Quatrième partie

Réception

Chapitre 19

Réception des trames

Nous allons étudier dans ce chapitre comment les trames sont reçues en prenant comme exemple notre carte Ethernet 3Com 501.

Une fois que l'interface est activée, lors de la réception d'une trame par la carte réseau, celle-ci lève une interruption pour indiquer qu'une trame est arrivée. Le gestionnaire d'interruption la vérifie, la rejette ou la place en mémoire vive en utilisant l'accès direct à la mémoire puis elle passe les données et la main à la couche supérieure (la couche réseau en général).

19.1 Action du gestionnaire d'interruption

L'itinéraire de toute trame qui n'a pas été générée en local dans l'ordinateur commence dans un adaptateur réseau (ou une interface comparable, comme le port parallèle). Cet adaptateur reçoit la trame dans sa mémoire interne propre et informe le noyau de son arrivée en déclenchant une interruption.

Prenons comme exemple celui de la carte 3c501 de 3Com.

19.1.1 Démultiplexage : réception ou émission

Le même numéro d'interruption est utilisé pour la réception et l'émission. La première action du gestionnaire de cette interruption est donc de déterminer la cause de l'interruption. Nous avons vu que le gestionnaire d'interruption de la carte 3Com 501 s'appelle `el_interrupt()`. Il est défini dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
496 /**
497  * el_interrupt :
498  * @irq : Numero d'interruption
499  * @dev_id : La 3c501 qui a fait un renvoi
500  * @regs : Registre des donnees (surplus a nos exigences)
501  *
502  * Gestionnaire des interruptions de l'interface ether. La 3c501 a besoin de beaucoup plus
503  * que la plupart des cartes. En particulier nous obtenons une interruption en emission par
504  * erreur due a une collision puisque le micrologiciel de la carte n'est pas capable de
505  * rembobiner ses propres pointeurs de tampon. Il peut cependant compter jusqu'a 16 pour nous.
506  *
507  * Du cote de la reception, la carte est egalement tres bete. Elle n'a pas de tampon pour
508  * indiquer. Nous tirons tout simplement le paquet hors de son tampon PIO (qui est lent)
509  * et le mettons dans une file d'attente a l'attention du noyau. Nous reinitialison alors
       la carte pour le paquet suivant.
510  *
511  * Nous obtenons quelquefois des interruptions suprises tardives, a la fois parce que la
512  * livraison des IRQ SMP est un passage de message et parce que la carte semble quelquefois
513  * les livrer en retard. Je pense que ceci est du en partie a la facon de receptionner et au
514  * fait que le mode est change lorsqu'il sort d'une reception et nous envoie une interruption.
515  * Nous avons a regrouper tous ces cas pour obtenir une performamce d'a peu pres 150
516  * koctets/seconde. Meme alors vous devez avoir une petite fenetre TCP.
517  */
518
519 static irqreturn_t el_interrupt(int irq, void *dev_id, struct pt_regs *regs)
520 {
521         struct net_device *dev = dev_id;
522         struct net_local *lp;
523         int ioaddr;
524         int axsr;                       /* Registre de statut aux. */
525
526         ioaddr = dev->base_addr;
527         lp = netdev_priv(dev);
528
529         spin_lock(&lp->lock);
530
531         /*
532          *      Qu'est ce qui se passe ?
533          */
534
535         axsr = inb(AX_STATUS);
536
537         /*
538          *      Inscrivons-le !
539          */
```

```
540
541         if (el_debug > 3)
542                 printk(KERN_DEBUG "%s: el_interrupt() aux=%#02x", dev->name, axsr);
543
544         if(lp->loading==1 && !lp->txing)
545                 printk(KERN_WARNING "%s: Inconsistent state loading while not in tx\n",
546                         dev->name);
547
548         if (lp->txing)
549         {
550
551                 /*
552                  *      La carte est en mode emission. Peut etre en train de charger.
[...]
633         }
634         else
635         {
636                 /*
637                  *      En mode reception.
638                  */
[...]
673         }
674
675         /*
676          *      Passer dans l'etat reception
677          */
678
679         outb(AX_RX, AX_CMD);
680         outw(0x00, RX_BUF_CLR);
681         inb(RX_STATUS);         /* Etre certain que les interruptions sont mises a zero. */
682         inb(TX_STATUS);
683         spin_unlock(&lp->lock);
684 out:
685         return IRQ_HANDLED;
686 }
```

Le type général `pt_regs` spécifie la façon dont les registres sont stockés dansla pile lors d'un appel système. Il dépend de l'architecture. Il est défini dans le fichier `linux/include/asm-i386/ptrace.h` dans le cas des microprocesseurs Intel :

Code Linux 2.6.10

```
23 /* Cette struct definit la facon dont les registres sont stockes sur la
24    pile lors d'un appel systeme. */
25
26 struct pt_regs {
27         long ebx;
28         long ecx;
29         long edx;
30         long esi;
31         long edi;
32         long ebp;
33         long eax;
34         int  xds;
35         int  xes;
36         long orig_eax;
37         long eip;
38         int  xcs;
39         long eflags;
40         long esp;
41         int  xss;
42 };
```

Venons-en au commentaire du code :

– On déclare une adresse de descripteur de périphérique réseau, que l'on initialise avec celle

passée en argument.
- On déclare une adresse de structure réseau locale, un port d'entrée-sortie et l'adresse du registre auxiliaire de statut.
- On initialise le port d'entrée-sortie avec celui fourni par le descripteur de périphérique passé en argument.
- On initialise l'adresse de structure de réseau locale avec celle de la partie privée du descripteur de périphérique passé en argument.
- On verrouille la structure de réseau locale.
- On récupère le contenu du registre auxiliaire de statut pour connaître la raison de l'interruption matérielle (début de réception ou fin d'émission?).
- Si le niveau de débogage est strictement supérieur à 3, on affiche un message système précisant le contenu de ce registre.
- Si la structure de réseau locale précise qu'on est en train de recevoir des données sans être dans l'état émission, on affiche un message noyau à propos de cette incohérence.
- On traite le cas de l'émission ou de la réception suivant le cas, de la façon détaillée plus loin.
- On passe en mode réception dans les deux cas.
- On déverrouille la structure locale de réseau et on renvoie `IRQ_HANDLED`.

Code Linux 2.6.10

Les constantes symboliques générales `IRQ_HANDLED` et `IRQ_NONE`, la macro `IRQ_RETVAL()` et le type `irqreturn_t` sont définis dans le fichier `linux/include/linux/interrupt.h`:

```
16 /*
17  * Pour une compatibilite avec 2.4.x, 2.4.x peut utiliser
18  *
19  *      typedef void irqreturn_t;
20  *      #define IRQ_NONE
21  *      #define IRQ_HANDLED
22  *      #define IRQ_RETVAL(x)
23  *
24  * Pour melanger l'ancien style et le nouveau style, la routine d'irq renvoie quelque
      chose.
25  *
26  * IRQ_NONE signifie que nous ne l'avons pas traitee.
27  * IRQ_HANDLED signifie que nous avions une interruption valide et qu'elle a ete traitee.
28  * IRQ_RETVAL(x) selectionne une des deux suivant que x est non nul (pour traite)
29  */
30 typedef int irqreturn_t;
31
32 #define IRQ_NONE        (0)
33 #define IRQ_HANDLED     (1)
34 #define IRQ_RETVAL(x)   ((x) != 0)
```

19.1.2 Traitement en cas de réception

Code Linux 2.6.10

Reprenons le code pour nous intéresser à la partie concernant la réception:

```
519 static irqreturn_t el_interrupt(int irq, void *dev_id, struct pt_regs *regs)
[...]
635         {
636                 /*
637                  *      En mode reception.
638                  */
639
640                 int rxsr = inb(RX_STATUS);
641                 if (el_debug > 5)
642                         printk(KERN_DEBUG " rxsr=%02x txsr=%02x rp=%04x", rxsr,
```

