Chapitre 27

Implémentation des fichiers de type socket

Nous avons vu au chapitre 6 que la programmation des réseaux se fait à travers un type de fichiers particulier, appelés sockets. Nous allons étudier l'implémentation de ce type de fichiers dans ce chapitre.

De même que les chapitres 13 et 24 portent plus sur la gestion de la mémoire que sur le sous-système réseau, ce chapitre porte sur la gestion des fichiers.

27.1 Implémentation générale des fichiers

Comme nous venons de le rappeler, sous UNIX tout est fichier, ce qui est donc également le cas des sockets, appartenant au type de fichiers qui nous intéresse. Commençons par donner une vue générale de l'implémentation des fichiers sous Linux.

27.1.1 Système de fichiers virtuel

Une des clés du succès de Linux est sa capacité à coexister aisément avec des systèmes de fichiers autres qui ceux qui sont natifs pour lui. Il est possible de monter, en toute transparence, des disques ou des partitions qui hébergent des formats de fichiers utilisés par Windows, d'autres systèmes UNIX ou des systèmes à faible part de marché comme Amiga.

27.1.1.1 Modèle de systèmes de fichiers commun

Linux gère le support de ces types multiples types de systèmes de fichiers au moyen d'un concept appelé **système de fichiers virtuel** (ou **VFS** pour l'anglais *Virtual File Sytem*), introduit par Kleiman en 1986 ([KLE–86]), avec une implémentation qui lui est propre.

Le concept majeur du `VFS` consiste à présenter un *modèle de fichier commun* capable de représenter tous les systèmes de fichiers pris en charge. L'idée du système de fichiers virtuel est que les entités internes représentant les fichiers et les systèmes de fichiers, situées dans la mémoire du noyau, renferment une vaste gamme d'informations de telle façon que toute opération fournie par un système de fichiers réel (compatible avec Linux) soit prise en charge par un champ du système virtuel. Pour toute fonction de lecture, d'écriture ou autre qui est appelée, le noyau substitue la fonction réelle adéquate.

Chaque implémentation d'un système de fichiers spécifique devra traduire son organisation physique dans le modèle de fichier commun du `VFS`.

27.1.1.2 Implémentation orientée objet

On peut considérer que le modèle de fichier commun est orienté objet, où un *objet* est une structure logicielle qui définit à la fois une structure de données et les méthodes qui opèrent à son niveau. Pour des raisons d'efficacité, Linux n'est pas programmé à l'aide d'un langage orienté objet tel que C++. Les objets sont implémentés comme des structures de données dont certains champs pointent sur les fonctions correspondant aux méthodes de l'objet.

27.1.1.3 Les composants du modèle de fichier commun

Le modèle de fichier commun se compose de quatre types d'objets, ceux-ci oncernant les fichiers (deux types), les répertoires et le disque :

Fichier. Les systèmes UNIX distinguent traditionnellement deux types de structures de données pour un fichier :

Nœud d'information et descripteur. Un nœud d'information (*inode* en anglais) sur le disque enregistre les informations générales sur un fichier donné (telles que le propriétaire du fichier et les droits d'accès). À ce nœud d'information, situé sur le disque pour les systèmes de fichiers natifs, correspond un **descripteur de nœud d'information**, situé en mémoire vive dans l'espace noyau, pour chaque fichier utilisé dans le système.

Descripteur de fichier. Les informations sur l'interaction entre un fichier ouvert et un processus sont stockées dans un descripteur de fichier [1]. Il s'agit des attributs du fichier, tel que le mode dans lequel le fichier peut être utilisé (lecture, écriture, lecture-écriture), ou la position en cours de la prochaine opération d'entrées-sorties. Ces informations existent seulement dans la mémoire du noyau et au cours de la période durant laquelle un processus accède à un fichier.

Entrée de répertoire. Un tel objet (en anglais *dentry* pour *Directory ENTRY*) stocke des informations sur la correspondance entre une entrée de répertoire et le fichier correspondant. Chaque système de fichiers sur disque enregistre ces informations sur un disque selon sa manière propre.

Super-bloc. Un tel objet enregistre des informations concernant un système de fichier monté, normalement une partition de disque dur ou un CD-ROM mais également une socket comme nous allons le voir.

Nous allons maintenant décrire les structures de données utilisées par Linux pour ces quatre types d'objets.

27.1.2 Super-bloc

27.1.2.1 Descripteur

Les descripteurs de super-bloc sont des entités du type `struct super_block`, défini dans le fichier `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
754 struct super_block {
755         struct list_head        s_list;         /* Garder ceci en premier */
756         dev_t                   s_dev;          /* index de recherche ; _pas_ kdev_t */
757         unsigned long           s_blocksize;
758         unsigned long           s_old_blocksize;
759         unsigned char           s_blocksize_bits;
760         unsigned char           s_dirt;
761         unsigned long long      s_maxbytes;     /* Taille max du fichier */
762         struct file_system_type *s_type;
763         struct super_operations *s_op;
764         struct dquot_operations *dq_op;
765         struct quotactl_ops     *s_qcop;
766         struct export_operations *s_export_op;
767         unsigned long           s_flags;
768         unsigned long           s_magic;
769         struct dentry           *s_root;
770         struct rw_semaphore     s_umount;
771         struct semaphore        s_lock;
772         int                     s_count;
773         int                     s_syncing;
774         int                     s_need_sync_fs;
775         atomic_t                s_active;
776         void                    *s_security;
777         struct xattr_handler    **s_xattr;
778
779         struct list_head        s_dirty;        /* noeud d'information sale */
780         struct list_head        s_io;           /* en reserve d'ecriture */
781         struct hlist_head       s_anon;         /* dentries anonymes a exporter (nfs) */
782         struct list_head        s_files;
783
784         struct block_device     *s_bdev;
```

1. Nous appellerons **numéro de fichier** ce qui est quelquefois appelé descripteur de fichier.

```
785         struct list_head        s_instances;
786         struct quota_info       s_dquot;        /* Options relatives au quota du disque */
787
788         int                     s_frozen;
789         wait_queue_head_t       s_wait_unfrozen;
790
791         char s_id[32];                          /* Nom donnant de l'information */
792
793         void                    *s_fs_info;     /* Infos privees du systeme de fichiers */
794
795         /*
796          * Le champ suivant interesse *exclusivement* VFS. Aucun systeme de fichiers n'a
797          * la possibilite de le verrouiller. Vous etes avertis.
798          */
799         struct semaphore s_vfs_rename_sem;      /* Kludge */
800 };
```

Nous nous permettons de renvoyer à notre livre [CEG–03] pour une description détaillée des champs dans le cas, plus simple, du noyau 0.01.

27.1.2.2 Opérations sur les super-blocs

Code Linux 2.6.10

Les opérations permises sur un super-bloc sont précisées par le champ de la ligne 763, à savoir `s_op` de type `super_operations`. Ce dernier est défini dans le même fichier en-tête :

