Chapitre 28

Création et fermeture des sockets

Nous avons vu au chapitre 6 un exemple de programmation avec l'API des sockets dans le cas d'une communication non connectée (ce qui correspond à UDP dans le cas de TCP/IP). Les opérations les plus importantes sont évidemment l'envoi et la réception de datagrammes mais avant cela il faut créer une socket, éventuellement initialiser dans le cas d'un serveur et après cela, fermer la socket. Nous allons étudier l'implémentation des appels systèmes de création, de semi-arrêt et de fermeture des sockets dans ce chapitre.

L'implémentation de chacun de ces appels système comprend une partie générale, indépendante de tout protocole, une partie dépendant de la famille d'adresses (ce qui correspond au protocole de la couche réseau) et une partie dépendant du type de communication (ce qui correspond au protocole de la couche transport). Nous choisirons toujours, à titre d'illustration, le cas de IPv4 pour la couche réseau et la cas UDP pour la couche transport.

28.1 Création d'une socket

La fonction utilisateur `socket()` est définie, comme pour tout appel système, grâce à la fonction d'appel `sys_socket()`. Celle-ci fait appel à la fonction générale `sock_create()` qui renvoie à une fonction de création d'une socket spécifique à la famille de protocoles et au type de communication indiqués.

28.1.1 Traitement général

28.1.1.1 Fonction d'appel

Code Linux 2.6.10

La fonction d'appel `sys_socket()` est définie dans le fichier `linux/net/socket.c`:

```
1187 asmlinkage long sys_socket(int family, int type, int protocol)
1188 {
1189         int retval;
1190         struct socket *sock;
1191
1192         retval = sock_create(family, type, protocol, &sock);
1193         if (retval < 0)
1194                 goto out;
1195
1196         retval = sock_map_fd(sock);
1197         if (retval < 0)
1198                 goto out_release;
1199
1200 out:
1201         /* Il peut y avoir deja un autre descripteur 8) Ce n'est pas le probleme du
               noyau. */
1202         return retval;
1203
1204 out_release:
1205         sock_release(sock);
1206         return retval;
1207 }
```

Autrement dit :

- On déclare une valeur de retour et un descripteur de socket.
- On fait appel à la fonction `sock_create()`, étudiée ci-après, qui a pour fonction d'instantier le descripteur de socket et de renvoyer 0 ou l'opposé d'un code d'erreur. S'il y a un problème, on renvoie l'opposé du code d'erreur fourni par cette fonction.
- On instantie un descripteur de fichier (de socket), grâce à la fonction `sock_map_fd()` étudiée ci-après, associé au descripteur de socket instantié ci-dessus. S'il y a un problème, on relâche le descripteur de socket et on renvoie l'opposé du code d'erreur adéquat. Sinon on renvoie le numéro de fichier.

28.1.1.2 Instantiation d'une socket

La fonction :

```
int sock_create(int family, int type, int protocol, struct socket **res);
```

vérifie que la famille de protocole `family` et le type de communication `type` sont implémentés, fait appel à la fonction spécifique de création de socket pour ceux-ci et place le descripteur de socket ainsi créé dans `res`. Elle renvoie 0 en cas de succès, l'opposé d'un code d'erreur sinon.

La fonction `int sock_create()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
1074 static int __sock_create(int family, int type, int protocol, struct socket **res, int kern)
1075 {
1076         int i;
1077         int err;
1078         struct socket *sock;
1079
1080         /*
1081          *      Verifie si le protocole est dans l'intervalle
1082          */
1083         if (family < 0 || family >= NPROTO)
1084                 return -EAFNOSUPPORT;
1085         if (type < 0 || type >= SOCK_MAX)
1086                 return -EINVAL;
1087
1088         /* Compatibilite.
1089
1090            Ce truc affreux a ete deplace de la couche INET jusqu'ici pour eviter
1091             une impasse lors du chargement du module.
1092          */
1093         if (family == PF_INET && type == SOCK_PACKET) {
1094                 static int warned;
1095                 if (!warned) {
1096                         warned = 1;
1097                         printk(KERN_INFO "%s uses obsolete (PF_INET,SOCK_PACKET)\n",
                                        current->comm);
1098                 }
1099                 family = PF_PACKET;
1100         }
1101
1102         err = security_socket_create(family, type, protocol, kern);
1103         if (err)
1104                 return err;
1105
1106 #if defined(CONFIG_KMOD)
1107         /* Essaie de charger un module de protocole si la recherche a echouee.
1008          *
1009          * 12/09/1996 Marcin : Mais ! ceci a REELLEMENT un sens si l'utilisateur
1010          * demande un support de reseau bien caracterise et reel lors de la configuration.
1011          * Sinon le support du module se brisera !
1112          */
1113         if (net_families[family]==NULL)
1114         {
1115                 request_module("net-pf-%d",family);
1116         }
1117 #endif
1118
1119         net_family_read_lock();
1120         if (net_families[family] == NULL) {
1121                 i = -EAFNOSUPPORT;
1122                 goto out;
1123         }
1124
1125 /*
1126  *      Alloue la socket et permet a la famille d'activer des choses. Si
1127  *      le protocole est 0, la famille a pour instruction de selectionner une
1128  *      valeur par defaut appropriee.
1129  */
1130
1131         if (!(sock = sock_alloc()))
1132         {
1133                 printk(KERN_WARNING "socket: no more sockets\n");
```

```
1134                 i = -ENFILE;            /* Ne concorde pas exactement, mais c'est la
1135                                            chose posix la plus proche */
1136                 goto out;
1137         }
1138
1139         sock->type  = type;
1140
1141         /*
1142          * Nous appellerons la fonction ->create, qui est certainement dans un module
1143          * chargeable, aussi avons-nous a tamponner le refcnt du module chargeable d'abord.
1144          */
1145         i = -EAFNOSUPPORT;
1146         if (!try_module_get(net_families[family]->owner))
1147                 goto out_release;
1148
1149         if ((i = net_families[family]->create(sock, protocol)) < 0)
1150                 goto out_module_put;
1151         /*
1152          * Tamponner maintenant le refcnt du module [chargeable] qui possede cette
1153          * socket. Au moment de sock_release, nous decrementons ce refcnt.
1154          */
1155         if (!try_module_get(sock->ops->owner)) {
1156                 sock->ops = NULL;
1157                 goto out_module_put;
1158         }
1159         /*
1160          * Maintenant que nous avons fini avec la fonction ->create, le module
1161          * [chargeable] peut avoir son refcnt decremente
1162          */
1163         module_put(net_families[family]->owner);
1164         *res = sock;
1165         security_socket_post_create(sock, family, type, protocol, kern);
1166
1167 out:
1168         net_family_read_unlock();
1169         return i;
1170 out_module_put:
1171         module_put(net_families[family]->owner);
1172 out_release:
1173         sock_release(sock);
1174         goto out;
1175 }
1176
1177 int sock_create(int family, int type, int protocol, struct socket **res)
1178 {
1179         return __sock_create(family, type, protocol, res, 0);
1180 }
```

Autrement dit :

- On déclare un descripteur de socket.
- Si le type de famille de protocoles n'appartient pas à l'intervalle adéquat, on renvoie l'opposé du code d'erreur `EAFNOSUPPORT`.
- Si le type de communication n'appartient pas à l'intervalle adéquat, on renvoie l'opposé du code d'erreur `EINVAL`.
- L'interface ayant évoluée, un petit message est indiqué si on utilise le couple "PF_INET" et "SOCK_PACKET" au lieu du couple "PF_PACQUET" et "SOCK_PACKET", puis le type de famille de protocoles voulu est postionné.
- On fait appel à la fonction `security_socket_create()`, sur laquelle nous allons revenir ci-dessous, et on s'arrête éventuellement en renvoyant le code d'erreur fourni par celle-ci.

- Si la famille de protocoles n'est pas installée, on essaie de charger le module correspondant si l'on permet les modules.
- Si la famille de protocoles n'est (toujours) pas installée, on renvoie l'opposé du code d'erreur `EAFNOSUPPORT`.

Les fonctions en ligne `net_family_read_lock()` et `net_family_read_unlock()` sont définies dans le fichier `linux/net/socket.c`:

Code Linux 2.6.10

