Troisième partie

Vue générale sur l'implémentation Linux

Chapitre 10

Implémentation générale de Linux

Les systèmes d'exploitation modernes, tels que Linux, implémentent tous des un sous-système réseau. Avant de passer à l'implémentation de celui-ci en ce qui concerne Linux, disons quelques mots de l'implémentation générale de Linux.

Il serait malheureusement trop long ne serait-ce que d'en faire un résumé en un chapitre. Heureusement il existe de bons livres de ce point de vue, entre autres [BOV–01]. Nous avons la faiblesse de penser que notre livre consacré à ce sujet [CEG–03], qui commente la toute première version 0.01 de Linux, est bien adapté. La suite de ce livre suppose une connaissance de l'implémentation de Linux, acquise dans [CEG–03] ou autre.

La connaissance du noyau 0.01 est largement suffisante pour comprendre l'implémentation réseau de Linux. Cependant certains points généraux ont évolué et influent sur celle-ci. Ce sont d'eux dont nous allons parler ici et, parsemé ici et là, sous le nom de *fonction générale*.

Les quatre sous-systèmes les plus importants de Linux sont : la gestion de la mémoire (`MM` pour l'anglais *Memory Management*), la gestion des processus (devenue gestion des activités avec la diversification de nature de celles-ci), la gestion des fichiers et la gestion des périphériques. Il existe d'autres sous-systèmes tels que le sous-système réseau, objet de cet ouvrage, la gestion du temps, la gestion des processeurs sur un système avec plusieurs microprocesseurs ou la gestion des modules.

Après une vue d'ensemble de l'organisation des fichiers source de Linux, nous consacrerons une section à la définition des types portables (pour les rendre indépendants du microprocesseur utilisé), puis à la gestion de la mémoire et à la gestion du temps. La gestion des fichiers sera revue au début du chapitre 27, consacré à l'implémentation du type de fichiers socket. Nous aborderons ensuite la gestion de plusieurs microprocesseurs, puis celle des activités. Nous consacrerons une section à l'implémentation des opérations atomiques, à celle des modules et enfin à l'initialisation du système Linux.

10.1 Organisation du code source

Nous renvoyons à [CEG–03] pour une description complète de l'organisation du code source d'un des noyaux de Linux, le tout premier. Évidemment ça se complique de version en version : les sources du noyau 2.4.18, par exemple, occupent 122 Mo, celles du noyau 2.6.0 pas moins de 212 Mo.

10.1.1 Premier niveau

Linux 2.6.10

Le premier niveau de l'aborescence du noyau 2.6.10 est assez simple :

- `./Documentation` contient de la documentation, en particulier sur les périphériques pris en compte.
- `./arch` concerne tout qui dépend de l'architecture du microprocesseur, Linux ayant été adapté à plusieurs microprocesseurs. C'est dans ce répertoire qu'on trouve ce qui concerne le démarrage du système.
- `./crypto` concerne la cryptographie.
- `./drivers` contient les divers pilotes de périphériques.
- `./fs` contient ce qui concerne les différents systèmes de fichiers.
- `./include` contient les fichiers en-tête, dont beaucoup dépendent d'une architecture de microprocesseur donnée.
- `./init` ne contient qu'un seul fichier `main.c`, le fichier principal du code.
- `./ipc` contient la mise en place d'un mode de communication entre processus appelé IPC V.
- `./kernel` contient tout ce qu'on pourrait appelé le micro-noyau ou cœur du système d'exploitation.
- `./lib` contient le code des appels systèmes qui seront utilisés lors du démarrage du système.
- `./mm` (pour *Memory Management*) contient le code de gestion de la mémoire.
- `./net` concerne la mise en place du sous-système réseau.
- `./scripts` contient un certain nombre de programmes de commandes (*script* en anglais).
- `./security` contient ce qui concerne la sécurité.
- `./sound` contient ce qui concerne le son.
- `./usr` contient des fichiers auxiliaires.

Nous allons évidemment nous intéresser dans ce livre plus particulièrement au répertoire `./net` et aux parties des répertoires `./include` et `./drivers` qui concernent les réseaux.

10.1.2 Répertoire concernant les réseaux

Le répertoire `./net` contient un grand nombre de sous-répertoires. Nous aborderons les répertoires suivants :

- `./core` concerne le code commun à l'accès des périphériques réseau, au filtrage, aux sockets, etc.
- `./ipv4` contient le code concernant IP version 4.
- `./sched` contient le code concernant l'ordonnancement des paquets, qu'ils soient IP ou non.
- `./packet` contient le code des familles de protocole d'accès aux paquets bruts.
- `./netlink` contient le code pour ce protocole.

- `./ethernet` contient le code général de l'implémentation d'Ethernet, le code spécifique à chaque carte Ethernet se trouvant dans `../drivers`.
- `./802` contient le code générique du protocole de la couche de liaison spécifique à IEEE 802.
- `./unix` contient l'implémentation des sockets de type local.
- `./llc` contient l'implémentation de la sous-couche `LLC`.

Décrivons en quelques mots les autres répertoires, ceux qui ne nous intéresseront pas dans cet ouvrage:

- `./ipv6` contient le code concernant IP version 6.
- `./sunrpc` contient l'implémentation de l'appel de procédure distante (*Remote Procedure Call*) Sun (utile pour NFS).
- `./wanrouter` contient le support de routeur multi-protocole pour les réseaux étendus.
- `./atm` contient le code générique du protocole de la couche de liaison/réseau du mode de transfert asynchrone, incluant MPOA et les spécificités IP.
- `./ax25` contient le code du protocole de liaison X.25 des paquets pour les radio-amateurs.
- `./x25` contient le code du protocole de réseau x.25.
- `./rose` contient le code du service des opérations distantes sur x.25, non IP.
- `./lapb` contient le code du protocole de liaison *Link Access Procedure, Balanced*.
- `./netrom` contient le code du périphérique IP appelé *netrom*.
- `./bridge` contient l'implémentation des ponts suivant la spécification IEEE 802.1d en dessous du niveau IP.
- `./ipx` contient l'implémentation du protocole réseau propriétaire de Novell appelé IPX.
- `./irda` contient l'implémentation des pilotes de périphériques infra-rouges (*Infra-Red Digital Access*).
- `./decnet` contient l'implémentation de l'ancienne suite de protocoles DECNET de *Digital Equipment Corporation*.
- `./econet` contient l'implémentation de l'ancienne suite de protocoles ECONET (non IP).
- `./appletalk` contient l'implémentation de la suite de protocoles qui fut conçue pour les MAC d'Apple (non IP).
- `./key` contient l'implémentation de `PF_KEY` version 2.
- `./rxrpc` contient l'implémentation de `Rx RPC`.
- `./sctp` contient l'implémentation de `SCTP` d'IBM.
- `./xfrm` contient l'implémentation de `XFRM`.
- `./8021q` contient l'implémentation générique de la couche *VLAN* de réseau local virtuel.
- `./bluetooth` contient l'implémentation de la couche physique pour les matériels utilisant le protocole Bluetooth.