```
969 /*
970  * NOTE : write_inode, delete_inode, clear_inode, put_inode peuvent etre appeles
971  * sans que le verrou du noyau ne soit detenu dans certains systemes de fichiers.
972  */
973 struct super_operations {
974         struct inode *(*alloc_inode)(struct super_block *sb);
975         void (*destroy_inode)(struct inode *);
976
977         void (*read_inode) (struct inode *);
978
979         void (*dirty_inode) (struct inode *);
980         int (*write_inode) (struct inode *, int);
981         void (*put_inode) (struct inode *);
982         void (*drop_inode) (struct inode *);
983         void (*delete_inode) (struct inode *);
984         void (*put_super) (struct super_block *);
985         void (*write_super) (struct super_block *);
986         int (*sync_fs)(struct super_block *sb, int wait);
987         void (*write_super_lockfs) (struct super_block *);
988         void (*unlockfs) (struct super_block *);
989         int (*statfs) (struct super_block *, struct kstatfs *);
990         int (*remount_fs) (struct super_block *, int *, char *);
991         void (*clear_inode) (struct inode *);
992         void (*umount_begin) (struct super_block *);
993
994         int (*show_options)(struct seq_file *, struct vfsmount *);
995 };
```

27.1.2.3 Opération sur le statut

La fonction `statfs()`, l'une des trois seules qui nous intéresseront dans le cas des sockets comme nous le verrons, doit fournir le statut du système de fichiers. Celui-ci est décrit dans une entité du type `struct statfs`. Ce type dépend de l'architecture du microprocesseur. Il est défini, en particulier pour les microprocesseurs Intel, dans le fichier en-tête `linux/include/asm-generic/statfs.h`:

Code Linux 2.6.10

```
1  #ifndef _GENERIC_STATFS_H
2  #define _GENERIC_STATFS_H
3
4  #ifndef __KERNEL_STRICT_NAMES
5  # include <linux/types.h>
6  typedef __kernel_fsid_t fsid_t;
7  #endif
8
9  struct statfs {
10         __u32 f_type;
11         __u32 f_bsize;
12         __u32 f_blocks;
13         __u32 f_bfree;
14         __u32 f_bavail;
15         __u32 f_files;
16         __u32 f_ffree;
17         __kernel_fsid_t f_fsid;
18         __u32 f_namelen;
19         __u32 f_frsize;
20         __u32 f_spare[5];
21 };
```

27.1.3 Nœud d'information

27.1.3.1 Descripteur

Les descripteurs de nœud d'information ont une structure, appelée `inode`, définie dans le fichier `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
429 struct inode {
430         struct hlist_node       i_hash;
431         struct list_head        i_list;
432         struct list_head        i_dentry;
433         unsigned long           i_ino;
434         atomic_t                i_count;
435         umode_t                 i_mode;
436         unsigned int            i_nlink;
437         uid_t                   i_uid;
438         gid_t                   i_gid;
439         dev_t                   i_rdev;
440         loff_t                  i_size;
441         struct timespec         i_atime;
442         struct timespec         i_mtime;
443         struct timespec         i_ctime;
444         unsigned int            i_blkbits;
445         unsigned long           i_blksize;
446         unsigned long           i_version;
447         unsigned long           i_blocks;
448         unsigned short          i_bytes;
449         unsigned char           i_sock;
450         spinlock_t              i_lock; /* i_blocks, i_bytes, peut-etre i_size */
451         struct semaphore        i_sem;
452         struct rw_semaphore     i_alloc_sem;
```

```
453         struct inode_operations *i_op;
454         struct file_operations  *i_fop; /* precedemment ->i_op->default_file_ops */
455         struct super_block      *i_sb;
456         struct file_lock        *i_flock;
457         struct address_space    *i_mapping;
458         struct address_space    i_data;
459 #ifdef CONFIG_QUOTA
460         struct dquot            *i_dquot[MAXQUOTAS];
461 #endif
462         /* Ces trois champs devraient probablement former une union */
463         struct list_head        i_devices;
464         struct pipe_inode_info  *i_pipe;
465         struct block_device     *i_bdev;
466         struct cdev             *i_cdev;
467         int                     i_cindex;
468
469         __u32                   i_generation;
470
471 #ifdef CONFIG_DNOTIFY
472         unsigned long           i_dnotify_mask; /* Evenements de notification repertoire */
473         struct dnotify_struct   *i_dnotify;     /* pour les notifications repertoire     */
474 #endif
475
476         unsigned long           i_state;
477         unsigned long           dirtied_when;   /* heure exacte (en jiffies) de
                                                       la premiere salissure */
478
479         unsigned int            i_flags;
480
481         atomic_t                i_writecount;
482         void                    *i_security;
483         union {
484                 void            *generic_ip;
485         } u;
486 #ifdef __NEED_I_SIZE_ORDERED
487         seqcount_t              i_size_seqcount;
488 #endif
489 };
```

Nous renvoyons à notre livre [CEG–03] pour la description de la structure associée aux descripteurs de nœud d'information dans le cas, plus simple mais suffisant pour avoir une idée générale, du noyau Linux 0.01.

Les deux champs qui nous intéresseront plus particulièrement sont le champ `i_fop` de la ligne 454 concernant les opérations possibles sur le type de fichier et le champ `u` de la ligne 485, spécifique aux sockets.

27.1.3.2 Opérations

Code Linux 2.6.10

Le type `struct file_operations` du champ `i_fop` est défini dans le fichier en-tête `linux/-include/linux/fs.h`:

```
903 /*
904  * NOTE :
905  * read, write, poll, fsync, readv, writev peuvent etre appeles
906  *   sans que le verrou noyau ne soit detenu pour certains systemes de fichiers.
907  */
908 struct file_operations {
909         struct module *owner;
910         loff_t (*llseek) (struct file *, loff_t, int);
911         ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
912         ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
```

```
913         ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
914         ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
915         int (*readdir) (struct file *, void *, filldir_t);
916         unsigned int (*poll) (struct file *, struct poll_table_struct *);
917         int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
918         int (*mmap) (struct file *, struct vm_area_struct *);
919         int (*open) (struct inode *, struct file *);
920         int (*flush) (struct file *);
921         int (*release) (struct inode *, struct file *);
922         int (*fsync) (struct file *, struct dentry *, int datasync);
923         int (*aio_fsync) (struct kiocb *, int datasync);
924         int (*fasync) (int, struct file *, int);
925         int (*lock) (struct file *, int, struct file_lock *);
926         ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
927         ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
928         ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
929         ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
930         unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                                               unsigned long, unsigned long);
931         int (*check_flags)(int);
932         int (*dir_notify)(struct file *filp, unsigned long arg);
933         int (*flock) (struct file *, int, struct file_lock *);
934 };
```

27.1.4 Descripteur de fichier

Les descripteurs de fichier ont une structure, appelée `file`, définie dans le fichier `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
577 struct file {
578         struct list_head        f_list;
579         struct dentry           *f_dentry;
580         struct vfsmount         *f_vfsmnt;
581         struct file_operations  *f_op;
582         atomic_t                f_count;
583         unsigned int            f_flags;
584         mode_t                  f_mode;
585         int                     f_error;
586         loff_t                  f_pos;
587         struct fown_struct      f_owner;
588         unsigned int            f_uid, f_gid;
589         struct file_ra_state    f_ra;
590
591         unsigned long           f_version;
592         void                    *f_security;
593
594         /* necessaire pour les peripheriques tty et peut-etre d'autres */
595         void                    *private_data;
596
597 #ifdef CONFIG_EPOLL
598         /* Utilise par fs/eventpoll.c pour lier toutes les ancres a ce fichier */
599         struct list_head        f_ep_links;
600         spinlock_t              f_ep_lock;
601 #endif /* #ifdef CONFIG_EPOLL */
602         struct address_space    *f_mapping;
603 };
```

Là encore nous renvoyons à notre livre [CEG–03] pour une étude de cette structure dans le cas plus simple du noyau Linux 0.01. Le seul champ qui nous intéressera, dans le cas des sockets, est celui de la ligne 581, à savoir `f_op` concernant sur les opérations permises et dont le type a été défini ci-dessus.

27.1.5 Répertoire

27.1.5.1 Descripteur

Code Linux 2.6.10

Les descripteurs d'entrées de répertoire sont des entités du type `struct dentry` défini dans le fichier en-tête `linux/include/linux/dcache.h`:

```
83  struct dentry {
84          atomic_t d_count;
85          unsigned int d_flags;           /* protege par d_lock */
86          spinlock_t d_lock;              /* un verrou par dentry */
87          struct inode *d_inode;          /* La a qui le nom appartient - NULL est
88                                           * negatif */
89          /*
90           * Les trois champs suivants sont affectes par __d_lookup. Les placer ici
91           * pour qu'ils soient tous dans une amplitude de 16 octets, avec un alignement
               de 16 octets.
92           */
93          struct dentry *d_parent;        /* repertoire parent */
94          struct qstr d_name;
95
96          struct list_head d_lru;         /* Liste LRU */
97          struct list_head d_child;       /* fils de la liste parent */
98          struct list_head d_subdirs;     /* nos enfants */
99          struct list_head d_alias;       /* liste d'alias des inode */
100         unsigned long d_time;           /* utilise par d_revalidate */
101         struct dentry_operations *d_op;
102         struct super_block *d_sb;       /* La racine de l'arbre du dentry */
103         void *d_fsdata;                 /* donnees specifiques au fs */
104         struct rcu_head d_rcu;
105         struct dcookie_struct *d_cookie; /* cafteur, s'il y en a */
106         struct hlist_node d_hash;       /* liste de hachage de consultation */
107         int d_mounted;
108         unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* petits noms */
109 };
```

27.1.5.2 Opérations

Code Linux 2.6.10

Les fonctions permises sur les entrées de répertoire sont définies par le champ `d_op` de la ligne 101, du type `struct dentry_operations`. Celui-ci est défini dans le même fichier en-tête:

```
111 struct dentry_operations {
112         int (*d_revalidate)(struct dentry *, struct nameidata *);
113         int (*d_hash) (struct dentry *, struct qstr *);
114         int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
115         int (*d_delete)(struct dentry *);
116         void (*d_release)(struct dentry *);
117         void (*d_iput)(struct dentry *, struct inode *);
118 };
```

27.1.6 Types de fichiers

Code Linux 2.6.10

Les types de fichiers acceptés par Linux sont définis dans le fichier `linux/include/linux/-fs.h`:

```
844 /*
845  * Types de fichiers
846  *
847  * NOTE : Ceci correspond aux bits 12..15 de stat.st_mode
848  * (par exemple "(i_mode >> 12) & 15").
849  */
```

```
850 #define DT_UNKNOWN      0
851 #define DT_FIFO         1
852 #define DT_CHR          2
853 #define DT_DIR          4
854 #define DT_BLK          6
855 #define DT_REG          8
856 #define DT_LNK          10
857 #define DT_SOCK         12
858 #define DT_WHT          14
```

à savoir les types inconnu, tube de communication, périphérique caractère, répertoire, périphérique bloc, régulier, lien, socket et blanc. Seul le type socket nous intéressera dans ce livre.

27.1.7 Déclaration d'un système de fichiers

Un système de fichier est caractérisé par une entité du type `struct file_system_type`, défini dans le fichier en-tête `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
1137 struct file_system_type {
1138         const char *name;
1139         int fs_flags;
1140         struct super_block *(*get_sb) (struct file_system_type *, int,
1141                                        const char *, void *);
1142         void (*kill_sb) (struct super_block *);
1143         struct module *owner;
1144         struct file_system_type * next;
1145         struct list_head fs_supers;
1146 };
```

qui spécifie en particulier le nom du système de fichiers et les fonctions permettant d'obtenir le super-bloc d'une instance de celui-ci et de libérer celle-ci.

27.1.8 Enregistrement d'un système de fichiers

L'enregistrement d'un (type de) système(s) de fichiers est effectué grâce à la fonction `register_filesystem`, définie dans le fichier `linux/fs/filesystems.c`:

Code Linux 2.6.10

```
53 /**
54  *      register_filesystem - enregistre un nouveau systeme de fichiers
55  *      @fs : la structure systeme de fichiers
56  *
57  *      Ajoute le systeme de fichiers passe a la liste des systemes de fichiers dont le noyau
58  *      est averti pour le montage et autres appels systeme. Renvoie 0 en cas de succes
59  *      ou un code errno negatif en cas d'erreur.
60  *
61  *      La &struct file_system_type qui est passee est liee aux structures du
62  *      noyau et ne doit pas etre liberee avant que le systeme de fichiers n'ait ete
63  *      retire.
64  */
65
66 int register_filesystem(struct file_system_type * fs)
67 {
68         int res = 0;
69         struct file_system_type ** p;
70
71         if (!fs)
72                 return -EINVAL;
73         if (fs->next)
74                 return -EBUSY;
75         INIT_LIST_HEAD(&fs->fs_supers);
76         write_lock(&file_systems_lock);
```

```
77         p = find_filesystem(fs->name);
78         if (*p)
79                 res = -EBUSY;
80         else
81                 *p = fs;
82         write_unlock(&file_systems_lock);
83         return res;
84 }
```

27.2 Descripteur de socket

Linux place les informations générales sur une socket, celles qui sont indépendantes de tout protocole, dans un **descripteur de socket**, entité du type `struct socket`.

27.2.1 Définition du type

Code Linux 2.6.10

Le type `struct socket` est défini dans le fichier `linux/include/linux/net.h`:

```
93  /**
94   *  struct socket - socket BSD generale
95   *  @state - etat de la socket (%SS_CONNECTED, etc)
96   *  @flags - drapeaux de la socket (%SOCK_ASYNC_NOSPACE, etc)
97   *  @ops - operations sur la socket specifiques au protocole
98   *  @fasync_list - liste d'eveil asynchrone
99   *  @file - pointeur de retour au fichier pour le gc
100  *  @sk - representation de la socket agnostique au protocole de reseau interne
101  *  @wait - attend une file d'attente pour plusieurs usages
102  *  @type - type de socket (%SOCK_STREAM, etc)
103  *  @passcred - pieces d'identite (utilisees seulement pour les sockets Unix (ou PF_LOCAL))
104  */
105 struct socket {
106         socket_state            state;
107         unsigned long           flags;
108         struct proto_ops        *ops;
109         struct fasync_struct    *fasync_list;
110         struct file             *file;
111         struct sock             *sk;
112         wait_queue_head_t       wait;
113         short                   type;
114         unsigned char           passcred;
115 };
```

Décrivons rapidement chacun des champs avant de revenir plus en détail sur certains d'entre eux :

- l'état de la socket, sur lequel nous allons revenir, est précisé par le champ `state` ;
- un vecteur de drapeaux `flags`, sur lequel nous allons également revenir ;
- les opérations `ops` permises par la famille de protocoles utilisée, que nous détaillerons ci-dessous ;
- la liste `fasync_list` des fichiers à réveiller, le type général `struct fasync_struct` étant défini dans le fichier `linux/include/linux/fs.h` ;
- le descripteur de fichier `file` associé, ce qui sert pour le ramasse-miettes ; il s'agit d'une référence inverse puisque c'est normalement le descripteur de fichier qui fait référence au descripteur de socket ;
- un descripteur de couche transport `sk` ;
- la file d'attente `wait` associée à la socket ;

- le type de communication `type` tel qu'il est passé en argument lors de la création d'une socket ;
- les pièces d'identité `passcred` de la socket, utiles lors de l'utilisation de données ancillaires.

