Chapitre 15

Détection d'une carte réseau (1) Aspect général

Comme nous l'avons vu, la première action pour mettre en place le sous-système réseau est de détecter la carte réseau, déterminer ses paramètres (port d'entrée-sortie, IRQ, canal DMA, adresse MAC...) et lui associer un pilote de périphérique.

Comme nous l'avons vu dans le chapitre précédent, un pilote de périphérique réseau est implémenté sous la forme d'une structure (pour ne pas dire d'une classe au sens de la programmation orientée objet) possédant un certain nombre de champs, qui sont soit des attributs, soit des méthodes. Lors de la phase de démarrage, le pilote est initialisé et enregistré dans le système d'exploitation. Ensuite on gère les entrées-sorties du périphérique avec les appels système et les interruptions.

Nous allons étudier l'initialisation dans ce chapitre, c'est-à-dire voir comment l'on associe un nom à une carte réseau, comment on lui associe un index et comment on place les adresses des fonctions qui seront utiles lors de la partie dynamique (envoi et réception de trames) dans le descripteur de périphérique réseau associé.

Nous avons au chapitre 14 qu'une carte réseau est décrite par un descripteur de périphérique réseau, de type `struct net_device`, et que les descripteurs de ce type sont placés dans une liste chaînée `dev_base`. La première étape consiste à créer et à initialiser la structure `net_device` associée au périphérique. Ceci peut se faire de deux manières différentes :

- S'il a été décidé lors de la configuration du noyau que le pilote de périphérique réseau doit être intégré au noyau de manière fixe, une structure `net_device` pour celui-ci existe dans le fichier `linux/drivers/net/Space.c`. Les descripteurs de périphérique prévus à l'amorçage pour les cartes réseau prédéfinies sont créées grâce à un mécanisme comportant des définitions de préprocesseur, en fonction de la configuration du noyau.

  Par exemple huit descripteurs de périphérique réseau, qui ne sont à ce moment attribués à aucune carte, sont créés pour les périphériques réseau Ethernet.
- Si le pilote est écrit en tant que module du noyau, celui-ci doit le créer chaque fois que le module est appelé.

15.1 Première étape

15.1.1 Cas des périphériques liés statiquement

15.1.1.1 Démarrage

Tout commence par le lancement de l'initialisation des périphériques prévu lors du démarrage du système grâce à un certain nombre d'appels à la fonction générale `device_initcall()` répartie tout au long du code. Pour les périphériques réseau, il s'agit de la ligne 457 du fichier `linux/drivers/net/Space.c`:

Code Linux 2.6.10

```
423 device_initcall(net_olddevs_init);
```

qui fait appel à une fonction `net_olddevs_init()` définie juste au-dessus dans le même fichier :

Code Linux 2.6.10

```
389 /*  Pilotes configures statiquement -- mettre un peu d'ordre ici. */
390 static int __init net_olddevs_init(void)
391 {
392         int num;
393
394         if (loopback_init()) {
395                 printk(KERN_ERR "Network loopback device setup failed\n");
396         }
397
398
399 #ifdef CONFIG_SBNI
400         for (num = 0; num < 8; ++num)
401                 sbni_probe(num);
402 #endif
403 #ifdef CONFIG_TR
404         for (num = 0; num < 8; ++num)
405                 if (!trif_probe(num))
406                         trif_probe2(num);
407 #endif
408         for (num = 0; num < 8; ++num)
409                 ethif_probe2(num);
410
411 #ifdef CONFIG_COPS
412         cops_probe(0);
413         cops_probe(1);
414         cops_probe(2);
415 #endif
416 #ifdef CONFIG_LTPC
417         ltpc_probe();
418 #endif
419
420         return 0;
421 }
```

dont le but est d'initialiser le périphérique en boucle, huit périphériques Ethernet (lignes 408–409) et éventuellement d'autres périphériques suivant les options de compilation du noyau.

Nous reviendrons plus tard sur les détails de l'initialisation des huit périphériques Ethernet. Intéressons-nous pour l'instant au périphérique en boucle (*loopback* en anglais) à titre d'exemple.

15.1.1.2 Descripteur du périphérique en boucle

L'instance de structure `net_device` pour le périphérique *loopback*, à savoir `loopback_dev`, est définie dans le fichier `linux/drivers/net/loopback.c`:

Code Linux 2.6.10

```
198 struct net_device loopback_dev = {
199         .name                   = "lo",
```

```
200         .mtu                    = (16 * 1024) + 20 + 20 + 12,
201         .hard_start_xmit        = loopback_xmit,
202         .hard_header            = eth_header,
203         .hard_header_cache      = eth_header_cache,
204         .header_cache_update    = eth_header_cache_update,
205         .hard_header_len        = ETH_HLEN,     /* 14   */
206         .addr_len               = ETH_ALEN,     /* 6    */
207         .tx_queue_len           = 0,
208         .type                   = ARPHRD_LOOPBACK,      /* 0x0001*/
209         .rebuild_header         = eth_rebuild_header,
210         .flags                  = IFF_LOOPBACK,
211         .features               = NETIF_F_SG|NETIF_F_FRAGLIST
212                                   |NETIF_F_NO_CSUM|NETIF_F_HIGHDMA
213                                   |NETIF_F_LLTX,
214         .ethtool_ops            = &loopback_ethtool_ops,
215 };
```

Les valeurs des champs sont presque toutes liées à Ethernet (`eth_header()`, `eth_header_cache()`, `eth_header_cache_update()`, `eth_rebuild_header()`, `ETH_HLEN` et `ETH_ALEN`). Nous y reviendrons à propos de l'implémentation d'Ethernet au fur et à mesure des besoins. Seuls le nom, la fonction `loopback_xmit()` et l'ensemble d'opérations `loopback_ethtool_ops` sont propres au périphérique en boucle.

15.1.1.3 Initialisation et enregistrement du périphérique en boucle

La fonction d'initialisation et d'enregistrement du périphérique en boucle, `loopback_init()` dont il est fait référence ci-dessus, est définie dans le même fichier :

Code Linux 2.6.10

```
217 /* Initialise et enregistre le peripherique LOOPBACK. */
218 int __init loopback_init(void)
219 {
220         struct net_device_stats *stats;
221
222         /* Peut survivre sans statistiques */
223         stats = kmalloc(sizeof(struct net_device_stats), GFP_KERNEL);
224         if (stats) {
225                 memset(stats, 0, sizeof(struct net_device_stats));
226                 loopback_dev.priv = stats;
227                 loopback_dev.get_stats = &get_stats;
228         }
229
230         return register_netdev(&loopback_dev);
231 };
```

Autrement dit :

- On déclare une structure pour les statistiques. On essaie de réserver de la place en mémoire centrale pour celle-ci. Si on n'y parvient pas, ce n'est pas grave.
- Si on y parvient, on initialise cet emplacement mémoire à zéro et on renseigne les deux champs du descripteur de périphérique concernant les statistiques.

  La fonction `get_stats()` pour le périphérique en boucle est définie dans le même fichier :

  Code Linux 2.6.10

```
161 static struct net_device_stats *get_stats(struct net_device *dev)
162 {
163         struct net_device_stats *stats = dev->priv;
164         int i;
165
166         if (!stats) {
167                 return NULL;
```

```
168         }
169
170         memset(stats, 0, sizeof(struct net_device_stats));
171
172         for (i=0; i < NR_CPUS; i++) {
173                 struct net_device_stats *lb_stats;
174
175                 if (!cpu_possible(i))
176                         continue;
177                 lb_stats = &per_cpu(loopback_stats, i);
178                 stats->rx_bytes   += lb_stats->rx_bytes;
179                 stats->tx_bytes   += lb_stats->tx_bytes;
180                 stats->rx_packets += lb_stats->rx_packets;
181                 stats->tx_packets += lb_stats->tx_packets;
182         }
183
184         return stats;
185 }
```

Autrement dit :

- On essaie de récupérer la partie privée du descripteur de périphérique en boucle. Si on n'y parvient pas, on renvoie `NULL`.
- On réinitialise à zéro cette partie de la mémoire vive.
- On récupère les données concernant le périphérique en boucle pour chaque microprocesseur, on les cumule et on renvoie l'adresse de la partie privée.