10.1.3 Structure d'un fichier source

Montrons à quoi ressemble un fichier de l'implémentation Linux à propos du fichier principal `socket.c` du répertoire `linux/net`. L'API des sockets pour Linux est due à un grand nombre d'auteurs. Ceux-ci sont cités au début de ce fichier:

Code Linux 2.6.10

```
1  /*
2   * NET          Une implementation du protocole d'acces au reseau SOCKET.
3   *
4   * Version :    @(#)socket.c    1.1.93  18/02/95
```

```
5   *
6   * Auteurs :    Orest Zborowski, <obz@Kodak.COM>
7   *              Ross Biro, <bir7@leland.Stanford.Edu>
8   *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
9   *
10  * Rectifications :
11  *              Anonyme          :       Nettoyage NOTSOCK/BADF. Erreur rectifiee dans
12  *                                       shutdown()
13  *              Alan Cox         :       verify_area() rectifie
14  *              Alan Cox         :       DDI enleve
15  *              Jonathan Kamens  :       Bogue sur reconnect SOCK_DGRAM
16  *              Alan Cox         :       Deplacement d'un grand nombre de verifications
17  *                                       au plus haut niveau.
18  *              Alan Cox         :       Deplacement des structures d'adresses vers/depuis le
19  *                                       mode utilisateur au-dessus des couches de protocole.
20  *              Rob Janssen      :       Permet d'envoyer une longueur 0.
21  *              Alan Cox         :       Support des E/S asynchrones (plaggie des
22  *                                       pilotes tty).
23  *              Niibe Yutaka     :       E/S asynchrones pour les ecritures (style 4.4BSD)
24  *              Jeff Uphoff      :       Rendu le nombre max de sockets configurable
25  *                                       en ligne de commande.
26  *              Matti Aarnio     :       Rendu le nombre de sockets dynamique,
27  *                                       a etre alloue lorsque necessaire. Le max de M.
28  *                                       Uphoff est utilise comme max
29  *                                       permis en allocation.
30  *              Linus            :       Argh. efface toutes les allocation de socket par
31  *                                       ailleurs : dans le noeud d'information maintenant.
32  *              Alan Cox         :       Rendu sock_alloc()/sock_release() public
33  *                                       pour NetROM et le type nfsd du futur
34  *                                       noyau.
35  *              Alan Cox         :       Bases de sendmsg/recvmsg.
36  *              Tom Dyas         :       Exporte les symboles du reseau.
37  *              Marcin Dalecki   :       Problemes rectifies avec CONFIG_NET="n".
38  *              Alan Cox         :       Ajoute le verrouillage des threads pour les appels
39  *                                       sys_* des sockets. Il peut y avoir des erreurs
40  *                                       en ce moment.
41  *              Kevin Buhr       :       Rectifie les erreurs betes ci-dessus.
42  *              Andi Kleen       :       Quelques petits nettoyages, optimisations
43  *                                       et rectifie un bogue dans copy_from_user().
44  *              Tigran Aivazian  :       sys_send(args) appelle sys_sendto(args, NULL, 0)
45  *              Tigran Aivazian  :       Rendu les verifications du nombre de clients dans
46  *                                       listen(2) independant du protocole
47  *
48  *
49  *              Ce programme est un logiciel libre ; vous pouvez le redistribuer et/ou
50  *              le modifier sous les termes de la Licence Publique Generale GNU
51  *              publiee par la Free Software Foundation ; soit la version
52  *              2 de cette licence, soit (a votre convenance) toute version ulterieure.
53  *
54  *
55  *      Ce module est effectivement l'interface de plus haut niveau pour le
56  *      paradigme des sockets BSD.
57  *
58  */
58  *      Fonde sur NET3.039 de la Swansea University Computer Society
59  */
```

Le fichier commence par un commentaire comprenant un titre et une description rapide de son objet (ligne 2) suivi de la version et de la date de celle-ci (ligne 4). On remarquera que l'implémentation de cette partie n'aurait pas changée depuis 1995. En fait les dates sont rarement à jour.

Les auteurs de la première version du fichier sont cités, suivis des auteurs des rectifications,

avec une description rapide du rôle de chacun d'eux.

Les lignes 49 à 52 sur la licence sont reprises dans la plupart des fichiers.

Dans cet ouvrage, nous traduirons les commentaires en français. On pourra retrouver la version originelle anglaise en consultant les sources.

10.1.4 Aide au parcours du code source

Dans la version 2.4.9, le noyau Linux compte 9 837 fichiers représentant un volume de 3 857 319 lignes de code source. Quant aux modifications intervenant d'une version à la suivante, elles peuvent représenter plusieurs méga-octets. L'important volume représenté par ce code source fait qu'il est très difficile de s'y retrouver et de naviguer à l'emplacement souhaité.

C'est pour simplifier la manipulation du code source du noyau Linux qu'a été développé à l'université d'Oslo le navigateur de code source **LXR** (pour *Linux cross (X) Reference*) accessible sur le site Internet *Cross-Referencing Linux*:

`http://lxr.linux.no/`

en cliquant sur "Browse the code" ou:

`http://fxr.watson.org/fxr/`

ou encore, mais le serveur est plus lent, *Linux Cross Reference*:

`http://www.iglu.org.il/lxr/`

10.2 Définition de types portables

Les systèmes d'exploitation modernes, tels que Linux, doivent être indépendant de l'architecture matérielle. Il existe une très petite part du noyau qui dépend de l'architecture. Linux définit un ensemble de types C pour utilisation dans le noyau (et ailleurs si on le désire).

10.2.1 Types Posix

Ceci commence par la définition des types POSIX pour lesquels on n'a pas besoin de connaître l'architecture, dans le fichier `linux/include/linux/posix_types.h` que nous reproduisons intégralement:

Code Linux 2.6.10

```
 1 #ifndef _LINUX_POSIX_TYPES_H
 2 #define _LINUX_POSIX_TYPES_H
 3
 4 #include <linux/stddef.h>
 5
 6 /*
 7  * On permet 1 024 descripteurs de fichier : si NR_OPEN est un jour augmente
 8  * au-dela, vous aurez a changer ceci egalement. Mais 1024 fd semble suffisant
 9  * pour des unix "reels" comme OSF/1, aussi peut-on esperer que cette
10  * limite n'aura pas a etre changee [une fois de plus].
11  *
12  * Noter que POSIX veut que FD_CLEAR(fd,fdsetp) soit defini dans
13  * <sys/time.h> (en ainsi dans <linux/time.h>) - mais ici est une place plus
14  * logique. Resolu en ayant une definition muette dans <sys/time.h>.
15  */
16
17 /*
18  * Ces macros peuvent avoir ete definies dans <gnu/types.h>. Mais nous
19  * utiliserons toujours celles qui sont ici.
20  */
```

```
21 #undef __NFDBITS
22 #define __NFDBITS       (8 * sizeof(unsigned long))
23
24 #undef __FD_SETSIZE
25 #define __FD_SETSIZE    1024
26
27 #undef __FDSET_LONGS
28 #define __FDSET_LONGS   (__FD_SETSIZE/__NFDBITS)
29
30 #undef __FDELT
31 #define __FDELT(d)      ((d) / __NFDBITS)
32
33 #undef __FDMASK
34 #define __FDMASK(d)     (1UL << ((d) % __NFDBITS))
35
36 typedef struct {
37         unsigned long fds_bits [__FDSET_LONGS];
38 } __kernel_fd_set;
39
40 /* Type d'une routine de signal.  */
41 typedef void (*__kernel_sighandler_t)(int);
42
43 /* Type d'une cle SYSV IPC.  */
44 typedef int __kernel_key_t;
45 typedef int __kernel_mqd_t;
46
47 #include <asm/posix_types.h>
48
49 #endif /* _LINUX_POSIX_TYPES_H */
```

Les types pour lesquels la définition repose sur l'architecture sont définis dans le répertoire `asm` adéquat. Par exemple pour les microprocesseurs Intel, on trouve dans le fichier `linux/include/asm-i386/posix_types.h`:

Code Linux 2.6.10

```
1  #ifndef __ARCH_I386_POSIX_TYPES_H
2  #define __ARCH_I386_POSIX_TYPES_H
3
4  /*
5   * Ce fichier est en general utilise par les logiciels au niveau utilisateur, aussi devez-vous
6   * etre un petit peu soigneux quant a la pollution de l'espace des noms etc. De plus, vous ne
7   * pouvez pas supposer que GCC l'utilise.
8   */
9
10 typedef unsigned long   __kernel_ino_t;
11 typedef unsigned short  __kernel_mode_t;
12 typedef unsigned short  __kernel_nlink_t;
13 typedef long            __kernel_off_t;
14 typedef int             __kernel_pid_t;
15 typedef unsigned short  __kernel_ipc_pid_t;
16 typedef unsigned short  __kernel_uid_t;
17 typedef unsigned short  __kernel_gid_t;
18 typedef unsigned int    __kernel_size_t;
19 typedef int             __kernel_ssize_t;
20 typedef int             __kernel_ptrdiff_t;
21 typedef long            __kernel_time_t;
22 typedef long            __kernel_suseconds_t;
23 typedef long            __kernel_clock_t;
24 typedef int             __kernel_timer_t;
25 typedef int             __kernel_clockid_t;
26 typedef int             __kernel_daddr_t;
27 typedef char *          __kernel_caddr_t;
28 typedef unsigned short  __kernel_uid16_t;
29 typedef unsigned short  __kernel_gid16_t;
```

```
30 typedef unsigned int    __kernel_uid32_t;
31 typedef unsigned int    __kernel_gid32_t;
32
33 typedef unsigned short  __kernel_old_uid_t;
34 typedef unsigned short  __kernel_old_gid_t;
35 typedef unsigned short  __kernel_old_dev_t;
36
37 #ifdef __GNUC__
38 typedef long long       __kernel_loff_t;
39 #endif
```

10.2.2 Types entiers

Linux utilise les types entiers suivants :

Type	Signification	Variantes
s8	entier relatif sur 8 bits	__s8, int8_t
u8	entier naturel sur 8 bits	__u8, u_int8_t, uint8_t, u_char, unchar
s16	entier relatif sur 16 bits	__s16, int16_t
u16	entier naturel sur 16 bits	__u16, u_int16_t, uint16_t, u_short, ushort
s32	entier relatif sur 32 bits	__s32, int32_t
u32	entier naturel sur 32 bits	__u32, u_int32_t, uint32_t, u_int, uint, u_long, ulong
s64	entier relatif sur 64 bits	__s64, int64_t
u64	entier naturel sur 64 bits	__u64, uint64_t, u_int64_t

avec 's' pour *signed* et 'u' pour *unsigned*.

La définition de base de ces types dépend de l'architecture. Pour les microprocesseurs Intel, par exemple, elle se trouve dans le fichier `linux/include/asm-i386/types.h`:

Code Linux 2.6.10

```
8  /*
9   * __xx est ok : il ne pollue pas l'espace des noms POSIX. Utiliser ceci dans les
10  * fichiers en-tete exportes vers l'espace utilisateur
11  */
12
13 typedef __signed__ char __s8;
14 typedef unsigned char __u8;
15
16 typedef __signed__ short __s16;
17 typedef unsigned short __u16;
18
19 typedef __signed__ int __s32;
20 typedef unsigned int __u32;
21
22 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
23 typedef __signed__ long long __s64;
24 typedef unsigned long long __u64;
25 #endif
26
27 #endif /* __ASSEMBLY__ */
28
29 /*
30  * Ceux-ci ne sont pas exportes en-dehors du noyau pour eviter des problemes avec l'espace
      des noms
31  */
32 #ifdef __KERNEL__
33
34 #define BITS_PER_LONG 32
35
```

```
36 #ifndef __ASSEMBLY__
37
38 #include <linux/config.h>
39
40 typedef signed char s8;
41 typedef unsigned char u8;
42
43 typedef signed short s16;
44 typedef unsigned short u16;
45
46 typedef signed int s32;
47 typedef unsigned int u32;
48
49 typedef signed long long s64;
50 typedef unsigned long long u64;
```

Code Linux 2.6.10

La définition des variantes se trouve dans le fichier `linux/include/linux/types.h`:

```
  1 #ifndef _LINUX_TYPES_H
  2 #define _LINUX_TYPES_H
  3
  4 #ifdef  __KERNEL__
  5 #include <linux/config.h>
[...]
 13 #include <linux/posix_types.h>
 14 #include <asm/types.h>
 15
 16 #ifndef __KERNEL_STRICT_NAMES
[...]
 92 /* bsd */
 93 typedef unsigned char           u_char;
 94 typedef unsigned short          u_short;
 95 typedef unsigned int            u_int;
 96 typedef unsigned long           u_long;
 97
 98 /* sysv */
 99 typedef unsigned char           unchar;
100 typedef unsigned short          ushort;
101 typedef unsigned int            uint;
102 typedef unsigned long           ulong;
103
104 #ifndef __BIT_TYPES_DEFINED__
105 #define __BIT_TYPES_DEFINED__
106
107 typedef         __u8            u_int8_t;
108 typedef         __s8            int8_t;
109 typedef         __u16           u_int16_t;
110 typedef         __s16           int16_t;
111 typedef         __u32           u_int32_t;
112 typedef         __s32           int32_t;
113
114 #endif /* !(__BIT_TYPES_DEFINED__) */
115
116 typedef         __u8            uint8_t;
117 typedef         __u16           uint16_t;
118 typedef         __u32           uint32_t;
119
120 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
121 typedef         __u64           uint64_t;
122 typedef         __u64           u_int64_t;
123 typedef         __s64           int64_t;
124 #endif
```

10.2.3 Tailles

Les types liés à des tailles sont définis dans le fichier `linux/include/linux/types.h`: Code Linux 2.6.10

```
54 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
55 typedef __kernel_loff_t         loff_t;
56 #endif
57
58 /*
59  * Les typedef suivants sont egalement proteges par des ifdef individuels pour
60  * des raisons historiques :
61  */
62 #ifndef _SIZE_T
63 #define _SIZE_T
64 typedef __kernel_size_t         size_t;
65 #endif
66
67 #ifndef _SSIZE_T
68 #define _SSIZE_T
69 typedef __kernel_ssize_t        ssize_t;
70 #endif
```

Il s'agit de la longueur d'un décalage et des types entiers utilisés pour exprimer les tailles.

10.3 Gestion de la mémoire

Rappelons ce qui est nécessaire pour la gestion de la mémoire (uniquement du point de vue du programmeur système) : réservation et libération de mémoire de l'espace noyau, copie entre l'espace d'adressage du noyau et celui de l'utilisateur et enfin les caches mémoire.

10.3.1 Réservation et libération de mémoire noyau

10.3.1.1 Réservation

La fonction :

`kmalloc(size, priority)`

essaie de réserver une zone de mémoire connexe de `size` octets dans l'espace mémoire du noyau. En fait quelques octets supplémentaires sont réservés à cause des caches mémoire dont nous reparlerons ci-dessous.

Le paramètre `priority` permet d'indiquer des options. Le préfixe `GFP_` indique qu'il faut éventuellement utiliser la fonction `get_free_pages()` en vue de réservation de mémoire :

- `GFP_USER` pour allouer l'emplacement mémoire dans l'espace utilisateur.
- `GFP_KERNEL` pour allouer l'emplacement mémoire dans l'espace noyau. L'activité qui en fait la demande peut être interrompue au cours de la réservation.
- `GFP_ATOMIC` est l'analogue de `GFP_KERNEL` mais tout l'emplacement mémoire doit être attribué de façon atomique (en une seule fois).
- `GFP_DMA` pour indiquer que l'emplacement mémoire doit être adéquat pour un accès direct en mémoire.

