sock.h

Go to the documentation of this file.[1]

```
00001 /*
00002  * INET         An implementation of the TCP/IP protocol suite for the LINUX
00003  *              operating system.  INET is implemented using the  BSD Socket
00004  *              interface as the means of communication with the user level.
00005  *
00006  *              Definitions for the AF_INET socket handler.
00007  *
00008  * Version:     @(#)sock.h      1.0.4   05/13/93
00009  *
00010  * Authors:     Ross Biro, <bir7@leland.Stanford.Edu>
00011  *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
00012  *              Corey Minyard <wf-rch!minyard@relay.EU.net>
00013  *              Florian La Roche <flla@stud.uni-sb.de>
00014  *
00015  * Fixes:
00016  *              Alan Cox        :       Volatiles in skbuff pointers. See
00017  *                                      skbuff comments. May be overdone,
00018  *                                      better to prove they can be removed
00019  *                                      than the reverse.
00020  *              Alan Cox        :       Added a zapped field for tcp to note
00021  *                                      a socket is reset and must stay shut up
00022  *              Alan Cox        :       New fields for options
00023  *      Pauline Middelink       :       identd support
00024  *              Alan Cox        :       Eliminate low level recv/recvfrom
00025  *              David S. Miller :       New socket lookup architecture.
00026  *              Steve Whitehouse:       Default routines for sock_ops
00027  *              Arnaldo C. Melo :       removed net_pinfo, tp_pinfo and made
00028  *                                      protinfo be just a void pointer, as the
00029  *                                      protocol specific parts were moved to
00030  *                                      respective headers and ipv4/v6, etc now
00031  *                                      use private slabcaches for its socks
00032  *              Pedro Hortas    :       New flags field for socket options
00033  *
00034  *
00035  *              This program is free software; you can redistribute it and/or
00036  *              modify it under the terms of the GNU General Public License
00037  *              as published by the Free Software Foundation; either version
00038  *              2 of the License, or (at your option) any later version.
00039  */
00040 #ifndef _SOCK_H
00041 #define _SOCK_H
00042
00043 #include <linux/config.h>
```

```
00044 #include <linux/list.h>
00045 #include <linux/timer.h>
00046 #include <linux/cache.h>
00047 #include <linux/module.h>
00048 #include <linux/netdevice.h>
00049 #include <linux/skbuff.h[2]>       /* struct sk_buff */
00050 #include <linux/security.h>
00051
00052 #include <linux/filter.h>
00053
00054 #include <asm/atomic.h>
00055 #include <net/dst.h>
00056
00057 /*
00058  * This structure really needs to be cleaned up.
00059  * Most of it is for TCP, and not used by any of
00060  * the other protocols.
00061  */
00062
00063 /* Define this to get the sk->sk_debug debugging facility. */
00064 //#define SOCK_DEBUGGING
00065 #ifdef SOCK_DEBUGGING
00066 #define SOCK_DEBUG(sk, msg...) do { if ((sk) && ((sk)->sk_debug)) \
00067                                         printk(KERN_DEBUG msg); } while (0)
00068 #else
00069 #define SOCK_DEBUG(sk, msg...) do { } while (0)
00070 #endif
00071
00072 /* This is the per-socket lock.  The spinlock provides a synchronization
00073  * between user contexts and software interrupt processing, whereas the
00074  * mini-semaphore synchronizes multiple users amongst themselves.
00075  */
00076 struct sock_iocb[3];
00077[4] typedef struct {
00078         spinlock_t              slock;
00079         struct sock_iocb[5]        *owner;
00080         wait_queue_head_t       wq;
00081 } socket_lock_t[6];
00082
00083 #define sock_lock_init(__sk) \
00084 do {    spin_lock_init(&((__sk)->sk_lock.slock)); \
00085         (__sk)->sk_lock.owner = NULL; \
00086         init_waitqueue_head(&((__sk)->sk_lock.wq)); \
00087 } while(0)
00088
00089 struct sock[7];
00090
00104[8] struct sock_common[9] {
00105         unsigned short          skc_family;
00106         volatile unsigned char  skc_state;
```

```
00107         unsigned char           skc_reuse;
00108         int                     skc_bound_dev_if;
00109         struct hlist_node       skc_node;
00110         struct hlist_node       skc_bind_node;
00111         atomic_t                skc_refcnt;
00112 };
00113
00177[10] struct sock[11] {
00178         /*
00179          * Now struct tcp_tw_bucket also uses sock_common, so please just
00180          * don't add nothing before this first member (__sk_common) --acme
00181          */
00182         struct sock_common[12]      __sk_common;
00183 #define sk_family               __sk_common.skc_family
00184 #define sk_state                __sk_common.skc_state
00185 #define sk_reuse                __sk_common.skc_reuse
00186 #define sk_bound_dev_if         __sk_common.skc_bound_dev_if
00187 #define sk_node                 __sk_common.skc_node
00188 #define sk_bind_node            __sk_common.skc_bind_node
00189 #define sk_refcnt               __sk_common.skc_refcnt
00190         volatile unsigned char  sk_zapped;
00191         unsigned char           sk_shutdown;
00192         unsigned char           sk_use_write_queue;
00193         unsigned char           sk_userlocks;
00194         socket_lock_t[13]           sk_lock;
00195         int                     sk_rcvbuf;
00196         wait_queue_head_t       *sk_sleep;
00197         struct dst_entry        *sk_dst_cache;
00198         rwlock_t                sk_dst_lock;
00199         struct xfrm_policy      *sk_policy[2];
00200         atomic_t                sk_rmem_alloc;
00201         struct sk_buff_head[14]     sk_receive_queue;
00202         atomic_t                sk_wmem_alloc;
00203         struct sk_buff_head[15]     sk_write_queue;
00204         atomic_t                sk_omem_alloc;
00205         int                     sk_wmem_queued;
00206         int                     sk_forward_alloc;
00207         unsigned int            sk_allocation;
00208         int                     sk_sndbuf;
00209         unsigned long           sk_flags;
00210         char                    sk_no_check;
00211         unsigned char           sk_debug;
00212         unsigned char           sk_rcvtstamp;
00213         unsigned char           sk_no_largesend;
00214         int                     sk_route_caps;
00215         unsigned long           sk_lingertime;
00216         int                     sk_hashent;
00217         struct sock[16]             *sk_pair;
00218         /*
00219          * The backlog queue is special, it is always used with
```

```
00220          * the per-socket spinlock held and requires low latency
00221          * access. Therefore we special case it's implementation.
