Chapitre 35

Désactivation et retrait d'un pilote de périphérique réseau

Nous avons retardé jusqu'à ce chapitre l'étude de l'implémentation de la désactivation et du retrait d'un pilote de périphérique non pas tant par esprit de symétrie mais pour ne pas surcharger les chapitres 18 et 15.

35.1 Désactivation

La désactivation d'un périphérique réseau s'effectue grâce à la fonction `dev_close()`, qui fait appel à une fonction spécifique à la carte réseau.

35.1.1 Cas général

35.1.1.1 Fonction de fermeture

Code Linux 2.6.10

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

```
875 /**
876  *      dev_close - desactive une interface.
877  *      @dev : peripherique a desactiver
878  *
879  *      Cette fonction fait passer un peripherique de l'etat actif a l'etat non actif. Un
880  *      %NETDEV_GOING_DOWN est envoye a la chaine de notification netdev. Le peripherique
881  *      est alors desactive puis un %NETDEV_DOWN est envoye a la chaine de
882  *      notification.
883  */
884 int dev_close(struct net_device *dev)
885 {
886         if (!(dev->flags & IFF_UP))
887                 return 0;
888
889         /*
890          *      Dire aux gens que nous sommes desactives, qu'ils peuvent se
891          *      preparer a notre mort, lorsque le peripherique est encore en train d'agir.
892          */
893         notifier_call_chain(&netdev_chain, NETDEV_GOING_DOWN, dev);
894
895         dev_deactivate(dev);
896
897         clear_bit(__LINK_STATE_START, &dev->state);
898
899         /* Synchroniser l'election dans l'ordonnanceur. Nous ne pouvons pas atteindre la liste
900          * d'election, elle peut meme etre sur une cpu differente. Donc juste effacer
901          * netif_running(), et attendre que l'election arrive reellement. Actuellement,
902          * la meilleure place pour ceci est a l'interieur de dev->stop() apres que le
903          * peripherique ait arrete sa machine irq, mais ceci necessite plusieurs
              changements dans les peripheriques. */
904
905         smp_mb__after_clear_bit(); /* Commit netif_running(). */
906         while (test_bit(__LINK_STATE_RX_SCHED, &dev->state)) {
907                 /* Ne pas se precipiter. */
908                 current->state = TASK_INTERRUPTIBLE;
909                 schedule_timeout(1);
910         }
911
912         /*
913          *      Appeler la fonction close specifique au peripherique. Ceci n'echouera pas.
914          *      Seulement si le peripherique est actif.
915          *
916          *      Nous permettons qu'elle soit appelee meme apres un evenement
917          *      de DETACHEMENT a chaud.
918          */
919         if (dev->stop)
920                 dev->stop(dev);
921
922         /*
923          *      Le peripherique est maintenant desactive.
924          */
```

```
925
926         dev->flags &= ~IFF_UP;
927
928         /*
929          * Dire aux gens que nous sommes desactive
930          */
931         notifier_call_chain(&netdev_chain, NETDEV_DOWN, dev);
932
933         return 0;
934 }
```

Autrement dit :

- Si le périphérique n'est pas ouvert, on renvoie 0.
- Toutes les méthodes de la chaîne de notification `netdev_chain` sont informées de la désactivation imminente du périphérique et peuvent agir en conséquence.
- On désactive le périphérique.
- On met à zéro le bit `__LINK_STATE_START` de l'état du descripteur de périphérique.
- On synchronise la liste d'élection et on change l'état du processus en cours.

  La fonction générale `schedule_timeout()` est définie dans le fichier `linux/kernel/-timer.c` :

