Chapitre 24

Les descripteurs de couche transport

Nous veons de voir comment un paquet entrant, dépecé de son en-tête IP, est transmis en tant que datagramme à une fonction de traitement de couche de transport, `udp_rcv()` dans le cas de UDP. Avant de poursuivre le traitement de celui-ci, étudions une nouvelle structure de données utile pour la couche transport.

On a besoin d'entreposer un certain nombre de renseignements utiles pour les couches de réseau (un peu) et transport (surtout), tels que l'adresse de destination et le numéro de port. Ceci s'effectue grâce à un **descripteur de couche transport**, de type `struct sock`.

Nous allons étudier la structure des descripteurs de couche transport et leur gestion dans ce chapitre qui, comme la gestion des descripteurs de tampon, ressort plus de la gestion de la mémoire que du réseau proprement dit.

24.1 Structure des descripteurs de couche transport

24.1.1 Définition du type

Code Linux 2.6.10

Le type `struct sock` est défini dans le fichier en-tête `linux/include/net/sock.h`:

```
115 /**
116   *     struct sock - representation de la couche reseau des sockets
117   *     @__sk_common - structure partagee avec tcp_tw_bucket
118   *     @sk_zapped - ax25 & ipx signifie !linked
119   *     @sk_shutdown - masque de %SEND_SHUTDOWN et/ou %RCV_SHUTDOWN
120   *     @sk_use_write_queue - si on doit appeler sk->sk_write_space dans sock_wfree
121   *     @sk_userlocks - positionnements de %SO_SNDBUF et de %SO_RCVBUF
122   *     @sk_lock -      synchronisateur
123   *     @sk_rcvbuf - taille du tampon de reception en octets
124   *     @sk_sleep - sock attend une file d'attente
125   *     @sk_dst_cache - cache de destination
126   *     @sk_dst_lock - verrou du cache de destination
127   *     @sk_policy - police de flux
128   *     @sk_rmem_alloc - octets de la file d'attente en reception remis
129   *     @sk_receive_queue - paquets entrants
130   *     @sk_wmem_alloc - octets de la file d'attente en emission remis
131   *     @sk_write_queue - File d'attente d'envoi des paquets
132   *     @sk_omem_alloc - "o" est "option" ou "other"
133   *     @sk_wmem_queued - taille de la file d'attente persistante
134   *     @sk_forward_alloc - espace alloue en avant
135   *     @sk_allocation - mode d'allocation
136   *     @sk_sndbuf - taille du tampon d'envoi en octets
137   *     @sk_flags - positionnements de %SO_LINGER (l_onoff), %SO_BROADCAST, %SO_KEEPALIVE,
                                     %SO_OOBINLINE
138   *     @sk_no_check - positionnement de %SO_NO_CHECK s'il y a ou non des paquets a verifier
139   *     @sk_debug - positionnement de %SO_DEBUG
140   *     @sk_rcvtstamp - positionnement de %SO_TIMESTAMP
141   *     @sk_no_largesend - si on doit envoyer de grands segments ou non
142   *     @sk_route_caps - possibilites de la route (par exemple %NETIF_F_TSO)
143   *     @sk_lingertime - positionnement de %SO_LINGER l_linger
144   *     @sk_hashent - entree de hachage dans plusieurs tables (par exemple tcp_ehash)
145   *     @sk_backlog - toujours utilise avec le verrou tournant per-socket detenu
146   *     @sk_callback_lock - utilise avec les retours d'appel a la fin de cette struct
147   *     @sk_error_queue - rarement utilise
148   *     @sk_prot - gestionnaires de protocole dans une famille de reseau
149   *     @sk_err - derniere erreur
150   *     @sk_err_soft - erreurs non vouees a l'echec mais qui sont la cause
           d'un echec persistant, pas juste 'delai ecoule'
151   *     @sk_ack_backlog - nombre de clients en attente possibles en cours
152   *     @sk_max_ack_backlog - positionne le nombre de clients en attente possible dans
      listen()
153   *     @sk_priority - positionnement de %SO_PRIORITY
154   *     @sk_type - type de socket (%SOCK_STREAM, etc)
155   *     @sk_localroute - route locale seulement, positionnement de %SO_DONTROUTE
156   *     @sk_protocol - protocole auquel cette socket appartient dans cette famille reseau
157   *     @sk_peercred - positionnement de %SO_PEERCRED
158   *     @sk_rcvlowat - positionnement de %SO_RCVLOWAT
159   *     @sk_rcvtimeo - positionnement de %SO_RCVTIMEO
160   *     @sk_sndtimeo - positionnement de %SO_SNDTIMEO
161   *     @sk_filter - instructions de filtage de la socket
162   *     @sk_protinfo - zone privee, propre a la famille reseau, lorsqu'on n'utilise pas
                         le slab
163   *     @sk_slab - le cache de slab a partir duquel cette instance a ete allouee
164   *     @sk_timer - minuteur de nettoyage de sock
165   *     @sk_stamp - estampille temporelle du dernier paquet recu
166   *     @sk_socket - Ident et report des signaux d'E/S
167   *     @sk_user_data - donnees privees de la couche RPC
```

```
168   *     @sk_owner - module qui possede cette socket
169   *     @sk_sndmsg_page - page en cache pour sendmsg
170   *     @sk_sndmsg_off - decalage en cache pour sendmsg
171   *     @sk_send_head - debut des choses a emettre
172   *     @sk_write_pending - une ecriture pour une socket de flux attend de demarrer
173   *     @sk_queue_shrunk - une file d'attente en ecriture a ete reduite recemment
174   *     @sk_state_change - appel pour indiquer le changment dans l'etat de la sock
175   *     @sk_data_ready - appel pour indiquer qu'il y a des donnees a traiter
176   *     @sk_write_space - appel pour indiquer qu'il y a de l'espace d'envoi disponible
177   *     @sk_error_report - appel pour indiquer les erreurs (par exemple %MSG_ERRQUEUE)
178   *     @sk_backlog_rcv - appel pour traiter le nombre de clients en attente
179   *     @sk_destruct - appele au moment de la liberation de sock, c'est-a-dire lorsque tous
          les refcnt == 0
180   */
181  struct sock {
182          /*
183           * struct tcp_tw_bucket utilise maintenant egalement sock_common, donc n'ajouter
184           * rien s'il vous plait avant ce premier membre (__sk_common) --acme
185           */
186          struct sock_common      __sk_common;
187  #define sk_family               __sk_common.skc_family
188  #define sk_state                __sk_common.skc_state
189  #define sk_reuse                __sk_common.skc_reuse
190  #define sk_bound_dev_if         __sk_common.skc_bound_dev_if
191  #define sk_node                 __sk_common.skc_node
192  #define sk_bind_node            __sk_common.skc_bind_node
193  #define sk_refcnt               __sk_common.skc_refcnt
194          volatile unsigned char  sk_zapped;
195          unsigned char           sk_shutdown;
196          unsigned char           sk_use_write_queue;
197          unsigned char           sk_userlocks;
198          socket_lock_t           sk_lock;
199          int                     sk_rcvbuf;
200          wait_queue_head_t       *sk_sleep;
201          struct dst_entry        *sk_dst_cache;
202          rwlock_t                sk_dst_lock;
203          struct xfrm_policy      *sk_policy[2];
204          atomic_t                sk_rmem_alloc;
205          struct sk_buff_head     sk_receive_queue;
206          atomic_t                sk_wmem_alloc;
207          struct sk_buff_head     sk_write_queue;
208          atomic_t                sk_omem_alloc;
209          int                     sk_wmem_queued;
210          int                     sk_forward_alloc;
211          unsigned int            sk_allocation;
212          int                     sk_sndbuf;
213          unsigned long           sk_flags;
214          char                    sk_no_check;
215          unsigned char           sk_debug;
216          unsigned char           sk_rcvtstamp;
217          unsigned char           sk_no_largesend;
218          int                     sk_route_caps;
219          unsigned long           sk_lingertime;
220          int                     sk_hashent;
221          /*
222           * La file d'attente des clients en attente est particuliere, elle est toujours
223           * utilisee avec le verrou tournant per-socket detenu et exige un acces a latence
224           * basse. D'ou le cas particulier de son implementation.
225           */
226          struct {
227                  struct sk_buff *head;
228                  struct sk_buff *tail;
```

```
229         } sk_backlog;
230         rwlock_t                sk_callback_lock;
231         struct sk_buff_head     sk_error_queue;
232         struct proto            *sk_prot;
233         int                     sk_err,
234                                 sk_err_soft;
235         unsigned short          sk_ack_backlog;
236         unsigned short          sk_max_ack_backlog;
237         __u32                   sk_priority;
238         unsigned short          sk_type;
239         unsigned char           sk_localroute;
240         unsigned char           sk_protocol;
241         struct ucred            sk_peercred;
242         int                     sk_rcvlowat;
243         long                    sk_rcvtimeo;
244         long                    sk_sndtimeo;
245         struct sk_filter        *sk_filter;
246         void                    *sk_protinfo;
247         kmem_cache_t            *sk_slab;
248         struct timer_list       sk_timer;
249         struct timeval          sk_stamp;
250         struct socket           *sk_socket;
251         void                    *sk_user_data;
252         struct module           *sk_owner;
253         struct page             *sk_sndmsg_page;
254         __u32                   sk_sndmsg_off;
255         struct sk_buff          *sk_send_head;
256         int                     sk_write_pending;
257         void                    *sk_security;
258         __u8                    sk_queue_shrunk;
259         /* three bytes hole, try to pack */
260         void                    (*sk_state_change)(struct sock *sk);
261         void                    (*sk_data_ready)(struct sock *sk, int bytes);
262         void                    (*sk_write_space)(struct sock *sk);
263         void                    (*sk_error_report)(struct sock *sk);
264         int                     (*sk_backlog_rcv)(struct sock *sk,
265                                                   struct sk_buff *skb);
266         void                    (*sk_destruct)(struct sock *sk);
267 };
```