00222          */
00223         struct {
00224                 struct sk_buff[17] *head;
00225                 struct sk_buff[18] *tail;
00226         } sk_backlog;
00227         rwlock_t                sk_callback_lock;
00228         struct sk_buff_head[19]     sk_error_queue;
00229         struct proto[20]            *sk_prot;
00230         int                     sk_err,
00231                                 sk_err_soft;
00232         unsigned short          sk_ack_backlog;
00233         unsigned short          sk_max_ack_backlog;
00234         __u32                   sk_priority;
00235         unsigned short          sk_type;
00236         unsigned char           sk_localroute;
00237         unsigned char           sk_protocol;
00238         struct ucred[21]            sk_peercred;
00239         int                     sk_rcvlowat;
00240         long                    sk_rcvtimeo;
00241         long                    sk_sndtimeo;
00242         struct sk_filter        *sk_filter;
00243         void                    *sk_protinfo;
00244         kmem_cache_t            *sk_slab;
00245         struct timer_list       sk_timer;
00246         struct timeval          sk_stamp;
00247         struct socket[22]           *sk_socket;
00248         void                    *sk_user_data;
00249         struct module           *sk_owner;
00250         void                    *sk_security;
00251         void                    (*sk_state_change)(struct sock[23] *sk);
00252         void                    (*sk_data_ready)(struct sock[24] *sk, int bytes);
00253         void                    (*sk_write_space)(struct sock[25] *sk);
00254         void                    (*sk_error_report)(struct sock[26] *sk);
00255         int                     (*sk_backlog_rcv)(struct sock[27] *sk,
00256                                                   struct sk_buff[28] *skb);
00257         void                    (*sk_create_child)(struct sock[29] *sk, struct sock[30] *newsk);
00258         void                    (*sk_destruct)(struct sock[31] *sk);
00259 };
00260
00261 /*
00262  * Hashed lists helper routines
00263  */
00264 static inline struct sock[32] *__sk_head(struct hlist_head *head)
00265 {
00266         return hlist_entry(head->first, struct sock[33], sk_node);
00267 }
```

```
00268
00269 static inline struct sock[34] *sk_head(struct hlist_head *head)
00270 {
00271         return hlist_empty(head) ? NULL : __sk_head(head);
00272 }
00273
00274 static inline struct sock[35] *sk_next(struct sock[36] *sk)
00275 {
00276         return sk->sk_node.next ?
00277                 hlist_entry(sk->sk_node.next, struct sock[37], sk_node) : NULL;
00278 }
00279
00280 static inline int sk_unhashed(struct sock[38] *sk)
00281 {
00282         return hlist_unhashed(&sk->sk_node);
00283 }
00284
00285 static inline int sk_hashed(struct sock[39] *sk)
00286 {
00287         return sk->sk_node.pprev != NULL;
00288 }
00289
00290 static __inline__ void sk_node_init(struct hlist_node *node)
00291 {
00292         node->pprev = NULL;
00293 }
00294
00295 static __inline__ void __sk_del_node(struct sock[40] *sk)
00296 {
00297         __hlist_del(&sk->sk_node);
00298 }
00299
00300 static __inline__ int __sk_del_node_init(struct sock[41] *sk)
00301 {
00302         if (sk_hashed(sk)) {
00303                 __sk_del_node(sk);
00304                 sk_node_init(&sk->sk_node);
00305                 return 1;
00306         }
00307         return 0;
00308 }
00309
00310 /* Grab socket reference count. This operation is valid only
00311    when sk is ALREADY grabbed f.e. it is found in hash table
00312    or a list and the lookup is made under lock preventing hash table
00313    modifications.
00314  */
00315
00316 static inline void sock_hold(struct sock[42] *sk)
00317 {
```

```
00318         atomic_inc(&sk->sk_refcnt);
00319 }
00320
00321 /* Ungrab socket in the context, which assumes that socket refcnt
00322    cannot hit zero, f.e. it is true in context of any socketcall.
