg(struct socket *sock, struct msghdr *msg, size_t size,
1578                         int flags);
1579 int sock_common_setsockopt(struct socket *sock, int level, int optname,
1580                                  char __user *optval, unsigned int optlen);
1581 int compat_sock_common_getsockopt(struct socket *sock, int level,
1582                 int optname, char __user *optval, int __user *optlen);
1583 int compat_sock_common_setsockopt(struct socket *sock, int level,
1584                 int optname, char __user *optval, unsigned int optlen);
1585
1586 void sk_common_release(struct sock *sk);
1587
1588 /*
1589  *      Default socket callbacks and setup code
1590  */
1591
1592 /* Initialise core socket variables */
1593 void sock_init_data(struct socket *sock, struct sock *sk);
1594
1595 /*
1596  * Socket reference counting postulates.
1597  *
1598  * * Each user of socket SHOULD hold a reference count.
1599  * * Each access point to socket (an hash table bucket, reference from a list,
1600  *   running timer, skb in flight MUST hold a reference count.
1601  * * When reference count hits 0, it means it will never increase back.
1602  * * When reference count hits 0, it means that no references from
1603  *   outside exist to this socket and current process on current CPU
1604  *   is last user and may/should destroy this socket.
1605  * * sk_free is called from any context: process, BH, IRQ. When
1606  *   it is called, socket has no references from outside -> sk_free
1607  *   may release descendant resources allocated by the socket, but
1608  *   to the time when it is called, socket is NOT referenced by any
1609  *   hash tables, lists etc.
1610  * * Packets, delivered from outside (from network or from another process)
1611  *   and enqueued on receive/error queues SHOULD NOT grab reference count,
```

```
1612  *   when they sit in queue. Otherwise, packets will leak to hole, when
1613  *   socket is looked up by one cpu and unhasing is made by another CPU.
1614  *   It is true for udp/raw, netlink (leak to receive and error queues), tcp
1615  *   (leak to backlog). Packet socket does all the processing inside
1616  *   BR_NETPROTO_LOCK, so that it has not this race condition. UNIX sockets
1617  *   use separate SMP lock, so that they are prone too.
1618  */
1619
1620 /* Ungrab socket and destroy it, if it was the last reference. */
1621 static inline void sock_put(struct sock *sk)
1622 {
1623         if (atomic_dec_and_test(&sk->sk_refcnt))
1624                 sk_free(sk);
1625 }
1626 /* Generic version of sock_put(), dealing with all sockets
1627  * (TCP_TIMEWAIT, TCP_NEW_SYN_RECV, ESTABLISHED...)
1628  */
1629 void sock_gen_put(struct sock *sk);
1630
1631 int sk_receive_skb(struct sock *sk, struct sk_buff *skb, const int nested);
1632
1633 static inline void sk_tx_queue_set(struct sock *sk, int tx_queue)
1634 {
1635         sk->sk_tx_queue_mapping = tx_queue;
1636 }
1637
1638 static inline void sk_tx_queue_clear(struct sock *sk)
1639 {
1640         sk->sk_tx_queue_mapping = -1;
1641 }
1642
1643 static inline int sk_tx_queue_get(const struct sock *sk)
1644 {
1645         return sk ? sk->sk_tx_queue_mapping : -1;
1646 }
1647
1648 static inline void sk_set_socket(struct sock *sk, struct socket *sock)
1649 {
1650         sk_tx_queue_clear(sk);
1651         sk->sk_socket = sock;
1652 }
1653
1654 static inline wait_queue_head_t *sk_sleep(struct sock *sk)
1655 {
1656         BUILD_BUG_ON(offsetof(struct socket_wq, wait) != 0);
1657         return &rcu_dereference_raw(sk->sk_wq)->wait;
1658 }
1659 /* Detach socket from process context.
1660  * Announce socket dead, detach it from wait queue and inode.
1661  * Note that parent inode held reference count on this struct sock,
1662  * we do not release it in this function, because protocol
1663  * probably wants some additional cleanups or even continuing
1664  * to work with this socket (TCP).