Code Linux 2.6.10

```
1087 /**
1088  * schedule_timeout - s'assoupir jusqu'a la fin du delai
1089  * @timeout : valeur du delai en jiffies
1090  *
1091  * Fait que la tache en cours s'assoupisse jusqu'a ce que @timeout jiffies se soient
1092  * ecoulees. La routine rendra la main immediatement a moins que
1093  * l'etat de la tache en cours ait ete positionne (voir set_current_state()).
1094  *
1095  * Vous pouvez positionner l'etat de la tache comme suit -
1096  *
1097  * %TASK_UNINTERRUPTIBLE - on est sur qu'au moins @timeout jiffies s'ecoulent
1098  * avant que la routine rende la main. La routine renverra 0.
1099  *
1100  * %TASK_INTERRUPTIBLE - la routine peut rendre la main plus tot si un signal est
1101  * delivre a la tache en cours. Dans ce cas, le temps restant
1102  * en jiffies sera renvoye ou 0 si le minuteur a termine.
1103  *
1104  * On est sur que l'etat de la tache en cours est TASK_RUNNING lorsque cette
1105  * routine rend la main.
1106  *
1107  * Specifier une valeur @timeout de %MAX_SCHEDULE_TIMEOUT appellera l'ordonnanceur
1108  * du CPU sans limite sur le delai. Dans ce cas, la valeur
1109  * renvoyee sera %MAX_SCHEDULE_TIMEOUT.
1110  *
1111  * Dans tous les cas, on est sur que la valeur renvoyee est positive ou nulle.
1112  */
1113 fastcall signed long __sched schedule_timeout(signed long timeout)
```

- On fait appel à la fonction de fermeture associée au périphérique.

  Nous étudierons ci-dessous le cas de la fonction correspondant à la carte 3Com 501.
- On change le vecteur de drapeaux du descripteur de périphérique pour indiquer que le périphérique est désactivé.
- L'arrêt du périphérique est communiqué à tous les protocoles concernés et on renvoie 0.

35.1.1.2 Désactivation de la stratégie de mise en file d'attente

La fonction `dev_deactivate()` de désactivation de la stratégie de mise en file d'attente associée au descripteur de périphérique passé en argument est définie dans le fichier `linux/net/sched/sch_generic.c`:

Code Linux 2.6.10

```
551 void dev_deactivate(struct net_device *dev)
552 {
553         struct Qdisc *qdisc;
554
555         spin_lock_bh(&dev->queue_lock);
556         qdisc = dev->qdisc;
557         dev->qdisc = &noop_qdisc;
558
559         qdisc_reset(qdisc);
560
561         spin_unlock_bh(&dev->queue_lock);
562
563         dev_watchdog_down(dev);
564
565         while (test_bit(__LINK_STATE_SCHED, &dev->state))
566                 yield();
567
568         spin_unlock_wait(&dev->xmit_lock);
569 }
```

La fonction `qdisc_reset()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

Code Linux 2.6.10

```
449 /* Avec dev->queue_lock et BH ! */
450
451 void qdisc_reset(struct Qdisc *qdisc)
452 {
453         struct Qdisc_ops *ops = qdisc->ops;
454
455         if (ops->reset)
456                 ops->reset(qdisc);
457 }
```

35.1.1.3 Arrêt du minuteur d'identification de problème en émission

La fonction `dev_watchdog_down()` arrête le minuteur dédié à l'identification de problèmes d'émission. Elle est définie dans le fichier `linux/net/sched/sch_generic.c`:

Code Linux 2.6.10

```
232 static void dev_watchdog_down(struct net_device *dev)
233 {
234         spin_lock_bh(&dev->xmit_lock);
235         if (del_timer(&dev->watchdog_timer))
236                 __dev_put(dev);
237         spin_unlock_bh(&dev->xmit_lock);
238 }
```

La macro `__dev_put()` est définie dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
695 #define __dev_put(dev) atomic_dec(&(dev)->refcnt)
```

35.1.2 Cas de la carte 3Com 501

Nous avons vu au chapitre 17 que la fonction spécifique de fermeture associée à la carte 3Com 501 s'appelle `el1_close()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
788 /**
789  * el1_close :
```

```
790  * @dev : carte 3c501 a fermer
791  *
792  * Ferme une carte 3c501. Le drapeau IFF_UP a ete efface par l'utilisateur via
793  * le ioctl SIOCSIFFLAGS. Nous arretons toute transmission encore presente dans la
794  * file d'attente et inhibons alors les interruptions. Nous reinitialisons finalement la puce.
795  * Les effets du reste seront nettoyes par #el1_open. Renvoie toujours 0, indiquant
796  * un succes.
797  */
798
799 static int el1_close(struct net_device *dev)
800 {
801         int ioaddr = dev->base_addr;
802
803         if (el_debug > 2)
804                 printk(KERN_INFO "%s: Shutting down Ethernet card at %#x.\n",
                               dev->name, ioaddr);
805
806         netif_stop_queue(dev);
807
808         /*
809          *      Libere et inhibe l'IRQ.
810          */
811
812         free_irq(dev->irq, dev);
813         outb(AX_RESET, AX_CMD);         /* Reinitialise la puce */
814
815         return 0;
816 }
```