27.2.2 Les états d'une socket

Le type `socket_state`, défini dans le fichier en-tête `linux/include/linux/net.h`: Code Linux 2.6.10

```
49 typedef enum {
50         SS_FREE = 0,                    /* non allouee                     */
51         SS_UNCONNECTED,                 /* non connectee a une autre socket */
52         SS_CONNECTING,                  /* en train d'etre connectee       */
53         SS_CONNECTED,                   /* connectee a une autre socket    */
54         SS_DISCONNECTING                /* en train d'etre deconnectee     */
55 } socket_state;
```

précise les états possibles :

- `SS_FREE` : descripteur de socket non alloué ;
- `SS_UNCONNECTED` : socket non connectée à une autre socket ;
- `SS_CONNECTING` : socket en train de se connecter à une autre socket ;
- `SS_CONNECTED` : socket connectée à une autre socket ;
- `SS_DISCONNECTING` : socket en train de se déconnecter.

27.2.3 Les drapeaux

Le trois valeurs possibles des drapeaux sont définies dans le fichier en-tête `linux/include/linux/net.h`: Code Linux 2.6.10

```
61 #define SOCK_ASYNC_NOSPACE      0
62 #define SOCK_ASYNC_WAITDATA     1
63 #define SOCK_NOSPACE            2
```

27.2.4 Le type opérations sur une socket

Le type `struct proto_ops` est défini dans le fichier en-tête `linux/include/linux/net.h`: Code Linux 2.6.10

```
124 struct proto_ops {
125         int             family;
126         struct module   *owner;
127         int             (*release)   (struct socket *sock);
128         int             (*bind)      (struct socket *sock,
129                                       struct sockaddr *myaddr,
130                                       int sockaddr_len);
131         int             (*connect)   (struct socket *sock,
132                                       struct sockaddr *vaddr,
133                                       int sockaddr_len, int flags);
134         int             (*socketpair)(struct socket *sock1,
135                                       struct socket *sock2);
136         int             (*accept)    (struct socket *sock,
137                                       struct socket *newsock, int flags);
138         int             (*getname)   (struct socket *sock,
139                                       struct sockaddr *addr,
140                                       int *sockaddr_len, int peer);
141         unsigned int    (*poll)      (struct file *file, struct socket *sock,
142                                       struct poll_table_struct *wait);
143         int             (*ioctl)     (struct socket *sock, unsigned int cmd,
144                                       unsigned long arg);
```

```
145         int             (*listen)    (struct socket *sock, int len);
146         int             (*shutdown)  (struct socket *sock, int flags);
147         int             (*setsockopt)(struct socket *sock, int level,
148                                       int optname, char __user *optval, int optlen);
149         int             (*getsockopt)(struct socket *sock, int level,
150                                       int optname, char __user *optval, int __user *optlen);
151         int             (*sendmsg)   (struct kiocb *iocb, struct socket *sock,
152                                       struct msghdr *m, size_t total_len);
153         int             (*recvmsg)   (struct kiocb *iocb, struct socket *sock,
154                                       struct msghdr *m, size_t total_len,
155                                       int flags);
156         int             (*mmap)      (struct file *file, struct socket *sock,
157                                       struct vm_area_struct * vma);
158         ssize_t         (*sendpage)  (struct socket *sock, struct page *page,
159                                       int offset, size_t size, int flags);
160 };
```

On reconnaît les noms des appels système concernant les sockets.

27.3 Déclaration du système de fichiers des sockets

27.3.1 La structure

Code Linux 2.6.10

Le nom, `sockfs`, du système de fichiers des sockets et le type associé, `sock_fs_type`, sont définis dans le fichier `linux/net/socket.c`:

```
335 static struct file_system_type sock_fs_type = {
336         .name =         "sockfs",
337         .get_sb =       sockfs_get_sb,
338         .kill_sb =      kill_anon_super,
339 };
```

Les fonctions d'obtention et de libération d'un super-bloc d'un système de fichiers de sockets s'appellent donc `sockfs_get_sb()` et `kill_anon_super()`. Nous allons les décrire dans les sous-sections suivantes.

Code Linux 2.6.10

Le nombre magique `SOCKFS_MAGIC` associé à ce système de fichiers est défini dans le même fichier :

```
272 #define SOCKFS_MAGIC 0x534F434B
```

27.3.2 Enregistrement

Code Linux 2.6.10

L'enregistrement du sytème de fichiers des sockets est effectué ligne 2061 de la définition de la fonction `sock_init()`, déjà rencontrée au chapitre 26 :

```
2032 void __init sock_init(void)
2033 {
[...]
2056         /*
2057          *      Initialise le modules des protocoles.
2058          */
2059
2060         init_inodecache();
2061         register_filesystem(&sock_fs_type);
2062         sock_mnt = kern_mount(&sock_fs_type);
2063         /* L'initialisation reelle des protocoles est effectuee lorsque
2064          *  do_initcalls est en action.
2065          */
[...]
2070 }
```

27.4 Système de fichiers de sockets

Une socket est un système de fichiers de sockets à elle toute seule. Un système de fichiers de sockets possède donc un super-bloc dont peu de champs sont significatifs et un seul répertoire (outre les deux fichiers point et point-point obligatoires) contenant un seul fichier.

La structure `sock_fs_type` définie ci-dessus précise le nom des deux fonctions d'obtention et de libération d'un super-bloc dans le cas des sockets.

27.4.1 Obtention du super-bloc

27.4.1.1 Fonction spécifique

Nous venons de voir que la fonction de lecture du super-bloc d'un système de fichiers de type socket s'appelle `sockfs_get_sb()`. Elle est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
327 static struct super_block *sockfs_get_sb(struct file_system_type *fs_type,
328         int flags, const char *dev_name, void *data)
329 {
330         return get_sb_pseudo(fs_type, "socket:", &sockfs_ops, SOCKFS_MAGIC);
331 }
```

et renvoie à la fonction `get_sb_pseudo()` qui concerne les pseudo-systèmes de fichiers, ceux qui ne sont pas en mémoire de masse.

On remarquera que les trois derniers paramètres ne sont pas pris en compte (ils ont un intérêt pour les systèmes de fichiers situés en mémoire de masse mais pas pour celui des sockets). Le premier paramètre est nécessairement `sock_fs_type`, aussi cette fonction ne possède-t-elle aucun paramètre utile. C'est tout à fait normal pour une socket, le système de fichiers étant plutôt une abstraction.

27.4.1.2 Cas des pseudo-systèmes de fichiers

La fonction générale `get_sb_pseudo()` est définie dans le fichier `linux/fs/libfs.c`: Code Linux 2.6.10

```
192 /*
193  * Routine d'aide commune aux pseudo-systemes de fichiers (sockfs, pipefs, bdev - choses qui
194  * ne seront jamais montees)
195  */
196 struct super_block *
197 get_sb_pseudo(struct file_system_type *fs_type, char *name,
198         struct super_operations *ops, unsigned long magic)
199 {
200         struct super_block *s = sget(fs_type, NULL, set_anon_super, NULL);
201         static struct super_operations default_ops = {.statfs = simple_statfs};
202         struct dentry *dentry;
203         struct inode *root;
204         struct qstr d_name = {.name = name, .len = strlen(name)};
205
206         if (IS_ERR(s))
207                 return s;
208
209         s->s_flags = MS_NOUSER;
210         s->s_maxbytes = ~0ULL;
211         s->s_blocksize = 1024;
212         s->s_blocksize_bits = 10;
213         s->s_magic = magic;
214         s->s_op = ops ? ops : &default_ops;
215         root = new_inode(s);
216         if (!root)
217                 goto Enomem;
```