1665  */
1666 static inline void sock_orphan(struct sock *sk)
1667 {
1668         write_lock_bh(&sk->sk_callback_lock);
1669         sock_set_flag(sk, SOCK_DEAD);
1670         sk_set_socket(sk, NULL);
1671         sk->sk_wq  = NULL;
1672         write_unlock_bh(&sk->sk_callback_lock);
1673 }
1674
1675 static inline void sock_graft(struct sock *sk, struct socket *parent)
1676 {
1677         write_lock_bh(&sk->sk_callback_lock);
1678         sk->sk_wq = parent->wq;
1679         parent->sk = sk;
```

```
1680         sk_set_socket(sk, parent);
1681         security_sock_graft(sk, parent);
1682         write_unlock_bh(&sk->sk_callback_lock);
1683 }
1684
1685 kuid_t sock_i_uid(struct sock *sk);
1686 unsigned long sock_i_ino(struct sock *sk);
1687
1688 static inline struct dst_entry *
1689 __sk_dst_get(struct sock *sk)
1690 {
1691         return rcu_dereference_check(sk->sk_dst_cache, sock_owned_by_user(sk) ||
1692                                                        lockdep_is_held(&sk->sk_lock.slock));
1693 }
1694
1695 static inline struct dst_entry *
1696 sk_dst_get(struct sock *sk)
1697 {
1698         struct dst_entry *dst;
1699
1700         rcu_read_lock();
1701         dst = rcu_dereference(sk->sk_dst_cache);
1702         if (dst && !atomic_inc_not_zero(&dst->__refcnt))
1703                 dst = NULL;
1704         rcu_read_unlock();
1705         return dst;
1706 }
1707
1708 static inline void dst_negative_advice(struct sock *sk)
1709 {
1710         struct dst_entry *ndst, *dst = __sk_dst_get(sk);
1711
1712         if (dst && dst->ops->negative_advice) {
1713                 ndst = dst->ops->negative_advice(dst);
1714
1715                 if (ndst != dst) {
1716                         rcu_assign_pointer(sk->sk_dst_cache, ndst);
1717                         sk_tx_queue_clear(sk);
1718                 }
1719         }
1720 }
1721
1722 static inline void
1723 __sk_dst_set(struct sock *sk, struct dst_entry *dst)
1724 {
1725         struct dst_entry *old_dst;
1726
1727         sk_tx_queue_clear(sk);
1728         /*
1729          * This can be called while sk is owned by the caller only,
1730          * with no state that can be checked in a rcu_dereference_check() cond
1731          */
1732         old_dst = rcu_dereference_raw(sk->sk_dst_cache);
1733         rcu_assign_pointer(sk->sk_dst_cache, dst);
1734         dst_release(old_dst);
1735 }
1736
1737 static inline void
1738 sk_dst_set(struct sock *sk, struct dst_entry *dst)
1739 {
1740         struct dst_entry *old_dst;
1741
1742         sk_tx_queue_clear(sk);
1743         old_dst = xchg((__force struct dst_entry **)&sk->sk_dst_cache, dst);
1744         dst_release(old_dst);
1745 }
1746
1747 static inline void
```

```
1748 __sk_dst_reset(struct sock *sk)
1749 {
1750         __sk_dst_set(sk, NULL);
1751 }
1752
1753 static inline void
1754 sk_dst_reset(struct sock *sk)
1755 {
1756         sk_dst_set(sk, NULL);
1757 }
1758
1759 struct dst_entry *__sk_dst_check(struct sock *sk, u32 cookie);
1760
1761 struct dst_entry *sk_dst_check(struct sock *sk, u32 cookie);
1762
1763 bool sk_mc_loop(struct sock *sk);
1764
1765 static inline bool sk_can_gso(const struct sock *sk)
1766 {
1767         return net_gso_ok(sk->sk_route_caps, sk->sk_gso_type);
1768 }
1769
1770 void sk_setup_caps(struct sock *sk, struct dst_entry *dst);
1771
1772 static inline void sk_nocaps_add(struct sock *sk, netdev_features_t flags)
1773 {
1774         sk->sk_route_nocaps |= flags;
1775         sk->sk_route_caps &= ~flags;
1776 }
1777
1778 static inline int skb_do_copy_data_nocache(struct sock *sk, struct sk_buff *skb,
1779                                            struct iov_iter *from, char *to,
1780                                            int copy, int offset)
1781 {
1782         if (skb->ip_summed == CHECKSUM_NONE) {
1783                 __wsum csum = 0;
1784                 if (csum_and_copy_from_iter(to, copy, &csum, from) != copy)
1785                         return -EFAULT;
1786                 skb->csum = csum_block_add(skb->csum, csum, offset);
1787         } else if (sk->sk_route_caps & NETIF_F_NOCACHE_COPY) {
1788                 if (copy_from_iter_nocache(to, copy, from) != copy)
1789                         return -EFAULT;
1790         } else if (copy_from_iter(to, copy, from) != copy)
1791                 return -EFAULT;
1792
1793         return 0;
1794 }
1795
1796 static inline int skb_add_data_nocache(struct sock *sk, struct sk_buff *skb,