```
                                inb(TX_STATUS),inw(RX_LOW));
643                 /*
644                  *      Juste lire rx_status cause beaucoup d'erreurs.
645                  */
646                 if (rxsr & RX_MISSED)
647                         lp->stats.rx_missed_errors++;
648                 else if (rxsr & RX_RUNT)
649                 {       /* Manipule pour eviter le verrouillage de la carte. */
650                         lp->stats.rx_length_errors++;
651                         if (el_debug > 5)
652                                 printk(KERN_DEBUG " runt.\n");
653                 }
654                 else if (rxsr & RX_GOOD)
655                 {
656                         /*
657                          *      La reception a marche.
658                          */
659                         el_receive(dev);
660                 }
661                 else
662                 {
663                         /*
664                          *      Rien ? Quelque chose a disparu !
665                          */
666                         if (el_debug > 2)
667                                 printk(KERN_DEBUG "%s: No packet seen,
                                               rxsr=%02x **resetting 3c501***\n",
668                                                dev->name, rxsr);
669                         el_reset(dev);
670                 }
671                 if (el_debug > 3)
672                         printk(KERN_DEBUG ".\n");
673         }
[...]
686 }
```

Autrement dit :

- Le gestionnaire d'interruption de cette carte récupère le contenu du registre de statut en réception.
- Si le niveau de débogage est supérieur à 6, on affiche un message noyau.
- Suivant le contenu du registre de statut en réception :
  - S'il indique qu'on a manqué quelque chose, on met à jour les statistiques à ce propos dans la structure locale de réseau.
  - S'il indique qu'il y a une erreur sur la longueur du paquet reçu, on met à jour les statistiques à ce propos dans la structure locale de réseau. Si le niveau de débogage est strictement supérieur à 5, on affiche un message noyau.
  - Si tout s'est bien déroulé, on fait appel à la fonction `el_receive()`, étudiée ci-après, de contrôle de la trame et de création d'un nouveau tampon de socket.
  - Sinon, quelque chose s'est mal déroulé: si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau; on réinitialise le périphérique.
- Si le niveau de débogage est supérieur à 3, on passe à la ligne dans les messages noyau.

19.2 Création d'un nouveau tampon de socket

Dans le cas d'une réception réussie, on doit vérifier la trame, allouer un nouveau tampon de socket, placer les données de la trame dans celui-ci et placer son descripteur dans une file d'attente.

Les trois premières actions sont l'objet de la fonction `el_receive()` dans le cas de la carte 3c501 de 3Com, qui est appelée par le gestionnaire d'interruption comme nous venons de le voir. Elle fait appel à une nouvelle fonction pour continuer le traitement. La fonction `el_receive()` est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
689 /**
690  * el_receive :
691  * @dev : [Descripteur de] peripherique depuis lequel extraire les paquets entrants
692  *
693  * Nous avons un bon paquet. En fait, pas exactement "bon", juste non perdu.
694  * Nous devons verifier chaque chose pour voir s'il est bon. En particulier nous
695  * obtenons un paquet brut convenant a la carte. Si le paquet semble sain, nous le retirons
696  * de la carte et le placons dans une file d'attente pour les couches de protocole.
697  */
698
699 static void el_receive(struct net_device *dev)
700 {
701         struct net_local *lp = netdev_priv(dev);
702         int ioaddr = dev->base_addr;
703         int pkt_len;
704         struct sk_buff *skb;
705
706         pkt_len = inw(RX_LOW);
707
708         if (el_debug > 4)
709                 printk(KERN_DEBUG " el_receive %d.\n", pkt_len);
710
711         if ((pkt_len < 60)  ||  (pkt_len > 1536))
712         {
713                 if (el_debug)
714                         printk(KERN_DEBUG "%s: bogus packet, length=%d\n",
                                dev->name, pkt_len);
715                 lp->stats.rx_over_errors++;
716                 return;
717         }
718
719         /*
720          *      Mode commande, ainsi nous pouvons vider le tampon
721          */
722
723         outb(AX_SYS, AX_CMD);
724         skb = dev_alloc_skb(pkt_len+2);
725
726         /*
727          *      Debut de la trame
728          */
729
730         outw(0x00, GP_LOW);
731         if (skb == NULL)
732         {
733                 printk(KERN_INFO "%s: Memory squeeze, dropping packet.\n",
                        dev->name);
734                 lp->stats.rx_dropped++;
735                 return;
736         }
737         else
```

```
738         {
739                 skb_reserve(skb,2);     /* Force l'alignement 16 octets */
740                 skb->dev = dev;
741                 /*
742                  *      La lecture incremente [l'adresse des] octets. Le gestionnaire
743                  *      d'interruption rectifiera le pointeur lorsqu'il reviendra en
744                  *      mode reception.
745                  */
746                 insb(DATAPORT, skb_put(skb,pkt_len), pkt_len);
747                 skb->protocol=eth_type_trans(skb,dev);
748                 netif_rx(skb);
749                 dev->last_rx = jiffies;
750                 lp->stats.rx_packets++;
751                 lp->stats.rx_bytes+=pkt_len;
752         }
753         return;
754 }
```

Autrement dit :

- On déclare une adresse de structure locale de réseau, que l'on initialise avec celle de la partie privée du descripteur de périphérique réseau passé en argument.
- On déclare un port d'entrée-sortie, que l'on initialise avec celui associé au descripteur de périphérique réseau passé en argument.
- On déclare une longueur de paquet et un descripteur de tampon de socket.
- On renseigne la longueur du paquet reçu à partir de la valeur du pointeur de réception de la carte 3Com 501.
- Si le niveau de débogage est strictement supérieur à 4, on affiche celle-ci comme message noyau.
- Si cette longueur est strictement inférieure à 60 ou strictement supérieure à 1 536, il ne s'agit pas d'une longueur de trame Ethernet permise : si le niveau de débogage est non nul, on affiche un message noyau à ce propos ; on met à jour les informations statistiques sur ce type d'erreur dans la structure locale de réseau et on a terminé.

  Remarquons qu'il aurait mieux vallu utiliser une constante plutôt qu'un nombre magique : 1 536, c'est 1 514 plus deux adresses MAC plus le CRC.
- On se place en mode commande.
- On essaie d'instantier un nouveau descripteur de tampon de socket, dans le champ données duquel on placera le paquet. Si on n'y parvient pas, on affiche un message noyau, on met à jour les informations statistiques sur ce type d'erreur dans la structure locale de réseau et on a terminé.
- On réinitialise le pointeur de tampon de la carte à zéro.
- On initialise le tampon de socket : on aligne sur un multiple de 16 octets, on détermine l'adresse de l'emplacement du champ des données que l'on communique au contrôleur de la carte, on récupère les données de la trame reçue que l'on place dans le champ données du tampon.
- On renseigne les champs du descripteur de tampon de socket : le descripteur de périphérique associé est celui passé en argument ; on positionne le type de protocole de la couche réseau, déterminé grâce à la fonction `eth_type_trans()` étudiée ci-dessous.
- On place le descripteur du nouveau tampon dans la file d'attente en réception du microprocesseur en cours, grâce à la fonction `netif_rx()` étudiée ci-dessous.
- On met à jour l'heure de la dernière réception de paquet par le périphérique ainsi que les informations statistiques dans la structure locale de réseau du descripteur de périphérique et on a terminé.

19.3 Détermination du protocole de couche réseau

La fonction `short eth_type_trans(skb, dev)` est l'un des éléments les plus importants de l'implémentation Linux de la sous-couche LLC. Deux tâches essentielles sont exécutées :

- on détermine le protocole de la couche réseau au moyen des paramètres placés dans la trame et on renvoie cette valeur ;
- on détermine le type de paquet (individuel, diffusion restreinte, diffusion générale) et on vérifie si le paquet est destiné à l'ordinateur présent.

La fonction `eth_type_trans()`, utilisable pour toutes les cartes réseau compatibles Ethernet, est en général appelée par le pilote de réseau dans la méthode de réception des paquets, ainsi que lors de l'émission dans le cas du périphérique en boucle. Tous les périphériques réseau d'un type de protocole MAC utilisent la même méthode `type_trans()`. Les périphériques *Token Ring* et FDDI disposent de méthodes correspondantes (`tr_type_trans()` et `fddi_type_trans()` respectivement).

Les réseaux Ethernet n'utilisent aucune des normes LLC. Elles transmettent directement l'identification de protocole de la couche réseau trouvée dans la trame MAC. Le seul mécanisme de protocole est le démultiplexage des différents protocoles de couche réseau. Par conséquent `eth_type_trans()` est assez simple et claire.