24.1.2 Attributs communs

Le premier champ, `__sk_common`, regroupe les attributs qui sont communs avec une autre structure, `struct tcp_tw_bucket`, concernant le protocole TCP. Le type `struct sock_common` de la partie commune est défini une peu plus haut dans le même fichier en-tête :

Code Linux 2.6.10

```
92  /**
93    *     struct sock_common - representation minimale de la couche reseau des sockets
94    *     @skc_family - famille d'adresses reseau
95    *     @skc_state - Etat de connexion
96    *     @skc_reuse - positionnement de %SO_REUSEADDR
97    *     @skc_bound_dev_if - index du peripherique lie si != 0
98    *     @skc_node - principal lien de hachage pour les tables de consultation de divers
                         protocoles
99    *     @skc_bind_node - lien de hachage bind pour les tables de consultation de divers
                           protocoles
100    *     @skc_refcnt - compteur des references
101   *
102   *     Ceci est la representation minimale pour la couche reseau des sockets, l'en-tete
103   *     de struct sock et de struct tcp_tw_bucket.
104   */
105 struct sock_common {
```

```
106         unsigned short          skc_family;
107         volatile unsigned char  skc_state;
108         unsigned char           skc_reuse;
109         int                     skc_bound_dev_if;
110         struct hlist_node       skc_node;
111         struct hlist_node       skc_bind_node;
112         atomic_t                skc_refcnt;
113 };
```

Les champs sont :

- Le numéro de la famille de protocoles (`AF_INET` pour ce qui nous intéresse).
- L'état de la connexion, les valeurs dépendant du type de communication choisi.