00323  */
00324 static inline void __sock_put(struct sock[43] *sk)
00325 {
00326         atomic_dec(&sk->sk_refcnt);
00327 }
00328
00329 static __inline__ int sk_del_node_init(struct sock[44] *sk)
00330 {
00331         int rc = __sk_del_node_init(sk);
00332
00333         if (rc) {
00334                 /* paranoid for a while -acme */
00335                 WARN_ON(atomic_read(&sk->sk_refcnt) == 1);
00336                 __sock_put(sk);
00337         }
00338         return rc;
00339 }
00340
00341 static __inline__ void __sk_add_node(struct sock[45] *sk, struct hlist_head *list)
00342 {
00343         hlist_add_head(&sk->sk_node, list);
00344 }
00345
00346 static __inline__ void sk_add_node(struct sock[46] *sk, struct hlist_head *list)
00347 {
00348         sock_hold(sk);
00349         __sk_add_node(sk, list);
00350 }
00351
00352 static __inline__ void __sk_del_bind_node(struct sock[47] *sk)
00353 {
00354         __hlist_del(&sk->sk_bind_node);
00355 }
00356
00357 static __inline__ void sk_add_bind_node(struct sock[48] *sk,
00358                                         struct hlist_head *list)
00359 {
00360         hlist_add_head(&sk->sk_bind_node, list);
00361 }
00362
00363 #define sk_for_each(__sk, node, list) \
00364         hlist_for_each_entry(__sk, node, list, sk_node)
00365 #define sk_for_each_from(__sk, node) \
00366         if (__sk && ({ node = &(__sk)->sk_node; 1; })) \
00367                 hlist_for_each_entry_from(__sk, node, sk_node)
```

```
00368 #define sk_for_each_continue(__sk, node) \
00369         if (__sk && ({ node = &(__sk)->sk_node; 1; })) \
00370                 hlist_for_each_entry_continue(__sk, node, sk_node)
00371 #define sk_for_each_safe(__sk, node, tmp, list) \
00372         hlist_for_each_entry_safe(__sk, node, tmp, list, sk_node)
00373 #define sk_for_each_bound(__sk, node, list) \
00374         hlist_for_each_entry(__sk, node, list, sk_bind_node)
00375
00376 /* Sock flags */
00377 enum sock_flags {
00378         SOCK_DEAD,
00379         SOCK_DONE,
00380         SOCK_URGINLINE,
00381         SOCK_KEEPOPEN,
00382         SOCK_LINGER,
00383         SOCK_DESTROY,
00384         SOCK_BROADCAST,
00385         SOCK_TIMESTAMP,
00386 };
00387
00388 static inline void sock_set_flag(struct sock[49] *sk, enum sock_flags flag)
00389 {
00390         __set_bit(flag, &sk->sk_flags);
00391 }
00392
00393 static inline void sock_reset_flag(struct sock[50] *sk, enum sock_flags flag)
00394 {
00395         __clear_bit(flag, &sk->sk_flags);
00396 }
00397
00398 static inline int sock_flag(struct sock[51] *sk, enum sock_flags flag)
00399 {
00400         return test_bit(flag, &sk->sk_flags);
00401 }
00402
00403 /* The per-socket spinlock must be held here. */
00404 #define sk_add_backlog(__sk, __skb)                             \
00405 do {    if (!(__sk)->sk_backlog.tail) {                         \
00406                 (__sk)->sk_backlog.head =                       \
00407                      (__sk)->sk_backlog.tail = (__skb);         \
00408         } else {                                                \
00409                 ((__sk)->sk_backlog.tail)->next = (__skb);      \
00410                 (__sk)->sk_backlog.tail = (__skb);              \
00411         }                                                       \
00412         (__skb)->next = NULL;                                   \
00413 } while(0)
00414
00415 /* IP protocol blocks we attach to sockets.
00416  * socket layer -> transport layer interface
00417  * transport -> network interface is defined by struct inet_proto
00418  */
```

```
00419[52] struct proto[53] {
00420         void                    (*close)(struct sock[54] *sk,
00421                                         long timeout);
00422         int                     (*connect)(struct sock[55] *sk,
00423                                         struct sockaddr[56] *uaddr,
00424                                         int addr_len);
00425         int                     (*disconnect)(struct sock[57] *sk, int flags);
00426
00427         struct sock[58] *           (*accept) (struct sock[59] *sk, int flags, int *err);
00428
00429         int                     (*ioctl)(struct sock[60] *sk, int cmd,
00430                                          unsigned long arg);
00431         int                     (*init)(struct sock[61] *sk);
00432         int                     (*destroy)(struct sock[62] *sk);
00433         void                    (*shutdown)(struct sock[63] *sk, int how);
00434         int                     (*setsockopt)(struct sock[64] *sk, int level,
00435                                         int optname, char *optval, int optlen);
00436         int                     (*getsockopt)(struct sock[65] *sk, int level,
00437                                         int optname, char *optval,
00438                                         int *option);
00439         int                     (*sendmsg)(struct kiocb *iocb, struct sock[66] *sk,
00440                                            struct msghdr[67] *msg, size_t len);
00441         int                     (*recvmsg)(struct kiocb *iocb, struct sock[68] *sk,
00442                                            struct msghdr[69] *msg,
00443                                         size_t len, int noblock, int flags,
00444                                         int *addr_len);
00445         int                     (*sendpage)(struct sock[70] *sk, struct page *page,
00446                                         int offset, size_t size, int flags);
00447         int                     (*bind)(struct sock[71] *sk,
00448                                         struct sockaddr[72] *uaddr, int addr_len);
00449
00450         int                     (*backlog_rcv) (struct sock[73] *sk,
00451                                                 struct sk_buff[74] *skb);
00452
00453         /* Keeping track of sk's, looking them up, and port selection methods. */
00454         void                    (*hash)(struct sock[75] *sk);
00455         void                    (*unhash)(struct sock[76] *sk);
00456         int                     (*get_port)(struct sock[77] *sk, unsigned short snum);
00457
00458         char                    name[32];
00459
00460         struct {
00461                 int inuse;
00462                 u8  __pad[SMP_CACHE_BYTES - sizeof(int)];
00463         } stats[NR_CPUS];
00464 };
```

```
00465
00466 static __inline__ void sk_set_owner(struct sock[78] *sk, struct module *owner)
00467 {
00468         /*
00469          * One should use sk_set_owner just once, after struct sock creation,
00470          * be it shortly after sk_alloc or after a function that returns a new
00471          * struct sock (and that down the call chain called sk_alloc), e.g. the
00472          * IPv4 and IPv6 modules share tcp_create_openreq_child, so if
00473          * tcp_create_openreq_child called sk_set_owner IPv6 would have to
00474          * change the ownership of this struct sock, with one not needed
00475          * transient sk_set_owner call.