```
171 static __inline__ void net_family_read_lock(void)
172 {
173         atomic_inc(&net_family_lockct);
174         spin_unlock_wait(&net_family_lock);
175 }
176
177 static __inline__ void net_family_read_unlock(void)
178 {
179         atomic_dec(&net_family_lockct);
180 }
181
182 #else
183 #define net_family_write_lock() do { } while(0)
184 #define net_family_write_unlock() do { } while(0)
185 #define net_family_read_lock() do { } while(0)
186 #define net_family_read_unlock() do { } while(0)
187 #endif
```

- On essaie d'instantier le descripteur de socket, grâce à la fonction `sock_alloc()` étudiée ci-après. Si on n'y parvient pas, on affiche un message d'erreur et on renvoie l'opposé du code d'erreur `ENFILE`.
- On renseigne le champ type de communication du descripteur de socket avec celui passé en argument.
- Si la famille de protocoles a ete chargée depuis un module, on essaie d'incrémenter le compteur de références de celui-ci. Si on n'y parvient pas, on libère le descripteur de socket, on déverrouille la famille et on renvoie l'opposé du code d'erreur `EAFNOSUPPORT`.

  La fonction en ligne générale `try_module_get()` est définie dans le fichier `linux/include/linux/module.h`.
- On essaie de renseigner les champs du descripteur de socket en utilisant la fonction de création spécifique à la famille de protocoles. Si on n'y parvient pas, on décrémente le compteur de référence du module de la famille de protocoles, on libère le descripteur de socket et on renvoie l'opposé du code d'erreur fourni par la fonction spécifique de création.

  La fonction en ligne générale `module_put()` est définie dans le fichier `linux/include/linux/module.h`.
- On essaie d'incrémenter le compteur de références du module correspondant à l'ensemble des opérations. Si on n'y parvient pas, on positionne à `NULL` le champ `ops` du descripteur de socket, on décrémente le compteur de référence au module de la famille de protocoles, on libère le descripteur de socket, on déverrouille la famille et on renvoie 0.
- On décrémente le compteur de référence du module de la famille de protocoles.
- Le résultat est renseigné avec le descripteur de socket obtenu.
- On fait appel à la fonction de sécurité `security_socket_post_create()`.
- On déverrouille la lecture pour la famille de protocoles et on renvoie 0.

28.1.1.3 Instantiation d'un descripteur de socket

Code Linux 2.6.10

La fonction `sock_alloc()` permet d'obtenir un nouveau descripteur de socket, paramétré grâce à la variable globale `sock_mnt`. Elle est définie dans le fichier `linux/net/socket.c` :

```
454 /**
455  *      sock_alloc      -       alloue une socket
456  *
457  *      Alloue un nouveau inode et un nouveau objet socket. Les deux sont lies ensemble
458  *      et initialises. La socket est alors renvoyee. Si nous sommes a court d'inodes,
459  *      NULL est renvoye.
460  */
461
462 static struct socket *sock_alloc(void)
463 {
464         struct inode * inode;
465         struct socket * sock;
466
467         inode = new_inode(sock_mnt->mnt_sb);
468         if (!inode)
469                 return NULL;
470
471         sock = SOCKET_I(inode);
472
473         inode->i_mode = S_IFSOCK|S_IRWXUGO;
474         inode->i_sock = 1;
475         inode->i_uid = current->fsuid;
476         inode->i_gid = current->fsgid;
477
478         get_cpu_var(sockets_in_use)++;
479         put_cpu_var(sockets_in_use);
480         return sock;
481 }
```

Autrement dit :

- Un descripteur de nœud d'information et un descripteur de socket sont déclarés.
- On essaie d'instantier le descripteur de nœud d'information et de positionner son point de montage, initialisé grâce à la variable `sock_mnt`. Si on n'y parvient pas, on renvoie `NULL`.

  Code Linux 2.6.10

  La variable `sock_mnt` est définie un peu plus haut dans le même fichier :

  ```
  333 static struct vfsmount *sock_mnt;
  ```

- L'adresse du descripteur de socket est initialisée avec celle du descripteur de socket associé au descripteur de nœud d'information.
- On renseigne un certain nombre de champs du descripteur de nœud d'information (son mode d'accès, le fait qu'il soit un nœud d'information de socket, l'utilisateur et le groupe d'utilisateur associés).
- On incrémente le nombre de sockets en utilisation sur le microprocesseur.

  Code Linux 2.6.10

  La variable `sockets_in_use`, une par microprocesseur, est définie plus haut dans le même fichier :

  ```
  190 /*
  191  *      Compteurs de statistiques des listes de sockets
  192  */
  193
  194 static DEFINE_PER_CPU(int, sockets_in_use) = 0;
  ```

- On renvoie l'adresse du descripteur de socket.

28.1.1.4 Points d'ancrage des opérations sécurisées

Nous allons rencontrer plusieurs fois des points d'ancrage à des opérations sécurisées dans la version 2.6.10 de Linux. Il s'agit d'une option de configuration du noyau. Par défaut ces opérations ne font rien. Nous ne nous intéresseront pas à elles dans une première étape. Nous nous contenterons de donner quelques indications.

Une variable `security_ops` est définie dans le fichier `linux/security/security.c`: Code Linux 2.6.10

```
27 struct security_operations *security_ops;      /* Initialise a NULL */
```

Le type `struct security_operations` est défini dans le fichier `linux/include/linux/security.h`: Code Linux 2.6.10