```
218         root->i_mode = S_IFDIR | S_IRUSR | S_IWUSR;
219         root->i_uid = root->i_gid = 0;
220         root->i_atime = root->i_mtime = root->i_ctime = CURRENT_TIME;
221         dentry = d_alloc(NULL, &d_name);
222         if (!dentry) {
223                 iput(root);
224                 goto Enomem;
225         }
226         dentry->d_sb = s;
227         dentry->d_parent = dentry;
228         d_instantiate(dentry, root);
229         s->s_root = dentry;
230         s->s_flags |= MS_ACTIVE;
231         return s;
232
233 Enomem:
234         up_write(&s->s_umount);
235         deactivate_super(s);
236         return ERR_PTR(-ENOMEM);
237 }
```

Autrement dit :

– Un descripteur de super-bloc est déclaré, que l'on essaie d'instantier. Si on n'y parvient pas, on renvoie le code fourni par la fonction `sget()`.

Code Linux 2.6.10

La fonction générale `sget()` est définie dans le fichier `linux/fs/super.c` :

```
264 /**
265  *      sget    -       trouver ou creer un super bloc
266  *      @type : type de systemes de fichiers auquel le super bloc doit appartenir
267  *      @test : rappel de comparaison
268  *      @set :  rappel d'initialisation
269  *      @data : argument pour chacun d'eux
270  */
271 struct super_block *sget(struct file_system_type *type,
272                         int (*test)(struct super_block *,void *),
273                         int (*set)(struct super_block *,void *),
274                         void *data)
275 {
276         struct super_block *s = NULL;
277         struct list_head *p;
278         int err;
279
280 retry:
281         spin_lock(&sb_lock);
282         if (test) list_for_each(p, &type->fs_supers) {
283                 struct super_block *old;
284                 old = list_entry(p, struct super_block, s_instances);
285                 if (!test(old, data))
286                         continue;
287                 if (!grab_super(old))
288                         goto retry;
289                 if (s)
290                         destroy_super(s);
291                 return old;
292         }
293         if (!s) {
294                 spin_unlock(&sb_lock);
295                 s = alloc_super();
296                 if (!s)
297                         return ERR_PTR(-ENOMEM);
298                 goto retry;
```

```
299         }
300
301         err = set(s, data);
302         if (err) {
303                 spin_unlock(&sb_lock);
304                 destroy_super(s);
305                 return ERR_PTR(err);
306         }
307         s->s_type = type;
308         strlcpy(s->s_id, type->name, sizeof(s->s_id));
309         list_add_tail(&s->s_list, &super_blocks);
310         list_add(&s->s_instances, &type->fs_supers);
311         spin_unlock(&sb_lock);
312         get_filesystem(type);
313         return s;
314 }
```

Nous ne décrirons pas en détail cette fonction puisqu'elle n'est pas spécifique au type de fichiers des sockets. Elle recherche un descripteur de super-bloc libre dans la liste des descripteurs du système de fichiers en question et elle en renvoie l'adresse, sinon elle en ajoute un et en renvoie l'adresse.

- On déclare un ensemble d'opérations par défaut sur les super-blocs, qui ne comprend que celle concernant le statut.
- On déclare une entrée de répertoire, un descripteur de nœud d'information (qui sera celui de la racine du système de fichiers monté) et un métanom.

  Le type général `struct qstr` des métanoms est défini dans le fichier `linux/include/linux/dcache.h`:

Code Linux 2.6.10

```
27 /*
28  * "quick string" -- facilite le passage des parametres, mais encore plus important,
29  * sauvegarde les "metadonnees" sur la chaine de caracteres (c'est-a-dire la longueur et
      le hachage).
30  *
31  * hash vient en premier, donc il est contre d_parent dans
32  * dentry.
33  */
34 struct qstr {
35         unsigned int hash;
36         unsigned int len;
37         const unsigned char *name;
38 };
```

- Si la valeur du pointeur de super-bloc n'est pas adéquate, on la renvoie.
- On initialise quelques champs du descripteur de super-bloc, en particulier la taille d'un bloc (1 024 octets), le nombre de bits pour cette taille (10 puisque $2^{10} = 1024$), le nombre magique de ce type de fichier (d'après la valeur passée en argument) et enfin l'ensemble des opérations sur les super-blocs de ce type de fichiers (d'après la valeur passée en argument ; `sockfs_ops` dans notre cas, étudié ci-dessous).
- On essaie d'instantier le descripteur de nœud d'information déclaré au début avec un descripteur qui n'est pas utilisé. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOMEM`. Sinon on indique le mode pour un nœud d'information de socket (pas de répertoire, lecture et écriture pour l'utilisateur), l'identificateur de son propriétaire et l'identificateur du groupe d'utilisateurs (tous les deux égaux à 0, donc se rapportant au super-utilisateur) ainsi que les différentes estampilles de temps (toutes initialisées à l'heure en cours).
- On essaie d'allouer une entrée de répertoire avec le nom passé en paramètre ("socket:" dans notre cas). Si on n'y parvient pas, on libère le descripteur de nœud d'information et on renvoie l'opposé du code d'erreur `ENOMEM`.

- On initialise ce répertoire avec les fichiers obligatoires point '.' et point-point '..'.
- On associe cette entrée de répertoire au descripteur de nœud d'information.
- On associe cette entrée de répertoire au descripteur de super-bloc et on renvoie l'adresse de celui-ci.

27.4.2 Libération d'un super-bloc

Nous avons vu ci-dessus que la fonction de libération du super-bloc d'un système de fichiers de type socket s'appelle `kill_anon_super()`. Cette fonction générale est définie dans le fichier `linux/fs/super.c`:

Code Linux 2.6.10

```
622 void kill_anon_super(struct super_block *sb)
623 {
624         int slot = MINOR(sb->s_dev);
625
626         generic_shutdown_super(sb);
627         spin_lock(&unnamed_dev_lock);
628         idr_remove(&unnamed_dev_idr, slot);
629         spin_unlock(&unnamed_dev_lock);
630 }
```

qui fait essentiellement appel à la fonction `generic_shutdown_super()` chargée de la partie de la libération indépendante du système de fichiers. Celle-ci est définie dans le même fichier:

Code Linux 2.6.10

```
213 /**
214  *      generic_shutdown_super  -       routine d'aide commune pour ->kill_sb()
215  *      @sb : super bloc a detruire
216  *
217  *      generic_shutdown_super() fait tout le travail independant du fs en fermeture de
218  *      super-bloc. Une ->kill_sb() typique devrait amasser tous les objets specifiques au
219  *      fs que necessite la destruction du super bloc, appeler generic_shutdown_super()
220  *      puis liberer les objets mentionnes ci-dessus. Note : on _doit_ prendre soin des
221  *      dentries et des inodes et ne pas avoir besoin de routine specifique.
222  */
223 void generic_shutdown_super(struct super_block *sb)
224 {
225         struct dentry *root = sb->s_root;
226         struct super_operations *sop = sb->s_op;
227
228         if (root) {
229                 sb->s_root = NULL;
230                 shrink_dcache_parent(root);
231                 shrink_dcache_anon(&sb->s_anon);
232                 dput(root);
233                 fsync_super(sb);
234                 lock_super(sb);
235                 lock_kernel();
236                 sb->s_flags &= ~MS_ACTIVE;
237                 /* mauvais nom - ce devrait etre evict_inodes() */
238                 invalidate_inodes(sb);
239
240                 if (sop->write_super && sb->s_dirt)
241                         sop->write_super(sb);
242                 if (sop->put_super)
243                         sop->put_super(sb);
244
245                 /* Oublier tous les inodes restants */
246                 if (invalidate_inodes(sb)) {
247                         printk("VFS: Busy inodes after unmount. "
248                            "Self-destruct in 5 seconds.  Have a nice day...\n");
249                 }
```