00476          */
00477         BUG_ON(sk->sk_owner != NULL);
00478
00479         sk->sk_owner = owner;
00480         __module_get(owner);
00481 }
00482
00483 /* Called with local bh disabled */
00484 static __inline__ void sock_prot_inc_use(struct proto[79] *prot)
00485 {
00486         prot->stats[smp_processor_id()].inuse++;
00487 }
00488
00489 static __inline__ void sock_prot_dec_use(struct proto[80] *prot)
00490 {
00491         prot->stats[smp_processor_id()].inuse--;
00492 }
00493
00494 /* About 10 seconds */
00495 #define SOCK_DESTROY_TIME (10*HZ)
00496
00497 /* Sockets 0-1023 can't be bound to unless you are superuser */
00498 #define PROT_SOCK       1024
00499
00500 #define SHUTDOWN_MASK   3
00501 #define RCV_SHUTDOWN    1
00502 #define SEND_SHUTDOWN   2
00503
00504 #define SOCK_SNDBUF_LOCK        1
00505 #define SOCK_RCVBUF_LOCK        2
00506 #define SOCK_BINDADDR_LOCK      4
00507 #define SOCK_BINDPORT_LOCK      8
00508
00509 /* sock_iocb: used to kick off async processing of socket ios */
00510[81] struct sock_iocb[82] {
00511         struct list_head        list;
00512
00513         int                     flags;
00514         int                     size;
```

```
00515         struct socket[83]           *sock[84];
00516         struct sock[85]             *sk;
00517         struct scm_cookie       *scm;
00518         struct msghdr[86]           *msg, async_msg;
00519         struct iovec            async_iov;
00520 };
00521
00522 static inline struct sock_iocb[87] *kiocb_to_siocb(struct kiocb *iocb)
00523 {
00524         BUG_ON(sizeof(struct sock_iocb[88]) > KIOCB_PRIVATE_SIZE);
00525         return (struct sock_iocb[89] *)iocb->private;
00526 }
00527
00528 static inline struct kiocb *siocb_to_kiocb(struct sock_iocb[90] *si)
00529 {
00530         return container_of((void *)si, struct kiocb, private);
00531 }
00532
00533[91] struct socket_alloc[92] {
00534         struct socket[93] socket[94];
00535         struct inode vfs_inode;
00536 };
00537
00538 static inline struct socket[95] *SOCKET_I(struct inode *inode)
00539 {
00540         return &container_of(inode, struct socket_alloc[96], vfs_inode)->socket;
00541 }
00542
00543 static inline struct inode *SOCK_INODE(struct socket[97] *socket[98])
00544 {
00545         return &container_of(socket[99], struct socket_alloc[100], socket[101])->vfs_inode;
00546 }
00547
00548 /* Used by processes to "lock" a socket state, so that
00549  * interrupts and bottom half handlers won't change it
00550  * from under us. It essentially blocks any incoming
00551  * packets, so that we won't get any new data or any
00552  * packets that change the state of the socket.
00553  *
00554  * While locked, BH processing will add new packets to
00555  * the backlog queue.  This queue is processed by the
00556  * owner of the socket lock right before it is released.
00557  *
00558  * Since ~2.3.5 it is also exclusive sleep lock serializing
00559  * accesses from user process context.
00560  */
00561 extern void __lock_sock(struct sock[102] *sk);
00562 extern void __release_sock(struct sock[103] *sk);
```

```
00563 #define sock_owned_by_user(sk)  ((sk)->sk_lock.owner)
00564
00565 extern void FASTCALL(lock_sock(struct sock[104] *sk));
00566 extern void FASTCALL(release_sock(struct sock[105] *sk));
00567
00568 /* BH context may only use the following locking interface. */
00569 #define bh_lock_sock(__sk)      spin_lock(&((__sk)->sk_lock.slock))
00570 #define bh_unlock_sock(__sk)    spin_unlock(&((__sk)->sk_lock.slock))
00571
00572 extern struct sock[106] *           sk_alloc(int family, int priority, int zero_it,
00573                                          kmem_cache_t *slab);
00574 extern void                     sk_free(struct sock[107] *sk);
00575
00576 extern struct sk_buff[108]          *sock_wmalloc(struct sock[109] *sk,
00577                                                   unsigned long size, int force,
00578                                                   int priority);
00579 extern struct sk_buff[110]          *sock_rmalloc(struct sock[111] *sk,
00580                                                   unsigned long size, int force,
00581                                                   int priority);
00582 extern void                     sock_wfree(struct sk_buff[112] *skb);
00583 extern void                     sock_rfree(struct sk_buff[113] *skb);
00584
00585 extern int                      sock_setsockopt(struct socket[114] *sock[115], int level,
00586                                                 int op, char __user *optval,
00587                                                 int optlen);
00588
00589 extern int                      sock_getsockopt(struct socket[116] *sock[117], int level,
00590                                                 int op, char __user *optval,
00591                                                 int __user *optlen);
00592 extern struct sk_buff[118]          *sock_alloc_send_skb(struct sock[119] *sk,
00593                                                          unsigned long size,
00594                                                          int noblock,
00595                                                          int *errcode);
00596 extern struct sk_buff[120]          *sock_alloc_send_pskb(struct sock[121] *sk,
00597                                                           unsigned long header_len,
00598                                                           unsigned long data_len,
00599                                                           int noblock,
00600                                                           int *errcode);
00601 extern void *sock_kmalloc(struct sock[122] *sk, int size, int priority);
00602 extern void sock_kfree_s(struct sock[123] *sk, void *mem, int size);
00603 extern void sk_send_sigurg(struct sock[124] *sk);
00604
00605 /*
00606  * Functions to fill in entries in struct proto_ops when a protocol
00607  * does not implement a particular function.