```
92  /**
93   * struct security_operations - structure principale pour la securite
94   *
[...]
650  * Points d'ancrage de securite pour le protocole reseau Unix.
651  *
652  * @unix_stream_connect :
653  *      Verifie les droits d'acces avant d'etablir une connexion de flux du domaine Unix
654  *      entre @sock et @other.
655  *      @sock contient la structure de socket.
656  *      @other contient la structure de socket paire.
657  *      Renvoie 0 si le droit d'acces est accorde.
658  * @unix_may_send :
659  *      Verifie les droits d'acces avant une connexion ou l'envoi de  datagrammes de @sock
660  *      a @other.
661  *      @sock contient la structure de socket.
662  *      @sock contient la structure de socket paire.
663  *      Renvoie 0 si le droit d'acces est accorde.
664  *
665  * Les points d'ancrage @unix_stream_connect et @unix_may_send sont necessaires puisque
666  * Linux fournit une alternative a l'espace de noms de fichier classique pour les
667  * sockets du domaine Unix. Bien que la liaison et la connexion des sockets dans l'espace des
668  * noms de fichier se fasse a travers les droits d'acces de fichiers typiques (et attrapes par
669  * mknod et les points d'ancrage de droits d'acces dans inode_security_ops), la liaison et la
670  * connexion aux sockets dans l'espace de noms abstrait se fait sans intermediaire aucun.
671  * Un controle suffisant des sockets du domaine Unix dans l'espace des noms abstrait n'est pas
672  * possible en utilisant seulement les points d'ancrage de la couche des sockets, puisque nous
673  * avons besoin de connaitre la socket cible, qui n'est pas consultee jusqu'a ce que nous
       soyons dans le code af_unix.
674  *
675  * Points d'ancrage de securite pour les operations sur les sockets.
676  *
677  * @socket_create :
678  *      Verifie les droits d'acces avant de creer une nouvelle socket.
679  *      @family contient la famille de protocoles requise.
680  *      @type contient le type de communication requis.
681  *      @protocol contient le protocole requis.
682  *      @kern positionne a 1 si c'est une socket noyau.
683  *      Renvoie 0 si le droit d'acces est accorde.
684  * @socket_post_create :
685  *      Ce point d'ancrage permet a un module de mettre a jour ou d'allouer une structure de
686  *      securite par socket. Noter que le champ de securite n'est pas ajoute directement a la
687  *      structure de socket, mais que l'information de securite de la socket est stockee
688  *      dans le inode associe. Typiquement le point d'ancrage d'inode alloc_security
689  *      allouera et attachera une information de securite a
690  *      sock->inode->i_security. Ce point d'ancrage peut etre utilise pour mettre a jour le
691  *      champ sock->inode->i_security avec des informations supplementaires qui ne sont pas
692  *      disponible lorsque l'inode a ete alloue.
693  *      @sock contient la structure de la socket nouvellement creee.
```

```
694  *      @family contient la famille de protocoles requise.
695  *      @type contient le type de communication requis.
696  *      @protocol contient le protocole requis.
697  *      @kern positionne a 1 si c'est une socket noyau.
698  * @socket_bind :
699  *      Verifies les droits d'acces avant que l'operation de liaison de la couche de protcole
700  *      des sockets soit effectuee et que la socket @sock soit liee a l'adresse specifiee dans
701  *      le parametre @address.
702  *      @sock contient la structure de socket.
703  *      @address contient l'adresse a laquelle lier.
704  *      @addrlen contient la longueur de l'adresse.
705  *      Renvoie 0 si le droit d'acces est accorde.
706  * @socket_connect :
707  *      Verifie les droits d'acces avant que l'operation de connexion de la couche protocole
708  *      des sockets essaie de connecter la socket @sock a une adresse distante, @address.
709  *      @sock contient la structure de socket.
710  *      @address contient l'adresse du point distant.
711  *      @addrlen contient la longueur de l'adresse.
712  *      Renvoie 0 si le droit d'acces est accorde.
713  * @socket_listen :
714  *      Verifie les droits d'acces avant l'operation d'ecoute de la couche protocole des
         sockets.
715  *      @sock contient la structure de socket.
716  *      @backlog contient la longueur maximum de la file d'attente de connection.
717  *      Renvoie 0 si le droit d'acces est accorde.
718  * @socket_accept :
719  *      Verifie les droits d'acces avant d'accepter une nouvelle connexion. Noter que la
720  *      nouvelle socket, @newsock, a ete creee et quelques informations copiees dans
721  *      celle-ci, mais l'operation d'acceptation n'a pas encore ete effectuee.
722  *      @sock contient la structure de socket d'ecoute.
723  *      @newsock contient la socket serveur nouvellement creee pour la connexion.
724  *      Renvoie 0 si le droit d'acces est accorde.
725  * @socket_post_accept :
726  *      Ce point d'ancrage permet a un module de securite de copier des informations
727  *      de securite dans l'inode de socket nouvellement cree.
728  *      @sock contient la structure de socket d'ecoute.
729  *      @newsock contient la socket serveur nouvellement creee pour la connexion.
730  * @socket_sendmsg :
731  *      Verifie les droits d'acces avant de transmettre un message a une autre socket.
732  *      @sock contient la structure de socket.
733  *      @msg contient le message a transmettre.
734  *      @size contient la taille du message.
735  *      Renvoie 0 si le droit d'acces est accorde.
736  * @socket_recvmsg :
737  *      Verifie les droits d'acces avant de recevoir un message d'une socket.
738  *      @sock contient la structure de socket.
739  *      @msg contient la structure de message.
740  *      @size contient la taille de la structure de message.
741  *      @flags contient les drapeaux operationnels.
742  *      Renvoie 0 si le droit d'acces est accorde.
743  * @socket_getsockname :
744  *      Verifie les droits d'acces avant que l'adresse locale (nom) de l'objet socket
745  *      @sock soit recuperee.
746  *      @sock contient la structure de socket.
747  *      Renvoie 0 si le droit d'acces est accorde.
748  * @socket_getpeername :
749  *      Verifie les droits d'acces avant que l'adresse distante (nom) de l'objet socket
750  *      @sock soit recuperee.
751  *      @sock contient la structure de socket.
752  *      Renvoie 0 si le droit d'acces est accorde.
753  * @socket_getsockopt :
754  *      Verifie les droits d'acces avant de recuperer les options associees a la socket
```