  Les états d'une connexion sont représentés par des constantes symboliques (de préfixe `TCP_` même s'il s'agit de UDP) définies dans le fichier `linux/include/linux/tcp.h`: Code Linux 2.6.10

```
59 enum {
60   TCP_ESTABLISHED = 1,
61   TCP_SYN_SENT,
62   TCP_SYN_RECV,
63   TCP_FIN_WAIT1,
64   TCP_FIN_WAIT2,
65   TCP_TIME_WAIT,
66   TCP_CLOSE,
67   TCP_CLOSE_WAIT,
68   TCP_LAST_ACK,
69   TCP_LISTEN,
70   TCP_CLOSING,   /* maintenant un etat valide */
71
72   TCP_MAX_STATES /* Laisser a la fin ! */
73 };
```

  Une connexion connaît plusieurs états durant sa durée de vie. Les états définis par [RFC 793] pour TCP sont les suivants :

  - `LISTEN` : la connexion reste en ettente d'une requête de connexion externe par une socket TCP distante. Cet état est atteint après une demande de connexion passive.
  - `SYN-SENT` : la connexion se met en attente d'une requête de connexion, après avoir elle-même envoyé une requête à un destinataire.
  - `SYN-RECEIVED` : les deux requêtes de connexion se sont croisées ; la connexion attend confirmation de son établissement.
  - `ESTABLISHED` : la connexion a été confirmée de part et d'autre et les données peuvent transiter sur la voie de communication. C'est l'état stable actif de la connexion.
  - `FIN-WAIT-1` : sur requête de déconnexion émise par l'application, la connexion demande la confirmation d'une requête de déconnexion qu'elle a elle-même émise vers la socket distante.
  - `FIN-WAIT-2` : la connexion se met en attente d'une requête de déconnexion par la socket distante, une fois reçue la confirmation de sa propre requête.
  - `CLOSE-WAIT` : la connexion se met en attente d'une requête de déconnexion émise par l'application.
  - `CLOSING` : la connexion attend la confirmation de sa requête de déconnexion par la socket TCP distante, laquelle avait auparavant émis sa propre requête de déconnexion.
  - `LAST-ACK` : la connexion attend la confirmation de sa requête de déconnexion, émise suite à une requête similaire à l'initiative de la socket distante.

  - `TIME-WAIT`: la connexion connaît un temps de latence avant fermeture complète du canal, pour s'assurer que toutes les confirmations ont bien été reçues.
  - `CLOSED`: la connexion n'existe plus. Il s'agit d'un pseudo-état dans la mesure où la connexion n'existe plus.

- Un paramètre permettant de spécifier si on réutilise le descripteur (utilisé dans la fonction `setsockopt()`).
- L'index de l'interface physique du périphérique.
- Une liste de hachage.
- Une seconde liste de hachage, utile pour certains protocoles.
- Un compteur de références, qui dénombre les entités qui utilisent le descripteur. Le descripteur ne pourra être libéré que si ce compteur est égal à zéro.

24.1.3 Attributs de contrôle

Les attributs de contrôle sont les suivants:

- Le champ `sk_zapped` est utile pour les suites de protocoles AX25 et IPX mais pas pour TCP/IP.
- Le champ `sk_shutdown` indique si on en train d'arrêter la connexion.
- Le champ `sk_use_write_queue` indique si on doit appeler `sk->sk_write_space`, utile pour la fonction `sock_vfree()`.
- Le champ `sk_userlocks` permet de spécifier ce qu'il faut synchroniser.
- Le champ `sk_lock` permet de synchroniser.

Code Linux 2.6.10

Le type `socket_lock_t` est défini dans le fichier `linux/include/net/sock.h`:

```
73 /* On a un verrou par socket. Ce verrou tournant fournit une synchronisation
74  * entre les contextes utilisateur et les processus d'interruptions logicielles, alors
75  * que le mini-semaphore synchronise les utilisateurs multiple entre eux.
76  */
77 struct sock_iocb;
78 typedef struct {
79         spinlock_t              slock;
80         struct sock_iocb        *owner;
81         wait_queue_head_t       wq;
82 } socket_lock_t;
```

- Le champ `sk_rcvbuf` indique la taille en octets du tampon de réception.
- Le champ `sk_sleep` spécifie l'adresse de la file d'attente des entités en attente d'opération sur le descripteur de couche transport.

Code Linux 2.6.10

Le type général `wait_queue_head_t` est défini dans le fichier `linux/include/linux/wait.h`:

```
51 struct __wait_queue_head {
52         spinlock_t lock;
53         struct list_head task_list;
54 };
55 typedef struct __wait_queue_head wait_queue_head_t;
```

- Le champ `sk_dst_cache` spécifie une entrée dans le cache de destination.
- Le verrou `sk_dst_lock` indique si on peut utiliser ce cache.

– Le champ `sk_policy[]` indique la politique pour chacun des deux flux d'entrée et de sortie.