– On enregistre le pilote grâce à la fonction `register_netdev()` étudiée ci-dessous dans la deuxième étape.

15.1.2 Cas des périphériques modularisés

15.1.2.1 Déclaration du descripteur de périphérique réseau

Dans le cas d'un périphérique modularisé, un descripteur de périphérique est déclaré quelque part. Par exemple le descripteur du périphérique réseau 3Com 501 est déclaré dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
891 #ifdef MODULE
892
893 static struct net_device *dev_3c501;
894
895 MODULE_PARM(io, "i");
896 MODULE_PARM(irq, "i");
897 MODULE_PARM_DESC(io, "EtherLink I/O base address");
898 MODULE_PARM_DESC(irq, "EtherLink IRQ number");
```

15.1.2.2 Module d'initialisation

Chaque pilote de carte réseau modularisé possède un module d'initialisation, appelé soit lors du démarrage du système, soit lors de l'appel d'un module.

Pour la carte 3Com 501, que nous prenons en exemple, ce module est situé à la fin du fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
900 /**
901  * init_module :
902  *
903  * Cette fonction est appelee lorsque le pilote est charge en tant que module. Nous
```

```
904  * improvisons une structure de peripherique avec l'adresse de base d'I/O et l'interruption
       positionnes comme si il avait ete appele
905  * depuis Space.c. Ceci minimise le code supplementaire qui serait requis
906  * sinon.
907  *
908  * Renvoie 0 en cas de succes et -EIO si la carte n'est pas trouvee. Renvoyer une
909  * erreur ici a pour consequence de ne pas charger le module
910  */
911
912 int init_module(void)
913 {
914         dev_3c501 = el1_probe(-1);
915         if (IS_ERR(dev_3c501))
916                 return PTR_ERR(dev_3c501);
917         return 0;
918 }
```

qui fait essentiellement appel à la fonction `el1_probe()`, sur laquelle nous reviendrons au chapitre 17. Retenons seulement, pour l'instant, qu'elle fait, comme le périphérique en boucle, appel à la fonction `register_netdev()`.

Les fonctions en ligne générales `IS_ERR()` et `PTR_ERR()` sont définies dans le fichier `linux/-include/linux/err.h`:

Code Linux 2.6.10

```
21 static inline long PTR_ERR(const void *ptr)
22 {
23         return (long) ptr;
24 }
25
26 static inline long IS_ERR(const void *ptr)
27 {
28         return unlikely((unsigned long)ptr > (unsigned long)-1000L);
29 }
```

15.2 Deuxième étape : allocation d'un nom

La deuxième étape de l'installation d'une carte réseau, après celle de la création du descripteur de périphérique, consiste à lui allouer un nom. Cette seconde étape est effectuée par la fonction `register_netdev()`, dont nous venons de voir qu'elle est appelée soit lors de l'initialisation (cas du périphérique en boucle par exemple), soit à partir du module d'initialisation dans le cas des pilotes modularisés.

15.2.1 Allocation du nom et passage à la troisième étape

15.2.1.1 Enregistrement d'un pilote de périphérique réseau

La fonction `register_netdev()` d'enregistrement d'un pilote (plus exactement d'un descripteur) de périphérique réseau est définie dans le fichier `linux/drivers/net/net_init.c`:

Code Linux 2.6.10

```
108 int register_netdev(struct net_device *dev)
109 {
110         int err;
111
112         rtnl_lock();
113
114         /*
115          *      Si le nom est une chaine de caracteres format, l'appelant veut que
116          *      nous allouions un nom
117          */
```

```
118
119         if (strchr(dev->name, '%'))
120         {
121                 err = dev_alloc_name(dev, dev->name);
122                 if (err < 0)
123                         goto out;
124         }
125
126         /*
127          *      Point d'ancrage de compatibilite ascendante. Enlever ceci dans 2.5
128          */
129
130         if (dev->name[0]==0 || dev->name[0]==' ')
131         {
132                 err = dev_alloc_name(dev, "eth%d");
133                 if (err < 0)
134                         goto out;
135         }
136
137         err = register_netdevice(dev);
138
139 out:
140         rtnl_unlock();
141         return err;
142 }
```

Elle a pour but d'allouer un nom au périphérique et d'appeler l'étape suivante. Plus précisément :

– On se réserve le maniement de la liste chaînée des périphériques réseau car on va en avoir besoin pour vérifier les noms existants.

Code Linux 2.6.10

Comme indiqué dans le fichier `linux/net/core/dev.c` :

```
166 /*
167  * La liste @dev_base est protegee par @dev_base_lock et le
168  * semaphore rtln.
169  *
170  * Pour les lectures pures, on doit detenir dev_base_lock pour lire.
171  *
172  * Pour les ecritures, on doit detenir le semaphore rtnl tant qu'on parcourt la
173  * liste dev_base, et detenir dev_base_lock pour ecrire lorsqu'on fait les
174  * mises a jour effectives. Ceci permet aux lecteurs pures d'acceder a la liste meme
175  * lorsqu'une ecriture se prepare a la mettre a jour.
176  *
177  * En d'autres mots, dev_base_lock est detenu pour l'ecriture seulement pour
178  * se proteger contre les lectuers pures ; le semaphore rtnl fournit la
179  * protection contre les autres ecritures.
180  *
181  * Voir, pour des exemples d'utilisation, register_netdevice() et
182  * unregister_netdevice(), qui doivent etre appeles avec le semaphore rtnl
183  * detenu.
184  */
```

l'écriture et la lecture dans cette liste sont protégées par le verrou rotatif `dev_base_lock` et le sémaphore `rtln_sem`.

Code Linux 2.6.10

Ceux-ci sont définis dans les fichiers `linux/net/core/dev.c` :

```
187 rwlock_t dev_base_lock = RW_LOCK_UNLOCKED;
```

Code Linux 2.6.10

et `linux/net/core/netlink.c` (`rtnl` signifiant *RouTing NetLink*) :

```
54 DECLARE_MUTEX(rtnl_sem);
55
```

```
56 void rtnl_lock(void)
57 {
58         rtnl_shlock();
59 }
```

La macro `rtnl_shlock()` est définie dans le fichier `linux/include/linux/netlink.h`: Code Linux 2.6.10

```
800 #define rtnl_shlock()           down(&rtnl_sem)
```

- Si le champ `name` du périphérique passé en argument ne comporte pas d'occurrence du caractère `'%'`, c'est que l'on veut que le contenu de ce champ soit le nom du périphérique, non suivi d'un numéro (cas de `lo` par exemple). On fait alors appel à la fonction `dev_alloc_name()`, étudiée ci-dessous, pour vérifier que ce nom n'est pas déjà attribué à un autre périphérique et le lui attribuer si c'est bien le cas. Sinon cette fonction renvoie une erreur, on ne se réserve plus alors le maniement de la liste chaînée des périphériques réseau et on renvoie le code d'erreur fourni par cette fonction.
- Si le champ `name` commence par `'0'`, autrement dit si aucun nom n'a été attribué, ou par une espace, on fait appel à la même fonction mais le nom du périphérique sera `'eth'` suivi d'un numéro. Si une erreur intervient au cours de cette allocation, on ne se réserve plus le maniement de la liste chaînée des périphériques réseau et on renvoie le code fourni par cette fonction.
- On passe à la troisième étape en faisant appel à la fonction `register_netdevice()` étudiée ci-dessous.
- On ne se réserve plus le maniement de la liste chaînée des périphériques réseau et on renvoie le code fourni par la fonction ci-dessus.