Autrement dit :

- On déclare un port d'entrée-sortie, que l'on initialise avec celui fourni par le descripteur de périphérique passé en argument.
- Si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau indiquant que l'on est en train de fermer la carte.
- On indique qu'il n'est plus possible de transmettre grâce à la fonction `netif_stop_queue()`.
- On libère l'IRQ associée au périphérique.
- On réinitialise le contrôleur du périphérique et on renvoie 0.

35.2 Retrait d'un périphérique réseau

Le retrait d'un périphérique réseau est effectué grâce à la fonction `unregister_netdev()` définie dans le fichier `linux/drivers/net/net_init.c` :

Code Linux 2.6.10

```
144 void unregister_netdev(struct net_device *dev)
145 {
146         rtnl_lock();
147         unregister_netdevice(dev);
148         rtnl_unlock();
149 }
```

qui fait appel à la fonction `unregister_netdevice()`, définie dans le fichier `linux/net/core/dev.c` :

Code Linux 2.6.10

```
2994 /**
2995  *      unregister_netdevice - supprime un peripherique du noyau
2996  *      @dev : peripherique
2997  *
2998  *      Cette fonction arrete une interface de peripherique et la supprime
```

```
2999  *      des tables du noyau. En cas de succes 0 est renvoye, en cas d'echec
3000  *      un code errno negatif est renvoye.
3001  *
3002  *      Les appelants doivent detenir le semaphore rtnl. Voir le commentaire a la
3003  *      fin de Space.c pour les details sur le verrouillage. Vous pouvez vouloir
3004  *      unregister_netdev() au lieu de cette fonction.
3005  */
3006
3007 int unregister_netdevice(struct net_device *dev)
3008 {
3009         struct net_device *d, **dp;
3010
3011         BUG_ON(dev_boot_phase);
3012         ASSERT_RTNL();
3013
3014         /* Certains peripheriques appellent sans s'enregistrer pour derouler
               l'initialisation. */
3015         if (dev->reg_state == NETREG_UNINITIALIZED) {
3016                 printk(KERN_DEBUG "unregister_netdevice: device %s/%p never "
3017                                   "was registered\n", dev->name, dev);
3018                 return -ENODEV;
3019         }
3020
3021         BUG_ON(dev->reg_state != NETREG_REGISTERED);
3022
3023         /* Si le peripherique est en marche, commencons par le fermer. */
3024         if (dev->flags & IFF_UP)
3025                 dev_close(dev);
3026
3027         /* Et enlevons-le de la chaine des peripheriques. */
3028         for (dp = &dev_base; (d = *dp) != NULL; dp = &d->next) {
3029                 if (d == dev) {
3030                         write_lock_bh(&dev_base_lock);
3031                         hlist_del(&dev->name_hlist);
3032                         hlist_del(&dev->index_hlist);
3033                         if (dev_tail == &dev->next)
3034                                 dev_tail = dp;
3035                         *dp = d->next;
3036                         write_unlock_bh(&dev_base_lock);
3037                         break;
3038                 }
3039         }
3040         if (!d) {
3041                 printk(KERN_ERR "unregister net_device: '%s' not found\n",
3042                        dev->name);
3043                 return -ENODEV;
3044         }
3045
3046         dev->reg_state = NETREG_UNREGISTERING;
3047
3048         synchronize_net();
3049
3050         /* Arret de la discipline de mise en file d'attente. */
3051         dev_shutdown(dev);
3052
3053
3054         /* Notifions aux protocoles que nous sommes en train de detruire
3055            ce peripherique. Ils doivent nettoyer toute chose.
3056         */
3057         notifier_call_chain(&netdev_chain, NETDEV_UNREGISTER, dev);
3058
3059         /*
```

```
3060          *      Vidons la chaine de multiffusion
3061          */
3062         dev_mc_discard(dev);
3063
3064         if (dev->uninit)
3065                 dev->uninit(dev);
3066
3067         /* La chaine de notification DOIT nous detacher du peripherique maitre. */
3068         BUG_TRAP(!dev->master);
3069
3070         free_divert_blk(dev);
3071
3072         /* Finir le traitement de retrait apres le deverrouillage */
3073         net_set_todo(dev);
3074
3075         synchronize_net();
3076
3077         dev_put(dev);
3078         return 0;
3079 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau et une liste de tels descripteurs.
- Si nous sommes en phase de démarrage, on reporte une erreur.
- On attend de pouvoir s'approprier le sémaphore RTNL.
- Si le descripteur de périphérique passé en argument n'est pas initialisé, on affiche un message noyau et on renvoie l'opposé du code d'erreur `ENODEV`.
- Si le descripteur de périphérique passé en argument n'a pas été installé, on reporte une erreur.
- Si le périphérique est en activité, on commence par le désactiver.
- On essaie de retirer le descripteur passé en argument de la liste des périphériques réseau. Si on ne le trouve pas dans cette liste, on affiche un message noyau et on renvoie l'opposé du code d'erreur `ENODEV`.
- On change son état d'enregistrement en "en train d'être retiré".
- On synchronise le réseau.
- On arrête la stratégie de mise en file d'attente du descripteur de périphérique passé en argument.

  La fonction `dev_shutdown()` est définie dans le fichier `linux/net/sched/sch_generic.c` :

Code Linux 2.6.10