00608  */
00609 extern int                      sock_no_release(struct socket[125] *);
00610 extern int                      sock_no_bind(struct socket[126] *,
```

```
00611                                           struct sockaddr[127] *, int);
00612 extern int                       sock_no_connect(struct socket[128] *,
00613                                                   struct sockaddr[129] *, int, int);
00614 extern int                       sock_no_socketpair(struct socket[130] *,
00615                                                     struct socket[131] *);
00616 extern int                       sock_no_accept(struct socket[132] *,
00617                                                 struct socket[133] *, int);
00618 extern int                       sock_no_getname(struct socket[134] *,
00619                                                   struct sockaddr[135] *, int *, int);
00620 extern unsigned int              sock_no_poll(struct file *, struct socket[136] *,
00621                                               struct poll_table_struct *);
00622 extern int                       sock_no_ioctl(struct socket[137] *, unsigned int,
00623                                                unsigned long);
00624 extern int                       sock_no_listen(struct socket[138] *, int);
00625 extern int                       sock_no_shutdown(struct socket[139] *, int);
00626 extern int                       sock_no_getsockopt(struct socket[140] *, int , int,
00627                                                     char __user *, int __user *);
00628 extern int                       sock_no_setsockopt(struct socket[141] *, int, int,
00629                                                     char __user *, int);
00630 extern int                       sock_no_sendmsg(struct kiocb *, struct socket[142] *,
00631                                                   struct msghdr[143] *, size_t);
00632 extern int                       sock_no_recvmsg(struct kiocb *, struct socket[144] *,
00633                                                   struct msghdr[145] *, size_t, int);
00634 extern int                       sock_no_mmap(struct file *file,
00635                                               struct socket[146] *sock[147],
00636                                               struct vm_area_struct *vma);
00637 extern ssize_t                   sock_no_sendpage(struct socket[148] *sock[149],
00638                                                   struct page *page,
00639                                                   int offset, size_t size,
00640                                                   int flags);
00641
00642 /*
00643  *      Default socket callbacks and setup code
00644  */
00645
00646 extern void sock_def_destruct(struct sock[150] *);
00647
00648 /* Initialise core socket variables */
00649 extern void sock_init_data(struct socket[151] *sock[152], struct sock[153] *sk);
00650
00665 static inline int sk_filter(struct sock[154] *sk, struct sk_buff[155] *skb, int needlock)
00666 {
00667         int err;
00668
00669         err = security_sock_rcv_skb(sk, skb);
00670         if (err)
```

```
00671                 return err;
00672
00673         if (sk->sk_filter) {
00674                 struct sk_filter *filter;
00675
00676                 if (needlock)
00677                         bh_lock_sock(sk);
00678
00679                 filter = sk->sk_filter;
00680                 if (filter) {
00681                         int pkt_len = sk_run_filter(skb, filter->insns,
00682                                                     filter->len);
00683                         if (!pkt_len)
00684                                 err = -EPERM;
00685                         else
00686                                 skb_trim(skb, pkt_len);
00687                 }
00688
00689                 if (needlock)
00690                         bh_unlock_sock(sk);
00691         }
00692         return err;
00693 }
00694
00703 static inline void sk_filter_release(struct sock[156] *sk, struct sk_filter *fp)
00704 {
00705         unsigned int size = sk_filter_len(fp);
00706
00707         atomic_sub(size, &sk->sk_omem_alloc);
00708
00709         if (atomic_dec_and_test(&fp->refcnt))
00710                 kfree(fp);
00711 }
00712
00713 static inline void sk_filter_charge(struct sock[157] *sk, struct sk_filter *fp)
00714 {
00715         atomic_inc(&fp->refcnt);
00716         atomic_add(sk_filter_len(fp), &sk->sk_omem_alloc);
00717 }
00718
00719 /*
00720  * Socket reference counting postulates.
00721  *
00722  * * Each user of socket SHOULD hold a reference count.
00723  * * Each access point to socket (an hash table bucket, reference from a list,
00724  *   running timer, skb in flight MUST hold a reference count.
00725  * * When reference count hits 0, it means it will never increase back.
00726  * * When reference count hits 0, it means that no references from
00727  *   outside exist to this socket and current process on current CPU
00728  *   is last user and may/should destroy this socket.
00729  * * sk_free is called from any context: process, BH, IRQ. When
```

```
00730  *   it is called, socket has no references from outside -> sk_free
00731  *   may release descendant resources allocated by the socket, but
00732  *   to the time when it is called, socket is NOT referenced by any
00733  *   hash tables, lists etc.