1797                                        struct iov_iter *from, int copy)
1798 {
1799         int err, offset = skb->len;
1800
1801         err = skb_do_copy_data_nocache(sk, skb, from, skb_put(skb, copy),
1802                                        copy, offset);
1803         if (err)
1804                 __skb_trim(skb, offset);
1805
1806         return err;
1807 }
1808
1809 static inline int skb_copy_to_page_nocache(struct sock *sk, struct iov_iter *from,
1810                                            struct sk_buff *skb,
1811                                            struct page *page,
1812                                            int off, int copy)
1813 {
1814         int err;
1815
```

```
1816         err = skb_do_copy_data_nocache(sk, skb, from, page_address(page) + off,
1817                                        copy, skb->len);
1818         if (err)
1819                 return err;
1820
1821         skb->len             += copy;
1822         skb->data_len        += copy;
1823         skb->truesize        += copy;
1824         sk->sk_wmem_queued   += copy;
1825         sk_mem_charge(sk, copy);
1826         return 0;
1827 }
1828
1829 /**
1830  * sk_wmem_alloc_get - returns write allocations
1831  * @sk: socket
1832  *
1833  * Returns sk_wmem_alloc minus initial offset of one
1834  */
1835 static inline int sk_wmem_alloc_get(const struct sock *sk)
1836 {
1837         return atomic_read(&sk->sk_wmem_alloc) - 1;
1838 }
1839
1840 /**
1841  * sk_rmem_alloc_get - returns read allocations
1842  * @sk: socket
1843  *
1844  * Returns sk_rmem_alloc
1845  */
1846 static inline int sk_rmem_alloc_get(const struct sock *sk)
1847 {
1848         return atomic_read(&sk->sk_rmem_alloc);
1849 }
1850
1851 /**
1852  * sk_has_allocations - check if allocations are outstanding
1853  * @sk: socket
1854  *
1855  * Returns true if socket has write or read allocations
1856  */
1857 static inline bool sk_has_allocations(const struct sock *sk)
1858 {
1859         return sk_wmem_alloc_get(sk) || sk_rmem_alloc_get(sk);
1860 }
1861
1862 /**
1863  * wq_has_sleeper - check if there are any waiting processes
1864  * @wq: struct socket_wq
1865  *
1866  * Returns true if socket_wq has waiting processes
1867  *
1868  * The purpose of the wq_has_sleeper and sock_poll_wait is to wrap the memory
1869  * barrier call. They were added due to the race found within the tcp code.
1870  *
1871  * Consider following tcp code paths:
1872  *
1873  * CPU1                  CPU2
1874  *
1875  * sys_select            receive packet
1876  *   ...                 ...
1877  *   __add_wait_queue    update tp->rcv_nxt
1878  *   ...                 ...
1879  *   tp->rcv_nxt check   sock_def_readable
1880  *   ...                 {
1881  *   schedule               rcu_read_lock();
1882  *                          wq = rcu_dereference(sk->sk_wq);
1883  *                          if (wq && waitqueue_active(&wq->wait))
```

```
1884  *                              wake_up_interruptible(&wq->wait)
1885  *                        ...
1886  *                      }
1887  *
1888  * The race for tcp fires when the __add_wait_queue changes done by CPU1 stay
1889  * in its cache, and so does the tp->rcv_nxt update on CPU2 side.  The CPU1
1890  * could then endup calling schedule and sleep forever if there are no more
1891  * data on the socket.
1892  *
1893  */
1894 static inline bool wq_has_sleeper(struct socket_wq *wq)
1895 {
1896         /* We need to be sure we are in sync with the
1897          * add_wait_queue modifications to the wait queue.
1898          *
1899          * This memory barrier is paired in the sock_poll_wait.
1900          */
1901         smp_mb();
1902         return wq && waitqueue_active(&wq->wait);
1903 }
1904
1905 /**
1906  * sock_poll_wait - place memory barrier behind the poll_wait call.
1907  * @filp:           file
1908  * @wait_address:   socket wait queue
1909  * @p:              poll_table
1910  *
1911  * See the comments in the wq_has_sleeper function.
1912  */
1913 static inline void sock_poll_wait(struct file *filp,
1914                 wait_queue_head_t *wait_address, poll_table *p)
1915 {
1916         if (!poll_does_not_wait(p) && wait_address) {
1917                 poll_wait(filp, wait_address, p);
1918                 /* We need to be sure we are in sync with the
1919                  * socket flags modification.