Le type `struct xfrm_policy`, qui ne nous intéressera pas ici, est défini dans le fichier `linux/include/net/xfrm.h`:

Code Linux 2.6.10

```
262 struct xfrm_policy
263 {
264         struct xfrm_policy      *next;
265         struct list_head        list;
266
267         /* Ce verrou affecte seulement les elements extirpes pour l'entree. */
268         rwlock_t                lock;
269         atomic_t                refcnt;
270         struct timer_list       timer;
271
272         u32                     priority;
273         u32                     index;
274         struct xfrm_selector    selector;
275         struct xfrm_lifetime_cfg lft;
276         struct xfrm_lifetime_cur curlft;
277         struct dst_entry       *bundles;
278         __u16                   family;
279         __u8                    action;
280         __u8                    flags;
281         __u8                    dead;
282         __u8                    xfrm_nr;
283         struct xfrm_tmpl        xfrm_vec[XFRM_MAX_DEPTH];
284 };
```

– Le champ `sk_rmem_alloc` permet d'indiquer que des données arrivent.
– Le champ `sk_receive_queue` spécifie la file d'attente des paquets en réception.
– Le champ `sk_wmem_alloc` permet d'indiquer que des données doivent être transmises.
– Le champ `sk_write_queue` spécifie la file d'attente des paquets qui attendent d'être envoyés.
– Le champ `sk_omem_alloc` permet d'indiquer que l'on attend quelque chose d'autre.
– Le champ `sk_wmem_queued` spécifie la taille de la file d'attente des paquets à envoyer.
– Le champ `sk_forward_alloc` spécifie l'espace mémoire alloué pour la retransmission.
– Le champ `sk_allocation` spécifie le mode d'allocation.
– Le champ `sk_sndbuf` spécifie la taille du tampon d'envoi, en octets.
– Le champ `sk_no_check` spécifie si on doit vérifier les paquets ou non.
– Le champ `sk_debug` spécifie si on doit déboguer.
– Le champ `sk_rcvtstamp` spécifie si on doit utiliser une estampille temporelle ou non.
– Le champ `sk_no_largesend` spécifie si on peut envoyer de grands segments ou non.
– Le champ `sk_route_caps` spécifie les capacités de routage.
– Le champ `sk_hashent` spécifie l'entrée dans la table de hachage.
– Le champ `sk_backlog` est un champ spécial.
– Le champ `sk_callback_lock` permet de synchroniser.
– Le champ `sk_error_queue` spécifie la file d'attente des erreurs. Il est rarement utilisé.
– Le champ `sk_prot` spécifie l'ensemble des opérations faisant intervenir ce type de descripteur pour la famille de protocoles et le type de communication de la socket.
– Le champ `sk_err` spécifie la dernière erreur intervenue.
– Le champ `sk_err_soft` spécifie le dernier avertissement.
– Le champ `sk_ack_backlog` spécifie le numéro de client en cours.
– Le champ `sk_max_ack_backlog` spécifie le nombre maximal de clients qui peuvent se connecter simultanémment.

- Le champ `sk_type` spécifie le type de communication (UDP pour ce qui nous intéresse).
- Le champ `sk_protocol` spécifie le protocole pour la famille d'adresses et le type de communication (toujours égal à 0 pour `AF_INET`).
- Le champ `sk_filter` spécifie les instructions de filtrage.

Code Linux 2.6.10

Le type `struct sk_filter` est défini dans le fichier `linux/include/linux/filter.h`:

```
21 /*
22  *      On essaie de garder ces valeurs et structures semblables a BSD, en particulier
23  *      pour les definitions du code BPF qui ont besoin de concorder, ainsi pouvez-vous
        partager les filtres
24  */
25
26 struct sock_filter      /* Bloc de filtre */
27 {
28         __u16   code;   /* Code de filtre actuel */
29         __u8    jt;     /* saut si vrai */
30         __u8    jf;     /* saut si faux */
31         __u32   k;      /* Champ multiusage generique */
32 };
[...]
40 #ifdef __KERNEL__
41 struct sk_filter
42 {
43         atomic_t                refcnt;
44         unsigned int            len;    /* Nombre de blocs de filtre */
45         struct sock_filter      insns[0];
46 };
```

- Le champ `sk_protinfo` est spécifique à la famille de protocole, utile lorsqu'on n'utilise pas le champ suivant.
- Le champ `sk_slab` spécifie où se trouve le cache.
- Le champ `sk_timer` spécifie le minuteur de nettoyage de ce descripteur.
- Le champ `sk_stamp` spécifie l'heure d'arrivée du dernier paquet.
- Le champ `sk_socket` spécifie le descripteur de socket associé à ce descripteur de couche de transport.
- Le champ `sk_user_data` spécifie l'emplacement des données privées de la couche RPC.
- Le champ `sk_owner` spécifie le module qui possède cette socket.

24.1.4 Attributs associés aux options des sockets

Nous avons vu que l'utilisateur pouvait choisir des options pour les sockets. L'état de celles-ci est reporté dans le descripteur de couche transport. Il s'agit des champs suivants :

- Le champ `sk_flags` est le vecteur des drapeaux, les valeurs étant celles utilisées par la fonction utilisateur `setsockopt()`.
- Le champ `sk_lingertime` spécifie si on a positionné `SO_LINGER` ou non.
- Le champ `sk_priority` spécifie si `SO_PRIORITY` est positionné ou non.
- Le champ `sk_localroute` spécifie si `SO_DONTROUTE` est positionné ou non.
- Le champ `sk_peercred` spécifie si `SO_PEERCRED` est positionné ou non.
- Le champ `sk_rcvlowat` spécifie si `SO_RCVLOWAT` est positionné ou non.
- Le champ `sk_rcvtimeo` spécifie si `SO_RCVTIMEO` est positionné ou non.
- Le champ `sk_sndtimeo` spécifie si `SO_SNDTIMEO` est positionné ou non.