```
755  *      @sock.
756  *      @sock contient la structure de socket.
757  *      @level contient le niveau de protocole duquel recuperer l'option.
758  *      @optname contient le nom de l'option a recuperer.
759  *      Renvoie 0 si le droit d'acces est accorde.
760  * @socket_setsockopt :
761  *      Verifie les droits d'acces avant de positionner les options associees a une socket
762  *      @sock.
763  *      @sock contient la structure de socket.
764  *      @level contient le niveau de protocole auquel positionner les options.
765  *      @optname contient le nom de l'option a positionner.
766  *      Renvoie 0 si le droit d'acces est accorde.
767  * @socket_shutdown :
768  *      Verifie les droits d'acces avant que tout ou partie d'une connexion a la socket
769  *      @sock soit fermee.
770  *      @sock contient la structure de socket.
771  *      @how contient le drapeau indiquant comment les envois et reception futurs seront
         manipules.
772  *      Renvoie 0 si le droit d'acces est accorde.
773  * @socket_sock_rcv_skb :
774  *      Verifie les droits d'acces sur les paquets reseau entrants. Ce point d'ancrage
775  *      est different des points d'ancrages d'entree IP de Netfilter puisque c'est la premiere
776  *      fois que le sk_buff entrant @skb est associe a une socket particuliere, @sk.
777  *      @sk contient le sock (non socket) associe au sk_buff entrant.
778  *      @skb contient les donnees reseau entrantes.
[...]
1018 * C'est la structure de securite principale.
1019 */
1020 struct security_operations {
[...]
1203 #ifdef CONFIG_SECURITY_NETWORK
1204         int (*unix_stream_connect) (struct socket * sock,
1205                                     struct socket * other, struct sock * newsk);
1206         int (*unix_may_send) (struct socket * sock, struct socket * other);
1207
1208         int (*socket_create) (int family, int type, int protocol, int kern);
1209         void (*socket_post_create) (struct socket * sock, int family,
1210                                     int type, int protocol, int kern);
1211         int (*socket_bind) (struct socket * sock,
1212                             struct sockaddr * address, int addrlen);
1213         int (*socket_connect) (struct socket * sock,
1214                                struct sockaddr * address, int addrlen);
1215         int (*socket_listen) (struct socket * sock, int backlog);
1216         int (*socket_accept) (struct socket * sock, struct socket * newsock);
1217         void (*socket_post_accept) (struct socket * sock,
1218                                     struct socket * newsock);
1219         int (*socket_sendmsg) (struct socket * sock,
1220                                struct msghdr * msg, int size);
1221         int (*socket_recvmsg) (struct socket * sock,
1222                                struct msghdr * msg, int size, int flags);
1223         int (*socket_getsockname) (struct socket * sock);
1224         int (*socket_getpeername) (struct socket * sock);
1225         int (*socket_getsockopt) (struct socket * sock, int level, int optname);
1226         int (*socket_setsockopt) (struct socket * sock, int level, int optname);
1227         int (*socket_shutdown) (struct socket * sock, int how);
1228         int (*socket_sock_rcv_skb) (struct sock * sk, struct sk_buff * skb);
1229         int (*socket_getpeersec) (struct socket *sock, char __user *optval,
                                      int __user *optlen, unsigned len);
1230         int (*sk_alloc_security) (struct sock *sk, int family, int priority);
1231         void (*sk_free_security) (struct sock *sk);
1232 #endif  /* CONFIG_SECURITY_NETWORK */
1233 };
```

28.1.1.5 Association d'un descripteur de fichier à un descripteur de socket

La fonction `sock_map_fd()` permet d'obtenir un nouveau descripteur de fichier (de type socket) et de l'associer au descripteur de socket entré en paramètre. Elle est définie dans le fichier `linux/net/socket.c`:

Code Linux 2.6.10

```
348 /*
349  *      Obtient le premier descripteur de fichier disponible et l'active pour l'utiliser.
350  *
351  *      Cette fonction cree une structure file et l'associe a l'espace des fd
352  *      du processus en cours. En cas de succes elle renvoie de descripteur de fichier
353  *      et la structure file est implicitement stockee dans sock->file.
354  *      Noter qu'un autre thread peut fermer le descripteur de fichier avant que nous
355  *      ne sortions de cette fonction. Nous utilisons le fait que nous ne faisons pas
356  *      reference a la socket immediatement apres l'association. Si un jour nous en avons
357  *      besoin, cette fonction incrementera le compteur de ref. du fichier de 1.
358  *
359  *      En tout cas le fd renvoye PEUT NE PAS ETRE valide !
360  *      Cette condition de course est inevitable
361  *      avec les espaces de fd partages, nous ne pouvons pas le resoudre dans le noyau,
362  *      mais nous prenons encore soin de la coherence interne.
363  */
364
365 int sock_map_fd(struct socket *sock)
366 {
367         int fd;
368         struct qstr this;
369         char name[32];
370
371         /*
372          *      Trouver un descripteur de fichier adequet a renvoyer a l'utilisateur.
373          */
374
375         fd = get_unused_fd();
376         if (fd >= 0) {
377                 struct file *file = get_empty_filp();
378
379                 if (!file) {
380                         put_unused_fd(fd);
381                         fd = -ENFILE;
382                         goto out;
383                 }
384
385                 sprintf(name, "[%lu]", SOCK_INODE(sock)->i_ino);
386                 this.name = name;
387                 this.len = strlen(name);
388                 this.hash = SOCK_INODE(sock)->i_ino;
389
390                 file->f_dentry = d_alloc(sock_mnt->mnt_sb->s_root, &this);
391                 if (!file->f_dentry) {
392                         put_filp(file);
393                         put_unused_fd(fd);
394                         fd = -ENOMEM;
395                         goto out;
396                 }
397                 file->f_dentry->d_op = &sockfs_dentry_operations;
398                 d_add(file->f_dentry, SOCK_INODE(sock));
399                 file->f_vfsmnt = mntget(sock_mnt);
400                 file->f_mapping = file->f_dentry->d_inode->i_mapping;
401
402                 sock->file = file;
403                 file->f_op = SOCK_INODE(sock)->i_fop = &socket_file_ops;
404                 file->f_mode = FMODE_READ | FMODE_WRITE;
```

```
405                 file->f_flags = O_RDWR;
406                 file->f_pos = 0;
407                 fd_install(fd, file);
408         }
409
410 out:
411         return fd;
412 }
```

Autrement dit :

- On déclare un numéro de fichier `fd`, un métanom `this` et une chaîne de caractères `name`.
- On essaie de trouver un numéro de fichier non utilisé. S'il n'y en a pas, on renvoie le code d'erreur fourni par la fonction générale `get_unused_fd()`.
- On essaie de trouver un descripteur de fichier (vide) non utilisé. Si on n'y parvient pas, on libère le numéro de fichier et on renvoie l'opposé du code d'erreur `ENFILE`.
- On instantie `name` avec le nunméro du descripteur de socket.

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

  ```
  654 static inline struct inode *SOCK_INODE(struct socket *socket)
  655 {
  656         return &container_of(socket, struct socket_alloc, socket)->vfs_inode;
  657 }
  ```

  Elle renvoie l'adresse de `socket->vfs_inode`.
- On renseigne les champs du métanom `this`.
- On essaie d'allouer une entrée de répertoire au descripteur de fichier. Si on n'y parvient pas, on libère le descripteur de fichier, on libère le numéro de fichier et on renvoie l'opposé du code d'erreur `ENOMEM`.
- On associe comme ensemble d'opérations sur les entrées de répertoire l'ensemble `sockfs_dentry_operations` vu au chapitre 27.
- On place comme descripteur de nœud d'information associé à l'entrée de répertoire initialisée ci-dessus le descripteur de nœud d'information associé au descripteur de socket.
- Le point de montage du descripteur de fichier est `sock_mnt`.
- Le descripteur de fichier associé au descripteur de socket sera celui que l'on vient d'initialiser.
- On initialise quelques champs du descripteur de fichier et on associe celui-ci au numéro de fichier `fd` que l'on renvoie.

28.1.2 Cas de la famille de protocoles IPv4

Nous avons ci-dessus que la fonction `sock_create()` fait appel à une fonction spécifique de création, dépendant de la famille de protocoles, pour renseigner les champs du descripteur de socket. L'instance `inet_family_ops` de `net_proto_family` montre que, dans le cas de IPv4, la fonction spécifique de création de socket s'appelle `inet_create()`.

28.1.2.1 Création d'une socket

La fonction `inet_create()` est définie dans le fichier `linux/net/ipv4/af_inet.c`: Code Linux 2.6.10