1920                  *
1921                  * This memory barrier is paired in the wq_has_sleeper.
1922                  */
1923                 smp_mb();
1924         }
1925 }
1926
1927 static inline void skb_set_hash_from_sk(struct sk_buff *skb, struct sock *sk)
1928 {
1929         if (sk->sk_txhash) {
1930                 skb->l4_hash = 1;
1931                 skb->hash = sk->sk_txhash;
1932         }
1933 }
1934
1935 /*
1936  *      Queue a received datagram if it will fit. Stream and sequenced
1937  *      protocols can't normally use this as they need to fit buffers in
1938  *      and play with them.
1939  *
1940  *      Inlined as it's very short and called for pretty much every
1941  *      packet ever received.
1942  */
1943
1944 static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
1945 {
1946         skb_orphan(skb);
1947         skb->sk = sk;
1948         skb->destructor = sock_wfree;
1949         skb_set_hash_from_sk(skb, sk);
1950         /*
1951          * We used to take a refcount on sk, but following operation
```

```
1952          * is enough to guarantee sk_free() wont free this sock until
1953          * all in-flight packets are completed
1954          */
1955         atomic_add(skb->truesize, &sk->sk_wmem_alloc);
1956 }
1957
1958 static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
1959 {
1960         skb_orphan(skb);
1961         skb->sk = sk;
1962         skb->destructor = sock_rfree;
1963         atomic_add(skb->truesize, &sk->sk_rmem_alloc);
1964         sk_mem_charge(sk, skb->truesize);
1965 }
1966
1967 void sk_reset_timer(struct sock *sk, struct timer_list *timer,
1968                     unsigned long expires);
1969
1970 void sk_stop_timer(struct sock *sk, struct timer_list *timer);
1971
1972 int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb);
1973
1974 int sock_queue_err_skb(struct sock *sk, struct sk_buff *skb);
1975 struct sk_buff *sock_dequeue_err_skb(struct sock *sk);
1976
1977 /*
1978  *      Recover an error report and clear atomically
1979  */
1980
1981 static inline int sock_error(struct sock *sk)
1982 {
1983         int err;
1984         if (likely(!sk->sk_err))
1985                 return 0;
1986         err = xchg(&sk->sk_err, 0);
1987         return -err;
1988 }
1989
1990 static inline unsigned long sock_wspace(struct sock *sk)
1991 {
1992         int amt = 0;
1993
1994         if (!(sk->sk_shutdown & SEND_SHUTDOWN)) {
1995                 amt = sk->sk_sndbuf - atomic_read(&sk->sk_wmem_alloc);
1996                 if (amt < 0)
1997                         amt = 0;
1998         }
1999         return amt;
2000 }
2001
2002 static inline void sk_wake_async(struct sock *sk, int how, int band)
2003 {
2004         if (sock_flag(sk, SOCK_FASYNC))
2005                 sock_wake_async(sk->sk_socket, how, band);
2006 }
2007
2008 /* Since sk_{r,w}mem_alloc sums skb->truesize, even a small frame might
2009  * need sizeof(sk_buff) + MTU + padding, unless net driver perform copybreak.
2010  * Note: for send buffers, TCP works better if we can build two skbs at
2011  * minimum.
2012  */
2013 #define TCP_SKB_MIN_TRUESIZE    (2048 + SKB_DATA_ALIGN(sizeof(struct sk_buff)))
2014
2015 #define SOCK_MIN_SNDBUF         (TCP_SKB_MIN_TRUESIZE * 2)
2016 #define SOCK_MIN_RCVBUF          TCP_SKB_MIN_TRUESIZE
2017
2018 static inline void sk_stream_moderate_sndbuf(struct sock *sk)
2019 {
```

```
2020         if (!(sk->sk_userlocks & SOCK_SNDBUF_LOCK)) {
2021                 sk->sk_sndbuf = min(sk->sk_sndbuf, sk->sk_wmem_queued >> 1);
2022                 sk->sk_sndbuf = max_t(u32, sk->sk_sndbuf, SOCK_MIN_SNDBUF);
2023         }
2024 }
2025
2026 struct sk_buff *sk_stream_alloc_skb(struct sock *sk, int size, gfp_t gfp,
2027                                     bool force_schedule);
2028
2029 /**
2030  * sk_page_frag - return an appropriate page_frag
2031  * @sk: socket
2032  *
2033  * If socket allocation mode allows current thread to sleep, it means its
2034  * safe to use the per task page_frag instead of the per socket one.