```
250
251                 unlock_kernel();
252                 unlock_super(sb);
253         }
254         spin_lock(&sb_lock);
255         /* devrait etre initialise pour __put_super_and_need_restart() */
256         list_del_init(&sb->s_list);
257         list_del(&sb->s_instances);
258         spin_unlock(&sb_lock);
259         up_write(&sb->s_umount);
260 }
```

27.5 Opérations sur les super-blocs

27.5.1 Ensemble des opérations

Nous avons vu que, pour obtenir un super-bloc de socket, on doit passer en argument l'ensemble des opérations sur les super-blocs. Celui-ci, `sockfs_ops`, est défini dans le fichier `linux/net/socket.c`:

Code Linux 2.6.10

```
321 static struct super_operations sockfs_ops = {
322         .alloc_inode =  sock_alloc_inode,
323         .destroy_inode =sock_destroy_inode,
324         .statfs =       simple_statfs,
325 };
```

Seules trois opérations sont donc nécessaires pour les super-blocs de socket sur les dix-huit prévues dans le cas général (comme vu ci-dessus) : celle qui alloue un descripteur de nœud d'information, celle qui le libère et celle qui renvoie le statut du système de fichiers.

27.5.2 Statut du système de fichiers

La fonction générale `simple_statfs()` est définie dans le fichier `linux/fs/libfs.c`:

Code Linux 2.6.10

```
21 int simple_statfs(struct super_block *sb, struct kstatfs *buf)
22 {
23         buf->f_type = sb->s_magic;
24         buf->f_bsize = PAGE_CACHE_SIZE;
25         buf->f_namelen = NAME_MAX;
26         return 0;
27 }
```

Autrement dit elle renseigne les champs nombre magique avec `SOCKFS_MAGIC`, taille d'un bloc avec 1 024 (octets) et nombre maximum de caractères avec 255 pour le nom d'une socket dans un tampon du type `struct kstatfs`.

Celui-ci est défini dans le fichier `linux/include/linux/statfs.h` que nous reproduisons intégralement :

Code Linux 2.6.10

```
1  #ifndef _LINUX_STATFS_H
2  #define _LINUX_STATFS_H
3
4  #include <linux/types.h>
5
6  #include <asm/statfs.h>
7
8  struct kstatfs {
9          long            f_type;
10         long            f_bsize;
11         sector_t        f_blocks;
```

```
12         sector_t        f_bfree;
13         sector_t        f_bavail;
14         sector_t        f_files;
15         sector_t        f_ffree;
16         __kernel_fsid_t f_fsid;
17         long            f_namelen;
18         long            f_frsize;
19         long            f_spare[5];
20 };
21
22 #endif
```

27.5.3 Allocation d'un descripteur de nœud d'information

Code Linux 2.6.10

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

```
276 static struct inode *sock_alloc_inode(struct super_block *sb)
277 {
278         struct socket_alloc *ei;
279         ei = (struct socket_alloc *)kmem_cache_alloc(sock_inode_cachep, SLAB_KERNEL);
280         if (!ei)
281                 return NULL;
282         init_waitqueue_head(&ei->socket.wait);
283
284         ei->socket.fasync_list = NULL;
285         ei->socket.state = SS_UNCONNECTED;
286         ei->socket.flags = 0;
287         ei->socket.ops = NULL;
288         ei->socket.sk = NULL;
289         ei->socket.file = NULL;
290         ei->socket.passcred = 0;
291
292         return &ei->vfs_inode;
293 }
```

Autrement dit:

- On déclare une entité du type `struct socket_alloc`.

  Code Linux 2.6.10

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

  ```
  644 struct socket_alloc {
  645         struct socket socket;
  646         struct inode vfs_inode;
  647 };
  ```

  et comprend deux champs: un descripteur de socket et un descripteur de nœud d'information.
- On essaie d'instantier cette entité en lui réservant de la place mémoire dans l'espace noyau. Si on n'y parvient pas, on renvoie `NULL`.

  Code Linux 2.6.10

  La variable `sock_inode_cachep` est définie dans le fichier `linux/net/sockect.c`:

  ```
  274 static kmem_cache_t * sock_inode_cachep;
  ```

- On renseigne les champs du descripteur de socket.
- On renvoie l'adresse du descripteur de nœud d'information.

Astuce

Remarquons que l'on utilise une astuce (est-ce bien venu?): le descripteur de socket associé à ce descripteur de nœud d'information est déterminé par son emplacement dans l'espace noyau (un nombre déterminé d'octets avant l'emplacement du descripteur de nœud d'information) au lieu d'un champ spécifique.

27.5.4 Libération d'un descripteur de nœud d'information

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

```
295 static void sock_destroy_inode(struct inode *inode)
296 {
297         kmem_cache_free(sock_inode_cachep,
298                         container_of(inode, struct socket_alloc, vfs_inode));
299 }
```

Il n'y a pas grand chose à en dire, sinon l'utilisation de l'astuce notée ci-dessus, mais maintenant à travers l'utilisation de la macro générale `container_of()`, définie dans le fichier `linux/include/linux/kernel.h`: Code Linux 2.6.10

```
232 /**
233  * container_of - convertit un membre d'une structure en dehors de la structure qui le
      contient
234  *
235  * @ptr :       le pointeur au membre.
236  * @type :      le type de la struct conteneur dans laquelle il est contenu.
237  * @member :    le nom du membre dans la struct.
238  *
239  */
240 #define container_of(ptr, type, member) ({                      \
241         const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
242         (type *)( (char *)__mptr - offsetof(type,member) );})
```

27.6 Opérations sur les répertoires de socket

Nous avons vu ci-dessus que, pour chaque type de fichiers, il faut définir l'ensemble des opérations permises sur les répertoires de ce type, utilisé par le système de fichier virtuel. Dans le cas des sockets cette structure, `sockfs_dentry_operations`, est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
344 static struct dentry_operations sockfs_dentry_operations = {
345         .d_delete =     sockfs_delete_dentry,
346 };
```

Une seule opération est donc nécessaire pour les répertoires de socket sur les six prévues dans le cas général, celle qui concerne la destruction d'une entrée de répertoire.

Cette fonction `sockfs_delete_dentry()` est définie dans le même fichier : Code Linux 2.6.10

```
340 static int sockfs_delete_dentry(struct dentry *dentry)
341 {
342         return 1;
343 }
```

Elle se contente de renvoyer qu'il y a une erreur (puisque la notion de répertoire n'a pas de sens pour les sockets).