```
226 /*
227  *      Cree une socket inet.
228  */
229
```

```
230 static int inet_create(struct socket *sock, int protocol)
231 {
232         struct sock *sk;
233         struct list_head *p;
234         struct inet_protosw *answer;
235         struct inet_opt *inet;
236         struct proto *answer_prot;
237         unsigned char answer_flags;
238         char answer_no_check;
239         int err;
240
241         sock->state = SS_UNCONNECTED;
242
243         /* Cherche le couple type/protocole requis. */
244         answer = NULL;
245         rcu_read_lock();
246         list_for_each_rcu(p, &inetsw[sock->type]) {
247                 answer = list_entry(p, struct inet_protosw, list);
248
249                 /* Verifie la concordance non-sauvage. */
250                 if (protocol == answer->protocol) {
251                         if (protocol != IPPROTO_IP)
252                                 break;
253                 } else {
254                         /* Verification pour les deux cas sauvages. */
255                         if (IPPROTO_IP == protocol) {
256                                 protocol = answer->protocol;
257                                 break;
258                         }
259                         if (IPPROTO_IP == answer->protocol)
260                                 break;
261                 }
262                 answer = NULL;
263         }
264
265         err = -ESOCKTNOSUPPORT;
266         if (!answer)
267                 goto out_rcu_unlock;
268         err = -EPERM;
269         if (answer->capability > 0 && !capable(answer->capability))
270                 goto out_rcu_unlock;
271         err = -EPROTONOSUPPORT;
272         if (!protocol)
273                 goto out_rcu_unlock;
274
275         sock->ops = answer->ops;
276         answer_prot = answer->prot;
277         answer_no_check = answer->no_check;
278         answer_flags = answer->flags;
279         rcu_read_unlock();
280
281         BUG_TRAP(answer_prot->slab != NULL);
282
283         err = -ENOBUFS;
284         sk = sk_alloc(PF_INET, GFP_KERNEL,
285                       answer_prot->slab_obj_size,
286                       answer_prot->slab);
287         if (sk == NULL)
288                 goto out;
289
290         err = 0;
291         sk->sk_prot = answer_prot;
```

```
292         sk->sk_no_check = answer_no_check;
293         if (INET_PROTOSW_REUSE & answer_flags)
294                 sk->sk_reuse = 1;
295
296         inet = inet_sk(sk);
297
298         if (SOCK_RAW == sock->type) {
299                 inet->num = protocol;
300                 if (IPPROTO_RAW == protocol)
301                         inet->hdrincl = 1;
302         }
303
304         if (ipv4_config.no_pmtu_disc)
305                 inet->pmtudisc = IP_PMTUDISC_DONT;
306         else
307                 inet->pmtudisc = IP_PMTUDISC_WANT;
308
309         inet->id = 0;
310
311         sock_init_data(sock, sk);
312         sk_set_owner(sk, sk->sk_prot->owner);
313
314         sk->sk_destruct    = inet_sock_destruct;
315         sk->sk_family      = PF_INET;
316         sk->sk_protocol    = protocol;
317         sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;
318
319         inet->uc_ttl    = -1;
320         inet->mc_loop   = 1;
321         inet->mc_ttl    = 1;
322         inet->mc_index  = 0;
323         inet->mc_list   = NULL;
324
325 #ifdef INET_REFCNT_DEBUG
326         atomic_inc(&inet_sock_nr);
327 #endif
328
329         if (inet->num) {
330                 /* On suppose que tout protocole qui permet a
331                  * l'utilisateur d'assigner un numero a une socket
332                  * partage automatiquement le temps de la
333                  * creation.
334                  */
335                 inet->sport = htons(inet->num);
336                 /* Ajoute aux chaines de hachage du protocole. */
337                 sk->sk_prot->hash(sk);
338         }
339
340         if (sk->sk_prot->init) {
341                 err = sk->sk_prot->init(sk);
342                 if (err)
343                         sk_common_release(sk);
344         }
345 out:
346         return err;
347 out_rcu_unlock:
348         rcu_read_unlock();
349         goto out;
350 }
```

Autrement dit :

– On déclare un descripteur de couche transport `sk`, une tête de liste, un protocole de com-

munication réponse, une option Internet, un protocole réponse, des drapeaux, une réponse de non vérification et un code de retour.

- On initialise l'état du descripteur de socket à non connecté.
- On essaie d'instantier le descripteur de type de communication en vérifiant que le couple famille de protocoles – type de communication est valide (lignes 243–263). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ESOCKNOTSUPPORT`.

Code Linux 2.6.10

La macro générale `list_for_each_rcu()` est définie dans le fichier `linux/include/linux/list.h`:

```
427 /**
428  * list_for_each_rcu    -       itere sur une liste rcu-protegee
429  * @pos :       la &struct list_head a utiliser comme compteur de boucle.
430  * @head :      la tete de votre liste.
431  *
432  * Cette primitive de parcours de liste peut de facon sure s'effectuer en meme temps
433  * que des primitives de mutation de listes _rcu telles que list_add_rcu()
434  * tant que le parcours est protege par rcu_read_lock().
435  */
436 #define list_for_each_rcu(pos, head) \
437         for (pos = (head)->next, prefetch(pos->next); pos != (head); \
438                 pos = rcu_dereference(pos->next), prefetch(pos->next))
```

Code Linux 2.6.10

La macro générale `list_entry()` est définie dans le même fichier général:

```
314 /**
315  * list_entry - recupere la struct pour cette entree
316  * @ptr :       le pointeur &struct list_head.
317  * @type :      le type de la struct dans laquelle ceci est immerge.
318  * @member :    le nom de la list_struct dans la struct.
319  */
320 #define list_entry(ptr, type, member) \
321         container_of(ptr, type, member)
```

- S'il y a un problème de capacité, on renvoie l'opposé du code d'erreur `EPERM`.
- S'il y a un problème avec la famille de protocoles, on renvoie l'opposé du code d'erreur `EPROTONOTSUPPORT`.
- On renseigne le champ ensemble des opérations du descripteur de socket.
- On essaie d'instantier le descripteur de couche transport. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOBUFS`.
- On renseigne les champs protocole, pas de vérification et réutilisation du descripteur de couche transport.
- On instantie l'option Internet.
- On renseigne certains champs de cette option Internet : le numéro de port local et le fait que les en-têtes sont incluses si le type de la couche transport est le type brut ; le champ `pmtudisc` et le compteur d'identification.

Code Linux 2.6.10

La structure `ipv4_config` est définie dans le fichier `linux/include/net/ip.h`:

```
145 struct ipv4_config
146 {
147         int     log_martians;
148         int     autoconfig;
149         int     no_pmtu_disc;
150 };
```

Code Linux 2.6.10

La variable `ipv4_config` est définie dans le fichier `linux/net/ipv4/sysctl_net_ipv4.c`:

```
57 struct ipv4_config ipv4_config;
```

Les constantes symboliques IP_PMTUDISC_DONT et IP_PMTUDISC_WANT sont définies dans le fichier linux/include/linux/ip.h: Code Linux 2.6.10