00734  * * Packets, delivered from outside (from network or from another process)
00735  *   and enqueued on receive/error queues SHOULD NOT grab reference count,
00736  *   when they sit in queue. Otherwise, packets will leak to hole, when
00737  *   socket is looked up by one cpu and unhasing is made by another CPU.
00738  *   It is true for udp/raw, netlink (leak to receive and error queues), tcp
00739  *   (leak to backlog). Packet socket does all the processing inside
00740  *   BR_NETPROTO_LOCK, so that it has not this race condition. UNIX sockets
00741  *   use separate SMP lock, so that they are prone too.
00742  */
00743
00744 /* Ungrab socket and destroy it, if it was the last reference. */
00745 static inline void sock_put(struct sock[158] *sk)
00746 {
00747         if (atomic_dec_and_test(&sk->sk_refcnt))
00748                 sk_free(sk);
00749 }
00750
00751 /* Detach socket from process context.
00752  * Announce socket dead, detach it from wait queue and inode.
00753  * Note that parent inode held reference count on this struct sock,
00754  * we do not release it in this function, because protocol
00755  * probably wants some additional cleanups or even continuing
00756  * to work with this socket (TCP).
00757  */
00758 static inline void sock_orphan(struct sock[159] *sk)
00759 {
00760         write_lock_bh(&sk->sk_callback_lock);
00761         sock_set_flag(sk, SOCK_DEAD);
00762         sk->sk_socket = NULL;
00763         sk->sk_sleep  = NULL;
00764         write_unlock_bh(&sk->sk_callback_lock);
00765 }
00766
00767 static inline void sock_graft(struct sock[160] *sk, struct socket[161] *parent)
00768 {
00769         write_lock_bh(&sk->sk_callback_lock);
00770         sk->sk_sleep = &parent->wait;
00771         parent->sk = sk;
00772         sk->sk_socket = parent;
00773         write_unlock_bh(&sk->sk_callback_lock);
00774 }
00775
00776 static inline int sock_i_uid(struct sock[162] *sk)
00777 {
00778         int uid;
00779
00780         read_lock(&sk->sk_callback_lock);
```

```
00781         uid = sk->sk_socket ? SOCK_INODE(sk->sk_socket)->i_uid : 0;
00782         read_unlock(&sk->sk_callback_lock);
00783         return uid;
00784 }
00785
00786 static inline unsigned long sock_i_ino(struct sock[163] *sk)
00787 {
00788         unsigned long ino;
00789
00790         read_lock(&sk->sk_callback_lock);
00791         ino = sk->sk_socket ? SOCK_INODE(sk->sk_socket)->i_ino : 0;
00792         read_unlock(&sk->sk_callback_lock);
00793         return ino;
00794 }
00795
00796 static inline struct dst_entry *
00797 __sk_dst_get(struct sock[164] *sk)
00798 {
00799         return sk->sk_dst_cache;
00800 }
00801
00802 static inline struct dst_entry *
00803 sk_dst_get(struct sock[165] *sk)
00804 {
00805         struct dst_entry *dst;
00806
00807         read_lock(&sk->sk_dst_lock);
00808         dst = sk->sk_dst_cache;
00809         if (dst)
00810                 dst_hold(dst);
00811         read_unlock(&sk->sk_dst_lock);
00812         return dst;
00813 }
00814
00815 static inline void
00816 __sk_dst_set(struct sock[166] *sk, struct dst_entry *dst)
00817 {
00818         struct dst_entry *old_dst;
00819
00820         old_dst = sk->sk_dst_cache;
00821         sk->sk_dst_cache = dst;
00822         dst_release(old_dst);
00823 }
00824
00825 static inline void
00826 sk_dst_set(struct sock[167] *sk, struct dst_entry *dst)
00827 {
00828         write_lock(&sk->sk_dst_lock);
00829         __sk_dst_set(sk, dst);
00830         write_unlock(&sk->sk_dst_lock);
```

```
00831 }
00832
00833 static inline void
00834 __sk_dst_reset(struct sock[168] *sk)
00835 {
00836         struct dst_entry *old_dst;
00837
00838         old_dst = sk->sk_dst_cache;
00839         sk->sk_dst_cache = NULL;
00840         dst_release(old_dst);
00841 }
00842
00843 static inline void
00844 sk_dst_reset(struct sock[169] *sk)
00845 {
00846         write_lock(&sk->sk_dst_lock);
00847         __sk_dst_reset(sk);
00848         write_unlock(&sk->sk_dst_lock);
00849 }
00850
00851 static inline struct dst_entry *
00852 __sk_dst_check(struct sock[170] *sk, u32 cookie)
00853 {
00854         struct dst_entry *dst = sk->sk_dst_cache;
00855
00856         if (dst && dst->obsolete && dst->ops->check(dst, cookie) == NULL) {
00857                 sk->sk_dst_cache = NULL;
00858                 return NULL;
00859         }
00860
00861         return dst;
00862 }
00863
00864 static inline struct dst_entry *
00865 sk_dst_check(struct sock[171] *sk, u32 cookie)
00866 {
00867         struct dst_entry *dst = sk_dst_get(sk);
00868
00869         if (dst && dst->obsolete && dst->ops->check(dst, cookie) == NULL) {
00870                 sk_dst_reset(sk);
00871                 return NULL;
00872         }
00873
00874         return dst;
00875 }
00876
00877
00878 /*
00879  *      Queue a received datagram if it will fit. Stream and sequenced
00880  *      protocols can't normally use this as they need to fit buffers in
00881  *      and play with them.
```

```
00882  *
00883  *      Inlined as it's very short and called for pretty much every
00884  *      packet ever received.