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

```
61 void rtnl_unlock(void)
62 {
63         rtnl_shunlock();
64
65         netdev_run_todo();
66 }
```

La macros `retnl_shunlock()` est définie dans le fichier `linux/include/linux/netlink.h`: Code Linux 2.6.10

```
803 #define rtnl_shunlock() do { up(&rtnl_sem); \
804                              if (rtnl && rtnl->sk_receive_queue.qlen) \
805                                      rtnl->sk_data_ready(rtnl, 0); \
806                          } while(0)
```

La variable `rtnl` est déclarée dans le fichier `linux/net/core/rtnetlink.c`: Code Linux 2.6.10

```
81 struct sock *rtnl;
```

dont le type `struct sock` sera étudié plus tard. Seule l'adresse est importante pour l'instant.

Enfin la fonction `netdev_run_todo()` correspond à la quatrième étape et sera étudiée ci-dessous.

15.2.2 Manipulation d'un périphérique désigné par un nom

15.2.2.1 Allocation en précisant le nom

La fonction `dev_alloc_name()` détermine un nom pour le périphérique à partir de celui qui est passé en argument et elle vérifie qu'il n'est pas déjà attribué. Elle est définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
655 /**
656  *      dev_alloc_name - alloue un nom a un peripherique
657  *      @dev : peripherique
658  *      @name: chaine de caracteres directive de format du nom
659  *
660  *      Passe une directive de format - par exemple "lt%d" - elle essaiera de trouver
670  *      un id convenable. Pas efficace pour certains peripheriques, ne pas appeler beaucoup.
671  *      L'appelant doit detenir le verrou dev_base ou rtnl pendant l'allocation du nom
672  *      et l'ajout du peripherique afin d'eviter les doublons. Renvoie le numero
673  *      de l'unite assignee ou un code errno negatif.
665  */
666
667 int dev_alloc_name(struct net_device *dev, const char *name)
668 {
669         int i = 0;
670         char buf[IFNAMSIZ];
671         const char *p;
672         const int max_netdevices = 8*PAGE_SIZE;
673         long *inuse;
674         struct net_device *d;
675
676         p = strnchr(name, IFNAMSIZ-1, '%');
677         if (p) {
678                 /*
679                  * Verifie la chaine de caracteres car elle peut provenir de
680                  * l'utilisateur. Elle doit soit contenir "%d" et pas d'autre caracteres "%",
681                  * ou aucun caractere "%" du tout.
682                  */
683                 if (p[1] != 'd' || strchr(p + 2, '%'))
684                         return -EINVAL;
685
686                 /* Utilise une page en tant que tableau de bits des slots possibles */
687                 inuse = (long *) get_zeroed_page(GFP_ATOMIC);
688                 if (!inuse)
689                         return -ENOMEM;
690
691                 for (d = dev_base; d; d = d->next) {
692                         if (!sscanf(d->name, name, &i))
693                                 continue;
694                         if (i < 0 || i >= max_netdevices)
695                                 continue;
696
697                         /*  evite les cas ou sscanf n'est pas l'inverse exact de printf */
698                         snprintf(buf, sizeof(buf), name, i);
699                         if (!strncmp(buf, d->name, IFNAMSIZ))
700                                 set_bit(i, inuse);
701                 }
702
703                 i = find_first_zero_bit(inuse, max_netdevices);
704                 free_page((unsigned long) inuse);
705         }
706
707         snprintf(buf, sizeof(buf), name, i);
708         if (!__dev_get_by_name(buf)) {
709                 strlcpy(dev->name, buf, IFNAMSIZ);
```

```
710                 return i;
711         }
712
713         /* Il est possible d'aller au-dela des slots possibles
714          * lorsque le nom est long et qu'il n'y a pas assez d'espace laisse
715          * pour les chiffres ou si tous les bits sont utilises.
716          */
717         return -ENFILE;
718 }
```

Autrement dit :

- On déclare le numéro du périphérique qui sera assigné, une chaîne de caractères qui contiendra le nom du périphérique, une chaîne de caractères qui contiendra le nom passé en argument, le nombre maximum de périphériques permis, l'adresse de la page qui sera utilisée en tant que tableau de bits des emplacements possibles et un descripteur de périphérique réseau.
- On recherche l'adresse de la première occurrence du caractère `'%'` dans la chaîne de caractères passée en argument. Si celle-ci existe :
  - si elle n'est pas suivie du caractère `'d'` ou s'il existe une autre occurrence de `'%'`, on renvoie l'opposé du code d'erreur `EINVAL`.
  - On recherche une page non utilisée (qui ne contient que des zéros). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOMEM`.
  - On décrit la liste chaînée des descripteurs de périphérique réseau, on place dans `buf` la chaîne de caractères obtenue à partir de `name` en remplaçant l'occurrence de `'%d'` par l'index `i`, s'il existe un périphérique portant le nom ainsi obtenu, on positionne le bit correspondant de la page à 1.
  - On détermine la position du premier bit nul de la page et on libère la page.
- On place dans `buf` la chaîne de caractères obtenue à partir de `name` en remplaçant l'occurrence de `'%d'` par l'index `i`.
- Si le nom obtenu est déjà utilisé par un autre périphérique, on renvoie l'opposé du code d'erreur `ENFILE`. La fonction `__dev_get_by_name()` sera étudiée ci-dessous.
- Sinon on renseigne le champ `name` du descripteur de périphérique passé en argument avec ce nom.

15.2.2.2 Recherche d'un périphérique réseau par son nom

La fonction `__dev_get_by_name()` renvoie l'adresse du descripteur du périphérique dont le nom est celui passé en argument s'il existe et `NULL` sinon. Elle est définie dans le fichier `linux/net/core/dev.c` :

Code Linux 2.6.10

```
476 /**
477  *      __dev_get_by_name       - trouver un peripherique grace a son nom
478  *      @name : nom a trouver
479  *
480  *      Trouver une interface grace a son nom. Doit etre appelee avec le semaphore RTNL
481  *      ou @dev_base_lock. Si le nom est trouve, un pointeur au peripherique
482  *      est renvoye. Si le nom n'est pas trouve alors %NULL est renvoye. Les
483  *      compteurs de reference ne sont pas incrementes, aussi l'appelant doit-il etre
484  *      soigneux avec les verrous.
485  */
486
487 struct net_device *__dev_get_by_name(const char *name)
```

```
488 {
489         struct hlist_node *p;
490
491         hlist_for_each(p, dev_name_hash(name)) {
492                 struct net_device *dev
493                         = hlist_entry(p, struct net_device, name_hlist);
494                 if (!strncmp(dev->name, name, IFNAMSIZ))
495                         return dev;
496         }
497         return NULL;
498 }
```

Autrement dit on décrit la liste chaînée des périphériques réseau d'index de hachage correspondant au nom passé en argument à la recherche d'un descripteur de périphérique réseau dont le nom coïncide avec celui passé en argument. Si on le trouve, on renvoie l'adresse de ce descripteur.

Remarquons l'utilisation d'une astuce: si on ne trouve pas le périphérique, la variable `dev` prendra la valeur `NULL` à la fin de la liste.

Code Linux 2.6.10

Les macros générales `hlist_for_each()` et `hlist_entry()` sont définies dans le fichier `linux/include/linux/list.h`:

```
631 #define hlist_entry(ptr, type, member) container_of(ptr,type,member)
632
633 #define hlist_for_each(pos, head) \
634         for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
635              pos = pos->next)
```

Code Linux 2.6.10

La fonction en ligne `dev_name_hash()` est définie dans le fichier `linux/net/core/dev.c`:

```
192 #define NETDEV_HASHBITS 8
193 static struct hlist_head dev_name_head[1<<NETDEV_HASHBITS];
194 static struct hlist_head dev_index_head[1<<NETDEV_HASHBITS];
195
196 static inline struct hlist_head *dev_name_hash(const char *name)
197 {
198         unsigned hash = full_name_hash(name, strnlen(name, IFNAMSIZ));
199         return &dev_name_head[hash & ((1<<NETDEV_HASHBITS)-1)];
200 }
```

La fonction générale:

`unsigned int full_name_hash(const unsigned char *name, unsigned int len)`

permet de calculer l'index de hachage d'une chaîne de caractères.

15.3 Attribution d'un index et insertion dans la liste

La troisième étape de l'installation d'un périphérique réseau consiste à lui attribuer un index et à insérer son descripteur dans la liste chaînée des descripteurs de périphériques réseau ainsi que dans la table de hachage.

La liste `dev_base` gère les périphériques réseau. Ils doivent tous être enregistrés dans cette liste, qu'ils soient ou non activés. Par contre, ils ne doivent être enregistrés que si la carte réseau correspondante est réellement disponible sur le système informatique. C'est pourquoi on vérifie avant l'insertion qu'il est possible de trouver une carte réseau pour le pilote. Pour ce faire, chacun des descripteurs de périphériques comporte une fonction `init()`. Elle est très spécifique à la carte. Les périphériques de réseau logiques comme *loopback* (`lo`) ou PPP (`ppp0`), qui ne sont pas assignés au matériel sous-jacent, constituent une exception.

Cette étape débute par l'appel à la fonction `register_netdevice()`, effectuée depuis la fonction `register_netdev()`. Avant de l'étudier, nous allons parler des chaînes de notification.

15.3.1 Chaînes de notification

Les périphériques réseau sont déclarés et retirés dynamiquement. De plus, leut état peut changer au fil du temps : un périphérique réseau peut par exemple modifier son adresse matérielle, voire son nom. Au niveau des périphériques réseau, une modification d'état n'entraîne aucun problème. Mais ceci n'est pas le cas pour les couches supérieures. En effet, pour des raisons d'efficacité et de simplicité, les protocoles de celles-ci stockent fréquemment des renvois aux périphériques réseau dans des caches. Si l'état d'un périphérique change, ces états stockés ne sont plus valides. Les instances de protocoles doivent donc être informées de tout changement d'état. Ceci s'effectue à l'aide des **chaînes de notification** (*notifier-chain* en anglais).

15.3.1.1 Type associé

Les chaînes de notification sont implémentées sous la forme d'une liste `netdev_chain` de structures `notifier_block`.

La chaîne `netdev_chain` est déclarée dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
207 /*
208  *      Notre liste de notification
209  */
210
211 static struct notifier_block *netdev_chain;
```

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

```
1 /*
2  *      Routines pour gerer les chaines de notification, pour passer les changements de
3  *      statut a toutes les routines interessees. Nous en avons besoin au lieu des listes
4  *      codees en dur car les modules peuvent s'en meler. Les peripheriques reseau
5  *      en ont besoin.
6  *
7  *                              Alan Cox <Alan.Cox@linux.org>
8  */
[...]
14 struct notifier_block
15 {
16         int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
17         struct notifier_block *next;
18         int priority;
19 };
```

- Le champ `notifier_call()` est un pointeur sur une routine de traitement chargée de gérer la communication au cours du changement d'état du périphérique réseau. Toute instance de protocole devrait comporter une telle fonction de stockage d'états des périphériques réseau.

  Si un changement d'état a lieu dans un périphérique réseau, toutes les routines de traitement stockées dans la liste `netdev_chain` sont appelées. On passe les arguments suivants :

  - un pointeur sur la structure `notifier_block` dont la routine de traitement vient d'être appelée ;
  - un pointeur sur le descripteur de périphérique réseau (`net_device`) concerné, dont l'état change ;
  - l'état qui est à l'origine de la notification.

- `priority` spécifie une priorité, à partir de laquelle la notification sera traitée.

– `next` désigne le prochain élément dans la liste.

On emploie le concept de chaînes de notification pour les périphériques réseau, ce qui nous intéresse ici, mais également pour d'autres états susceptibles de changer. Cela explique pourquoi l'implémentation de ces chaînes de notification est générique. Il existe, par exemple, une chaîne `reboot_notifier_list` pour informer d'un redémarrage imminent du système.

Dans le domaine des réseaux, la liste suivante montre les causes possibles :

– `NETDEV_UP` : un périphérique réseau est activé (*via* `dev_open()`).
– `NETDEV_DOWN` : un périphérique réseau est désactivé.
– `NETDEV_CHANGE` : un périphérique réseau change d'état.
– `NETDEV_REGISTER` : un périphérique réseau a été enregistré mais aucune instance n'a encore été ouverte.
– `NETDEV_UNREGISTER` : un pilote de périphérique réseau a été retiré.
– `NETDEV_CHANGEMTU` : la MTU a été changée.
– `NETDEV_CHANGEADDR` : l'adresse matérielle d'un périphérique réseau a été changée.
– `NETDEV_CHANGENAME` : la désignation d'un périphérique réseau a été changée.

Code Linux 2.6.10

Ces constantes symboliques sont définies dans le fichier en-tête `linux/include/linux/notifier.h` :

```
45 /* chaines de notification des peripheriques reseau */
46 #define NETDEV_UP         0x0001  /* A partir de maintenant vous ne pouvez pas empecher
                                       un peripherique d'etre active/desactive */
47 #define NETDEV_DOWN       0x0002
48 #define NETDEV_REBOOT     0x0003  /* Dit a une pile de protocoles qu'une interface reseau
49                                       a detecte un accident materiel et qu'il a redemarre
50                                       - nous pouvons par exemple utiliser ceci pour bombarder
51                                       des sessions tcp une fois fait */
52 #define NETDEV_CHANGE     0x0004  /* Notifie un changement d'etat de peripherique */
53 #define NETDEV_REGISTER   0x0005
54 #define NETDEV_UNREGISTER 0x0006
55 #define NETDEV_CHANGEMTU  0x0007
56 #define NETDEV_CHANGEADDR 0x0008
57 #define NETDEV_GOING_DOWN 0x0009
58 #define NETDEV_CHANGENAME 0x000A
```

15.3.1.2 Appel de la chaîne de notification

Code Linux 2.6.10

L'appel de la chaîne de notification est mis en œuvre à l'aide de la fonction générale `notifier_call_chain()`, définie dans le fichier `linux/kernel/sys.c` :

```
149 /**
150  *      notifier_call_chain - Fonction d'appel d'une chaine de notification
151  *      @n : Pointeur au pointeur racine de la chaine de notification
152  *      @val : Valeur passee sans modification a la fonction de notification
153  *      @v : Pointeur passe sans modification a la fonction de notification
154  *
155  *      Appelle chaque fonction l'une apres l'autre dans une chaine de notification.
156  *
157  *      Si la valeur de retour de la notification peut etre mise en conjonction
158  *      avec %NOTIFY_STOP_MASK, alors notifier_call_chain
159  *      la renverra immediatement, avec la valeur de retour de
160  *      la fonction de notification qui a stoppe l'execution.
161  *      Sinon la valeur de retour est la valeur de retour
162  *      de la derniere fonction de notification appelee.
163  */
```

```
164
165 int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
166 {
167         int ret=NOTIFY_DONE;
168         struct notifier_block *nb = *n;
169
170         while(nb)
171         {
172                 ret=nb->notifier_call(nb,val,v);
173                 if(ret&NOTIFY_STOP_MASK)
174                 {
175                         return ret;
176                 }
177                 nb=nb->next;
178         }
179         return ret;
180 }
```

Elle informe toutes les méthodes de traitement enregistrées dans la liste `n` (`netdev_chain` pour nous) de l'événement `val`. Toutes les routines sont appelées les unes après les autres. Si une de ces routines renvoie la constante `NOTIFY_STOP_MASK`, la notification est interrompue. Les routines enregistrées décident si elles entreprennent quoi que ce soit à la suite de l'avertissement du changement d'état.