```
78 /* Valeurs de IP_MTU_DISCOVER */
79 #define IP_PMTUDISC_DONT                0       /* Ne jamais envoyer de trames DF */
80 #define IP_PMTUDISC_WANT                1       /* Indications de route par
                                                      l'utilisateur */
81 #define IP_PMTUDISC_DO                  2       /* Toujours DF            */
```

- On associe les descripteurs de socket et de couche de transport.
- On s'approprie le descripteur de couche transport.

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

```
577 static __inline__ void sk_set_owner(struct sock *sk, struct module *owner)
578 {
579         /*
580          * On ne devrait utiliser sk_set_owner qu'une seule fois, apres la creation de
               struct sock,
581          * faites-le rapidement apres sk_alloc ou apres une fonction qui renvoie une
582          * nouvelle struct sock (et ceci abaisse la chaine d'appel appelee sk_alloc),
583          * par exemple les modules IPv4 et IPv6 partagent tcp_create_openreq_child, donc
584          * si tcp_create_openreq_child appelait sk_set_owner IPv6 devrait changer
585          * le proprietaire de cette struct sock, le remplacant par un qui n'est pas
586          * necessairement un appel sk_set_owner ephemere.
587          */
588         BUG_ON(sk->sk_owner != NULL);
589
590         sk->sk_owner = owner;
591         __module_get(owner);
592 }
```

- On renseigne des champs du descripteur de couche transport et de l'option Internet.
- Si le numéro de port local de l'option Internet est non nul, on le place comme numéro de port source de celle-ci et on ajoute le descripteur de couche de transport aux chaînes de hachage du protocole.
- Si une fonction d'initialisation est associée au descripteur de couche transport, on fait appel à celle-ci. Si tout ne se passe pas correctement, on libère le descripteur de couche de transport, grâce à la fonction sk_common_release() étudiée ci-dessous. Sinon on renvoie 0.

Les actions essentielles consistent donc à spécifier les deux ensembles d'opérations et à associer un descripteur de couche transport au descripteur général de socket.

28.1.2.2 Libération d'un descripteur de couche de transport

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

```
1297 void sk_common_release(struct sock *sk)
1298 {
1299         if (sk->sk_prot->destroy)
1300                 sk->sk_prot->destroy(sk);
1301
1302         /*
1303          * Observation : lorsque sock_common_release est appelee, les processus n'ont
1304          * pas acces a la socket. Mais le reseau l'a encore.
1305          * Etape un, la detacher du reseau :
1306          *
1307          * A. Retirer des tables de hachage.
```

```
1308          */
1309
1310         sk->sk_prot->unhash(sk);
1311
1312         /*
1313          * A ce point, la socket ne peut plus recevoir de nouveaux paquets, mais il est
1314          * possible que des paquets soient en route puisque certains CPU effectuent des
1315          * receptions et ont consulte la table de hachage avant que nous n'en retirions
1316          * la socket. Ils termineront la file d'attente de reception et seront purges
                par le destructeur de la socket.
1317          *
1318          * Ainsi nous avons encore des paquets en suspens sur la file d'attente en
1319          * reception et probablement nos propres paquets attendront dans les files
                d'attente des peripheriques. sock_destroy drainera
1320          * la file d'attente en reception, mais les paquets transmis retarderont la
1321          * destruction de la socket jusqu'a ce que la derniere reference soit liberee.
1322          */
1323
1324         sock_orphan(sk);
1325
1326         xfrm_sk_free_policy(sk);
1327
1328 #ifdef INET_REFCNT_DEBUG
1329         if (atomic_read(&sk->sk_refcnt) != 1)
1330                 printk(KERN_DEBUG "Destruction of the socket %p delayed, c=%d\n",
1331                        sk, atomic_read(&sk->sk_refcnt));
1332 #endif
1333         sock_put(sk);
1334 }
```

Code Linux 2.6.10

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

```
903 /* Detache la socket du constexte processus.
904  * Annonce la mort de la socket, la detache de la file d'attente et du noeud d'information.
905  * Noter que le noeud d'information parent tient la compteur de reference sur cette struct
906  * sock, nous ne devons pas la liberer dans cette fonction, puisque le protocole
907  * veut probablemnt des nettoyages supplementaires ou meme continuer
908  * a travailler avec cette socket (TCP).
909  */
910 static inline void sock_orphan(struct sock *sk)
911 {
912         write_lock_bh(&sk->sk_callback_lock);
913         sock_set_flag(sk, SOCK_DEAD);
914         sk->sk_socket = NULL;
915         sk->sk_sleep  = NULL;
916         write_unlock_bh(&sk->sk_callback_lock);
917 }
```

28.2 Semi-arrêt d'une socket

28.2.1 Traitement général

28.2.1.1 Fonction d'appel

Code Linux 2.6.10

La fonction `sys_shutdown()` est définie dans le fichier `linux/net/socket.c`:

```
1663 /*
1664  *      Ferme une socket.
1665  */
1666
1667 asmlinkage long sys_shutdown(int fd, int how)
```

```
1668 {
1669         int err;
1670         struct socket *sock;
1671
1672         if ((sock = sockfd_lookup(fd, &err))!=NULL)
1673         {
1674                 err = security_socket_shutdown(sock, how);
1675                 if (err) {
1676                         sockfd_put(sock);
1677                         return err;
1678                 }
1679
1680                 err=sock->ops->shutdown(sock, how);
1681                 sockfd_put(sock);
1682         }
1683         return err;
1684 }
```

Autrement dit :

- On déclare un code d'erreur et un descripteur de socket.
- On essaie de d'instantier le descripteur de socket avec celui associé au numéro de fichier (de socket) passé en argument. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur fourni par la fonction `sockfd_lookup()`, étudiée ci-dessous.
- On fait appel à la fonction de sécurité `security_socket_shutdown()`, dont nous ne nous occuperons pas ici. Si elle n'accorde pas le droit d'accès, on libère le descripteur de fichier associé à la socket et on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction d'arrêt spécifique à la famille d'adresses. On libère le descripteur de fichier associé à la socket et on renvoie 0 si tout s'est bien passé ou l'opposé du code d'erreur fourni par cette fonction spécifique sinon.

28.2.1.2 Descripteur de socket associé à un numéro de fichier

La fonction `sockfd_lookup()` permet d'obtenir le descripteur de socket associé à un numéro de fichier (de socket). Elle est définie dans le fichier `linux/net/socket.c` : Code Linux 2.6.10

```
414 /**
415  *      sockfd_lookup   -       passe d'un numero de fichier a son emplacement de socket
416  *      @fd  : numero de fichier
417  *      @err : pointeur sur un renvoi de code d'erreur
418  *
419  *      Le numero de fichier passe en argument est verrouille et la socket a laquelle il
420  *      est lie est renvoyee. Si une erreur apparait, le pointeur err est surcharge
421  *      avec un code errno negatif et NULL est renvoye. La fonction verifie
422  *      a la fois les numeros non valides et le passage d'un numero qui n'est pas une socket.
423  *
424  *      En cas de succes, le pointeur de l'objet socket est renvoye.
425  */
426
427 struct socket *sockfd_lookup(int fd, int *err)
428 {
429         struct file *file;
430         struct inode *inode;
431         struct socket *sock;
432
433         if (!(file = fget(fd)))
434         {
435                 *err = -EBADF;
436                 return NULL;
```