00885  */
00886
00887 static inline void skb_set_owner_w(struct sk_buff[172] *skb, struct sock[173] *sk)
00888 {
00889         sock_hold(sk);
00890         skb->sk = sk;
00891         skb->destructor = sock_wfree;
00892         atomic_add(skb->truesize, &sk->sk_wmem_alloc);
00893 }
00894
00895 static inline void skb_set_owner_r(struct sk_buff[174] *skb, struct sock[175] *sk)
00896 {
00897         skb->sk = sk;
00898         skb->destructor = sock_rfree;
00899         atomic_add(skb->truesize, &sk->sk_rmem_alloc);
00900 }
00901
00902 static inline int sock_queue_rcv_skb(struct sock[176] *sk, struct sk_buff[177] *skb)
00903 {
00904         int err = 0;
00905         int skb_len;
00906
00907         /* Cast skb->rcvbuf to unsigned... It's pointless, but reduces
00908            number of warnings when compiling with -W --ANK
00909          */
00910         if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
00911             (unsigned)sk->sk_rcvbuf) {
00912                 err = -ENOMEM;
00913                 goto out;
00914         }
00915
00916         /* It would be deadlock, if sock_queue_rcv_skb is used
00917            with socket lock! We assume that users of this
00918            function are lock free.
00919         */
00920         err = sk_filter(sk, skb, 1);
00921         if (err)
00922                 goto out;
00923
00924         skb->dev = NULL;
00925         skb_set_owner_r(skb, sk);
00926
00927         /* Cache the SKB length before we tack it onto the receive
00928          * queue.  Once it is added it no longer belongs to us and
00929          * may be freed by other threads of control pulling packets
00930          * from the queue.
00931          */
00932         skb_len = skb->len;
```

```
00933
00934         skb_queue_tail(&sk->sk_receive_queue, skb);
00935
00936         if (!sock_flag(sk, SOCK_DEAD))
00937                 sk->sk_data_ready(sk, skb_len);
00938 out:
00939         return err;
00940 }
00941
00942 static inline int sock_queue_err_skb(struct sock[178] *sk, struct sk_buff[179] *skb)
00943 {
00944         /* Cast skb->rcvbuf to unsigned... It's pointless, but reduces
00945            number of warnings when compiling with -W --ANK
00946          */
00947         if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
00948             (unsigned)sk->sk_rcvbuf)
00949                 return -ENOMEM;
00950         skb_set_owner_r(skb, sk);
00951         skb_queue_tail(&sk->sk_error_queue, skb);
00952         if (!sock_flag(sk, SOCK_DEAD))
00953                 sk->sk_data_ready(sk, skb->len);
00954         return 0;
00955 }
00956
00957 /*
00958  *      Recover an error report and clear atomically
00959  */
00960
00961 static inline int sock_error(struct sock[180] *sk)
00962 {
00963         int err = xchg(&sk->sk_err, 0);
00964         return -err;
00965 }
00966
00967 static inline unsigned long sock_wspace(struct sock[181] *sk)
00968 {
00969         int amt = 0;
00970
00971         if (!(sk->sk_shutdown & SEND_SHUTDOWN)) {
00972                 amt = sk->sk_sndbuf - atomic_read(&sk->sk_wmem_alloc);
00973                 if (amt < 0)
00974                         amt = 0;
00975         }
00976         return amt;
00977 }
00978
00979 static inline void sk_wake_async(struct sock[182] *sk, int how, int band)
00980 {
00981         if (sk->sk_socket && sk->sk_socket->fasync_list)
00982                 sock_wake_async(sk->sk_socket, how, band);
00983 }
```

```
00984
00985 #define SOCK_MIN_SNDBUF 2048
00986 #define SOCK_MIN_RCVBUF 256
00987
00988 /*
00989  *      Default write policy as shown to user space via poll/select/SIGIO
00990  */
00991 static inline int sock_writeable(struct sock[183] *sk)
00992 {
00993         return atomic_read(&sk->sk_wmem_alloc) < (sk->sk_sndbuf / 2);
00994 }
00995
00996 static inline int gfp_any(void)
00997 {
00998         return in_softirq() ? GFP_ATOMIC : GFP_KERNEL;
00999 }
01000
01001 static inline long sock_rcvtimeo(struct sock[184] *sk, int noblock)
01002 {
01003         return noblock ? 0 : sk->sk_rcvtimeo;
01004 }
01005
01006 static inline long sock_sndtimeo(struct sock[185] *sk, int noblock)
01007 {
01008         return noblock ? 0 : sk->sk_sndtimeo;
01009 }
01010
01011 static inline int sock_rcvlowat(struct sock[186] *sk, int waitall, int len)
01012 {
01013         return (waitall ? len : min_t(int, sk->sk_rcvlowat, len)) ? : 1;
01014 }
01015
01016 /* Alas, with timeout socket operations are not restartable.
01017  * Compare this to poll().