Code Linux 2.6.10

La fonction `eth_type_trans()` est définie dans le fichier `linux/net/ethernet/eth.c`:

```
153 /*
154  *      Determine l'identificateur de protocole du paquet. La regle est de considerer ici
155  *      qu'il s'agit de 802.3 si le champ type est suffisamment court pour etre
156  *      une taille. C'est la pratique normale et cela marche pour les protocoles
         'actuellement en usage'.
157  */
158
159 unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev)
160 {
161         struct ethhdr *eth;
162         unsigned char *rawp;
163
164         skb->mac.raw=skb->data;
165         skb_pull(skb,ETH_HLEN);
166         eth = eth_hdr(skb);
167         skb->input_dev = dev;
168
169         if(*eth->h_dest&1)
170         {
171                 if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
172                         skb->pkt_type=PACKET_BROADCAST;
173                 else
174                         skb->pkt_type=PACKET_MULTICAST;
175         }
176
177         /*
178          *      Cette verification ALLMULTI devrait etre redondante d'apres 1.4
179          *      aussi n'oubliez pas de l'enlever.
180          *
181          *      Il semblerait que vous ayez oublie de l'enlever. Tous les
182          *      peripheriques stupides semblent positionner IFF_PROMISC.
183          */
184
185         else if(1 /*dev->flags&IFF_PROMISC*/)
186         {
187                 if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
```

```
188                         skb->pkt_type=PACKET_OTHERHOST;
189         }
190
191         if (ntohs(eth->h_proto) >= 1536)
192                 return eth->h_proto;
193
194         rawp = skb->data;
195
196         /*
197          *      Ceci est un truc magique pour reperer les paquets IPX. Les anciens Novell ne
199          *      respectent pas le protocole et realisent IPX sur 802.3 sans couche LLC 802.2.
199          *      Nous recherchons FFFF qui n'est pas un SSAP/DSAP 802.2 utilise. Ceci ne
200          *      marchera pas pour les reseaux tolerant les fautes mais marchera pour les
                 autres.
201          */
202         if (*(unsigned short *)rawp == 0xFFFF)
203                 return htons(ETH_P_802_3);
204
205         /*
206          *      Vrai LLC 802.2
207          */
208         return htons(ETH_P_802_2);
209 }
```

Autrement dit :

- On déclare une adresse d'en-tête Ethernet et une chaîne de caractères.
- On positionne le début des données brutes du tampon de socket au début des données du tampon de socket passé en argument.
- On tronque le début du tampon de la taille d'un en-tête Ethernet.
- On initialise l'adresse d'en-tête Ethernet avec celle du champ en-tête Ethernet du tampon de socket.

  La fonction en ligne `eth_hdr()` est définie dans le fichier `linux/include/linux/if_ether.h` :

  Code Linux 2.6.10

```
108 static inline struct ethhdr *eth_hdr(const struct sk_buff *skb)
109 {
110         return (struct ethhdr *)skb->mac.raw;
111 }
```

- On renseigne le champ périphérique d'entrée du descripteur de tampon avec le descripteur de périphérique passé en argument.
- Si le type du paquet est diffusion générale, multidiffusion ou mode promiscuité, ceci est consigné dans le champ adéquat du descripteur de tampon passé en argument.
- Le protocole réseau du paquet admis est ensuite déterminé :
  - Lorsqu'il existe dans le champ longueur et/ou protocole de l'en-tête de la trame Ethernet une valeur supérieure à la longueur de la trame maximale (1 536 octets), on part du principe qu'il s'agit d'une carte Ethernet compatible avec la norme 802.3. Nous avons vu que le protocole 802.3 intègre le type de protocole de la couche réseau dans le champ protocole de la trame 802.3. Par conséquent, la valeur du champ est tout simplement renvoyée.
  - Les cartes plus anciennes ne stockent pas l'identification du protocole de couche réseau dans le champ longueur et/ou protocole, mais la longueur de la trame. Linux renvoie tout simplement `ETH_P_802_2` à titre d'identification de protocole de couche réseau car 802.2 est plutôt un cas exceptionnel. Le processus de démultiplexage proprement dit aura lieu dans la routine de traitement du protocole 802.2, à savoir `llc_rcv()`.

19.4 Mise du nouveau tampon en file d'attente

Les paquets reçus par l'ordinateur sont placés dans une file d'attente, une par microprocesseur dans le cas d'un système multiprocesseurs. Commençons par étudier celles-ci.

19.4.1 File d'attente d'un microprocesseur

19.4.1.1 Structure de données

Code Linux 2.6.10

Ces files d'attente sont des entités du type `struct softnet_data`, défini dans le fichier `linux/include/linux/netdevice.h`:

```
563 /*
564  * Les paquets recus sont places dans des files d'attente, une par microprocesseur,
565  * donc aucun verrouillage n'est necessaire.
566  */
567
568 struct softnet_data
569 {
570         int                     throttle;
571         int                     cng_level;
572         int                     avg_blog;
573         struct sk_buff_head     input_pkt_queue;
574         struct list_head        poll_list;
575         struct net_device       *output_queue;
576         struct sk_buff          *completion_queue;
577
578         struct net_device       backlog_dev;    /* Desole. 8) */
579 };
```

dont les champs ont la signification suivante:

- Une valeur non nulle pour `throttle` signifie que la file d'attente est pleine. On ne peut plus y placer de nouvel élément.
- Le champ `cng_level` spécifie le niveau de congestion de la file d'attente. Cette information est utile avant d'en arriver au cas extrême représenté par le champ précédent.

  Les valeurs possibles sont représentées par l'une des trois constantes symboliques parmi les suivantes:

  - `NET_RX_SUCCESS` en cas de réception réussie;
  - `NET_RX_DROP` si le paquet a été détruit ('écarté' est certes plus beau, mais c'est la même chose) pour une raison quelconque, par exemple s'il n'est pas destiné à l'ordinateur;
  - `NET_RX_CN_LOW` (avec `CN` pour *CoNgested*) lorsque le canal commence à être congestionné (mais on a quand même reçu le paquet);
  - `NET_RX_CN_MOD` (avec `MOD` pour *MODerate*) lorsque le canal commence vraiment à être congestionné (on a eu du mal à récupérer le paquet);
  - `NET_RX_CN_HIGH` lorsque le canal est vraiment congestionné;
  - `NET_RX_BAD` lorsque le paquet a été détruit à cause d'une erreur concernant le noyau.

  Comme on le voit, cette série de constantes symboliques sert également à d'autres propos.

  Code Linux 2.6.10

  Ces constantes symboliques spécifiant l'état de la réception sont définies dans le fichier `linux/include/linux/netdevice.h`:

```
62 /* Niveaux de congestion d'accumulation */
63 #define NET_RX_SUCCESS          0   /* Bien arrive, cherie */
64 #define NET_RX_DROP             1   /* paquet ecarte */
```

```
65 #define NET_RX_CN_LOW           2   /* alerte de tempete, on ne sait jamais */
66 #define NET_RX_CN_MOD           3   /* La tempete arrive */
67 #define NET_RX_CN_HIGH          4   /* La tempete est la */
68 #define NET_RX_BAD              5   /* paquet ecarte du a une erreur du noyau */
```

- Le champ `input_pkt_queue` est la file d'attente de descripteurs de tampon de socket proprement dite. Il s'agit donc du champ le plus important de cette structure. Les autres champs correspondent à un descripteur de cette file d'attente.
- Le champ `poll_list` est une liste d'élection.
- Le champ `output_queue` est un pointeur sur le périphérique d'où provient l'élément de la file d'attente en train d'être traité.
- Le champ `completion_queue` est un pointeur sur le descripteur du tampon de socket.
- Le périphérique (virtuel) `backlog_dev` permet de passer la main à la couche supérieure, c'est-à-dire à la couche réseau, pour continuer le traitement du paquet, grâce à sa fonction `poll()`, comme nous le verrons.

19.4.1.2 Déclaration

Une file d'attente de ce type est créée pour chaque microprocesseur, comme le montre l'extrait suivant du fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
581 DECLARE_PER_CPU(struct softnet_data,softnet_data);
```

La taille maximale d'une telle file d'attente est limitée à 300. Cette constante est représentée par la constante symbolique (en fait la variable) `netdev_max_backlog` définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1340 int netdev_max_backlog = 300;
```

Son nom provient de ce qu'elle désignait le nombre de clients pouvant se connecter simultanément à un serveur (*backlog* en anglais) dans les versions antérieures du noyau.