2035  */
2036 static inline struct page_frag *sk_page_frag(struct sock *sk)
2037 {
2038         if (sk->sk_allocation & __GFP_WAIT)
2039                 return &current->task_frag;
2040
2041         return &sk->sk_frag;
2042 }
2043
2044 bool sk_page_frag_refill(struct sock *sk, struct page_frag *pfrag);
2045
2046 /*
2047  *      Default write policy as shown to user space via poll/select/SIGIO
2048  */
2049 static inline bool sock_writeable(const struct sock *sk)
2050 {
2051         return atomic_read(&sk->sk_wmem_alloc) < (sk->sk_sndbuf >> 1);
2052 }
2053
2054 static inline gfp_t gfp_any(void)
2055 {
2056         return in_softirq() ? GFP_ATOMIC : GFP_KERNEL;
2057 }
2058
2059 static inline long sock_rcvtimeo(const struct sock *sk, bool noblock)
2060 {
2061         return noblock ? 0 : sk->sk_rcvtimeo;
2062 }
2063
2064 static inline long sock_sndtimeo(const struct sock *sk, bool noblock)
2065 {
2066         return noblock ? 0 : sk->sk_sndtimeo;
2067 }
2068
2069 static inline int sock_rcvlowat(const struct sock *sk, int waitall, int len)
2070 {
2071         return (waitall ? len : min_t(int, sk->sk_rcvlowat, len)) ? : 1;
2072 }
2073
2074 /* Alas, with timeout socket operations are not restartable.
2075  * Compare this to poll().
2076  */
2077 static inline int sock_intr_errno(long timeo)
2078 {
2079         return timeo == MAX_SCHEDULE_TIMEOUT ? -ERESTARTSYS : -EINTR;
2080 }
2081
2082 struct sock_skb_cb {
2083         u32 dropcount;
2084 };
2085
2086 /* Store sock_skb_cb at the end of skb->cb[] so protocol families
2087  * using skb->cb[] would keep using it directly and utilize its
```

```
2088  * alignement guarantee.
2089  */
2090 #define SOCK_SKB_CB_OFFSET ((FIELD_SIZEOF(struct sk_buff, cb) - \
2091                              sizeof(struct sock_skb_cb)))
2092
2093 #define SOCK_SKB_CB(__skb) ((struct sock_skb_cb *)((__skb)->cb + \
2094                             SOCK_SKB_CB_OFFSET))
2095
2096 #define sock_skb_cb_check_size(size) \
2097         BUILD_BUG_ON((size) > SOCK_SKB_CB_OFFSET)
2098
2099 static inline void
2100 sock_skb_set_dropcount(const struct sock *sk, struct sk_buff *skb)
2101 {
2102         SOCK_SKB_CB(skb)->dropcount = atomic_read(&sk->sk_drops);
2103 }
2104
2105 void __sock_recv_timestamp(struct msghdr *msg, struct sock *sk,
2106                            struct sk_buff *skb);
2107 void __sock_recv_wifi_status(struct msghdr *msg, struct sock *sk,
2108                              struct sk_buff *skb);
2109
2110 static inline void
2111 sock_recv_timestamp(struct msghdr *msg, struct sock *sk, struct sk_buff *skb)
2112 {
2113         ktime_t kt = skb->tstamp;
2114         struct skb_shared_hwtstamps *hwtstamps = skb_hwtstamps(skb);
2115
2116         /*
2117          * generate control messages if
2118          * - receive time stamping in software requested
2119          * - software time stamp available and wanted
2120          * - hardware time stamps available and wanted
2121          */
2122         if (sock_flag(sk, SOCK_RCVTSTAMP) ||
2123             (sk->sk_tsflags & SOF_TIMESTAMPING_RX_SOFTWARE) ||
2124             (kt.tv64 && sk->sk_tsflags & SOF_TIMESTAMPING_SOFTWARE) ||
2125             (hwtstamps->hwtstamp.tv64 &&
2126              (sk->sk_tsflags & SOF_TIMESTAMPING_RAW_HARDWARE)))
2127                 __sock_recv_timestamp(msg, sk, skb);
2128         else
2129                 sk->sk_stamp = kt;
2130
2131         if (sock_flag(sk, SOCK_WIFI_STATUS) && skb->wifi_acked_valid)
2132                 __sock_recv_wifi_status(msg, sk, skb);
2133 }
2134
2135 void __sock_recv_ts_and_drops(struct msghdr *msg, struct sock *sk,
2136                               struct sk_buff *skb);
2137