Les constantes symboliques générales `NOTIFY_DONE` et `NOTIFY_STOP_MASK` sont définies dans le fichier `linux/include/linux/notifier.h`:

Code Linux 2.6.10

```
28 #define NOTIFY_DONE             0x0000          /* Ne pas en tenir compte */
29 #define NOTIFY_OK               0x0001          /* Me convient */
30 #define NOTIFY_STOP_MASK        0x8000          /* Ne pas rappeler plus tard */
31 #define NOTIFY_BAD              (NOTIFY_STOP_MASK|0x0002)       /* Action Mauvais/Veto */
32 /*
33  * Nettoyer le chemin pour revenir du notificateur et arreter les appels utlerieurs.
34  */
35 #define NOTIFY_STOP             (NOTIFY_OK|NOTIFY_STOP_MASK)
```

15.3.2 Fonction principale

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

Code Linux 2.6.10

```
2687 /**
2688  *      register_netdevice      - enregistre un peripherique reseau
2689  *      @dev : peripherique a enregistrer
2690  *
2691  *      Prend une structure de peripherique reseau completee et l'ajoute aux interfaces
2692  *      du noyau. Un message %NETDEV_REGISTER est envoye a la chaine de notification
2693  *      netdev. 0 est renvoye en cas de succes. Un code errno negatif est renvoye
2694  *      en cas d'echec d'installation du peripherique ou si le nom est un doublon.
2695  *
2696  *      Les appelants doivent posseder le semaphore rtnl. Voir le commentaire a la
2697  *      fin de Space.c pour les details sur le verrouillage. Vous pouvez vouloir
2698  *      register_netdev() au lieu de celle-ci.
2699  *
2700  *      BOGUES :
2701  *      Le verrouillage apparait insuffisant pour garantir que deux registres paralleles
2702  *      ne porteront pas le meme nom.
2703  */
2704
2705 int register_netdevice(struct net_device *dev)
2706 {
2707         struct hlist_head *head;
2708         struct hlist_node *p;
```

```
2709         int ret;
2710
2711         BUG_ON(dev_boot_phase);
2712         ASSERT_RTNL();
2713
2714         /* Lorsque des net_device sont persistants, ceci sera fatal. */
2715         BUG_ON(dev->reg_state != NETREG_UNINITIALIZED);
2716
2717         spin_lock_init(&dev->queue_lock);
2718         spin_lock_init(&dev->xmit_lock);
2719         dev->xmit_lock_owner = -1;
2720 #ifdef CONFIG_NET_CLS_ACT
2721         spin_lock_init(&dev->ingress_lock);
2722 #endif
2723
2724         ret = alloc_divert_blk(dev);
2725         if (ret)
2726                 goto out;
2727
2728         dev->iflink = -1;
2729
2730         /* Initialiser, si cette fonction est disponible */
2731         if (dev->init) {
2732                 ret = dev->init(dev);
2733                 if (ret) {
2734                         if (ret > 0)
2735                                 ret = -EIO;
2736                         goto out_err;
2737                 }
2738         }
2739
2740         if (!dev_valid_name(dev->name)) {
2741                 ret = -EINVAL;
2742                 goto out_err;
2743         }
2744
2745         dev->ifindex = dev_new_index();
2746         if (dev->iflink == -1)
2747                 dev->iflink = dev->ifindex;
2748
2749         /* Verification de l'existence du nom */
2750         head = dev_name_hash(dev->name);
2751         hlist_for_each(p, head) {
2752                 struct net_device *d
2753                         = hlist_entry(p, struct net_device, name_hlist);
2754                 if (!strncmp(d->name, dev->name, IFNAMSIZ)) {
2755                         ret = -EEXIST;
2756                         goto out_err;
2757                 }
2758         }
2759
2760         /* Rectifier les combinaisons SG+CSUM illegales. */
2761         if ((dev->features & NETIF_F_SG) &&
2762             !(dev->features & (NETIF_F_IP_CSUM |
2763                                NETIF_F_NO_CSUM |
2764                                NETIF_F_HW_CSUM))) {
2765                 printk("%s: Dropping NETIF_F_SG since no checksum feature.\n",
2766                        dev->name);
2767                 dev->features &= ~NETIF_F_SG;
2768         }
2769
2770         /* TSO exige que SG soit egalement present. */
```

```
2771         if ((dev->features & NETIF_F_TSO) &&
2772             !(dev->features & NETIF_F_SG)) {
2773                 printk("%s: Dropping NETIF_F_TSO since no SG feature.\n",
2774                        dev->name);
2775                 dev->features &= ~NETIF_F_TSO;
2776         }
2777
2778         /*
2779          *      routine rebuild_header par defaut,
2780          *      qui ne devrait jamais etre appelee et seulement utilisee lors du debogage.
2781          */
2782
2783         if (!dev->rebuild_header)
2784                 dev->rebuild_header = default_rebuild_header;
2785
2786         /*
2787          *      L'etat initial par defaut lors de l'enregistrement est que le
2788          *      peripherique est present.
2789          */
2790
2791         set_bit(__LINK_STATE_PRESENT, &dev->state);
2792
2793         dev->next = NULL;
2794         dev_init_scheduler(dev);
2795         write_lock_bh(&dev_base_lock);
2796         *dev_tail = dev;
2797         dev_tail = &dev->next;
2798         hlist_add_head(&dev->name_hlist, head);
2799         hlist_add_head(&dev->index_hlist, dev_index_hash(dev->ifindex));
2800         dev_hold(dev);
2801         dev->reg_state = NETREG_REGISTERING;
2802         write_unlock_bh(&dev_base_lock);
2803
2804         /* Notifie aux protocoles qu'un nouveau peripherique est apparu. */
2805         notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);
2806
2807         /* Termine l'enregistrement apres le deverrouillage */
2808         net_set_todo(dev);
2809         ret = 0;
2810
2811 out:
2812         return ret;
2813 out_err:
2814         free_divert_blk(dev);
2815         goto out;
2816 }
```

Autrement dit :

- On déclare une liste de hachage, un élément d'une telle liste et un code de retour.
- On ne permet pas d'enregistrer deux périphériques réseau en même temps ; si on est déjà en train d'en enregistrer un autre, on attend jusqu'à ce qu'on ait terminé l'enregistrement de ce dernier.

  La variable `dev_boot_phase` est déclarée un peu plus haut dans le même fichier : Code Linux 2.6.10

  ```
  2674 static int dev_boot_phase = 1;
  ```

- De même on attend si on ne dispose pas du sémaphore RTNL.

  La macro `ASSERT_RTNL()` est définie dans le fichier `linux/include/linux/rtnetlink.h` : Code Linux 2.6.10

  ```
  812 #define ASSERT_RTNL() do { \
  ```

```
813         if (unlikely(down_trylock(&rtnl_sem) == 0)) { \
814                 up(&rtnl_sem); \
815                 printk(KERN_ERR "RTNL: assertion failed at %s (%d)\n", \
816                        __FILE__,  __LINE__); \
817                 dump_stack(); \
818         } \
819 } while(0)
```

- On attend également si l'état d'enregistrement n'est pas "non enregistré", car sinon on risquerait de se placer dans une boucle infinie.
- On verrouille les files d'attente en lecture et en écriture associées au descripteur de périphérique réseau.
- On indique qu'aucun paquet en émission n'a été transmis.
- On essaie d'allouer une trame de diversion au descripteur de périphérique. Si on n'y parvient pas, on renvoie le code fourni par la fonction `alloc_divert_blk()`, étudiée ci-dessous.
- On positionne le champ `iflink` du descripteur de périphérique à -1.
- Si une fonction spécifique d'initialisation est associée au descripteur de périphérique, on fait appel à elle. Si une erreur intervient lors de son exécution, on renvoie soit le code fourni par cette fonction spécifique, soit l'opposé du code d'erreur `EIO`.