27.7 Opérations sur les fichiers de type socket

Nous avons vu ci-dessus que, pour chaque type de fichiers, il faut définir une structure des opérations sur les fichiers de ce type, utilisée par le système de fichier virtuel. Cette structure, `socket_file_ops`, est définie dans le fichier `linux/net/socket.c` dans le cas des sockets, précédée de la liste des fichiers inclus nécessaires et des prototypes des douze fonctions nécessaires pour définir cette structure : Code Linux 2.6.10

```
61  #include <linux/config.h>
62  #include <linux/mm.h>
63  #include <linux/smp_lock.h>
64  #include <linux/socket.h>
65  #include <linux/file.h>
66  #include <linux/net.h>
67  #include <linux/interrupt.h>
68  #include <linux/netdevice.h>
69  #include <linux/proc_fs.h>
70  #include <linux/seq_file.h>
71  #include <linux/wanrouter.h>
72  #include <linux/if_bridge.h>
73  #include <linux/init.h>
74  #include <linux/poll.h>
75  #include <linux/cache.h>
76  #include <linux/module.h>
77  #include <linux/highmem.h>
78  #include <linux/divert.h>
79  #include <linux/mount.h>
80  #include <linux/security.h>
81  #include <linux/syscalls.h>
82  #include <linux/compat.h>
83  #include <linux/kmod.h>
84
85  #ifdef CONFIG_NET_RADIO
86  #include <linux/wireless.h>             /* Note : WIRELESS_EXT a definir */
87  #endif  /* CONFIG_NET_RADIO */
88
89  #include <asm/uaccess.h>
90  #include <asm/unistd.h>
91
92  #include <net/compat.h>
93
94  #include <net/sock.h>
95  #include <linux/netfilter.h>
96
97  static int sock_no_open(struct inode *irrelevant, struct file *dontcare);
98  static ssize_t sock_aio_read(struct kiocb *iocb, char __user *buf,
99                               size_t size, loff_t pos);
100 static ssize_t sock_aio_write(struct kiocb *iocb, const char __user *buf,
101                               size_t size, loff_t pos);
102 static int sock_mmap(struct file *file, struct vm_area_struct * vma);
103
104 static int sock_close(struct inode *inode, struct file *file);
105 static unsigned int sock_poll(struct file *file,
106                               struct poll_table_struct *wait);
107 static int sock_ioctl(struct inode *inode, struct file *file,
108                       unsigned int cmd, unsigned long arg);
109 static int sock_fasync(int fd, struct file *filp, int on);
110 static ssize_t sock_readv(struct file *file, const struct iovec *vector,
111                           unsigned long count, loff_t *ppos);
112 static ssize_t sock_writev(struct file *file, const struct iovec *vector,
113                           unsigned long count, loff_t *ppos);
114 static ssize_t sock_sendpage(struct file *file, struct page *page,
115                              int offset, size_t size, loff_t *ppos, int more);
116
117
118 /*
119  *      Les fichiers socket possedent un ensemble d'operations 'speciales' ainsi que
        des operations de fichier generique. Celles-ci n'apparaissent pas
120  *      dans la structure des operations mais sont effectuees directement a travers le
        multiplexeur socketcall().
```

```
121  */
122
123 static struct file_operations socket_file_ops = {
124         .owner =        THIS_MODULE,
125         .llseek =       no_llseek,
126         .aio_read =     sock_aio_read,
127         .aio_write =    sock_aio_write,
128         .poll =         sock_poll,
129         .ioctl =        sock_ioctl,
130         .mmap =         sock_mmap,
131         .open =         sock_no_open,   /* code open special pour refuser open via /proc */
132         .release =      sock_close,
133         .fasync =       sock_fasync,
134         .readv =        sock_readv,
135         .writev =       sock_writev,
136         .sendpage =     sock_sendpage
137 };
```

On remarquera que seules douze des vingt et une fonctions prévues dans le cas général sont nécessaires pour les sockets.

27.8 Premières implémentations de fonctions

Un certain nombre de fonctions sur les fichiers de type socket sont implémentées mais n'ont pas vraiment d'intérêt. Voyons-les à la fin de ce chapitre, que l'on peut passer sans problème.

27.8.1 Implémentation du positionnement

La fonction générale `no_llseek()` est définie dans le fichier `linux/fs/read_write.c`: Code Linux 2.6.10

```
81 loff_t no_llseek(struct file *file, loff_t offset, int origin)
82 {
83         return -ESPIPE;
84 }
```

Elle se contente de renvoyer l'opposé d'un code d'erreur puisqu'il n'y a pas d'accès direct pour les tubes de communication, plus exactement pour les sockets dans notre cas.

27.8.2 Implémentation de la scrutation

27.8.2.1 Fonction principale

La fonction `sock_poll()`, qui permet de déterminer s'il existe des données disponibles sur la socket, est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
932 /* Pas de verrou noyau detenu - parfait */
933 static unsigned int sock_poll(struct file *file, poll_table * wait)
934 {
935         struct socket *sock;
936
937         /*
938          *      Nous ne devons pas renvoyer d'erreurs a poll, aussi est-ce oui ou non.
939          */
940         sock = SOCKET_I(file->f_dentry->d_inode);
941         return sock->ops->poll(file, sock, wait);
942 }
```

Autrement dit :

- on charge le descripteur de socket associé au descripteur de fichier passé en paramètre, grâce à la fonction auxiliaire `SOCKET_I()` étudiée ci-après ;
- on utilise la fonction de scrutation spécifique à la famille de protocoles et on renvoie ce qui est transmis par celle-ci.

27.8.2.2 Descripteur de socket associé à un nœud d'information de socket

La fonction :

```
struct socket *SOCKET_I(struct inode *inode);
```

permet de connaître le descripteur de socket auquel un nœud d'information (de socket) est associé. Bien entendu s'il ne s'agit pas d'un nœud d'information de socket, `NULL` est renvoyé.

Cette fonction en ligne est définie, d'une façon on ne peut plus simple, dans le fichier `linux/include/net/sock.h` :

Code Linux 2.6.10

```
649 static inline struct socket *SOCKET_I(struct inode *inode)
650 {
651         return &container_of(inode, struct socket_alloc, vfs_inode)->socket;
652 }
```

27.8.2.3 Cas de IPv4/UDP

Nous avons vu au chapitre 26 que, dans le cas de IPv4/UDP, l'ensemble des opérations sur les descripteurs de socket s'appelle `inet_dgram_ops`. Cet ensemble est défini dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
803 struct proto_ops inet_dgram_ops = {
804         .family =       PF_INET,
805         .owner =        THIS_MODULE,
806         .release =      inet_release,
807         .bind =         inet_bind,
808         .connect =      inet_dgram_connect,
809         .socketpair =   sock_no_socketpair,
810         .accept =       sock_no_accept,
811         .getname =      inet_getname,
812         .poll =         udp_poll,
813         .ioctl =        inet_ioctl,
814         .listen =       sock_no_listen,
815         .shutdown =     inet_shutdown,
816         .setsockopt =   sock_common_setsockopt,
817         .getsockopt =   sock_common_getsockopt,
818         .sendmsg =      inet_sendmsg,
819         .recvmsg =      sock_common_recvmsg,
820         .mmap =         sock_no_mmap,
821         .sendpage =     inet_sendpage,
822 };
```

Le nom de la fonction spécifique de la fonction de scrutation dans le cas du couple IPv4/UDP est donc `datagram_poll()`, mais cela ne nous intéressera pas ici.

27.8.3 Implémentation du non mappage en mémoire vive

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

Code Linux 2.6.10

```
1055 int sock_no_mmap(struct file *file, struct socket *sock, struct vm_area_struct *vma)
1056 {
```

```
1057         /* Code d'erreur de la methode mmap manquant */
1058         return -ENODEV;
1059 }
```

qui renvoie l'opposé du code d'erreur `ENODEV`. En effet, cette fonction est prévue pour copier le contenu intégral d'un fichier en mémoire vive pour accélérer les opérations sur celui-ci, ce qui n'est pas possible pour une socket.