19.4.1.3 Initialisation des files d'attente

Ces files d'attente sont initialisées lors du démarrage du système de la façon indiquée dans la fonction `net_dev_init()`, définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
3139 static int __init net_dev_init(void)
3140 {
[...]
3163         /*
3164          *      Initialise les files d'attente des paquets en reception.
3165          */
3166
3167         for (i = 0; i < NR_CPUS; i++) {
3168                 struct softnet_data *queue;
3169
3170                 queue = &per_cpu(softnet_data, i);
3171                 skb_queue_head_init(&queue->input_pkt_queue);
3172                 queue->throttle = 0;
3173                 queue->cng_level = 0;
3174                 queue->avg_blog = 10; /* non nul arbitraire */
3175                 queue->completion_queue = NULL;
3176                 INIT_LIST_HEAD(&queue->poll_list);
3177                 set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
3178                 queue->backlog_dev.weight = weight_p;
3179                 queue->backlog_dev.poll = process_backlog;
3180                 atomic_set(&queue->backlog_dev.refcnt, 1);
```

```
3181         }
[...]
3199 }
```

Code Linux 2.6.10

La constante (en fait variable) `weight_p` est définie dans le même fichier :

```
1341 int weight_p = 64;            /* Ancien nombre maximum de clients */
```

Nous reviendrons ci-dessous sur l'importante fonction `process_backlog()`.

19.4.2 Fonction de mise en file d'attente

La fonction `netif_rx(skb)` place le descripteur de tampon de socket passé en argument dans la file d'attente en réception du microprocesseur en cours, passe la main à la couche supérieure et renvoie une indication sur le niveau de congestion de la file d'attente, correspondant à l'une des constantes symboliques vues ci-dessus.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/net/core/dev.c` :

```
1405 /**
1406  *      netif_rx        -       met a disposition le tampon au code du reseau
1407  *      @skb : [descripteur de] tampon a mettre a disposition
1408  *
1409  *      Cette fonction recoit un paquet d'un peripherique reseau et le place dans une file
1410  *      d'attente pour etre traitee par les couches (de protocole) superieures. Elle
1411  *      reussit toujours. Le tampon peut etre ecarte durant le traitement par le
1412  *      controle de congestion ou par les couches de protocole.
1413  *
1414  *      valeurs de retour :
1415  *      NET_RX_SUCCESS  (pas de congestion)
1416  *      NET_RX_CN_LOW   (congestion basse)
1417  *      NET_RX_CN_MOD   (congestion moderee)
1418  *      NET_RX_CN_HIGH  (congestion importante)
1419  *      NET_RX_DROP     (le paquet a ete ecarte)
1420  *
1421  */
1422
1423 int netif_rx(struct sk_buff *skb)
1424 {
1425         int this_cpu;
1426         struct softnet_data *queue;
1427         unsigned long flags;
1428
1429 #ifdef CONFIG_NETPOLL
1430         if (skb->dev->netpoll_rx && netpoll_rx(skb)) {
1431                 kfree_skb(skb);
1432                 return NET_RX_DROP;
1433         }
1434 #endif
1435
1436         if (!skb->stamp.tv_sec)
1437                 net_timestamp(&skb->stamp);
1438
1439         /*
1440          * Le code est arrange de telle sorte que le chemin soit le plus
1441          * court lorsque le microprocesseur est congestionne, mais encore operationnel.
1442          */
1443         local_irq_save(flags);
1444         this_cpu = smp_processor_id();
1445         queue = &__get_cpu_var(softnet_data);
1446
1447         __get_cpu_var(netdev_rx_stat).total++;
```

```
1448         if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
1449                 if (queue->input_pkt_queue.qlen) {
1450                         if (queue->throttle)
1451                                 goto drop;
1452
1453 enqueue:
1454                         dev_hold(skb->dev);
1455                         __skb_queue_tail(&queue->input_pkt_queue, skb);
1456 #ifndef OFFLINE_SAMPLE
1457                         get_sample_stats(this_cpu);
1458 #endif
1459                         local_irq_restore(flags);
1460                         return queue->cng_level;
1461                 }
1462
1463                 if (queue->throttle)
1464                         queue->throttle = 0;
1465
1466                 netif_rx_schedule(&queue->backlog_dev);
1467                 goto enqueue;
1468         }
1469
1470         if (!queue->throttle) {
1471                 queue->throttle = 1;
1472                 __get_cpu_var(netdev_rx_stat).throttled++;
1473         }
1474
1475 drop:
1476         __get_cpu_var(netdev_rx_stat).dropped++;
1477         local_irq_restore(flags);
1478
1479         kfree_skb(skb);
1480         return NET_RX_DROP;
1481 }
```

Autrement dit :

- On déclare un numéro de processeur, utile dans le cas d'un système multi-processeurs.
- On déclare une file d'attente en réception de microprocesseur.
- On déclare un vecteur de drapeaux.
- Si l'heure de réception du tampon dont le descripteur est passé en argument n'est pas spécifiée, on lui attribue l'heure en cours.
- On sauvegarde les valeurs des registres du microprocesseur en cours dans le vecteur de drapeaux.
- On détermine le numéro du microprocesseur en cours.
- On instantie la file d'attente avec celle associée à ce microprocesseur.
- On incrémente l'information sur le nombre total de trames reçues par ce microprocesseur.
- Si la taille de la file d'attente en réception du microprocesseur est inférieure à la taille maximale (autrement dit si cette file d'attente n'est pas pleine) :
  - Si la taille de la file d'attente en réception du microprocesseur est non nulle (autrement dit si la file d'attente n'est pas vide) :
    - S'il est indiqué que la file d'attente est pleine, on détruit le paquet : on incrémente l'information sur le nombre de paquets détruits pour ce microprocesseur, on restaure les valeurs des registres du microprocesseur, on libère le tampon de socket et on renvoie `NET_RX_DROP`.

- Sinon (c'est-à-dire dans le cas normal) :
  - On incrémente le compteur de références au périphérique.
  - On ajoute le descripteur de tampon passé en argument à la fin de la file d'attente en réception du microprocesseur.
  - On met éventuellement à jour les informations statistiques concernant le microprocesseur en cours.
  - On restaure les valeurs des registres du microprocesseur.
  - On renvoie le niveau de congestion du microprocesseur, tel qu'indiqué dans la file d'attente.
- Si la taille de la file d'attente en réception du microprocesseur est nulle (autrement dit si la file d'attente est vide) :
  - Si cette file d'attente est marquée comme pleine, on rectifie cette indication et, éventuellement, on réveille la file d'attente.
  - On fait appel à la fonction `netif_rx_schedule()`, étudiée dans la section suivante, en passant en argument le descripteur de périphérique virtuel, afin de passer à la couche supérieure.
  - On essaie à nouveau de placer le descripteur de tampon dans la file d'attente.
- Si la taille de la file d'attente est supérieure à la constante (autrement dit si elle est pleine) :
  - S'il n'est pas indiqué que la file d'attente est pleine, on le fait et on met à jour les informations statistiques à ce propos pour le microprocesseur.
  - On écarte le paquet.

19.5 Passage à la couche supérieure

La fonction de passage à la couche supérieure considère un à un les descripteurs de tampon de socket placés dans la file d'attente du microprocesseur. Elle les communique à la fonction de traitement associée au protocole indiqué dans le champs créé à cet effet. Avant d'entrer dans le détail de l'action, voyons les structures de données nécessaires à ce passage.

19.5.1 Type de paquet

Pour remettre un paquet à la couche supérieure, on regarde le champ `protocol` de son descripteur et on communique son descripteur à une fonction de traitement dépendant de ce champ. La détermination de cette fonction s'effectue en parcourant deux listes de *types de paquet*.

19.5.1.1 Structure de données de type de paquet

Le type d'un paquet est une instance du type `struct packet_type` défini dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
509 struct packet_type {
510         unsigned short          type;   /* Il s'agit de htons(ether_type).          */
511         struct net_device       *dev;   /* NULL comme caractere de remplacement ici */
512         int                     (*func) (struct sk_buff *, struct net_device *,
513                                          struct packet_type *);
514         void                    *af_packet_priv;
515         struct list_head        list;
516 };
```

dont la signification des champs est la suivante :

- Le champ `type` spécifie l'identificateur de protocole. Il s'agit de l'une des constantes vues au chapitre 16.
- Le champ `dev` peut être occupé par un pointeur sur un périphérique réseau. Dans ce cas, seuls les paquets reçus par ce périphérique sont transmis au protocole. Si plusieurs adaptateurs réseau, mais pas tous, doivent bénéficier de cette priorité, il faut que le protocole soit enregistré séparément pour chaque périphérique réseau. Lorsque `dev` contient `NULL`, ce qui est le cas usuel, le périphérique réseau d'entrée ne joue aucun rôle dans le choix du protocole.
- Le champ `func()` est la routine de traitement prévue pour commencer le traitement du protocole.
- Le champ `af_packet_priv` peut contenir des données supplémentaires pour ce protocole. Il n'est pas utilisé en général.
- Le champ `list` sert à chaîner les différentes instances de cette structure.