```
437         }
438
439         inode = file->f_dentry->d_inode;
440         if (!inode->i_sock || !(sock = SOCKET_I(inode)))
441         {
442                 *err = -ENOTSOCK;
443                 fput(file);
444                 return NULL;
445         }
446
447         if (sock->file != file) {
448                 printk(KERN_ERR "socki_lookup: socket file changed!\n");
449                 sock->file = file;
450         }
451         return sock;
452 }
```

Autrement dit :

- On déclare un descripteur de fichier, un descripteur de nœud d'information et un descripteur de socket.
- On essaie d'instantier le descripteur de fichier avec celui associé au numéro de fichier passé en argument. Si on n'y parvient pas, on positionne le code de retour à l'opposé du code d'erreur `EBADF` et on renvoie `NULL`.
- On instantie le descripteur de nœud d'information avec celui associé au descripteur de fichier.
- On essaie d'instantier le descripteur de socket avec celui associé à ce descripteur de nœud d'information. Si le descripteur de fichier n'est pas du type socket ou si l'on ne parvient pas à instantier le descripteur de socket, on libère le descripteur de fichier et on positionne le code de retour avec l'opposé du code d'erreur `ENOTSOCK` et on renvoie `NULL`.
- Si le descripteur de fichier associé au descripteur de socket n'est pas le descripteur de fichier instantié ci-dessus, on affiche un message d'erreur noyau et on le modifie.
- On renvoie l'adresse du descripteur de socket.

28.2.2 Cas de IPv4

Nous avons vu au chapitre 27, à propos de `inet_dgram_ops` pour UDP, que la fonction spécifique à IPv4 d'arrêt de socket s'appelle `inet_shutdown()`. Cette fonction est définie dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
676 int inet_shutdown(struct socket *sock, int how)
677 {
678         struct sock *sk = sock->sk;
679         int err = 0;
680
681         /* On doit vraiment verifier pour s'assurer que
682          * la socket est une socket TCP . (POURQUOI AC...)
683          */
684         how++; /* transforme 0->1 a l'avantage de faire du bit 1 receptions et
685                               1->2 du bit 2 envois.
686                               2->3 */
687         if ((how & ~SHUTDOWN_MASK) || !how)     /* MAXINT->0 */
688                 return -EINVAL;
689
690         lock_sock(sk);
691         if (sock->state == SS_CONNECTING) {
692                 if ((1 << sk->sk_state) &
```

```
693                     (TCPF_SYN_SENT | TCPF_SYN_RECV | TCPF_CLOSE))
694                         sock->state = SS_DISCONNECTING;
695                 else
696                         sock->state = SS_CONNECTED;
697         }
698
699         switch (sk->sk_state) {
700         case TCP_CLOSE:
701                 err = -ENOTCONN;
702                 /* Force a reveiller les autres qui sont a l'ecoute, qui peuvent elire
703                    POLLHUP, meme par exemple pour les sockets UDP non connectees -- RR */
704         default:
705                 sk->sk_shutdown |= how;
706                 if (sk->sk_prot->shutdown)
707                         sk->sk_prot->shutdown(sk, how);
708                 break;
709
710         /* Les deux branches restantes sont des solutions temporaires pour le close()
711          * manquant dans un environnement multithread. Ce n'est _pas_ une bonne idee,
712          * mais nous n'avons pas le choix jusqu'a ce que close() soit repare au niveau VFS.
713          */
714         case TCP_LISTEN:
715                 if (!(how & RCV_SHUTDOWN))
716                         break;
717                 /* Passer a travers */
718         case TCP_SYN_SENT:
719                 err = sk->sk_prot->disconnect(sk, O_NONBLOCK);
720                 sock->state = err ? SS_DISCONNECTING : SS_UNCONNECTED;
721                 break;
722         }
723
724         /* Reveiller tout ce qui est assoupi pour l'election. */
725         sk->sk_state_change(sk);
726         release_sock(sk);
727         return err;
728 }
```

Autrement dit :

- On déclare un descripteur de couche transport, que l'on instantie avec celui associé au descripteur de socket passé en argument.
- Si la valeur de l'argument `how` n'est pas située dans le bon intervalle, on renvoie l'opposé du code d'erreur `EINVAL`.
- On verrouille le descripteur de couche transport.
- On change l'état du descripteur de socket si la socket était en train de se connecter.
- Si la socket était déjà fermée, on renvoie l'opposé du code d'erreur `ENOTCONN`.
- Si la socket avait envoyé un message de synchronisation pour se connecter, on se déconnecte grâce à la fonction spécifique à la couche transport et on change d'état.
- On change l'état du descripteur de couche de transport et on libère celui-ci.

28.2.3 Fonction de déconnexion spécifique à UDP

Nous avons vu que, dans le cas de IPv4/UDP, l'ensemble des opérations au niveau de la couche de transport s'appelle `udp_prot`. Cet ensemble est défini dans le fichier `linux/net/ipv4/udp.c`: Code Linux 2.6.10

```
1353 struct proto udp_prot = {
1354          .name =         "UDP",
1355          .owner =        THIS_MODULE,
```

```
1356         .close =        udp_close,
1357         .connect =      ip4_datagram_connect,
1358         .disconnect =   udp_disconnect,
1359         .ioctl =        udp_ioctl,
1360         .destroy =      udp_destroy_sock,
1361         .setsockopt =   udp_setsockopt,
1362         .getsockopt =   udp_getsockopt,
1363         .sendmsg =      udp_sendmsg,
1364         .recvmsg =      udp_recvmsg,
1365         .sendpage =     udp_sendpage,
1366         .backlog_rcv =  udp_queue_rcv_skb,
1367         .hash =         udp_v4_hash,
1368         .unhash =       udp_v4_unhash,
1369         .get_port =     udp_v4_get_port,
1370         .slab_obj_size = sizeof(struct udp_sock),
1371 };
```

Cette structure `udp_prot` indique que la fonction de déconnexion spécifique à UDP s'appelle `udp_disconnect()` et qu'il n'y a pas de fonction spécifique `shutdown()`.