```
582 void dev_shutdown(struct net_device *dev)
583 {
584         struct Qdisc *qdisc;
585
586         qdisc_lock_tree(dev);
587         qdisc = dev->qdisc_sleeping;
588         dev->qdisc = &noop_qdisc;
589         dev->qdisc_sleeping = &noop_qdisc;
590         qdisc_destroy(qdisc);
591 #if defined(CONFIG_NET_SCH_INGRESS) || defined(CONFIG_NET_SCH_INGRESS_MODULE)
592         if ((qdisc = dev->qdisc_ingress) != NULL) {
593                 dev->qdisc_ingress = NULL;
594                 qdisc_destroy(qdisc);
595         }
596 #endif
597         BUG_TRAP(!timer_pending(&dev->watchdog_timer));
```

```
598         qdisc_unlock_tree(dev);
599 }
```

– On notifie aux protocoles que nous sommes en train de retirer ce périphérique.
– On force la chaîne de multidiffusion.

Code Linux 2.6.10

La multidiffusion ne nous intéresse pas dans ce livre. La fonction `dev_mc_discard()`, qui vide une liste, est définie dans le fichier `linux/net/core/dev_mcast.c`:

```
202 /*
203  *      Detruit la liste de multidiffusion lorsqu'un peripherique est arrete
204  */
205
206 void dev_mc_discard(struct net_device *dev)
207 {
208         spin_lock_bh(&dev->xmit_lock);
209
210         while (dev->mc_list != NULL) {
211                 struct dev_mc_list *tmp = dev->mc_list;
212                 dev->mc_list = tmp->next;
213                 if (tmp->dmi_users > tmp->dmi_gusers)
214                         printk("dev_mc_discard: multicast leakage! dmi_users=%d\n",
                                tmp->dmi_users);
215                 kfree(tmp);
216         }
217         dev->mc_count = 0;
218
219         spin_unlock_bh(&dev->xmit_lock);
220 }
```

– Si le descripteur de périphérique fait apparaître une méthode `uninit()`, ce qui n'est le cas d'aucun descripteur de périphérique à l'heure actuelle, on fait appel à elle.
– Si la chaîne de notification ne nous a pas détachée du périphérique maître, on reporte une erreur.
– On libère la trame de diversion éventuelle.

Code Linux 2.6.10

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

```
78 /*
79  * Libere un divert_blk alloue par la fonction ci-dessus, si il a ete
80  * alloue sur ce peripherique.
81  */
82 void free_divert_blk(struct net_device *dev)
83 {
84         if (dev->divert) {
85                 kfree(dev->divert);
86                 dev->divert=NULL;
87                 dev_put(dev);
88                 printk(KERN_DEBUG "divert: freeing divert_blk for %s\n",
89                         dev->name);
90         } else {
91                 printk(KERN_DEBUG "divert: no divert_blk to free, %s not ethernet\n",
92                         dev->name);
93         }
94 }
```

– On termine le traitement du retrait en ajoutant le champ `dev->todo_list` du descripteur de périphérique réseau à la liste `net_todo_list`.
– On décrémente le compteur d'utilisation de ce descripteur de périphérique.

Code Linux 2.6.10

La fonction `dev_put()` est définie dans le fichier `linux/include/linux/netdevice.h`:

```
690 static inline void dev_put(struct net_device *dev)
```

```
691 {
692         atomic_dec(&dev->refcnt);
693 }
```

35.2.1 Démontage du système de fichier

La fonction `netdev_run_todo()`, étudiée au chapitre 15, montre que dans le cas d'un retrait on démonte le système de fichiers associé grâce à la fonction `netdev_unregister_sysfs()` étudiée ci-après, on change l'état en "désinstallé", on attend jusqu'à ce que toutes les références aient terminées (en faisant appel à la fonction `netdev_wait_allrefs()` étudiée ci-dessous), on vérifie quelques points, puis on fait appel à la fonction spécifique de destruction pour ce périphérique si elle existe.

Dans le cas de la carte 3Com 501, nous avons vu qu'il n'existe pas de fonction de destrucion spécifique.

35.2.2 Démontage du système de fichiers

La fonction `netdev_unregister_sysfs()` est définie dans le fichier `linux/net/core/net-sysfs.c`: Code Linux 2.6.0

```
386 void netdev_unregister_sysfs(struct net_device * net)
387 {
388         struct class_device * class_dev = &(net->class_dev);
389
390         if (net->get_stats)
391                 sysfs_remove_group(&class_dev->kobj, &netstat_group);
392
393 #ifdef WIRELESS_EXT
394         if (net->get_wireless_stats)
395                 sysfs_remove_group(&class_dev->kobj, &wireless_group);
396 #endif
397         class_device_del(class_dev);
398
399 }
```

35.2.3 Attente des références

La fonction `netdev_wait_allrefs()` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.0

```
2730 /*
2731  * netdev_wait_allrefs - attend jusqu'a ce que toutes les references s'en sont allees.
2732  *
2733  * Ceci est appele lors de l'enregistrement des peripheriques reseau.
2734  *
2735  * Tout protocole ou peripherique qui a une a reference doit s'enregistrer
2736  * pour la notification des peripheriques reseau, puis nettoyer et retirer la
2737  * reference si elle recoit un evenement UNREGISTER.
2738  * Nous pouvons avoir des pretentieux ici si des protocoles bogues n'appellent pas
2739  * correctement dev_put.
2740  */
2741 static void netdev_wait_allrefs(struct net_device *dev)
2742 {
2743         unsigned long rebroadcast_time, warning_time;
2744
2745         rebroadcast_time = warning_time = jiffies;
2746         while (atomic_read(&dev->refcnt) != 0) {
2747                 if (time_after(jiffies, rebroadcast_time + 1 * HZ)) {
2748                         rtnl_shlock();
```

```
2749                        rtnl_exlock();
2750
2751                        /* Renvoyons la notification de retrait */
2752                        notifier_call_chain(&netdev_chain,
2753                                            NETDEV_UNREGISTER, dev);
2754
2755                        if (test_bit(__LINK_STATE_LINKWATCH_PENDING,
2756                                     &dev->state)) {
2757                                /* Nous ne devons pas avoir d'evenements linkwatch
2758                                 * en attente de retrait. Si cela
2759                                 * arrive, nous ne passons pas la file d'attente
2760                                 * a l'ordonnanceur, d'ou un resultat de noop
2761                                 * pour ce peripherique.
2762                                 */
2763                                linkwatch_run_queue();
2764                        }
2765
2766                        rtnl_exunlock();
2767                        rtnl_shunlock();
2768
2769                        rebroadcast_time = jiffies;
2770                }
2771
2772                current->state = TASK_INTERRUPTIBLE;
2773                schedule_timeout(HZ / 4);
2774
2775                if (time_after(jiffies, warning_time + 10 * HZ)) {
2776                        printk(KERN_EMERG "unregister_netdevice: "
2777                               "waiting for %s to become free. Usage "
2778                               "count = %d\n",
2779                               dev->name, atomic_read(&dev->refcnt));
2780                        warning_time = jiffies;
2781                }
2782        }
2783 }
```