Par exemple, le type de paquet associé à IP s'appelle `ip_packet_type` et est défini dans le fichier `linux/net/ipv4/ip_output.c` : Code Linux 2.6.10

```
1323 /*
1324  *      Initialiseur de la couche de protocole IP
1325  */
1326
1327 static struct packet_type ip_packet_type = {
1328         .type = __constant_htons(ETH_P_IP),
1329         .func = ip_rcv,
1330 };
```

qui nous donne `ip_rcv()` comme fonction de traitement.

19.5.1.2 Listes des types de paquet

Il existe deux structures de données concernant les types de paquets, comme le montre la figure 19.1 ([WPRMB–02], p.148) :

- L'une, une liste simplement chaînée appelée `ptype_all`, stocke les protocoles qui doivent recevoir tous les tampons de sockets entrants. En règle générale, aucun protocole n'est enregistré dans cette liste. Toutefois elle convient parfaitement à l'insertion d'outils d'analyse comme les renifleurs.
- L'autre, une table de hachage appelée `ptype_base[]`, comprend les protocoles qui ne reçoivent que les paquets comportant une identification de protocole correcte, par exemple `0x08 00` pour le protocole IP.

Ces structures de données sont définies dans le le fichier `linux/net/core/dev.c` : Code Linux 2.6.10

```
129 /*
130  *      La liste des types des paquets que nous recevons (par opposition a ceux abandonnes)
131  *      et les routines a invoquer.
132  *
133  *      Pourquoi 16 ? Parce qu'avec 16 le seul chevauchement que nous obtiendrons avec un
134  *      hachage du demi-octet de poids faible de la valeur du protocole est RARP/SNAP/X.25.
135  *
136  *      NOTE : Ceci n'est plus vrai avec l'addition des valeurs de VLAN.
137  *             Ce qui arrivera en premier n'est pas sur, mais je mise qu'il n'y aura
138  *             pas beaucoup de difference si nous utilisons les VLAN. La bonne nouvelle
139  *             est que ce protocole ne sera pas dans la liste a moins d'etre compile, donc
```

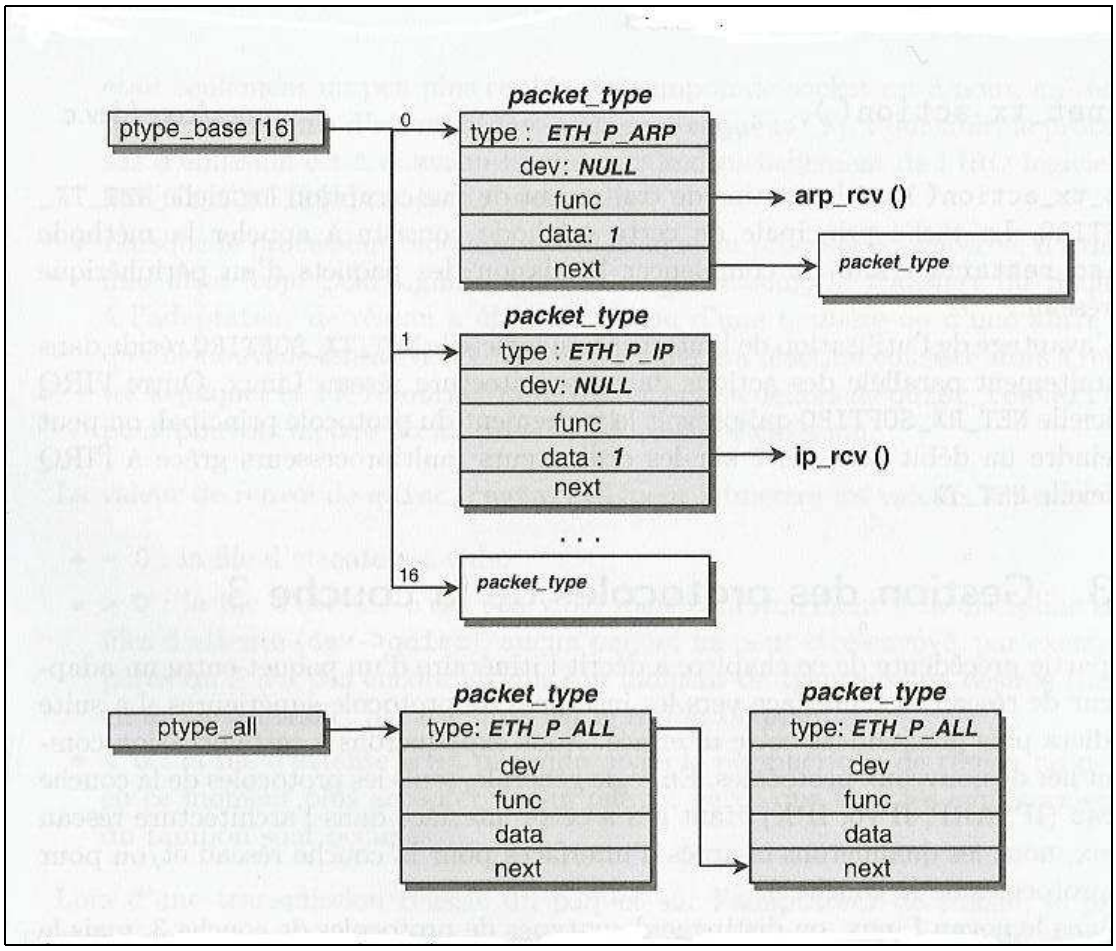

FIG. 19.1 – *Gestion des protocoles de réseau*

```
140  *             l'utilisateur moyen ne sera pas defavorablement affecte.
141  *             --BLG
142  *
143  *              0800    IP
144  *              8100    802.1Q VLAN
145  *              0001    802.3
146  *              0002    AX.25
147  *              0004    802.2
148  *              8035    RARP
149  *              0005    SNAP
150  *              0805    X.25
151  *              0806    ARP
152  *              8137    IPX
153  *              0009    Localtalk
154  *              86DD    IPv6
155  */
156
157 static spinlock_t ptype_lock = SPIN_LOCK_UNLOCKED;
158 static struct list_head ptype_base[16];  /* liste de hachage a 16 possibilites */
159 static struct list_head ptype_all;       /* Mise sur ecoute [Taps] */
```

Ces structures de données sont initialisées lors du démarrage du système de la façon indiquée dans la fonction `net_dev_init()`, définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
3139 static int __init net_dev_init(void)
3140 {
[...]
3153         INIT_LIST_HEAD(&ptype_all);
3154         for (i = 0; i < 16; i++)
3155                 INIT_LIST_HEAD(&ptype_base[i]);
[...]
3199 }
```

19.5.1.3 Ajout et retrait d'un type de paquet

La gestion des types de paquet est effectuée grâce aux deux méthodes suivantes:

- `dev_add_pack(packet_type)` place le type de paquet dans l'une des deux structures de données. Si le champ `type` contient la valeur `ETH_P_ALL`, il est ajouté à la liste `ptype_all`. Sinon il est inséré sur la ligne correspondante de la table de hachage `ptype_base`.
- `dev_remove_pack(packet_type)` retire le type de paquet de la structure de données dans laquelle il se trouve.

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