  On a vu, par exemple, que dans le cas de la carte 3Com 501, cette fonction spécifique d'initialisation s'appelle `el1_probe()`.
- On vérifie la validité du nom du périphérique. Si ce n'est pas le cas, on renvoie l'opposé du code d'erreur `EINVAL`.

Code Linux 2.6.10

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

```
640 /**
641  *      dev_valid_name - verifie si le nom est ok pour le peripherique reseau
642  *      @name : chaine de caracteres nom
643  *
644  *      On a besoin que les noms des peripheriques reseau soient des noms de fichiers
645  *      valides pour permettre a sysfs de marcher
646  */
647 int dev_valid_name(const char *name)
648 {
649         return !(*name == '\0'
650                  || !strcmp(name, ".")
651                  || !strcmp(name, "..")
652                  || strchr(name, '/'));
653 }
```

- On attribue un index à ce périphérique, en faisant appel à la fonction `dev_new_index()` étudiée ci-dessous.
- Si le champ `iflink` n'a pas été renseigné (autrement dit si sa valeur est -1) par la fonction spécifique d'initialisation, on lui attribue la valeur de l'index.
- On parcourt la liste de hachage des périphériques réseau d'index associé au nom pour s'assurer qu'il n'avait pas déjà été déclaré; s'il l'avait été, on renvoie l'opposé du code d'erreur `EEXIST`.
- On vérifie la cohérence du champ des caractéristiques et on le modifie éventuellement.
- Si le champ `rebuild_header` n'est pas déjà renseigné, on lui attribue la fonction `default_rebuild_header()` comme valeur. Celle-ci est définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
799 static int default_rebuild_header(struct sk_buff *skb)
800 {
801         printk(KERN_DEBUG "%s: default_rebuild_header called -- BUG!\n",
```

```
802                skb->dev ? skb->dev->name : "NULL!!!");
803         kfree_skb(skb);
804         return 1;
805 }
```

- On renseigne l'état du périphérique en spécifiant qu'il est présent.
- On met le champ `next` à `NULL` puisqu'il s'agira du dernier de la liste des périphériques réseau.
- On indique l'existence du périphérique au gestionnaire des tâches pour qu'il en tienne compte, grâce à la fonction `dev_init_scheduler()` étudiée ci-dessous.
- On verrouille la liste des périphériques réseau en écriture et on ajoute le périphérique à la fin de celle-ci ainsi que dans la liste de hachage.
- On initialise le nombre d'utilisateurs de ce périphérique.

  La macro `dev_hold()` est définie dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
678 #define dev_hold(dev) atomic_inc(&(dev)->refcnt)
```

- L'état d'enregistrement du périphérique prend maintenant la valeur "en train d'être enregistré".
- On déverrouille la liste des périphériques réseau.
- On notifie aux protocoles qu'un nouveau périphérique est présent.
- On termine l'installation en ajoutant l'adresse du champ `dev->todo_list` du descripteur de périphérique réseau à la liste `net_todo_list`.

  Cette liste est déclarée et instantiée dans le fichier `linux/net/core/dev.c`. La fonction `set_set_todo()` est définie dans le même fichier : Code Linux 2.6.10

```
2678 static struct list_head net_todo_list = LIST_HEAD_INIT(net_todo_list);
2679
2680 static inline void net_set_todo(struct net_device *dev)
2681 {
2682         spin_lock(&net_todo_list_lock);
2683         list_add_tail(&dev->todo_list, &net_todo_list);
2684         spin_unlock(&net_todo_list_lock);
2685 }
```

- On renvoie 0.

15.3.3 Allocation d'une trame de déroutement

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

```
6   *              Deroutement generique de trame
7   *
8   * Auteurs :
9   *              Benoit LOCHER :  integration initiale dans le noyau avec support pour ethernet
10  *              Dave Miller :    amelioration du code (correction, performance et
                                     fichiers source)
11  *
12  */
[...]
48 /*
49  * Alloue un divert_blk a un peripherique, qui doit etre une carte ethernet.
50  */
51 int alloc_divert_blk(struct net_device *dev)
52 {
53         int alloc_size = (sizeof(struct divert_blk) + 3) & ~3;
54
```

```
55         if (dev->type == ARPHRD_ETHER) {
56                 printk(KERN_DEBUG "divert: allocating divert_blk for %s\n",
57                        dev->name);
58
59                 dev->divert = (struct divert_blk *)
60                         kmalloc(alloc_size, GFP_KERNEL);
61                 if (dev->divert == NULL) {
62                         printk(KERN_DEBUG "divert: unable to allocate divert_blk for %s\n",
63                                dev->name);
64                         return -ENOMEM;
65                 } else {
66                         memset(dev->divert, 0, sizeof(struct divert_blk));
67                 }
68                 dev_hold(dev);
69         } else {
70                 printk(KERN_DEBUG "divert: not allocating divert_blk for
                       non-ethernet device %s\n",
71                        dev->name);
72
73                 dev->divert = NULL;
74         }
75         return 0;
76 }
```

Autrement dit :

- On calcule la taille de l'emplacement pour l'allocation.
- Si le périphérique est du type carte réseau Ethernet, on indique qu'on va allouer une trame de déroutement pour ce périphérique et on essaie d'instantier une telle trame dont on place l'adresse dans le champ `divert` du descripteur de périphérique passé en argument. Si on n'y parvient pas, on l'indique par un message d'erreur et on renvoie l'opposé du code d'erreur `ENOMEM`. Si on y parvient, on met à zéro cet emplacement mémoire.
- S'il ne s'agit pas d'une carte réseau Ethernet, on affiche un message noyau indiquant qu'on ne va pas allouer de trame de déroutement pour ce périphérique et on renseigne le champ `divert` avec `NULL`.
- On renvoie 0.

15.3.4 Attribution d'un index

Code Linux 2.6.10

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

```
2656 /**
2657  *      dev_new_index   -       alloue un ifindex
2658  *
2659  *      Renvoie une valeur unique convenable pour un nouveau numero d'interface de
2660  *      peripherique. L'appelant doit avoir le semaphore rtnl ou le verrou
2661  *      dev_base_lock pour etre sur qu'il soit unique.
2662  */
2663 static int dev_new_index(void)
2664 {
2665         static int ifindex;
2666         for (;;) {
2667                 if (++ifindex <= 0)
2668                         ifindex = 1;
2669                 if (!__dev_get_by_index(ifindex))
2670                         return ifindex;
2671         }
2672 }
```

qui fait essentiellement appel à la fonction `__dev_get_by_index()`. Celle-ci est définie au début du même fichier :

Code Linux 2.6.10

```
523 /**
524  *      __dev_get_by_index - trouve un peripherique grace a son ifindex
525  *      @ifindex : index du peripherique
526  *
527  *      Cherche une interface grace a son index. Renvoie %NULL si le peripherique
528  *      n'est pas trouve, un pointeur au peripherique sinon. Le peripherique n'a pas eu
529  *      son compteur de reference accru, aussi l'appelant doit-il etre soigneux
530  *      sur le verrouillage. L'appelant doit avoir soit le semaphore RTNL,
531  *      soit @dev_base_lock.
532  */
533
534 struct net_device *__dev_get_by_index(int ifindex)
535 {
536         struct hlist_node *p;
537
538         hlist_for_each(p, dev_index_hash(ifindex)) {
539                 struct net_device *dev
540                         = hlist_entry(p, struct net_device, index_hlist);
541                 if (dev->ifindex == ifindex)
542                         return dev;
543         }
544         return NULL;
545 }
```

qui est l'analogue de `__dev_get_by_name()` pour l'index au lieu du nom.