Ces valeurs générales sont commentées dans le fichier `linux/mm/slab.c`: Code Linux 2.6.10

```
2406 /**
2407  * kmalloc - alloue de la memoire
2408  * @size : le nombre d'octets requis.
2409  * @flags: le type de memoire a allouer.
```

```
2410  *
2411  * kmalloc est la methode normale d'allocation de la memoire
2412  * du noyau.
2413  *
2414  * L'argument @flags est l'un des suivants :
2415  *
2416  * %GFP_USER - Alloue de la memoire pour l'utilisateur. Peut s'assoupir.
2417  *
2418  * %GFP_KERNEL - Alloue de la ram noyau normale. Peut s'assoupir.
2419  *
2420  * %GFP_ATOMIC - L'allocation ne s'assoupira pas. A utiliser pour les routines
                    d'interruption.
2421  *
2422  * De plus, le drapeau %GFP_DMA peut etre positionne pour indiquer que le memoire
2423  * doit etre adequate pour la DMA. Ceci peut signifier des choses differentes suivant
2424  * la plateforme. Par exemple, sur i386, ceci signifie que la memoire doit provenir
2425  * des 16 premiers Mo.
2426  */
```

La valeur de renvoi de `malloc()` est un pointeur sur la zone de mémoire réservée avec succès ou `NULL` s'il n'y a pas de mémoire disponible.

10.3.1.2 Libération

La fonction :

`kfree(objp)`

libère la zone de mémoire réservée à l'adresse `objp`. Cette zone de mémoire doit avoir été réservée au préalable à l'aide de `kmalloc()`.

10.3.2 Copie entre espace noyau et espace utilisateur

Les fonctions suivantes sont utilisables pour échanger des données entre l'espace d'adressage noyau et l'espace d'adressage utilisateur :

- `copy_from_user(to, from, count)` copie `count` octets de la partie mémoire de l'espace utilisateur d'adresse `from` vers la partie mémoire de l'espace noyau d'adresse `to`.
- `copy_to_user(to, from, count)` copie `count` octets de la partie mémoire de l'espace noyau d'adresse `from` vers la partie mémoire de l'espace utilisateur d'adresse `to`.

10.3.3 Caches mémoire

10.3.3.1 Notion

La réservation de zones de mémoire à l'aide de l'appel système `kmalloc()` peut durer un certain temps. Si on prévoit qu'une zone mémoire de même taille sera nécessaire dans un très proche avenir, on a intérêt à ne pas libérer la zone mémoir avec l'appel système `kfree()` après utilisation, mais à enregistrer le fait dans une liste puis à la réutiliser à la demande.

Ce mode d'action est implémenté sous Linux par les **caches par plaque** (*slab* en anglais).

10.3.3.2 Création d'un cache par plaque

La fonction :

`kmem_cache_create(name, size, offset, flags, ctor, dstor)`

crée un cache par plaque pour les zones de mémoire de `size` octets. Le paramètre `name` pointe sur une chaîne de caractères contenant le nom du cache. Le paramètre `offset` permet d'indiquer

le décalage de la première zone de mémoire dans une page de mémoire (la plupart du temps de valeur nulle). Le paramètre `flags` sert à préciser ce qu'on souhaite pour la réservation des zones de mémoire, par exemple :

- `SLAB_HWCACHE_ALIGN` our s'aligner sur la taille du cache du premier niveau du processeur ;
- `SLAB_NO_REAP` pour emêcher la réduction du cache lorsque le noyau a besoin de mémoire ;
- `SLAB_CACHE_DMA` pour que les zones de mémoire réservée se trouvent dans la zone compatible DMA.

Les paramètres `ctor` et `dtor` permettent d'indiquer un constructeur et un destructeur pour la zone de mémoire. Ils servent à l'initialisation et au déblaiement des zones de mémoire réservées.

La valeur de renvoi est un pointeur sur la structure de cache de plaque, de type `kmem_cache_t`.

10.3.3.3 Allocation

La fonction :

`kmem_cache_alloc(cachep, flags)`

permet de demander une zone de mémoire à partir du cache par plaque `cachep`. S'il s'en trouve un de disponible, un pointeur est immédiatement renvoyé à l'appelant. Lorsque le cache est plein, `kmalloc()` réserve une nouvelle zone de mémoire ; les valeurs `GFP_` sont alors utilisées à titre de drapeaux.

10.3.3.4 Libération

La fonction :

`kmem_cache_free(kmem_cache_t *cachep, void *objp)`

renvoie au cache `cachep` la zone de mémoire commençant à l'adresse `objp`, réservée au préalable avec `kmem_cache_alloc()`.

10.4 La gestion du temps dans le noyau Linux

10.4.1 Durée

Une durée est exprimée soit en *jiffies*, soit en seconde et microseconde à travers la structure `struct timeval`. Cette dernière est définie dans le fichier `linux/include/linux/time.h`: Code Linux 2.6.10

```
18 struct timeval {
19         time_t          tv_sec;         /* secondes */
20         suseconds_t     tv_usec;        /* microsecondes */
21 };
```

Les types `time_t` et `suseconds_t` sont définis dans le fichier `linux/include/linux/types.h`: Code Linux 2.6.10

```
29 typedef __kernel_suseconds_t    suseconds_t;
[...]
77 #ifndef _TIME_T
78 #define _TIME_T
79 typedef __kernel_time_t         time_t;
80 #endif
```

qui renvoie, pour les microprocesseurs Intel, au fichier `linux/include/asm-i386/posix_types.h`: Code Linux 2.6.10

```
21 typedef long           __kernel_time_t;
22 typedef long           __kernel_suseconds_t;
```

10.4.2 File d'attente de minuteurs

Les ordinateurs ne possèdent en général que d'un seul minuteur alors qu'on en a besoin de plusieurs en programmation système. On résout de problème en utilisant une **file d'attente de minuteurs** : lorsqu'une interruption d'horloge se produit, la routine de gestion de celle-ci ne se contente pas d'actualiser la variable `jiffies` ; elle vérifie aussi la file d'attente de minuteurs à la recherche d'éventuelles routines de traitement.

La file d'attente de minuteurs est constituée de structures `timer_list` représentant une fonction (routine de traitement) devant être exécutée à un certain moment (`expires`). Ce type est défini dans le fichier `linux/include/linux/timer.h` :

Code Linux 2.6.10

```
11 struct timer_list {
12         struct list_head entry;
13         unsigned long expires;
14
15         spinlock_t lock;
16         unsigned long magic;
17
18         void (*function)(unsigned long);
19         unsigned long data;
20
21         struct tvec_t_base_s *base;
22 };
```

Voici les fonctions disponibles pour la gestion de la file d'attente des minuteurs :

- `add_timer()` insère une structure `timer_list` dans la file d'attente.
- `del_timer()` supprime une structure `timer_list` de la file d'attente.
- `init_timer()` initialise une structure `timer_list`. Cette fonction doit toujours être appelée lorsqu'une structure `timer_list` a été générée. Elle est définie dans le fichier `linux/include/linux/timer.h` :

Code Linux 2.6.10

```
35 /***
36  * init_timer - initialise un minuteur.
37  * @timer : le minuteur a initialiser
38  *
39  * init_timer() doit etre appliquee a un minuteur avant d'appeler *toutes* les
40  * autres fonctions de minutage.
41  */
42 static inline void init_timer(struct timer_list * timer)
43 {
44         timer->base = NULL;
45         timer->magic = TIMER_MAGIC;
46         spin_lock_init(&timer->lock);
47 }
```

10.5 SMP

Linux est l'un des rares systèmes d'exploitation à prendre en charge plusieurs processeurs, depuis la version 2.0 du noyau pour les processeurs Intel et Sparc, fonction améliorée dans les versions ultérieures. Il s'agit du **SMP** (pour l'anglais *Symetric Multi-Processing*).

Cependant nous ne commenterons pas le code qui dépend de SMP dans cet ouvrage.

10.6 Gestion des activités dans le noyau Linux

Linux est un système multitâche. Pendant plusieurs versions du noyau Linux, les seules **activités** possibles étaient les processus, les appels systèmes et les interruptions matérielles. De nos

jours, on a un plus grand choix :

- processus ;
- appels systèmes ;
- interruptions matérielles ;
- interruptions logicielles ;
- tasklets ;
- parties basses.