```
242 /*
243  *      Ajoute un identificateur de protocole a la liste. Maintenant que la routine d'entree
244  *      est plus maligne, nous pouvons nous dispenser de toutes les choses compliquees
245  *      qui ont ete utilisees jusqu'ici.
246  *
247  *      ATTENTION !!! Les routines de protocole mutilant les paquets d'entree
248  *      DOIVENT ETRE les dernieres dans les chaines de hachage. Verifier que les
249  *      routines de protocole DOIVENT demarrer par la chaine confuse ptype_all dans net_bh.
250  *      C'est vrai ici, ne pas changer.
251  *      Des explications suivent : si une routine de protocole mutilant les paquets etait
252  *      la premiere sur la liste, elle ne serait pas capable de savoir que le paquet
253  *      doit etre clone et copie, aussi serait-il change
254  *      et les lectures ulterieures obtiendraient des paquets incomplets.
255  *                                                      --ANK (980803)
256  */
257
258 /**
259  *      dev_add_pack - ajoute une routine de paquet
260  *      @pt : declaration du type de paquet
261  *
262  *      Ajoute une routine de protocole a la pile reseau. Le &packet_type passe
263  *      est lie dans les listes du noyau et ne peut pas etre libere avant qu'il ne soit
264  *      retire des listes du noyau.
265  *
266  *      Cet appel ne s'assoupit pas, aussi ne peut-il pas
267  *      garantir que tous les microprocesseurs qui sont en train de recevoir des paquets
268  *      verront le nouveau type de paquet (jusqu'au paquet recu suivant).
269  */
270
271 void dev_add_pack(struct packet_type *pt)
272 {
273         int hash;
274
275         spin_lock_bh(&ptype_lock);
276         if (pt->type == htons(ETH_P_ALL)) {
277                 netdev_nit++;
278                 list_add_rcu(&pt->list, &ptype_all);
```

```
279         } else {
280                 hash = ntohs(pt->type) & 15;
281                 list_add_rcu(&pt->list, &ptype_base[hash]);
282         }
283         spin_unlock_bh(&ptype_lock);
284 }
```

Code Linux 2.6.10

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

```
290 /**
291  *      __dev_remove_pack        - retire une routine de paquet
292  *      @pt : declaration du type de paquet
293  *
294  *      Retire une routine de protocole qui a ete precedemment ajoutee aux routines de
295  *      protocole du noyau avec dev_add_pack(). Le &packet_type passe est retire
296  *      des listes du noyau et peut ainsi etre libere ou reutilise une fois que cette
297  *      fonction est terminee.
298  *
299  *      Le type de paquet peut encore etre en train d'etre utilise par les recepteurs
300  *      et ne doit pas etre libere avant que tous les microprocesseurs soient entres
301  *      dans un etat latent.
302  */
303 void __dev_remove_pack(struct packet_type *pt)
304 {
305         struct list_head *head;
306         struct packet_type *pt1;
307
308         spin_lock_bh(&ptype_lock);
309
310         if (pt->type == htons(ETH_P_ALL)) {
311                 netdev_nit--;
312                 head = &ptype_all;
313         } else
314                 head = &ptype_base[ntohs(pt->type) & 15];
315
316         list_for_each_entry(pt1, head, list) {
317                 if (pt == pt1) {
318                         list_del_rcu(&pt->list);
319                         goto out;
320                 }
321         }
322
323         printk(KERN_WARNING "dev_remove_pack: %p not found.\n", pt);
324 out:
325         spin_unlock_bh(&ptype_lock);
326 }
327 /**
328  *      dev_remove_pack  - retire une routine de paquet
329  *      @pt : declaration du type de paquet
330  *
331  *      Retire une routine de protocole qui avait ete anterieurement ajoutee aux routines
332  *      de protocole du noyau avec dev_add_pack(). Le &packet_type passe est retire
333  *      des listes du noyau et peut ainsi etre libere ou reutilise une fois que cette
334  *      fonction est terminee.
335  *
336  *      Cet appel s'assoupit afin de garantir qu'aucun microprocesseur n'est a la recherche
337  *      de type de paquet apres retour de la fonction.
338  */
339 void dev_remove_pack(struct packet_type *pt)
340 {
341         __dev_remove_pack(pt);
342
343         synchronize_net();
```

```
344 }
```

19.5.2 Processus de passage à la couche supérieure

Le passage à la couche supérieure a lieu par l'intermédiaire de l'interruption logicielle `NET_RX_SOFTIRQ`. Celle-ci n'a pas besoin de bloquer les interruptions et peut se faire au gré de l'ordonnanceur.

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

```
826 /* Essaie de reprogrammer l'election. Appelee par une routine d'interruption. */
827
828 static inline void netif_rx_schedule(struct net_device *dev)
829 {
830         if (netif_rx_schedule_prep(dev))
831                 __netif_rx_schedule(dev);
832 }
```

Elle prépare au passage puis elle passe la main à la couche réseau de façon proprement dite.

19.5.2.1 Préparation

La fonction de préparation `netif_rx_schedule_prep()` est définie dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
799 /* Programme l'interruption de reception maintenant ? */
800
801 static inline int netif_rx_schedule_prep(struct net_device *dev)
802 {
803         return netif_running(dev) &&
804                 !test_and_set_bit(__LINK_STATE_RX_SCHED, &dev->state);
805 }
```

Autrement dit:

- On vérifie que la carte est active et peut être utilisée.
  La fonction `netif_running()` est définie plus haut dans le même fichier: Code Linux 2.6.10

  ```
  635 static inline int netif_running(const struct net_device *dev)
  636 {
  637         return test_bit(__LINK_STATE_START, &dev->state);
  638 }
  ```

- On positionne l'état du descripteur de peripherique reseau pour indiquer que l'on va passer à l'interruption logicielle.

19.5.2.2 Appel de l'interruption logicielle

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

```
807 /* Ajoute une interface en debut de liste d'election en reception. Ceci suppose que _prep a
808  * deja ete appele et qu'il a renvoye 1.
809  */
810
811 static inline void __netif_rx_schedule(struct net_device *dev)
812 {
813         unsigned long flags;
814
```

```
815         local_irq_save(flags);
816         dev_hold(dev);
817         list_add_tail(&dev->poll_list, &__get_cpu_var(softnet_data).poll_list);
818         if (dev->quota < 0)
819                 dev->quota += dev->weight;
820         else
821                 dev->quota = dev->weight;
822         __raise_softirq_irqoff(NET_RX_SOFTIRQ);
823         local_irq_restore(flags);
824 }
```

Autrement dit :

- On déclare un vecteur de drapeaux.
- On sauvegarde les valeurs des registres du microprocesseur en cours.
- On incrémente le compteur d'utilisation du descripteur de périphérique réseau passé en argument.
- On ajoute les éléments de la file d'attente du descripteur de périphérique passé en entrée `dev->poll_list` à la liste de files d'attente en réception du microprocesseur en cours.

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

Code Linux 2.6.10

```
70 /**
71  * list_add_tail - ajoute une nouvelle entree
72  * @new : nouvelle entree a etre ajoutee
73  * @head : tete de la liste a laquelle il faut ajouter
74  *
75  * Insere une nouvelle entree avant la tete specifiee.
76  * Ceci est utile pour implementer les files d'attente.
77  */
78 static inline void list_add_tail(struct list_head *new, struct list_head *head)
79 {
80         __list_add(new, head->prev, head);
81 }
```

- On met à jour le quota du descripteur de périphérique réseau.
- On fait appel à l'interruption logicielle `NET_RX_SOFTIRQ` pour passer au descripteur de périphérique virtuel.

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

Code Linux 2.6.10

```
108 #define __raise_softirq_irqoff(nr) do { local_softirq_pending() |= 1UL << (nr); }
                                        while (0)
```

- On restaure les valeurs des registres du microprocesseur.

19.5.2.3 Routine de service de l'interruption logicielle

La routine de service de l'interruption logicielle `NET_RX_SOFTIRQ` est précisée dans la fonction `net_dev_init()`, définie dans le fichier `linux/net/core/dev.c` :

Code Linux 2.6.10

```
3139 static int __init net_dev_init(void)
3140 {
[...]
3190         open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
3191         open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
[...]
3199 }
```