28.2.3.1 Fonction de déconnexion

Code Linux 2.6.10

La fonction `udp_disconnect()` de déconnexion spécifique à UDP est définie dans le fichier `linux/net/ipv4/udp.c`:

```
865 int udp_disconnect(struct sock *sk, int flags)
866 {
867         struct inet_opt *inet = inet_sk(sk);
868         /*
869          *      1003.1g - briser l'association.
870          */
871
872         sk->sk_state = TCP_CLOSE;
873         inet->daddr = 0;
874         inet->dport = 0;
875         sk->sk_bound_dev_if = 0;
876         if (!(sk->sk_userlocks & SOCK_BINDADDR_LOCK))
877                 inet_reset_saddr(sk);
878
879         if (!(sk->sk_userlocks & SOCK_BINDPORT_LOCK)) {
880                 sk->sk_prot->unhash(sk);
881                 inet->sport = 0;
882         }
883         sk_dst_reset(sk);
884         return 0;
885 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de couche de transport passé en argument.
- On indique, grâce à son descripteur de couche de transport, que l'état de la socket est fermé.
- On positionne l'adresse et le numéro de port de destination de l'option Internet à zéro.
- On positionne le numéro d'interface physique du descripteur de couche de transport à zéro.
- Si on n'est pas verrouillé en utilisation, on met à zéro l'adresse et le port source de l'option Internet.

Les drapeaux verrou SOCK_BINDADDR_LOCK et SOCK_BINDPORT_LOCK sont définis dans le fichier linux/include/net/sock.h: Code Linux 2.6.10

```
615 #define SOCK_SNDBUF_LOCK        1
616 #define SOCK_RCVBUF_LOCK        2
617 #define SOCK_BINDADDR_LOCK      4
618 #define SOCK_BINDPORT_LOCK      8
```

La fonction en ligne inet_reset_saddr() est définie dans le fichier linux/include/net/ip.h: Code Linux 2.6.10

```
236 static __inline__ void inet_reset_saddr(struct sock *sk)
237 {
238         inet_sk(sk)->rcv_saddr = inet_sk(sk)->saddr = 0;
239 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
240         if (sk->sk_family == PF_INET6) {
241                 struct ipv6_pinfo *np = inet6_sk(sk);
242
243                 memset(&np->saddr, 0, sizeof(np->saddr));
244                 memset(&np->rcv_saddr, 0, sizeof(np->rcv_saddr));
245         }
246 #endif
247 }
```

La fonction unhash() spécifique à UDP s'appelle udp_v4_unhash() comme le montre udp_prot. Cette fonction est définie dans le fichier linux/net/ipv4/udp.c: Code Linux 2.6.10

```
209 static void udp_v4_unhash(struct sock *sk)
210 {
211         write_lock_bh(&udp_hash_lock);
212         if (sk_del_node_init(sk)) {
213                 inet_sk(sk)->num = 0;
214                 sock_prot_dec_use(sk->sk_prot);
215         }
216         write_unlock_bh(&udp_hash_lock);
217 }
```

– On réinitialise le cache de destination, grâce à la fonction sk_dst_reset() étudiée ci-dessous, et on renvoie 0.

28.2.3.2 Réinitialisation de l'entrée de cache de destination

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

```
968 static inline void
969 __sk_dst_reset(struct sock *sk)
970 {
971         struct dst_entry *old_dst;
972
973         old_dst = sk->sk_dst_cache;
974         sk->sk_dst_cache = NULL;
975         dst_release(old_dst);
976 }
977
978 static inline void
979 sk_dst_reset(struct sock *sk)
980 {
981         write_lock(&sk->sk_dst_lock);
982         __sk_dst_reset(sk);
983         write_unlock(&sk->sk_dst_lock);
984 }
```

Autrement dit :

- on verrouille le champ entrée de cache de destination du descripteur de couche de transport en écriture, pour que personne d'autre ne puisse y accéder ;
- on fait appel à la fonction `__sk_dst_reset()` ; celle-ci met à `NULL` le champ entrée de cache de destination et libère l'ancienne entrée de cache ;
- on déverrouille le champ entrée de cache en écriture, pour qu'on puisse y accéder à nouveau.

28.3 Fermeture d'une socket

Code Linux 2.6.10

La fonction d'appel générale `sys_close()` est définie dans le fichier `linux/fs/open.c` :

```
1018 /*
1019  * Soyez soigneux ici ! Nous testons si le pointeur de fichier est NULL avant
1020  * de liberer le fd. Ceci nous assure qu'une tache clone ne peut pas liberer
1021  * un fd pendant qu'un autre clone est en train de l'ouvrir.
1022  */
1023 asmlinkage long sys_close(unsigned int fd)
1024 {
1025         struct file * filp;
1026         struct files_struct *files = current->files;
1027
1028         spin_lock(&files->file_lock);
1029         if (fd >= files->max_fds)
1030                 goto out_unlock;
1031         filp = files->fd[fd];
1032         if (!filp)
1033                 goto out_unlock;
1034         files->fd[fd] = NULL;
1035         FD_CLR(fd, files->close_on_exec);
1036         __put_unused_fd(files, fd);
1037         spin_unlock(&files->file_lock);
1038         return filp_close(filp, files);
1039
1040 out_unlock:
1041         spin_unlock(&files->file_lock);
1042         return -EBADF;
1043 }
```

Autrement dit :

- On déclare une adresse de descripteur de fichier.
- On déclare une table des fichiers ouverts, que l'on instantie avec celle associée au processus en cours.

  Le type général `struct files_struct` est défini dans le fichier `linux/include/linux/file.h` :

  Code Linux 2.6.10

```
19 /*
20  * Structure table des fichiers ouverts
21  */
22 struct files_struct {
23         atomic_t count;
24         spinlock_t file_lock;     /* Protege tous les membres ci-dessous. Se niche
                                       dans tsk->alloc_lock */
25         int max_fds;
26         int max_fdset;
27         int next_fd;
28         struct file ** fd;        /* tableau des fd en cours */
29         fd_set *close_on_exec;
```

```
30         fd_set *open_fds;
31         fd_set close_on_exec_init;
32         fd_set open_fds_init;
33         struct file * fd_array[NR_OPEN_DEFAULT];
34 };
```

- On verrouille la table des fichiers ouverts.
- Si le numéro de fichier passé en argument est supérieur au nombre maximum de fichiers ouverts, on déverrouille la table et on renvoie l'opposé du code d'erreur `EBADF`.
- On essaie d'instantier le descripteur de fichier avec celui associé au numéro de fichier passé en argument. Si on n'y parvient pas, on déverrouille la table et on renvoie l'opposé du code d'erreur `EBADF`.
- On indique dans la table que le descripteur de fichier associé au numéro de fichier est nul.
- On indique qu'on n'aura plus à fermer le fichier associé au numéro à la fin de l'exécution du processus en cours.
- On déverrouille la table des fichiers ouverts.
- On fait appel à la fonction `filp_close()` et on renvoie le code d'erreur fourni.
  Cette fonction générale est définie un peu plus haut dans le même fichier : Code Linux 2.6.10

```
986  /*
987   * "id" est l'ID du thread POSIX. Nous utilisons le
988   * pointeur de fichiers a la place...
989   */
990  int filp_close(struct file *filp, fl_owner_t id)
991  {
992          int retval;
993
994          /* Reporter et effacer les erreurs exceptionnelles */
995          retval = filp->f_error;
996          if (retval)
997                  filp->f_error = 0;
998
999          if (!file_count(filp)) {
1000                 printk(KERN_ERR "VFS: Close: file count is 0\n");
1001                 return retval;
1002         }
1003
1004         if (filp->f_op && filp->f_op->flush) {
1005                 int err = filp->f_op->flush(filp);
1006                 if (!retval)
1007                         retval = err;
1008         }
1009
1010         dnotify_flush(filp, id);
1011         locks_remove_posix(filp, id);
1012         fput(filp);
1013         return retval;
1014 }
```

Remarquons que rien de spécifique aux sockets n'intervient dans le cas de la fermeture.