27.8.4 Implémentation de l'ouverture de fichier

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

```
483 /*
484  *      En theorie vous ne pouvez pas ouvrir cet inode, mais /proc fournit
485  *      une porte derobee. Rappelons qu'il faut la garder fermee, sinon vous allez y
486  *      deposer des betes rampantes.
487  */
488
489 static int sock_no_open(struct inode *irrelevant, struct file *dontcare)
490 {
491         return -ENXIO;
492 }
```

Elle se contente de renvoyer l'opposé du code d'erreur `ENXIO`, puisqu'on ne peut pas réouvrir une socket. Elle est ouverte lors de sa création, un point c'est tout.

27.8.5 Implémentation des événements asynchrones

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

```
968  /*
969   *      Met a jour la liste async de la socket
970   *
971   *      Strategie de verrouillage de fasync_list.
972   *
973   *      1. fasync_list ne peut etre modifiee que par un processus ayant le verrou de
974   *         la socket c'est-a-dire avec semaphore.
975   *      2. fasync_list est utilise avec read_lock(&sk->sk_callback_lock)
976   *         ou le verrou de la socket.
977   *      3. fasync_list peut etre utilisee dans un contexte softirq, donc cette
978   *         modification avec verrou de socket doit etre effectuee avec
979   *         write_lock_bh(&sk->sk_callback_lock).
980   *                                                      --ANK (990710)
981   */
982
983  static int sock_fasync(int fd, struct file *filp, int on)
984  {
985          struct fasync_struct *fa, *fna=NULL, **prev;
986          struct socket *sock;
987          struct sock *sk;
988
989          if (on)
990          {
991                  fna=(struct fasync_struct *)kmalloc(sizeof(struct fasync_struct),
                        GFP_KERNEL);
992                  if(fna==NULL)
993                          return -ENOMEM;
994          }
995
996          sock = SOCKET_I(filp->f_dentry->d_inode);
997
```

```
998          if ((sk=sock->sk) == NULL) {
999                  if (fna)
1000                         kfree(fna);
1001                 return -EINVAL;
1002         }
1003
1004         lock_sock(sk);
1005
1006         prev=&(sock->fasync_list);
1007
1008         for (fa=*prev; fa!=NULL; prev=&fa->fa_next,fa=*prev)
1009                 if (fa->fa_file==filp)
1010                         break;
1011
1012         if(on)
1013         {
1014                 if(fa!=NULL)
1015                 {
1016                         write_lock_bh(&sk->sk_callback_lock);
1017                         fa->fa_fd=fd;
1018                         write_unlock_bh(&sk->sk_callback_lock);
1019
1020                         kfree(fna);
1021                         goto out;
1022                 }
1023                 fna->fa_file=filp;
1024                 fna->fa_fd=fd;
1025                 fna->magic=FASYNC_MAGIC;
1026                 fna->fa_next=sock->fasync_list;
1027                 write_lock_bh(&sk->sk_callback_lock);
1028                 sock->fasync_list=fna;
1029                 write_unlock_bh(&sk->sk_callback_lock);
1030         }
1031         else
1032         {
1033                 if (fa!=NULL)
1034                 {
1035                         write_lock_bh(&sk->sk_callback_lock);
1036                         *prev=fa->fa_next;
1037                         write_unlock_bh(&sk->sk_callback_lock);
1038                         kfree(fa);
1039                 }
1040         }
1041
1042 out:
1043         release_sock(sock->sk);
1044         return 0;
1045 }
```

27.8.6 Implémentation de la libération de fichier

27.8.6.1 Fonction principale

Code Linux 2.6.10

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

```
951 int sock_close(struct inode *inode, struct file *filp)
952 {
953         /*
954          *      Il est possible que le inode soit NULL. Nous serions alors
955          *      en train de fermer une socket indefinie.
956          */
957
```

```
958         if (!inode)
959         {
960                 printk(KERN_DEBUG "sock_close: NULL inode\n");
961                 return 0;
962         }
963         sock_fasync(-1, filp, 0);
964         sock_release(SOCKET_I(inode));
965         return 0;
966 }
```

Autrement dit on libère les ressources adéquates :

- si aucun descripteur de nœud d'information n'est affecté à la socket, c'est que celle-ci a déjà été fermée ; on affiche donc un message d'erreur et on renvoie 0 ;
- sinon on libère le descripteur de fichier, le descripteur de socket et on renvoie 0.

27.8.6.2 Libération d'un descripteur de socket

La fonction `sock_release()`, fonction interne de libération d'un descripteur de socket, est définie dans le fichier `linux/net/socket.c` :

Code Linux 2.6.10

```
499 /**
500  *      sock_release    -       ferme une socket
501  *      @sock : socket a fermer
502  *
503  *      La socket est enlevee de la pile de protocoles si elle a un rappel de liberation,
504  *      puis l'inode est alors libere si la socket est liee a un
505  *      inode et non a un fichier.
506  */
507
508 void sock_release(struct socket *sock)
509 {
510         if (sock->ops) {
511                 struct module *owner = sock->ops->owner;
512
513                 sock->ops->release(sock);
514                 sock->ops = NULL;
515                 module_put(owner);
516         }
517
518         if (sock->fasync_list)
519                 printk(KERN_ERR "sock_release: fasync list not empty!\n");
520
521         get_cpu_var(sockets_in_use)--;
522         put_cpu_var(sockets_in_use);
523         if (!sock->file) {
524                 iput(SOCK_INODE(sock));
525                 return;
526         }
527         sock->file=NULL;
528 }
```

Autrement dit :

- s'il existe une opération spécifique de libération de socket associée à la famille de protocole, on fait appel à elle ;
- si la liste `fasync` n'est pas vide, on affiche un message d'erreur ;
- on décrémente le nombre de sockets en utilisation sur le microprocesseur concerné ;
- on libère le nœud d'information associé à la socket.

27.8.6.3 Cas de IPv4

Comme nous l'avons vu à propos de `inet_dgram_ops`, le nom de la fonction spécifique de libération d'un descripteur de socket dans le cas de IPv4, que ce soit pour TCP ou pour UDP, est `inet_release()`. Elle est définie dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
353 /*
354  *      La socket paire devrait toujours etre NULL (ou autre). Lorsque nous appelons cette
355  *      fonction, nous sommes en train de detruire l'objet et donc plus personne ne
356  *      peut lui faire reference.
357  */
358 int inet_release(struct socket *sock)
359 {
360         struct sock *sk = sock->sk;
361
362         if (sk) {
363                 long timeout;
364
365                 /* Les applications oublient de quitter les groupes avant de sortir */
366                 ip_mc_drop_socket(sk);
367
368                 /* Si linger est positionne, nous ne renverrons rien avant que la fermeture
369                  * soit complete. Sinon nous renvoyons immediatement. Actuellement la
370                  * fermeture est faite de la meme facon.
371                  *
372                  * Si la fermeture est due a la mort du processus, nous n'avons
373                  * jamais linger...
374                  */
375                 timeout = 0;
376                 if (sock_flag(sk, SOCK_LINGER) &&
377                     !(current->flags & PF_EXITING))
378                         timeout = sk->sk_lingertime;
379                 sock->sk = NULL;
380                 sk->sk_prot->close(sk, timeout);
381         }
382         return 0;
383 }
```