01018  */
01019 static inline int sock_intr_errno(long timeo)
01020 {
01021         return timeo == MAX_SCHEDULE_TIMEOUT ? -ERESTARTSYS : -EINTR;
01022 }
01023
01024 static __inline__ void
01025 sock_recv_timestamp(struct msghdr[187] *msg, struct sock[188] *sk, struct sk_buff[189] *skb)
01026 {
01027         struct timeval *stamp = &skb->stamp;
01028         if (sk->sk_rcvtstamp) {
01029                 /* Race occurred between timestamp enabling and packet
01030                    receiving.  Fill in the current time for now. */
01031                 if (stamp->tv_sec == 0)
01032                         do_gettimeofday(stamp);
01033                 put_cmsg(msg, SOL_SOCKET, SO_TIMESTAMP, sizeof(struct timeval),
```

```
01034                           stamp);
01035         } else
01036                 sk->sk_stamp = *stamp;
01037 }
01038
01039 extern atomic_t netstamp_needed;
01040 extern void sock_enable_timestamp(struct sock[190] *sk);
01041 extern void sock_disable_timestamp(struct sock[191] *sk);
01042
01043 static inline void net_timestamp(struct timeval *stamp)
01044 {
01045         if (atomic_read(&netstamp_needed))
01046                 do_gettimeofday(stamp);
01047         else {
01048                 stamp->tv_sec = 0;
01049                 stamp->tv_usec = 0;
01050         }
01051 }
01052
01053 extern int sock_get_timestamp(struct sock[192] *, struct timeval *);
01054
01055 /*
01056  *      Enable debug/info messages
01057  */
01058
01059 #if 0
01060 #define NETDEBUG(x)     do { } while (0)
01061 #define LIMIT_NETDEBUG(x) do {} while(0)
01062 #else
01063 #define NETDEBUG(x)     do { x; } while (0)
01064 #define LIMIT_NETDEBUG(x) do { if (net_ratelimit()) { x; } } while(0)
01065 #endif
01066
01067 /*
01068  * Macros for sleeping on a socket. Use them like this:
01069  *
01070  * SOCK_SLEEP_PRE(sk)
01071  * if (condition)
01072  *      schedule();
01073  * SOCK_SLEEP_POST(sk)
01074  *
01075  * N.B. These are now obsolete and were, afaik, only ever used in DECnet
01076  * and when the last use of them in DECnet has gone, I'm intending to
01077  * remove them.
01078  */
01079
01080 #define SOCK_SLEEP_PRE(sk)      { struct task_struct *tsk = current; \
01081                                 DECLARE_WAITQUEUE(wait, tsk); \
01082                                 tsk->state = TASK_INTERRUPTIBLE; \
01083                                 add_wait_queue((sk)->sk_sleep, &wait); \
01084                                 release_sock(sk);
```

```
01085
01086 #define SOCK_SLEEP_POST(sk)     tsk->state = TASK_RUNNING; \
01087                                 remove_wait_queue((sk)->sk_sleep, &wait); \
01088                                 lock_sock(sk); \
01089                                 }
01090
01091 static inline void sock_valbool_flag(struct sock[193] *sk, int bit, int valbool)
01092 {
01093         if (valbool)
01094                 sock_set_flag(sk, bit);
01095         else
01096                 sock_reset_flag(sk, bit);
01097 }
01098
01099 extern __u32 sysctl_wmem_max;
01100 extern __u32 sysctl_rmem_max;
01101
01102 int siocdevprivate_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg);
01103
01104 #endif  /* _SOCK_H */
```

Links

1. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/sock_8h.html
2. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/skbuff_8h.html
3. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
4. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket__lock__t.html
5. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
6. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket__lock__t.html
7. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
8. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__common.html
9. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__common.html
10. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
11. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
12. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__common.html
13. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket__lock__t.html
14. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff__head.html
15. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff__head.html
16. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
17. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
18. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
19. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff__head.html
20. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structproto.html
21. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structucred.html

22. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
23. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
24. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
25. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
26. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
27. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
28. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
29. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
30. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
31. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
32. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
33. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
34. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
35. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
36. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
37. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
38. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
39. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
40. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
41. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
42. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
43. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
44. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
45. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
46. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
47. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
48. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
49. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
50. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
51. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
52. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structproto.html
53. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structproto.html
54. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
55. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
56. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsockaddr.html
57. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
58. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
59. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html

60. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
61. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
62. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
63. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
64. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
65. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
66. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
67. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structmsghdr.html
68. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
69. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structmsghdr.html
70. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
71. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
72. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsockaddr.html
73. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
74. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
75. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
76. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
77. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
78. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
79. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structproto.html
80. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structproto.html
81. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
82. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
83. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
84. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
85. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
86. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structmsghdr.html
87. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
88. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
89. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
90. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock__iocb.html
91. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket__alloc.html
92. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket__alloc.html
93. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
94. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
95. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
96. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket__alloc.html
97. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
98. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html

99. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
100. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket__alloc.html
101. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
102. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
103. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
104. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
105. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
106. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
107. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
108. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
109. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
110. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
111. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
112. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
113. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
114. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
115. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
116. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
117. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
118. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
119. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
120. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
121. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
122. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
123. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
124. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
125. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
126. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
127. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsockaddr.html
128. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
129. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsockaddr.html
130. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
131. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
132. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
133. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
134. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
135. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsockaddr.html
136. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
137. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html

138. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
139. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
140. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
141. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
142. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
143. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structmsghdr.html
144. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
145. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structmsghdr.html
146. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
147. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
148. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
149. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
150. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
151. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
152. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
153. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
154. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
155. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
156. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
157. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
158. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
159. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
160. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
161. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsocket.html
162. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
163. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
164. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
165. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
166. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
167. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
168. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
169. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
170. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
171. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
172. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
173. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
174. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
175. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
176. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html

177. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
178. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
179. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
180. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
181. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
182. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
183. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
184. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
185. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
186. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
187. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structmsghdr.html
188. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
189. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsk__buff.html
190. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
191. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
192. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html
193. http://www.cse.scu.edu/~dclark/am_256_graph_theory/linux_2_6_stack/structsock.html