2138 static inline void sock_recv_ts_and_drops(struct msghdr *msg, struct sock *sk,
2139                                           struct sk_buff *skb)
2140 {
2141 #define FLAGS_TS_OR_DROPS ((1UL << SOCK_RXQ_OVFL)                       | \
2142                            (1UL << SOCK_RCVTSTAMP))
2143 #define TSFLAGS_ANY       (SOF_TIMESTAMPING_SOFTWARE                    | \
2144                            SOF_TIMESTAMPING_RAW_HARDWARE)
2145
2146         if (sk->sk_flags & FLAGS_TS_OR_DROPS || sk->sk_tsflags & TSFLAGS_ANY)
2147                 __sock_recv_ts_and_drops(msg, sk, skb);
2148         else
2149                 sk->sk_stamp = skb->tstamp;
2150 }
2151
2152 void __sock_tx_timestamp(const struct sock *sk, __u8 *tx_flags);
2153
2154 /**
2155  * sock_tx_timestamp - checks whether the outgoing packet is to be time stamped
```

```
2156  * @sk:         socket sending this packet
2157  * @tx_flags:   completed with instructions for time stamping
2158  *
2159  * Note : callers should take care of initial *tx_flags value (usually 0)
2160  */
2161 static inline void sock_tx_timestamp(const struct sock *sk, __u8 *tx_flags)
2162 {
2163         if (unlikely(sk->sk_tsflags))
2164                 __sock_tx_timestamp(sk, tx_flags);
2165         if (unlikely(sock_flag(sk, SOCK_WIFI_STATUS)))
2166                 *tx_flags |= SKBTX_WIFI_STATUS;
2167 }
2168
2169 /**
2170  * sk_eat_skb - Release a skb if it is no longer needed
2171  * @sk: socket to eat this skb from
2172  * @skb: socket buffer to eat
2173  *
2174  * This routine must be called with interrupts disabled or with the socket
2175  * locked so that the sk_buff queue operation is ok.
2176 */
2177 static inline void sk_eat_skb(struct sock *sk, struct sk_buff *skb)
2178 {
2179         __skb_unlink(skb, &sk->sk_receive_queue);
2180         __kfree_skb(skb);
2181 }
2182
2183 static inline
2184 struct net *sock_net(const struct sock *sk)
2185 {
2186         return read_pnet(&sk->sk_net);
2187 }
2188
2189 static inline
2190 void sock_net_set(struct sock *sk, struct net *net)
2191 {
2192         write_pnet(&sk->sk_net, net);
2193 }
2194
2195 static inline struct sock *skb_steal_sock(struct sk_buff *skb)
2196 {
2197         if (skb->sk) {
2198                 struct sock *sk = skb->sk;
2199
2200                 skb->destructor = NULL;
2201                 skb->sk = NULL;
2202                 return sk;
2203         }
2204         return NULL;
2205 }
2206
2207 /* This helper checks if a socket is a full socket,
2208  * ie _not_ a timewait or request socket.
2209  */
2210 static inline bool sk_fullsock(const struct sock *sk)
2211 {
2212         return (1 << sk->sk_state) & ~(TCPF_TIME_WAIT | TCPF_NEW_SYN_RECV);
2213 }
2214
2215 void sock_enable_timestamp(struct sock *sk, int flag);
2216 int sock_get_timestamp(struct sock *, struct timeval __user *);
2217 int sock_get_timestampns(struct sock *, struct timespec __user *);
2218 int sock_recv_errqueue(struct sock *sk, struct msghdr *msg, int len, int level,
2219                        int type);
2220
2221 bool sk_ns_capable(const struct sock *sk,
2222                    struct user_namespace *user_ns, int cap);
2223 bool sk_capable(const struct sock *sk, int cap);
```

```
2224 bool sk_net_capable(const struct sock *sk, int cap);
2225
2226 extern __u32 sysctl_wmem_max;
2227 extern __u32 sysctl_rmem_max;
2228
2229 extern int sysctl_tstamp_allow_data;
2230 extern int sysctl_optmem_max;
2231
2232 extern __u32 sysctl_wmem_default;
2233 extern __u32 sysctl_rmem_default;
2234
2235 #endif  /* _SOCK_H */
2236
```

This page was automatically generated by LXR 0.3.1 (source). • Linux is a registered trademark of Linus Torvalds • Contact us

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