24.1.5 Fonctions membre

Rappelons que Linux est conçu comme un programme orienté objet dont les structures comprennent des attributs et des méthodes. Les méthodes sont les suivantes:

- la fonction `sk_state_change()` permet d'indiquer un changement d'état;
- la fonction `sk_data_ready()` permet de savoir s'il y a des données à traiter;
- la fonction `sk_write_space()` permet de savoir s'il y a de la place dans la file d'attente d'envoi;
- la fonction `sk_error_report()` permet d'indiquer les erreurs;
- la fonction `sk_backlog_rcv()` permet de traiter les clients en attente;
- la fonction `sk_destruct()` permet de détruire le descripteur.

24.2 Allocation et libération

Les descripteurs de couche transport sont instantiés grâce à la fonction `sk_alloc()`. Chaque descripteur maintient un compteur de référence: les fonctions `sock_hold()` et `sock_put()` permettent d'incrémenter et de décrémenter celui-ci. Lorsque que le compteur de référence atteint zéro, le descripteur peut être libéré grâce à la fonction `sk_free()`.

24.2.1 Allocation

La fonction `sk_alloc()` est définie dans le fichier `linux/net/core/sock.c`: Code Linux 2.6.10

```
606 static kmem_cache_t *sk_cachep;
607
608 /**
609  *      sk_alloc - Tous les objets socket sont alloues ici
610  *      @family - famille de protocoles
611  *      @priority - pour l'allocation (%GFP_KERNEL, %GFP_ATOMIC, etc)
612  *      @zero_it - met a zero le descripteur alloue
613  *      @slab - slab alternatif
614  *
615  *      Tous les objets socket sont alloues ici. Si @zero_it est non nul,
616  *      elle doit contenir la taille de ce qui doit etre mis a zero, puisque les
617  *      caches slab prives ont des tailles differentes de la struct sock generique.
618  *      1 a ete garde comme une facon de dire sizeof(struct sock).
619  */
620 struct sock *sk_alloc(int family, int priority, int zero_it, kmem_cache_t *slab)
621 {
622         struct sock *sk = NULL;
623
624         if (!slab)
625                 slab = sk_cachep;
626         sk = kmem_cache_alloc(slab, priority);
627         if (sk) {
628                 if (zero_it) {
629                         memset(sk, 0,
630                                zero_it == 1 ? sizeof(struct sock) : zero_it);
631                         sk->sk_family = family;
632                         sock_lock_init(sk);
633                 }
634                 sk->sk_slab = slab;
635
636                 if (security_sk_alloc(sk, family, priority)) {
637                         kmem_cache_free(slab, sk);
638                         sk = NULL;
```

```
639                 }
640         }
641         return sk;
642 }
```

Autrement dit :

- On déclare une adresse de descripteur de couche transport, que l'on initialise à `NULL`.
- Si l'emplacement de l'antémémoire n'est pas spécifié, on choisit celle par défaut, correspondant à l'emplacement de la variable `sk_cachep`, déclarée juste avant la définition de la fonction.

  Cette variable est initialisée dans la fonction `sk_init()`, définie dans le fichier `linux/net/core/sock.c` :

Code Linux 2.6.10

```
669 void __init sk_init(void)
670 {
671         sk_cachep = kmem_cache_create("sock", sizeof(struct sock), 0,
672                                       SLAB_HWCACHE_ALIGN, NULL, NULL);
673         if (!sk_cachep)
674                 printk(KERN_CRIT "sk_init: Cannot create sock SLAB cache!");
675
676         if (num_physpages <= 4096) {
677                 sysctl_wmem_max = 32767;
678                 sysctl_rmem_max = 32767;
679                 sysctl_wmem_default = 32767;
680                 sysctl_rmem_default = 32767;
681         } else if (num_physpages >= 131072) {
682                 sysctl_wmem_max = 131071;
683                 sysctl_rmem_max = 131071;
684         }
685 }
```

  Les variables `sysctl_wmem_max`, `sysctl_rmem_max`, `sysctl_rmem_default` et `sysctl_wmem_default` sont définies plus haut dans le même fichier :

Code Linux 2.6.10

```
130 /* Prendre en consideration la taille de l'en-tete de struct sk_buff dans la
131  * determination de ces valeurs, puisque ceci n'est pas constant suivant la
132  * plate-forme. Ceci fait que le comportement de la mise en file d'attente des sockets
133  * et la performance de dependent pas de telles differences.
134  */
135 #define _SK_MEM_PACKETS         256
136 #define _SK_MEM_OVERHEAD        (sizeof(struct sk_buff) + 256)
137 #define SK_WMEM_MAX             (_SK_MEM_OVERHEAD * _SK_MEM_PACKETS)
138 #define SK_RMEM_MAX             (_SK_MEM_OVERHEAD * _SK_MEM_PACKETS)
139
140 /* Parametres ajustable en execution. */
141 __u32 sysctl_wmem_max     = SK_WMEM_MAX;
142 __u32 sysctl_rmem_max     = SK_RMEM_MAX;
143 __u32 sysctl_wmem_default = SK_WMEM_MAX;
144 __u32 sysctl_rmem_default = SK_RMEM_MAX;
```

  dont l'initialisation ici ne sert à rien puisque nous venons de voir que les valeurs des variables sont surchargées dans la fonction `sk_init()`.
- On essaie de réserver de la place en mémoire vive pour le descripteur de couche transport. Si on n'y parvient pas, on renvoie la valeur `NULL`.
- Si on a spécifié en argument de mettre à zéro :
  - On initialise cet emplacement mémoire à zéro.
  - On renseigne le champ famille de protocoles de ce descripteur, grâce à la valeur passée en paramètre (UDP pour ce qui nous intéresse).