10.6.1 Activités de base

Les trois types d'activités de base existent depuis le tout premier noyau Linux et sont décrits, par exemple, dans [CEG–03]. Nous n'insisterons pas ici.

10.6.1.1 Processus

Un **processus** est une activité démarrée dans le but d'exécuter une instance d'une application donnée. Un processus est décrit par une entité du type `struct task_struct`.

10.6.1.2 Appels système

Les processus n'agissent que dans l'espace d'adressage utilisateur. Si un processus veut accéder aux pilotes ou utiliser une fonctionnalité du noyau du système d'exploitation, il doit avoir recours à un **appel système**. Tout n'est pas permis mais seulement ce qui a été prévu par le concepteur du noyau.

10.6.1.3 Interruptions matérielles

Les périphériques informent un système d'exploitation de la survenue d'un événement à l'aide d'une **interruption matérielle** (**HW-IRQ** pour *HardWare Interrupt ReQuest*). Ceci provoque l'interruption de l'activité en cours sur un des microprocesseurs et l'exécution d'une **routine de traitement d'interruption**.

La routine de traitement d'une interruption peut être décrite statiquement dans le noyau ou enregistrée au moment de l'exécution à l'aide de la fonction `request_irq()`. Dans ce dernier cas, elle peut être retirée grâce à la fonction `free_irq()`.

La fonction générale `request_irq()` est commentée dans le fichier `linux/kernel/irq/manage.c` :

Code Linux 2.6.10

```
279 /**
280  *      request_irq - alloue une ligne d'interruption
281  *      @irq : Ligne d'interruption a allouer
282  *      @handler : Fonction a appeler lorsque l'IRQ survient
283  *      @irqflags : Drapeaux du type d'interruption
284  *      @devname : Un nom ascii pour le peripherique reclamant
285  *      @dev_id : Un cafteur passe a la routine de gestion
286  *
287  *      Cet appel alloue des ressources d'interruption et active la
288  *      ligne d'interruption et le traitement de l'IRQ. Pour ce dernier point,
289  *      cet appel donne la routine de gestion qui doit etre invoquee. Puisque
290  *      votre routine de gestion peut effacer toute interruption que la carte
291  *      a leve, vous devez prendre soin a la fois d'initialiser votre materiel
292  *      et de positionner la routine d'interruption dans le bon ordre.
293  *
294  *      Dev_id doit etre globalement unique. Normalement l'adresse de la
```

```
295  *      structure de donnees du peripherique est utilisee comme cafteur. Puisque la
296  *      routine recoit cette valeur, il n'est pas bete de l'utiliser.
297  *
298  *      Si votre interruption est partagee, vous devez passer un dev_id non NULL
299  *      comme ceci est exige lorsqu'on libere l'interruption.
300  *
301  *      Drapeaux :
302  *
303  *      SA_SHIRQ                L'interruption est partagee
304  *      SA_INTERRUPT            Inhibe les interruptions locales lors du traitement
305  *      SA_SAMPLE_RANDOM        L'interruption peut etre utilisee pour l'entropie
306  *
307  */
308 int request_irq(unsigned int irq,
309                 irqreturn_t (*handler)(int, void *, struct pt_regs *),
310                 unsigned long irqflags, const char * devname, void *dev_id)
```

Code Linux 2.6.10

La fonction générale `free_irq()` est commentée dans le même fichier :

```
218 /**
219  *      free_irq - libere une interruption
220  *      @irq : ligne d'interruption a liberer
221  *      @dev_id : identite du peripherique a liberer
222  *
223  *      Retire un gestionnaire d'interruption. Le gestionnaire est retire et si
224  *      la ligne d'interruption n'est plus utilisee par aucun peripherique, elle es inhibee.
225  *      Pour une IRQ partagee, l'appelant doit s'assurer que l'interruption est inhibee
226  *      sur la carte qu'il pilote avant d'appeler cette fonction. La fonction
227  *      ne renvoie rien jusqu'a ce que toutes les interruptions en cours pour cette IRQ
228  *      soient traitees.
229  *
230  *      Cette fonction ne peut pas etre appelee dans un contexte d'interruption.
231  */
232 void free_irq(unsigned int irq, void *dev_id)
```

Le noyau Linux distingue deux types d'interruptions matérielles :

- Les **interruptions rapides** comportent une routine de traitement d'interruption très courtes et interrompent l'activité en cours très rapidement. Toutes les autres interruptions sont verrouillées au cours de leur exécution. La routine ne doit pas être interrompue.

  Les interruptions rapides sont identifiées lors de l'enregistrement par `request_irq()` à l'aide du drapeau `SA_INTERRUPT`.
- Les **interruptions lentes** peuvent être interrompues au cours de leur exécution par d'autres interruptions. Elles comportent souvent une routine de traitement d'interruption plus longue. Dans ce cas on distingue une partie haute et une partie basse.

10.6.2 Parties basses et tasklets

10.6.2.1 Parties hautes et parties basses

Les routines de traitement d'interruptions doivent s'exécuter dès que possible après le déclenchement de l'interruption et n'interrompre l'activité en cours que le plus brièvement possible. Toutefois, il est impossible d'exécuter certaines tâches avec peu d'instructions. Ceci a conduit à diviser certaines routines en deux parties :

- Dans la **partie haute** (*top half* en anglais), seules les tâches les plus importantes sont réalisées après déclenchement d'une interruption. La partie haute correspond à la routine d'interruption proprement dite.

– Dans la **partie basse** (*bottom half* en anglais, abrégé en **BH**) s'exécutent toutes les opérations pour lesquelles le temps n'est pas la préoccupation majeure et qu'il n'est pas nécessaire de réaliser dans la routine de traitement d'interruption proprement dite. La partie basse correspond à une activité qui sera prise en compte par l'ordonnanceur des tâches.

Dans les noyaux antérieurs à la version 2.4, les parties basses constituaient la forme d'activité principale au sein du noyau. La `NET_BH` était par exemple responsable du traitement du sous-système réseau et de l'envoi des paquets. Les parties basses ont ensuite été remplacées par les tasklets.

10.6.2.2 Tasklets

Les **tasklets**, introduites pour remplacer les anciennes parties basses, possèdent les propriétés suivantes :

– Elles possèdent une instantiation du type `struct tasklet_struct`.
– Elles sont mises en activité grâce à la fonction `tasklet_schedule(&tasklet_struct)`.
– Il est garanti qu'une tasklet ne peut s'exécuter à un instant donné que sur un seul microprocesseur.
– Différentes tasklets peuvent s'exécuter simultanément sur plusieurs microprocesseurs.

10.6.3 Interruptions logicielles

Les interruptions logicielles ne sont pas des interruptions à proprement parler mais des activités dont l'exécution est démarrée à l'aide de la fonction `do_softirq()` lorsque l'ordonnanceur des tâches lui passe la main.