19.5.2.4 Appel du descripteur de périphérique virtuel

La routine de service `net_rx_action()` de l'interruption logicielle `NET_RX_SOFTIRQ` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1764 static void net_rx_action(struct softirq_action *h)
1765 {
1766         struct softnet_data *queue = &__get_cpu_var(softnet_data);
1767         unsigned long start_time = jiffies;
1768         int budget = netdev_max_backlog;
1769
1770
1771         local_irq_disable();
1772
1773         while (!list_empty(&queue->poll_list)) {
1774                 struct net_device *dev;
1775
1776                 if (budget <= 0 || jiffies - start_time > 1)
1777                         goto softnet_break;
1778
1779                 local_irq_enable();
1780
1781                 dev = list_entry(queue->poll_list.next,
1782                                  struct net_device, poll_list);
1783
1784                 if (dev->quota <= 0 || dev->poll(dev, &budget)) {
1785                         local_irq_disable();
1786                         list_del(&dev->poll_list);
1787                         list_add_tail(&dev->poll_list, &queue->poll_list);
1788                         if (dev->quota < 0)
1789                                 dev->quota += dev->weight;
1790                         else
1791                                 dev->quota = dev->weight;
1792                 } else {
1793                         dev_put(dev);
1794                         local_irq_disable();
1795                 }
1796         }
1797 out:
1798         local_irq_enable();
1799         return;
1800
1801 softnet_break:
1802         __get_cpu_var(netdev_rx_stat).time_squeeze++;
1803         __raise_softirq_irqoff(NET_RX_SOFTIRQ);
1804         goto out;
1805 }
```

Autrement dit:

- On déclare une file d'attente en réception de microprocesseur, que l'on instantie avec celle du microprocesseur en cours.
- On déclare une heure de début, que l'on instantie avec l'heure en cours.
- On déclare un nombre de paquets restants à traiter, que l'on initialise avec le nombre maximum d'éléments permis dans une telle file d'attente.
- On inhibe les interruptions sur le microprocesseur en cours.
- Tant que la liste d'élection de la file d'attente en réception du microprocesseur n'est pas vide:
  - On déclare un descripteur de périphérique réseau.

- Si le nombre de paquets à traiter est (négatif ou) nul ou si un tick s'est écoulé depuis le début, on donne la main au gestionnaire des tâches pour qu'une autre activité puisse se dérouler. On évite ainsi que le traitement bloque le reste des activités de l'ordinateur. De façon détaillée : on incrémente l'information statistique sur le nombre de pincements de la file d'attente du microproceseur en cours, on lève `NET_RX_SOFTIRQ` et enfin on restaure les interruptions.
- On accepte à nouveau les interruptions sur ce microprocesseur.
- On instantie le descripteur de périphérique réseau avec le descripteur de périphérique réseau virtuel de la file d'attente du microprocesseur en cours.
- On fait appel à la fonction d'élection du descripteur de périphérique virtuel.

  Nous avons vu lors de l'initialisation des files d'attente en réception de microprocesseur que cette fonction d'élection est `proces_backlog()`. Nous l'étudierons dans la section suivante.
- Si le quota associé au descripteur de périphérique réseau virtuel est (négatif ou) nul ou si la fonction d'élection renvoie une valeur non nulle : on inhibe les interruptions sur ce microprocesseur, on retire un élément de la liste d'élection du descripteur de périphérique réseau virtuel, on fait passer un élément de la liste d'élection de la file d'attente en réception du microprocesseur vers la liste d'élection du descripteur de périphérique réseau virtuel et on met à jour le quota du descripteur de périphérique réseau virtuel.
- Sinon on libère le descripteur de périphérique réseau virtuel et on inhibe les interruptions.

- On permet à nouveau les interruptions et on a terminé.

19.6 Préparation au traitement dans la couche supérieure

19.6.1 Détermination du paquet à traiter

Nous venons de voir que la fonction précédente fait appel à la fonction `process_backlog()`. Celle-ci est définie dans le fichier `linux/net/core/dev.c` :

Code Linux 2.6.10

```
1716 static int process_backlog(struct net_device *backlog_dev, int *budget)
1717 {
1718         int work = 0;
1719         int quota = min(backlog_dev->quota, *budget);
1720         struct softnet_data *queue = &__get_cpu_var(softnet_data);
1721         unsigned long start_time = jiffies;
1722
1723         for (;;) {
1724                 struct sk_buff *skb;
1725                 struct net_device *dev;
1726
1727                 local_irq_disable();
1728                 skb = __skb_dequeue(&queue->input_pkt_queue);
1729                 if (!skb)
1730                         goto job_done;
1731                 local_irq_enable();
1732
1733                 dev = skb->dev;
1734
1735                 netif_receive_skb(skb);
1736
1737                 dev_put(dev);
```

```
1738
1739                work++;
1740
1741                if (work >= quota || jiffies - start_time > 1)
1742                        break;
1743
1744        }
1745
1746        backlog_dev->quota -= work;
1747        *budget -= work;
1748        return -1;
1749
1750 job_done:
1751        backlog_dev->quota -= work;
1752        *budget -= work;
1753
1754        list_del(&backlog_dev->poll_list);
1755        smp_mb__before_clear_bit();
1756        netif_poll_enable(backlog_dev);
1757
1758        if (queue->throttle)
1759                queue->throttle = 0;
1760        local_irq_enable();
1761        return 0;
1762 }
```

Autrement dit :

- On déclare un nombre de paquets traités, que l'on initialise à zéro.
- On déclare un quota, que l'on initialise au minimum entre le quota du périphérique réseau virtuel passé en argument et le budget passé en argument.
- On déclare une file d'attente en réception de microprocesseur, que l'on instantie avec celle du microprocesseur en cours.
- On déclare une heure de début, que l'on initialise avec l'heure en cours.
- On entre dans une boucle (pseudo-)infinie :
  - On déclare un descripteur de tampon de socket et un descripteur de périphérique réseau.
  - On inhibe les interruptions matérielles du microprocesseur en cours.
  - On essaie d'instantier le descripteur de tampon avec le premier élément de la file d'attente en réception du microprocesseur.
  - Si on n'y parvient pas, c'est que cette file d'attente est vide. Dans ce cas :
    - On diminue le quota du périphérique réseau virtuel et le budget passé en argument du nombre de paquets traités.
    - On retire le premier élément de la liste d'élection du périphérique réseau virtuel.
    - On effectue un traitement dans le cas d'un système à plusieurs microprocesseurs.
    - On met en service l'élection pour le périphérique réseau virtuel.

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

      Code Linux 2.6.10

```
879 static inline void netif_poll_enable(struct net_device *dev)
880 {
881         clear_bit(__LINK_STATE_RX_SCHED, &dev->state);
882 }
```

  - Si la file d'attente en réception du microprocesseur est indiquée comme pleine, on rectifie cette indication.
  - On permet à nouveau les interruptions matérielles sur le microprocesseur en cours et on renvoie 0.

- On permet à nouveau les interruptions matérielles sur le microprocesseur en cours.
- On instantie le descripteur de périphérique avec celui associé au descripteur de tampon de socket.
- On traite le paquet, grâce à la fonction `netif_receive_skb()` étudiée dans la sous-section ci-dessous.
- On libère le descripteur de tampon de socket.
- On incrémente le nombre de paquets traités.
- Si le nombre de paquets traités est inférieur au quota et si cela fait moins d'un tick que la fonction a débuté, on revient au début de la boucle pour traiter le paquet suivant.

- On diminue le quota du périphérique réseau virtuel et le budget passé en argument du nombre de paquets traités.
- On renvoie -1 pour indiquer qu'il reste des paquets à traiter.

19.6.2 Traitement d'un paquet

Code Linux 2.6.10

On vient de voir que la fonction précédente fait appel à la fonction `netif_receive_skb()` pour le traitement d'un paquet. Celle-ci est définie dans le fichier `linux/net/core/dev.c`:

```
1625 int netif_receive_skb(struct sk_buff *skb)
1626 {
1627         struct packet_type *ptype, *pt_prev;
1628         int ret = NET_RX_DROP;
1629         unsigned short type;
1630
1631 #ifdef CONFIG_NETPOLL
1632         if (skb->dev->netpoll_rx && skb->dev->poll && netpoll_rx(skb)) {
1633                 kfree_skb(skb);
1634                 return NET_RX_DROP;
1635         }
1636 #endif
1637
1638         if (!skb->stamp.tv_sec)
1639                 net_timestamp(&skb->stamp);
1640
1641         skb_bond(skb);
1642
1643         __get_cpu_var(netdev_rx_stat).total++;
1644
1645         skb->h.raw = skb->nh.raw = skb->data;
1646         skb->mac_len = skb->nh.raw - skb->mac.raw;
1647
1648         pt_prev = NULL;
1649
1650         rcu_read_lock();
1651
1652 #ifdef CONFIG_NET_CLS_ACT
1653         if (skb->tc_verd & TC_NCLS) {
1654                 skb->tc_verd = CLR_TC_NCLS(skb->tc_verd);
1655                 goto ncls;
1656         }
1657 #endif
```