– On verrouille le descripteur pour qu'il ne puisse plus être initialisé à nouveau.

La macro `sock_lock_init()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
84 #define sock_lock_init(__sk) \
85 do {    spin_lock_init(&((__sk)->sk_lock.slock)); \
86         (__sk)->sk_lock.owner = NULL; \
87         init_waitqueue_head(&((__sk)->sk_lock.wq)); \
88 } while(0)
```

– On renseigne le champ du descripteur spécifiant l'emplacement de l'antémémoire.
– On renvoie l'adresse du descripteur ainsi créé.

24.2.2 Gestion du compteur de référence

24.2.2.1 Incrémentation

La fonction `sock_hold()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
318 /* Met la main sur le compteur de reference d'une socket. Cette operation n'est valide que
319    lorsque sk a DEJA ete agrippe, par exemple elle est trouvee dans une table de hachage
320    ou dans une liste et la consultation est faite sous verrouillage empechant les
321    modifications de la table de hachage.
322  */
323
324 static inline void sock_hold(struct sock *sk)
325 {
326         atomic_inc(&sk->sk_refcnt);
327 }
[...]
871 /*
872  * Postulats de denombrement des references a une socket.
873  *
874  * * Chaque utilisateur d'une socket DOIT detenir un compteur de reference.
875  * * Chaque point d'acces a une socket (une chaine d'une table de hachage, une reference a
       une liste,
876  *   un minuteur en fonctionnement, un skb en vol) DOIT detenir un compteur de reference.
877  * * Lorsque le compteur de reference atteint 0, ceci signifie qu'il ne s'accroitra plus.
878  * * Lorsque le compteur de reference atteint 0, ceci signifie qu'aucune references a
879  *   la socket provenant de l'exterieur n'existe et que le processus en cours sur le
880  *   microprocesseur en cours en est le dernier utilisateur et qu'il peut/doit detruire cette
       socket.
881  * * sk_free est appelee depuis les contextes suivants : processus, BH, IRQ. Lorsqu'elle
882  *   est appelee, la socket n'a aucune reference provenant de l'exterieur -> sk_free
883  *   peut liberer les ressources descendantes allouees par la socket, mais
884  *   au moment ou elle est appelee, la socket N'est referencee par aucune
885  *   table de hachage, liste etc.
886  * * Les paquets provenant de l'exterieur (par le reseau ou provenant d'un autre processus)
887  *   et mis dans la file d'attente en reception/erreur NE DEVRAIENT PAS mettre la main
       sur le compteur de reference
888  *   lorsqu'ils s'inserent dans la file. Autrement les paquets s'echapperont par le trou
889  *   lorsque la socket sera consultee par une cpu et que la decrementation est faite par une
       autre CPU.
890  *   Ceci est vrai pour udp/raw, netlink (fuite pour les files d'attente en reception et en
       erreur), tcp
891  *   (fuite en nombre de clients). Une socket de paquet effectue toutes ses taches dans
892  *   BR_NETPROTO_LOCK, aussi n'a-t-elle pas cette condition de course. Les sockets UNIX
893  *   utilisent un verrou SMP independant, aussi en sont-elles egalement sujettes.
894  */
```

Autrement dit on incrémente de 1 le compteur de référence du descripteur de couche réseau en prenant les précautions pour que l'opération soit atomique.

24.2.2.2 Décrémentation

Code Linux 2.6.10

La fonction `sock_put()` est définie dans le fichier `linux/include/net/sock.h`:

```
896 /* Se detacher d'une socket et la detruire, si c'etait la derniere reference. */
897 static inline void sock_put(struct sock *sk)
898 {
899         if (atomic_dec_and_test(&sk->sk_refcnt))
900                 sk_free(sk);
901 }
```

Autrement dit on décrémente de 1 le compteur de référence du descripteur de couche réseau.

Nous allons étudier la fonction `sk_free()` dans la sous-section suivante.

24.2.3 Libération

24.2.3.1 Fonction de libération

Code Linux 2.6.10

La fonction `sk_free()` est définie dans le fichier `linux/net/core/sock.c`:

```
644 void sk_free(struct sock *sk)
645 {
646         struct sk_filter *filter;
647         struct module *owner = sk->sk_owner;
648
649         if (sk->sk_destruct)
650                 sk->sk_destruct(sk);
651
652         filter = sk->sk_filter;
653         if (filter) {
654                 sk_filter_release(sk, filter);
655                 sk->sk_filter = NULL;
656         }
657
658         sock_disable_timestamp(sk);
659
660         if (atomic_read(&sk->sk_omem_alloc))
661                 printk(KERN_DEBUG "%s: optmem leakage (%d bytes) detected.\n",
662                        __FUNCTION__, atomic_read(&sk->sk_omem_alloc));
663
664         security_sk_free(sk);
665         kmem_cache_free(sk->sk_slab, sk);
666         module_put(owner);
667 }
```

Autrement dit :

- On déclare un filtre de socket.
- On déclare un propriétaire du module, que l'on initialise avec le propriétaire du descripteur de couche transport passé en argument.
- Si une fonction spécifique de destruction a été déclarée dans le descripteur de couche transport, on fait appel à elle.
- On instantie le filtre avec celui du descripteur de couche transport passé en argument. S'il est non nul, on le libère grâce à la fonction auxiliaire `sk_filter_release()` et on met à zéro le champ filtre du descripteur.