On peut définir au maximum 32 interruptions logicielles dans le noyau Linux. Il n'en existe que six dans le noyau 2.6 dont `NET_RX_SOFTIRQ` (pour *NET ReCeive SOFT IRQ*), `NET_TX_SOFTIRQ` (pour *NET TransmiT SOFT IRQ*), qui nous intéressent tout particulièrement dans ce livre, et `TASKLET_SOFTIRQ` utilisée en vue de l'implémentation des tasklets décrites ci-dessus, comme le montre le fichier `linux/include/linux/interrupt.h`:

Code Linux 2.6.10

```
79 /* Evitez, S'IL VOUS PLAIT, d'ajouter de nouvelles interruptions logicielles, si vous
80    n'avez pas _reellement_ besoin d'ordonnancer des travaux a traiter a haute
      frequence. Pour presque tous les propos, les tasklets sont plus que
81    suffisantes. Par exemple tous les BH des peripheriques serie et analogues
82    devraient etre converties en tasklets, pas en interruptions logicielles.
83  */
84
85 enum
86 {
87         HI_SOFTIRQ=0,
88         TIMER_SOFTIRQ,
89         NET_TX_SOFTIRQ,
90         NET_RX_SOFTIRQ,
91         SCSI_SOFTIRQ,
92         TASKLET_SOFTIRQ
93 };
```

Voici les propriétés les plus importantes des interruptions logicielles :

– Une interruption logicielle peut s'exécuter simultanément sur plusieurs microprocesseurs. Par conséquent lorsqu'il existe des zones critiques dans une interruption logicielle, elles doivent être protégées par un verrou, ainsi que toutes les variables globales auxquelles elle accède.

- Une interruption logicielle ne peut pas s'interrompre au cours de son exécution sur un microprocesseur.
- Une interruption logicielle ne peut, au cours de son traitement sur un microprocesseur, être interrompue que par une interruption matérielle.

10.7 Opérations atomiques

Plusieurs formes d'activités différentes, capables de s'interrompre mutuellement, peuvent opérer en pseudo-parallélisme. Sur les systèmes multiprocesseurs, différentes activités opèrent même en vrai parallèlisme. Tant que les activités opèrent séparément dans le noyau Linux, il n'y a aucun problème. Toutefois, dès que plusieurs activités accèdent aux mêmes structures de données, des effets indésirables sont susceptibles de se produire.

On peut, par exemple, commencer par tester un champ avec un processus A. Le processus B change la valeur de celui-ci. Le processus A agit alors sur la structure de données avec une mauvaise information.

Il est extrêmement important, pour la stabilité du système, de s'assurer que ces opérations parallèles se déroulent sans effet secondaire indésirable. Pour éviter le genre de problèmes cité ci-dessus lors de l'opération de plusieurs activités sur une structure de données communes, appelée **zone critique**, ces activités doivent avoir lieu de manière **atomique**, c'est-à-dire qu'une opération comportant plusieurs étapes est exécutée de manière indivisible. Au cours d'une opération atomique, aucune autre instance ne peut agir sur la structure de données.

La plupart des microprocesseurs prévoient l'atomicité des opérations sur les bits et les entiers, encore faut-il les encapsuler dans des macros et fonctions de Linux. Pour compléter ceci, Linux utilise des verrous rotatifs et des sémaphores.

10.7.1 Opérations atomiques sur les bits

Les opérations atomiques sur les bits constituent la base des concepts de verrous rotatifs et des sémaphores. Par conséquent, les instructions atomiques "Test and Set" sont nécessaires. Ces opérations portent les noms suivants dans Linux :

- `set_bit(nr, void *addr)` positionne à 1 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `clear_bit(nr, void *addr)` positionne à 0 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `change_bit(nr, void *addr)` change la valeur du bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `test_bit(nr, void *addr)` renvoie la valeur du bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `test_and_set_bit(nr, void *addr)` positionne à 1 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`. La valeur précédente de ce bit est renvoyée.
- `test_and_clear_bit(nr, void *addr)` positionne à 0 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`. La valeur précédente de ce bit est renvoyée.
- `test_and_change_bit(nr, void *addr)` change la valeur du bit numéro `nr` de l'emplacement mémoire d'adresse `addr`. La valeur précédente de ce bit est renvoyée.

Ces fonctions sont définies, dans le cas des microprocesseurs Intel, dans le fichier `linux/include/asm-386/bitops.h`:

Code Linux 2.6.10

```
27 /**
28  * set_bit - Positionne un bit en memoire de facon atomique
29  * @nr : le bit a positionner
30  * @addr : l'adresse a partir de laquelle le decompter
31  *
32  * Cette fonction est atomique et ne peut pas etre rearrangee. Voir __set_bit()
33  * si vous n'avez pas besoin d'une garantie d'atomicite.
34  *
35  * Note : il n'y a aucune garantie que cette fonction ne sera pas rearrangee
36  * sur les architectures non x86, donc si vous etes en train d'ecrire du code portable,
37  * assurez-vous de ne pas vous fier a cette garantie.
38  *
39  * Noter que @nr peut etre presque arbitrairement aussi grand que l'on veut ; cette fonction
40  * n'est pas restreinte a agir sur un mot.
41  */
42 static inline void set_bit(int nr, volatile unsigned long * addr)
43 {
44         __asm__ __volatile__( LOCK_PREFIX
45                 "btsl %1,%0"
46                 :"=m" (ADDR)
47                 :"Ir" (nr));
48 }
[...]
130 /**
131  * test_and_set_bit - Positionne un bit et renvoie son ancienne valeur
132  * @nr : Bit a positionner
133  * @addr : Adresse a partir de laquelle decompter
134  *
135  * Cette operation est atomique et ne peut pas etre rearrangee.
136  * Elle peut etre rearrangee sur des architectures autres que x86.
137  * Elle implique egalement une barriere de memoire.
138  */
139 static inline int test_and_set_bit(int nr, volatile unsigned long * addr)
140 {
141         int oldbit;
142
143         __asm__ __volatile__( LOCK_PREFIX
144                 "btsl %2,%1\n\tsbbl %0,%0"
145                 :"=r" (oldbit),"=m" (ADDR)
146                 :"Ir" (nr) : "memory");
147         return oldbit;
148 }
```

10.7.2 Opérations atomiques sur les entiers

Pour exécuter des opérations atomiques sur les entiers, il faut utiliser le type `atomic_t`. Ce type dépend de l'architecture; il est défini dans le fichier `linux/include/asm-i386/atomic.h` pour les microprocesseurs Intel:

Code Linux 2.6.10

```
 8 /*
 9  * Operations atomiques que le C ne peut pas nous garantir. Utile pour
10  * le denombrement de ressources etc.
11 */
[...]
19 /*
20  * Assurez-vous que gcc n'essaie pas d'etre intelligent en deplacant des choses autour
21  * de nous. Nous avons besoin d'utiliser _exactement_ l'adresse que l'utilisateur
22  * nous a donnee et non quelque alias qui contient les memes informations.
23  */
```

```
24 typedef struct { volatile int counter; } atomic_t;
```

On dispose des opérations suivantes :

- `ATOMIC_INIT()` permet d'initialiser un entier lors de sa déclaration.
- `atomic_set(atomic_t *var, int i)` affecte de façon atomique la valeur `i` à l'entier d'adresse `var`.
- `atomic_read(atomic_t *var)` renvoie de façon atomique la valeur de l'entier d'adresse `var`.
- `atomic_add(int i, atomic_t *var)` et `atomic_sub(int i, atomic_t *var)` s'utilisent pour l'addition et la soustraction atomiques.
- `atomic_inc(atomic_t *var)` et `atomic_dec(atomic_t *var)` s'utilisent pour l'incrémentation et la décrémentation atomiques.
- `atomic_..._and_test()` sont les opérations analogues à celles sur les bits.