15.3.5 Rattachement au gestionnaire des tâches

La fonction `dev_init_scheduler()` met en place la procédure d'ordonnancement du nouveau périphérique réseau sur le mécanisme FIFO par défaut. Elle est définie dans le fichier `linux/net/sched/sch_generic.c` :

Code Linux 2.6.10

```
571 void dev_init_scheduler(struct net_device *dev)
572 {
573         qdisc_lock_tree(dev);
574         dev->qdisc = &noop_qdisc;
575         dev->qdisc_sleeping = &noop_qdisc;
576         INIT_LIST_HEAD(&dev->qdisc_list);
577         qdisc_unlock_tree(dev);
578
579         dev_watchdog_init(dev);
580 }
```

Autrement dit :

– On verrouille l'accès en écriture à l'arbre des stratégies de mise en file d'attente.

Le verrou `qdisc_tree_lock` et la fonction `qdisc_lock_tree()` sont définis plus haut dans le même fichier :

Code Linux 2.6.10

```
40 /* Verrou principal de structure qdisc.
41
42    Cependant les modifications
43    aux donnees participant a l'ordonnancement doivent etre protegees
44    en plus avec le verrou tournant dev->queue_lock.
45
46    L'idee est la suivante :
47    - empiler et depiler sont serialises via le verrou tournant du
48      peripherique dev->queue_lock.
```

```
49    - le parcours de l'arbre est protege par read_lock_bh(qdisc_tree_lock)
50      et ce verrou est utilise dans le contexte des processus.
51    - les mises a jour de l'arbre sont faites sous semaphore rtnl ou
52      dans un contexte softirq (__qdisc_destroy rcu-callback),
53      donc ce verrou a besoin d'inhiber les bh locales.
54
55    qdisc_tree_lock doit etre attrape AVANT dev->queue_lock !
56  */
57 rwlock_t qdisc_tree_lock = RW_LOCK_UNLOCKED;
58
59 void qdisc_lock_tree(struct net_device *dev)
60 {
61         write_lock_bh(&qdisc_tree_lock);
62         spin_lock_bh(&dev->queue_lock);
63 }
```

– On renseigne le champ de stratégie de mise en file d'attente du descripteur de périphérique avec la stratégie qui ne fait rien.

Code Linux 2.6.10

La stratégie `noop_qdisc` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
267 struct Qdisc_ops noop_qdisc_ops = {
268         .next           =       NULL,
269         .cl_ops         =       NULL,
270         .id             =       "noop",
271         .priv_size      =       0,
272         .enqueue        =       noop_enqueue,
273         .dequeue        =       noop_dequeue,
274         .requeue        =       noop_requeue,
275         .owner          =       THIS_MODULE,
276 };
277
278 struct Qdisc noop_qdisc = {
279         .enqueue        =       noop_enqueue,
280         .dequeue        =       noop_dequeue,
281         .flags          =       TCQ_F_BUILTIN,
282         .ops            =       &noop_qdisc_ops,
283         .list           =       LIST_HEAD_INIT(noop_qdisc.list),
284 };
```

– On initialise la liste de la stratégie de mise en file d'attente du descripteur de périphérique.

Code Linux 2.6.10

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

```
37 #define INIT_LIST_HEAD(ptr) do { \
38         (ptr)->next = (ptr); (ptr)->prev = (ptr); \
39 } while (0)
```

– On déverrouille le verrou de file d'attente associé au descripteur de périphérique passé en argument.

Code Linux 2.6.10

La fonction `qdisc_unlock_tree()` est définie plus haut dans le même fichier que la fonction principale:

```
65 void qdisc_unlock_tree(struct net_device *dev)
66 {
67         spin_unlock_bh(&dev->queue_lock);
68         write_unlock_bh(&qdisc_tree_lock);
69 }
```

– on initialise le minuteur en vue de l'identification des problèmes d'émission.

15.3.6 Initialisation du minuteur

La fonction `dev_watchdog_init()` est définie dans le fichier `linux/net/sched/sch_generic.c`: Code Linux 2.6.10

```
208 static void dev_watchdog_init(struct net_device *dev)
209 {
210         init_timer(&dev->watchdog_timer);
211         dev->watchdog_timer.data = (unsigned long)dev;
212         dev->watchdog_timer.function = dev_watchdog;
213 }
```

Autrement dit:

- On initialise le minuteur associé au descripteur de périphérique passé en argument.
- L'adresse des données est l'adresse du descripteur de périphérique elle-même.
- La fonction à appliquer lorsque le délai est écoulé est `dev_watchdog()`, que nous étudierons au moment opportun.

15.4 Mise en fonctionnement des périphériques réseau

La quatrième étape, appelée par la fonction `rtln_unlock()` comme nous l'avons vu ci-dessus, correspond à la remise en fonctionnement de tous les périphériques réseau.

15.4.1 Fonction principale

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

```
2893 void netdev_run_todo(void)
2894 {
2895         struct list_head list = LIST_HEAD_INIT(list);
2896         int err;
2897
2898
2899         /* Exige de prendre garde aux autres cpu. */
2900         down(&net_todo_run_mutex);
2901
2902         /* Pas sain de le faire en dehors du semaphore. Nous ne devons rien renvoyer
2903          * avant que tous les evenements de retrait invoques par le processeur local
2904          * aient ete accomplis (soit par ce todo run, soit par un
2905          * autre cpu).
2906          */
2907         if (list_empty(&net_todo_list))
2908                 goto out;
2909
2910         /* Liste instantanee, permet les requetes ulterieures */
2911         spin_lock(&net_todo_list_lock);
2912         list_splice_init(&net_todo_list, &list);
2913         spin_unlock(&net_todo_list_lock);
2914
2915         while (!list_empty(&list)) {
2916                 struct net_device *dev
2917                         = list_entry(list.next, struct net_device, todo_list);
2918                 list_del(&dev->todo_list);
2919
2920                 switch(dev->reg_state) {
2921                 case NETREG_REGISTERING:
2922                         err = netdev_register_sysfs(dev);
2923                         if (err)
```

```
2924                                    printk(KERN_ERR "%s: failed sysfs registration (%d)\n",
2925                                           dev->name, err);
2926                            dev->reg_state = NETREG_REGISTERED;
2927                            break;
2928
2929                    case NETREG_UNREGISTERING:
2930                            netdev_unregister_sysfs(dev);
2931                            dev->reg_state = NETREG_UNREGISTERED;
2932
2933                            netdev_wait_allrefs(dev);
2934
2935                            /* paranoia */
2936                            BUG_ON(atomic_read(&dev->refcnt));
2937                            BUG_TRAP(!dev->ip_ptr);
2938                            BUG_TRAP(!dev->ip6_ptr);
2939                            BUG_TRAP(!dev->dn_ptr);
2940
2941
2942                            /* Cela doit etre la toute derniere action,
2943                             * apres cela 'dev' peut pointer sur de la memoire liberee.
2944                             */
2945                            if (dev->destructor)
2946                                    dev->destructor(dev);
2947                            break;
2948
2949                    default:
2950                            printk(KERN_ERR "network todo '%s' but state %d\n",
2951                                   dev->name, dev->reg_state);
2952                            break;
2953                    }
2954            }
2955
2956 out:
2957            up(&net_todo_run_mutex);
2958 }
```

Autrement dit :

- On déclare une liste que l'on instantie.
- On s'assure qu'aucun autre microprocesseur ne va agir.

Code Linux 2.6.10

Le sémaphore `net_todo_run_mutex` est défini juste au-dessus dans le même fichier :