  La fonction en ligne `sk_filter_release()` est définie dans le fichier `linux/include/net/sock.h`:

  Code Linux 2.6.10

  ```
  847 /**
  ```

```
848  *      sk_filter_release : supprime un filtre de socket
849  *      @sk : socket
850  *      @fp : filtre a supprimer
851  *
852  *      Supprime un filtre d'une socket et libere ses ressources.
853  */
854
855 static inline void sk_filter_release(struct sock *sk, struct sk_filter *fp)
856 {
857         unsigned int size = sk_filter_len(fp);
858
859         atomic_sub(size, &sk->sk_omem_alloc);
860
861         if (atomic_dec_and_test(&fp->refcnt))
862                 kfree(fp);
863 }
```

La fonction en ligne `sk_filter_len()` est définie dans le fichier `linux/include/linux/filter.h`: Code Linux 2.6.10

```
48 static inline unsigned int sk_filter_len(struct sk_filter *fp)
49 {
50         return fp->len*sizeof(struct sock_filter) + sizeof(*fp);
51 }
```

– On inhibe le minuteur du descripteur de couche transport.

La fonction `sock_disable_timestamp()` est définie dans le fichier `linux/net/core/sock.c`: Code Linux 2.6.10

```
178 static void sock_disable_timestamp(struct sock *sk)
179 {
180         if (sock_flag(sk, SOCK_TIMESTAMP)) {
181                 sock_reset_flag(sk, SOCK_TIMESTAMP);
182                 net_disable_timestamp();
183         }
184 }
```

Les fonctions en ligne `sock_flag()` et `sock_reset_flag()` sont définies dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
401 static inline void sock_reset_flag(struct sock *sk, enum sock_flags flag)
402 {
403         __clear_bit(flag, &sk->sk_flags);
404 }
405
406 static inline int sock_flag(struct sock *sk, enum sock_flags flag)
407 {
408         return test_bit(flag, &sk->sk_flags);
409 }
```

La fonction `net_disable_timestamp()` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1004 /* Lorsque > 0 il y a des consommateurs d'estampillles temporelle skb rx */
1005 static atomic_t netstamp_needed = ATOMIC_INIT(0);
[...]
1012 void net_disable_timestamp(void)
1013 {
1014          atomic_dec(&netstamp_needed);
1015 }
```

– Si le champ `sk_omem_alloc` du descripteur de couche transport est non nul, on ne devrait pas libérer le descripteur. On affiche donc un message noyau d'erreur.

– On fait éventuellement appel à la fonction de sécurité de libération du descripteur de couche transport. La fonction `security_sk_free()` ne nous intéressera pas dans cet ouvrage.
– On libère la mémoire vive qui était occupée par le descripteur.
– On décrémente le nombre d'utilisateurs du module propriétaire, pour qu'il puisse être libéré si besoin est.

24.2.3.2 Macro de libération

Code Linux 2.6.10

La fonction `release_sock()` est définie dans le fichier `linux/net/core/sock.c`:

```
1217 void fastcall release_sock(struct sock *sk)
1218 {
1219         spin_lock_bh(&(sk->sk_lock.slock));
1220         if (sk->sk_backlog.tail)
1221                 __release_sock(sk);
1222         sk->sk_lock.owner = NULL;
1223         if (waitqueue_active(&(sk->sk_lock.wq)))
1224                 wake_up(&(sk->sk_lock.wq));
1225         spin_unlock_bh(&(sk->sk_lock.slock));
1226 }
```

Code Linux 2.6.10

qui renvoie à la fonction `__release_sock()` définie dans le même fichier :

```
932 static void __release_sock(struct sock *sk)
933 {
934         struct sk_buff *skb = sk->sk_backlog.head;
935
936         do {
937                 sk->sk_backlog.head = sk->sk_backlog.tail = NULL;
938                 bh_unlock_sock(sk);
939
940                 do {
941                         struct sk_buff *next = skb->next;
942
943                         skb->next = NULL;
944                         sk->sk_backlog_rcv(sk, skb);
945                         skb = next;
946                 } while (skb != NULL);
947
948                 bh_lock_sock(sk);
949         } while((skb = sk->sk_backlog.head) != NULL);
950 }
```