Ces macros dépendent de l'architecture. Pour les microprocesseurs Intel, elles sont définies dans le fichier en-tête `linux/include/asm-i386/atomic.h` :

Code Linux 2.6.10

```
26 #define ATOMIC_INIT(i)  { (i) }
27
28 /**
29  * atomic_read - lit une variable atomique
30  * @v : pointeur du type atomic_t
31  *
32  * Lit atomiquement la valeur de @v.
33  */
34 #define atomic_read(v)          ((v)->counter)
35
36 /**
37  * atomic_set - positionne une variable atomique
38  * @v : pointeur du type atomic_t
39  * @i : valeur requise
40  *
41  * Positionne atomiquement la valeur de @v a @i.
42  */
43 #define atomic_set(v,i)         (((v)->counter) = (i))
44
45 /**
46  * atomic_add - ajoute un entier a une variable atomique
47  * @i : valeur entiere a ajouter
48  * @v : pointeur du type atomic_t
49  *
50  * Ajoute atomiquement @i a @v.
51  */
52 static __inline__ void atomic_add(int i, atomic_t *v)
53 {
54         __asm__ __volatile__(
55                 LOCK "addl %1,%0"
56                 :"=m" (v->counter)
57                 :"ir" (i), "m" (v->counter));
58 }
```

10.7.3 Verrous rotatifs

10.7.3.1 Notion

Si une portion critique de programme commence et si une variable de verrouillage, appelée **verrou rotatif** (*busy wait lock* en anglais), est déjà positionnée (par un autre processus), le processus attend que celle-ci se libère en la vérifiant dans une boucle sans fin (scrutation).

On gaspille ainsi du temps machine car le processus vérifie la variable de verrouillage continuellement. Il peut toutefois être plus efficace, sur un système multiprocesseur, d'effectuer une courte vérification au lieu d'appeler l'ordonnanceur. Il faut évidemment appliquer les verrous rotatifs pour de petites portions de programme.

10.7.3.2 Déclaration et initialisation

Sous Linux, les variables de verrouillage sont du type `spinlock_t`. Celui-ci est défini dans le fichier `linux/include/linux/spinlock.h`: Code Linux 2.6.10

```
89 #define SPINLOCK_MAGIC  0x1D244B3C
90 typedef struct {
91         unsigned long magic;
92         volatile unsigned long lock;
93         volatile unsigned int babble;
94         const char *module;
95         char *owner;
96         int oline;
97 } spinlock_t;
```

L'utilisation d'un verrou rotatif exige la création et l'initialisation d'une variable du type précédent de l'une des deux façons :

```
#include <linux/spinlock.h>
spinlock_t s = SPIN_LOCK_UNLOCKED;
```

ou :

```
#include <linux/spinlock.h>
spinlock_t s;
spinlock_init(&s);
```

La constante et la macro ci-dessus sont définis dans le fichier `linux/include/linux/spinlock.h`: Code Linux 2.6.10

```
98  #define SPIN_LOCK_UNLOCKED (spinlock_t) { SPINLOCK_MAGIC, 0, 10, __FILE__ , NULL, 0}
99
100 #define spin_lock_init(x) \
101         do { \
102                 (x)->magic = SPINLOCK_MAGIC; \
103                 (x)->lock = 0; \
104                 (x)->babble = 5; \
105                 (x)->module = __FILE__; \
106                 (x)->owner = NULL; \
107                 (x)->oline = 0; \
108         } while (0)
```

10.7.3.3 Gestion des verrous rotatifs

On peut demander un verrou rotatif au moyen des macros suivantes :

- `spin_lock(spinlock_t *s)` essaie d'occuper le verrou rotatif `s`. S'il n'est pas libre, il est mis en attente ou testé jusqu'à ce qu'il le soit. Le verrou rotatif est occupé dès qu'il se libère.
- `spin_lock_irqsave(spinlock_t *s, unsigned long flags)` fonctionne de la même façon que `spin_lock()`, mais en outre les interruptions sont bloquées et la valeur en cours du registre d'état du processeur est enregistrée dans la variable `flags`.

- `spin_lock_irq(spinlock_t *s)` fonctionne de la même façon que `spin_lock_irqsave()` mais la valeur du registre d'état du processeur n'est pas mise en sécurité. On part du principe que les interruptions sont déjà bloquées.
- `spin_lock_bh(spinlock_t *s)` essaie d'occuper le verrou de la même manière que `spin_lock()`, bien que l'exécution simultanée des parties basses soit bloquée.

Les macros suivantes permettent d'identifier la fin de la zone critique selon la zone d'utilisation proprement dite. Elles libèrent le verrou rotatif actuellement occupé :

- `spin_unlock(spinlock_t *s)` libère un verrou rotatif occupé.
- `spin_unlock_irqrestore(spinlock *s, unsigned long flags)` libère le verrou rotatif et autorise les interruptions, si elles étaient activées lors de la mise en sécurité du registre d'état du processeur dans la variable `flags`, sinon non.
- `spin_unlock_irq(spinlock_t *s)` libère le verrou rotatif et autorise les interruptions.
- `spin_unlock_bh(spinlock_t *s)` libère le verrou et autorise le traitement immédiat des parties basses.

Les macros suivantes permettent d'accéder aux verrous rotatifs :

- `spin_is_locked(spinlock_t *s)` interroge l'état en cours du verrou, sans le modifier. La valeur renvoyée est non nulle lorsque le verrou est en place, nulle s'il est libre.
- `spin_trylock(spinlock_t *s)` occupe le verrou rotatif s'il ne l'était pas déjà lors de l'appel. Sinon une valeur non nulle est renvoyée.
- `spin_unlock_wait(spinlock_t *s)` attend la libération d'un verrou occupé. Celui-ci n'est pas occupé cependant.

10.7.4 Verrous rotatifs de lecture-écriture

Les verrous rotatifs constituent un élément simple et utile de protection contre les effets secondaires indésirables lors d'opérations parallèles sur la plupart des structures de données. Toutefois, ils freinent la progression des activités, parce que celles-ci doivent attendre la libération des verrous par scrutation. Dans certaines situations, l'attente active n'est pas toujours nécessaire. Il existe, par exemple, des structures de données à lesquelles on accède très souvent en lecture, mais rarement en écriture. Il ne s'agit pas d'arrêter les activités en lecture seule si aucune activité d'écriture n'opère à ce moment sur la structure de données.

Dans ce cas, on utilise des **verrous rotatifs de lecture-écriture** qui autorisent plusieurs activités de lecture à pénétrer dans une zone critique si aucune activité d'écriture n'est en cours. Cependant, dès qu'une activité occupe le verrou dans un but d'écriture, aucune activité de lecture ou autre activité d'écriture ne doit se trouver dans la zone critique jusqu'à ce que le verrou ait été libéré.

Les verrous de lecture-écriture sont implémentés grâce au type `rwlock_t`. Celui-ci, qui dépend de l'architecture, est défini dans le fichier `linux/include/asm-i386/spinlock.h`, dans le cas des microprocesseurs Intel :

Code Linux 2.6.10

```
154 /*
155  * Verrous rotatifs de lecture-ecriture, permettant plusieurs lectures
156  * mais une seule ecriture.
157  *
158  * NOTE ! il est relativement frequent d'avoir plusieurs lectures dans une interruption
159  * mais pas d'ecriture. Pour ces raisons nous
160  * pouvons "mixer" des verrous irq-safe - toute ecriture necessitant d'obtenir
```

```
161  * un verrou en ecriture irq-safe, mais les lectures peuvent obtenir
162  * des verrous en lecture non-irqsafe.
163  */
164 typedef struct {
165         volatile unsigned int lock;
166 #ifdef CONFIG_DEBUG_SPINLOCK
167         unsigned magic;
168 #endif
169 } rwlock_t;
170
171 #define RWLOCK_MAGIC    0xdeaf1eed
```

L'utilisation d'un verrou rotatif de lecture-écriture exige la création et l'initialisation d'une variable du type précédent de l'une des deux façons :

```
#include <linux/spinlock.h>
rwlock_t s = RW_LOCK_UNLOCKED;
```

ou :

```
#include <linux/spinlock.h>
rwlock_t s;
rwlock_init(&s);
```

La constante et la macro ci-dessus dépendent de l'architecture. Elles sont définies dans le fichier `linux/include/asm-i386/spinlock.h` dans le cas des microprocesseurs Intel : Code Linux 2.6.10

```
173 #ifdef CONFIG_DEBUG_SPINLOCK
174 #define RWLOCK_MAGIC_INIT       , RWLOCK_MAGIC
175 #else
176 #define RWLOCK_MAGIC_INIT       /* */
177 #endif
178
179 #define RW_LOCK_UNLOCKED (rwlock_t) { RW_LOCK_BIAS RWLOCK_MAGIC_INIT }
180
181 #define rwlock_init(x)  do { *(x) = RW_LOCK_UNLOCKED; } while(0)
```