```
2892 static DECLARE_MUTEX(net_todo_run_mutex);
```

- Si la liste `net_todo_list` est vide, on a terminé puisqu'il n'y a aucun périphérique à mettre en fonctionnement.
- On agit sur un certain nombre de listes pour préparer les requêtes ultérieures.

Code Linux 2.6.10

Le verrou rotatif liste `net_todo_list_lock` est également déclaré et initialisé dans le même fichier :

```
2676 /* Enregistrement/retrait retarde */
2677 static spinlock_t net_todo_list_lock = SPIN_LOCK_UNLOCKED;
```

Code Linux 2.6.10

La fonction générale `list_splice_init()` est également déclaré et initialisé dans le fichier `linux/include/linux/list.h`:

```
298 /**
299  * list_splice_init - joint deux listes et reinitialise la liste videe.
300  * @list : la nouvelle liste a ajouter.
301  * @head : l'emplacement ou l'ajouter dans la premiere liste.
302  *
```

```
303  * La liste @list est reinitialisee
304  */
305 static inline void list_splice_init(struct list_head *list,
306                                      struct list_head *head)
```

- Pour chaque élément de la liste `net_todo_list` :
  - On récupère le descripteur de périphérique réseau associé.
  - On le retire de la liste.
  - Suivant la valeur de l'état d'enregistrement de ce périphérique:
    - S'il est en train de s'enregistrer, on monte le système de fichiers associé grâce à la fonction `netdev_register_sysfs()` étudiée ci-après et on change l'état en "enregistré".
    - Nous étudierons le cas en train d'être désinstallé au chapitre 35.
    - Dans les autres cas, on affiche un message indiquant qu'on ne fait rien.
- On lève le sémaphore `net_todo_run_mutex`.

15.4.2 Montage du système de fichiers

15.4.2.1 Fonction principale

La fonction `netdev_register_sysfs()` est définie dans le fichier `linux/net/core/sys-fs.c`: Code Linux 2.6.10

```
412 /* Cree les entrees sysfs pour un peripherique reseau. */
413 int netdev_register_sysfs(struct net_device *net)
414 {
415         struct class_device *class_dev = &(net->class_dev);
416         int i;
417         struct class_device_attribute *attr;
418         int ret;
419
420         class_dev->class = &net_class;
421         class_dev->class_data = net;
422
423         strlcpy(class_dev->class_id, net->name, BUS_ID_SIZE);
424         if ((ret = class_device_register(class_dev)))
425                 goto out;
426
427         for (i = 0; (attr = net_class_attributes[i]) != NULL; i++) {
428                 if ((ret = class_device_create_file(class_dev, attr)))
429                     goto out_unreg;
430         }
431
432
433         if (net->get_stats &&
434             (ret = sysfs_create_group(&class_dev->kobj, &netstat_group)))
435                 goto out_unreg;
436
437 #ifdef WIRELESS_EXT
438         if (net->get_wireless_stats &&
439             (ret = sysfs_create_group(&class_dev->kobj, &wireless_group)))
440                 goto out_cleanup;
441
442         return 0;
443 out_cleanup:
444         if (net->get_stats)
445                 sysfs_remove_group(&class_dev->kobj, &netstat_group);
```

```
446 #else
447         return 0;
448 #endif
449
450 out_unreg:
451         printk(KERN_WARNING "%s: sysfs attribute registration failed %d\n",
452                net->name, ret);
453         class_device_unregister(class_dev);
454 out:
455         return ret;
456 }
```

Autrement dit :

- On déclare une entité du type général `struct class_device` que l'on initialise avec le champ adéquat du périphérique réseau passé en argument.
- On déclare une entité du type général `struct class_device_attribute`.
- On renseigne le champ `class` de `class_dev` avec l'adresse de l'instance `net_class`. Cette dernière est définie dans le fichier `linux/net/core/net-sysfs.c` :

Code Linux 2.6.10

```
389 static struct class net_class = {
390         .name = "net",
391         .release = netdev_release,
392 #ifdef CONFIG_HOTPLUG
393         .hotplug = netdev_hotplug,
394 #endif
395 };
```

- On renseigne le champ `class_data` de `class_dev` avec l'adresse passée en argument.
- On renseigne le champ `class_id` de `class_dev` avec le nom du périphérique passé en argument.
  La constante symbolique `BUS_ID_SIZE` est définie dans le fichier `linux/include/linux/-device.h` :

Code Linux 2.6.10

```
28 #define BUS_ID_SIZE             KOBJ_NAME_LEN
```

- On esaie d'enregistrer la classe, grâce à la fonction générale `class_device_register()`. Si on n'y parvient pas, on renvoie le code fourni par cette dernière fonction.
- On essaie d'enregistrer les attributs, grâce à la fonction générale `class_device_create_-file()`. Si on n'y parvient pas pour l'un des attributs, on affiche un message d'erreur, on désintalle la classe de pilotes (grâce à la fonction générale `class_device_unregister()`) et on renvoie le code fourni par la fonction d'enregistrement.

  Le tableau `net_class_attributes[]` est défini dans le fichier `linux/net/core/net--sysfs.c` :

Code Linux 2.6.10

```
188 static struct class_device_attribute *net_class_attributes[] = {
189         &class_device_attr_ifindex,
190         &class_device_attr_iflink,
191         &class_device_attr_addr_len,
192         &class_device_attr_tx_queue_len,
193         &class_device_attr_features,
194         &class_device_attr_mtu,
195         &class_device_attr_flags,
196         &class_device_attr_type,
197         &class_device_attr_address,
198         &class_device_attr_broadcast,
199         &class_device_attr_carrier,
200         NULL
201 };
```

– On essaie de créer le groupe, grâce à la fonction générale `sysfs_create_group()`. Si on n'y parvient pas, on affiche un message d'erreur, on désintalle la classe de pilotes et on renvoie le code fourni par cette dernière fonction.

L'instance `netstat_group` est définie dans le fichier `linux/net/core/net-sysfs.c`: Code Linux 2.6.10

```
286 static struct attribute_group netstat_group = {
287         .name  = "statistics",
288         .attrs  = netstat_attrs,
289 };
```

L'instance `netstat_attrs` est définie dans le fichier `linux/net/core/net-sysfs.c`: Code Linux 2.6.10

```
258 static struct attribute *netstat_attrs[] = {
259         &class_device_attr_rx_packets.attr,
260         &class_device_attr_tx_packets.attr,
261         &class_device_attr_rx_bytes.attr,
262         &class_device_attr_tx_bytes.attr,
263         &class_device_attr_rx_errors.attr,
264         &class_device_attr_tx_errors.attr,
265         &class_device_attr_rx_dropped.attr,
266         &class_device_attr_tx_dropped.attr,
267         &class_device_attr_multicast.attr,
268         &class_device_attr_collisions.attr,
269         &class_device_attr_rx_length_errors.attr,
270         &class_device_attr_rx_over_errors.attr,
271         &class_device_attr_rx_crc_errors.attr,
272         &class_device_attr_rx_frame_errors.attr,
273         &class_device_attr_rx_fifo_errors.attr,
274         &class_device_attr_rx_missed_errors.attr,
275         &class_device_attr_tx_aborted_errors.attr,
276         &class_device_attr_tx_carrier_errors.attr,
277         &class_device_attr_tx_fifo_errors.attr,
278         &class_device_attr_tx_heartbeat_errors.attr,
279         &class_device_attr_tx_window_errors.attr,
280         &class_device_attr_rx_compressed.attr,
281         &class_device_attr_tx_compressed.attr,
282         NULL
283 };
```

– Nous ne nous intéresserons pas aux réseaux sans fil dans cet ouvrage.

– On renvoie 0 pour indiquer que tout s'est bien déroulé.