24.3 Gestion des données associées

24.3.1 Initialisation des données

Code Linux 2.6.10

La fonction `sock_init_data()` est définie dans le fichier `linux/net/core/sock.c`:

```
1153 void sock_init_data(struct socket *sock, struct sock *sk)
1154 {
1155         skb_queue_head_init(&sk->sk_receive_queue);
1156         skb_queue_head_init(&sk->sk_write_queue);
1157         skb_queue_head_init(&sk->sk_error_queue);
1158
1159         sk->sk_send_head        =       NULL;
1160
1161         init_timer(&sk->sk_timer);
1162
1163         sk->sk_allocation       =       GFP_KERNEL;
```

```
1164        sk->sk_rcvbuf           =       sysctl_rmem_default;
1165        sk->sk_sndbuf           =       sysctl_wmem_default;
1166        sk->sk_state            =       TCP_CLOSE;
1167        sk->sk_zapped           =       1;
1168        sk->sk_socket           =       sock;
1169
1170        if(sock)
1171        {
1172                sk->sk_type     =       sock->type;
1173                sk->sk_sleep    =       &sock->wait;
1174                sock->sk        =       sk;
1175        } else
1176                sk->sk_sleep    =       NULL;
1177
1178        rwlock_init(&sk->sk_dst_lock);
1179        rwlock_init(&sk->sk_callback_lock);
1180
1181        sk->sk_state_change     =       sock_def_wakeup;
1182        sk->sk_data_ready       =       sock_def_readable;
1183        sk->sk_write_space      =       sock_def_write_space;
1184        sk->sk_error_report     =       sock_def_error_report;
1185        sk->sk_destruct         =       sock_def_destruct;
1186
1187        sk->sk_sndmsg_page      =       NULL;
1188        sk->sk_sndmsg_off       =       0;
1189
1190        sk->sk_peercred.pid     =       0;
1191        sk->sk_peercred.uid     =       -1;
1192        sk->sk_peercred.gid     =       -1;
1193        sk->sk_write_pending    =       0;
1194        sk->sk_rcvlowat         =       1;
1195        sk->sk_rcvtimeo         =       MAX_SCHEDULE_TIMEOUT;
1196        sk->sk_sndtimeo         =       MAX_SCHEDULE_TIMEOUT;
1197        sk->sk_owner            =       NULL;
1198
1199        sk->sk_stamp.tv_sec     = -1L;
1200        sk->sk_stamp.tv_usec    = -1L;
1201
1202        atomic_set(&sk->sk_refcnt, 1);
1203 }
```

Autrement dit :

- On initialise les trois champs files d'attente du descripteur de couche transport passé en argument.
- On renseigne le champ `sk_send_head` avec l'adresse nulle.
- On initialise le champ liste des minuteurs.
- On indique que ce descripteur doit se trouver dans l'espace mémoire noyau.
- On indique où se trouveront les données du tampon, en réception et en émission.
- On spécifie l'état de la socket, qui est fermée pour l'instant.
- On spécifie que le descripteur est zappé (rappelons que ce champ n'a d'intérêt que pour les familles de protocoles AX25 et IPX).
- On spécifie comme descripteur de socket associé au descripteur de couche transport celui dont l'adresse est passée en paramètre :
  - Si cette adresse est non nulle :
    - On renseigne le champ type du descripteur de couche transport avec celui du descripteur de socket.

- La file des entités en attente d'opérations sur le descripteur de couche transport est instantiée avec celle associée au descripteur de socket.
- On indique que le descripteur de couche transport associé au descripteur de socket est celui que l'on est en train de renseigner.

– sinon le champ `sk_sleep` du descripteur de couche transport est initialisé à zéro.

– On initialise les autres champs du descripteur de couche transport.

La constante symbolique générale `MAX_SCHEDULE_TIMEOUT` est définie dans le fichier `linux/include/linux/sched.h`:

Code Linux 2.6.10

```
180 #define MAX_SCHEDULE_TIMEOUT    LONG_MAX
```

– On initialise le compteur de référence du descripteur de couche réseau à 1.

24.3.2 Verrouillage

La fonction `lock_sock()` est définie dans le fichier `linux/net/core/sock.c`:

Code Linux 2.6.10

```
1205 void fastcall lock_sock(struct sock *sk)
1206 {
1207         might_sleep();
1208         spin_lock_bh(&(sk->sk_lock.slock));
1209         if (sk->sk_lock.owner)
1210                 __lock_sock(sk);
1211         sk->sk_lock.owner = (void *)1;
1212         spin_unlock_bh(&(sk->sk_lock.slock));
1213 }
```

précédemment définie dans le fichier `linux/include/net/sock.h`, précédée d'un commentaire:

Code Linux 2.6.0

```
529 /* Utilise par les processus pour "verrouiller" un etat de socket de facon a ce que
530  * les routines d'interruptions et de partie basse ne puissent pas le changer
531  * a notre insu. Il bloque essentiellement tous les paquets entrants
532  * de facon a ce que nous n'obtenions pas de nouvelles donnees ou de
533  * nouveaux paquets qui changent l'etat de la socket.
534  *
535  * Tant qu'il est verrouille, le processus BH ajoutera les nouveaux paquets a
536  * la file d'attente de nouveaux clients. Cette file d'attente est manipulee par le
537  * proprietaire de la socket verrouillee juste avant qu'elle ne soit liberee.
538  *
539  * Depuis ~2.3.5 elle est egalement verrouillee pour les acces en serie
540  * dans le contexte des processus utilisateur.
541  */
```

Autrement dit:

– On indique que cette activitée peut être assoupie.
– On verrouille l'état de la socket.
– Si le module propriétaire n'est pas verrouillé, on fait appel à la fonction auxiliaire `__lock_sock()`.
– On verrouille le module.
– On déverrouille l'état de la socket.

La fonction auxiliaire `__lock_sock()` est définie plus haut dans le même fichier:

Code Linux 2.6.10

```
916 static void __lock_sock(struct sock *sk)
917 {
918         DEFINE_WAIT(wait);
```

```
919
920         for(;;) {
921                 prepare_to_wait_exclusive(&sk->sk_lock.wq, &wait,
922                                           TASK_UNINTERRUPTIBLE);
923                 spin_unlock_bh(&sk->sk_lock.slock);
924                 schedule();
925                 spin_lock_bh(&sk->sk_lock.slock);
926                 if(!sock_owned_by_user(sk))
927                         break;
928         }
929         finish_wait(&sk->sk_lock.wq, &wait);
930 }
```

qui fait essentiellement appel à l'ordonnanceur.

24.3.3 Initialisation du délai

La fonction `sock_sndtimeo()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
1228 static inline long sock_sndtimeo(const struct sock *sk, int noblock)
1229 {
1230         return noblock ? 0 : sk->sk_sndtimeo;
1231 }
```