Les fonctions disponibles pour l'occupation et la libération des verrous de lecture-écriture sont les suivantes :

- `read_lock()` essaie d'entrer dans une zone critique à des fins de lecture. La pénétration dans cette zone s'effectue immédiatement s'il ne s'y trouve aucune activité ou des activités de lecture seule. Si une activité d'écriture se trouve dans la zone critique, l'activité attend que le verrou soit libéré.
- `read_unlock()` quitte la zone critique dans laquelle elle était entrée en lecture seule.
- `write_lock()` essaie d'entrer dans une zone critique à des fins d'écriture. Si une activité (en écriture ou en lecture) se trouve dans la zone critique, l'activité attend que toutes les activités aient quitté la zone. L'occupation de la zone critique s'effectue alors de façon exclusive.
- `write_unlock()` fournit le verrou (en écriture) et libère ainsi la zone critique.

Il existe également différentes expressions des fonctions de verrous de lecture-écriture tenant compte de l'utilisation des interruptions et des parties basses (`..._irq()`, `.../bh()`, rtc.).

10.7.5 Sémaphore

L'inconvénient des verrous est qu'on utilise du temps machine pour la scrutation. Une autre façon est de faire appel à l'ordonnanceur. Ce concept est connu en informatique sous le nom de **sémaphore**. On parle de **mutex** dans le cas binaire (deux positions seulement).

Les sémaphores sont réalisés à l'aide de la structure de données `semaphore`. Le type, qui dépend de l'architecture, est défini dans le fichier `linux/include/asm-i386/semaphore.h`, pour les microprocesseurs Intel :

Code Linux 2.6.10

```
44 struct semaphore {
45         atomic_t count;
46         int sleepers;
47         wait_queue_head_t wait;
48 };
```

On initialise un sémaphore grâce à la macro `DECLARE_MUTEX()`.

On abaisse et on lève un sémaphore, au sens d'une barrière, grâce aux fonctions en ligne `down()` et `up()`.

10.8 Modules

Linux est un système d'exploitation à couches mais sans micro-noyau, considérant que les communications entre les extensions du micro-noyau sont beaucoup trop onéreuses en temps. Par contre il utilise la notion de **modules**, c'est-à-dire des portions de code du noyau qui peuvent être ajoutées ou retirées sans avoir à redémarrer le système, depuis la version 2.0.

Pour la suite, nous n'avons pas besoin de savoir comment ces modules sont gérés en interne, mais nous avons besoin de connaître comment concevoir et mettre en place de tels modules.

Le source d'un module s'écrit en langage C. Écrivons un module, de nom `hello.c`, qui se contente d'afficher 'Bonjour' lorsqu'on le charge et 'Au revoir' lorsqu'on le retire :

```
#define MODULE
#include <linux/module.h>

int init_module(void)
{
  printk("Bonjour\n");
}

void cleanup_module(void)
{
  printk("Au revoir\n");
}
```

La compilation du module se fait de façon habituelle, en ne générant le code objet :

```
root# gcc -c hello.c
```

On obtient ainsi un module de nom `hello.o`.

Le chargement d'un module s'effectue grâce à la commande `insmod` (pour *INSert MODule*) :

```
root# insmod ./hello.c
Bonjour
root#
```

Le retrait d'un module s'effectue grâce à la commande `rmmod` (pour *ReMove MODule*) :

```
root# rmmod hello
Au revoir
root#
```

On retrouvera souvent, dans la définition des structures, un champ déclaré de la façon suivante :

```
struct lodule *owner;
```

Il représente un pointeur sur le module qui est "propriétaire" de cette structure. Il est utilisé par le noyau pour conserver le comptage d'utilisation du module.

Le type `struct module` est défini dans le fichier en-tête `linux/scripts/modpost.h`: Code Linux 2.6.10

```
71 struct module {
72         struct module *next;
73         const char *name;
74         struct symbol *unres;
75         int seen;
76         int skip;
77         int has_init;
78         int has_cleanup;
79         struct buffer dev_table_buf;
80         char         srcversion[25];
81 };
```

10.9 Initialisation du système Linux

10.9.1 Fonctions d'initialisation

Les fonctions qui ne servent que pour l'initialisation du système Linux sont indiquées grâce au modificateur `__init`, comme indiqué dans le fichier `linux/include/linux/init.h`: Code Linux 2.6.10

```
7  /* Ces macros sont utilisees pour designer certaines fonctions ou
8   * donnees initialisees (ne s'applique pas aux donnees non initialisees)
9   * comme fonctions d''initialisation'. Le noyau peut prendre ceci comme
10  * une indication que la fonction est utilisee seulement durant la phase d'initialisation
11  * et liberer la memoire utilisee par les ressources apres.
12  *
13  * Utilisation :
14  * Pour les fonctions :
15  *
16  * Vous devez ajouter __init immediatement avant le nom de la fonction, comme ceci :
17  *
18  * static void __init initme(int x, int y)
19  * {
20  *    extern int z; z = x * y;
21  * }
22  *
23  * Si la fonction a un prototype quelque part, vous devez aussi ajouter
24  * __init entre la parenthese fermante du prototype et la virgule :
25  *
26  * extern int initialize_foobar_device(int, int, int) __init;
27  *
28  * Pour les donnees initialisees :
29  * Vous devez inserer __initdata entre le nom de la variable name et le signe
30  * d'egalite suivi par la valeur, par exemple :
31  *
32  * static int init_variable __initdata = 0;
33  * static char linux_logo[] __initdata = { 0x32, 0x36, ... };
34  *
35  * N'oubliez pas d'initialiser la donnee non pas dans la portee, c'est-a-dire dans une
36  * fonction, car sinon gcc place la donnee dans la section bss section et non dans
37  * la section init.
38  *
39  * Noter aussi que cette donnee ne peut pas etre "const".
40  */
```

10.9.2 Code d'initialisation

Lorsqu'on rencontre `module_init()` dans une partie du code source, la fonction indiquée entre les deux parenthèses sera placée dans un fichier spécial qui servira à initialiser le système lors du démarrage. Cette macro est définie dans le fichier `linux/include/linux/init.h`:

Code Linux 2.6.10

```
148 /**
149  * module_init() - point d'entree d'initialisation d'un peripherique
150  * @x : fonction a faire tourner lors du demarrage du noyau ou de l'insertion du module
151  *
152  * module_init() sera appele soit durant do_initcalls (si
153  * construit) ou au moment de l'insertion du module (s'il s'agit d'un module).
154  * Il peut seulement y en avoir une par module.
155  */
156 #define module_init(x)  __initcall(x);
```

La macro auxiliaire `__initcall()` est définie dans le même fichier :

Code Linux 2.6.10

```
78 /* Les initcalls sont maintenant groupes par fonctionalites dans des sous-sections
79  * separees. L'ordre a l'interieur des sous-sections est determine
80  * par l'ordre des liens.
81  * Par compatibilite ascendante, initcall() place l'appel dans la
82  * sous-section init du peripherique.
83  */
84
85 #define __define_initcall(level,fn) \
86         static initcall_t __initcall_##fn __attribute_used__ \
87         __attribute__((__section__(".initcall" level ".init"))) = fn
[...]
94 #define device_initcall(fn)             __define_initcall("6",fn)
[...]
97 #define __initcall(fn) device_initcall(fn)
```