```
1658
1659         list_for_each_entry_rcu(ptype, &ptype_all, list) {
1660                 if (!ptype->dev || ptype->dev == skb->dev) {
1661                         if (pt_prev)
1662                                 ret = deliver_skb(skb, pt_prev);
1663                         pt_prev = ptype;
1664                 }
1665         }
1666
1667 #ifdef CONFIG_NET_CLS_ACT
1668         if (pt_prev) {
1669                 ret = deliver_skb(skb, pt_prev);
1670                 pt_prev = NULL; /* noone else should process this after*/
1671         } else {
1672                 skb->tc_verd = SET_TC_OK2MUNGE(skb->tc_verd);
1673         }
1674
1675         ret = ing_filter(skb);
1676
1677         if (ret == TC_ACT_SHOT || (ret == TC_ACT_STOLEN)) {
1678                 kfree_skb(skb);
1679                 goto out;
1680         }
1681
1682         skb->tc_verd = 0;
1683 ncls:
1684 #endif
1685
1686         handle_diverter(skb);
1687
1688         if (handle_bridge(&skb, &pt_prev, &ret))
1689                 goto out;
1690
1691         type = skb->protocol;
1692         list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
1693                 if (ptype->type == type &&
1694                     (!ptype->dev || ptype->dev == skb->dev)) {
1695                         if (pt_prev)
1696                                 ret = deliver_skb(skb, pt_prev);
1697                         pt_prev = ptype;
1698                 }
1699         }
1700
1701         if (pt_prev) {
1702                 ret = pt_prev->func(skb, skb->dev, pt_prev);
1703         } else {
1704                 kfree_skb(skb);
1705                 /* Jamal, maintenant vous n'echapperez pas a une explication sur
1706                  * la facon dont vous utilisez ceci. :-)
1707                  */
1708                 ret = NET_RX_DROP;
1709         }
1710
1711 out:
1712         rcu_read_unlock();
1713         return ret;
1714 }
```

Autrement dit :

- On déclare un type de paquet et un type de paquet précédent.
- On déclare une valeur de retour, que l'on initialise avec `NET_RX_DROP` pour le cas d'un paquet

écarté.

- On déclare un type de protocole.
- Si l'estampille temporelle n'est pas spécifiée pour le descripteur de tampon de socket passé en argument, on la renseigne avec l'heure en cours.
- On renseigne le champ périphérique réel du descripteur de tampon passé en argument.

Code Linux 2.6.10

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

```
1498 static __inline__ void skb_bond(struct sk_buff *skb)
1499 {
1500         struct net_device *dev = skb->dev;
1501
1502         if (dev->master) {
1503                 skb->real_dev = skb->dev;
1504                 skb->dev = dev->master;
1505         }
1506 }
```

- On met à jour l'information statistique sur le nombre total de paquets reçus et traités par le microprocesseur.
- On renseigne les champs en-tête matériel et en-tête réseau du descripteur de périphérique pour le cas d'un paquet brut.
- On renseigne le champ longueur d'en-tête matériel du descripteur de périphérique.
- On verrouille le processus de lecture et mise à jour.

Code Linux 2.6.10

Les macros générales `rcu_read_lock()` et `rcu_read_unlock()` sont définies dans le fichier `linux/include/linux/rcupdate.h`:

```
157 /**
158  * rcu_read_lock - marque le debut d'une section critique du cote lecture RCU.
159  *
160  * Lorsque synchronize_kernel() est invoque sur un CPU alors que d'autres CPU
161  * sont dans des sections critiques du cote lecture RCU,
162  * on est assure que synchronize_kernel() bloque jusqu'a ce que tous les autres
163  * CPU sortent de leurs sections critiques. De meme, si call_rcu() est invoque
164  * sur un CPU alors que d'autres CPU sont dans des sections critiques du cote lecture
165  * RCU, l'invocation de l'appel RCU correspondant est retarde
166  * jusqu'a ce que tous les autres CPU sortent de leurs sections critiques.
167  *
168  * Noter, cependant, qu'on permet que les appels RCU tournent de facon concurrente
169  * avec des sections critiques du cote lecture RCU. Une facon dont ceci peut arriver
170  * est via la sequence suivante d'evenements : (1) CPU 0 entre dans une section
171  * critique du cote lecture RCU, (2) CPU 1 invoque call_rcu() pour enregistrer
172  * un appel RCU, (3) CPU 0 sort de la section critique du cote lecture RCU,
173  * (4) CPU 2 entre dans une section lecture du cote lecture RCU, (5) l'appel RCU
174  * est invoque. Ceci est legal, puisque la section critique du cote lecture RCU
175  * qui est en train de tourner en meme temps que call_rcu() (et que ainsi
176  * pourrait etre en train de referencer quelque chose que l'appel RCU correspondant
177  * pourrait liberer) a termine avant que l'appel RCU correspondant
178  * soit invoque.
179  *
180  * Les sections critiques du cote lecture RCU peuvent etre emboitees. Toutes les actions
181  * retardees le seront jusqu'a ce que la section critique du cote lecture RCU la plus
182  * exterieure ait terminee.
183  *
184  * Il est illegal de bloquer alors qu'on est dans une section critique du cote
       lecture RCU.
185  */
186 #define rcu_read_lock()         preempt_disable()
187
188 /**
```

```
189  * rcu_read_unlock - marque la fin d'une section critique du cote lecture RCU.
190  *
191  * Voir rcu_read_lock() pour plus d'informations.
192  */
193 #define rcu_read_unlock()       preempt_enable()
```

- On cherche si le type du tampon apparaît dans la liste des types de paquet de traitement de tous les paquets. Si c'est le cas on dépose le tampon à l'intention de la couche supérieure, grâce à la fonction `deliver_skb()` étudiée dans la sous-section suivante.
- Si le déroutement est supporté par le périphérique, on traite celui-ci.

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

Code Linux 2.6.10

```
111 #ifdef CONFIG_NET_DIVERT
112 #include <linux/netdevice.h>
113
114 int alloc_divert_blk(struct net_device *);
115 void free_divert_blk(struct net_device *);
116 int divert_ioctl(unsigned int cmd, struct divert_cf __user *arg);
117 void divert_frame(struct sk_buff *skb);
118 static inline void handle_diverter(struct sk_buff *skb)
119 {
120         /* si le deroutement est supporte sur le peripherique, alors allons-y */
121         if (skb->dev->divert && skb->dev->divert->divert)
122                 divert_frame(skb);
123 }
124
125 #else
126 # define alloc_divert_blk(dev)          (0)
127 # define free_divert_blk(dev)           do {} while (0)
128 # define divert_ioctl(cmd, arg)         (-ENOPKG)
129 # define handle_diverter(skb)           do {} while (0)
130 #endif
```

  La fonction `divert_frame()` de déroutement proprement dit ne nous intéressera pas ici.
- Lorsque l'ordinateur a été configuré en tant que passerelle (`CONFIG_BRIDGE`), le paquet est transmis à la méthode `handle_bridge()`, ce qui ne nous intéressera pas ici.
- On cherche si le type du tampon apparaît dans la liste des types de paquet de base. Si c'est le cas on dépose le tampon à l'intention de la couche supérieure adéquate.

  Dans le cas qui nous intéresse, on trouvera le type de paquet IP.
- Si on ne trouve aucun type adéquat, on libère le descripteur de tampon.
- On déverrouille le dispositif d'écriture et mise à jour et on renvoie `NET_RX_DROP` ou la valeur renvoyée par la fonction `deliver_skb()`.

19.6.3 Livraison à la couche supérieure

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

Code Linux 2.6.10

```
1554 static __inline__ int deliver_skb(struct sk_buff *skb,
1555                                     struct packet_type *pt_prev)
1556 {
1557         atomic_inc(&skb->users);
1558         return pt_prev->func(skb, skb->dev, pt_prev);
1559 }
```

Elle se contente d'incrémenter le nombre d'utilisateurs du descripteur de tampon et de faire appel à la routine de traitement du protocole en question, `ip_rcv()` rappelons-le dans le cas de IPv4.

19.7 Récupération des informations statistiques

19.7.1 Cas général

Nous avons vu lors de l'étude des fonctions précédentes qu'un certain nombre d'informations statistiques étaient placées soit dans une variable dépendant du microprocesseur actif, soit dans la partie privée du decripteur de périphérique dans le cas de la carte 3Com 501. La fonction `get_stats()` spécifique à la carte permet de récupérer ces dernières.

19.7.2 Cas de la carte 3Com 501

Comme nous l'avons vu au chapitre 17, la fonction `get_stats()` spécifique à la carte 3Com 501 s'appelle `el1_get_stats()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
818 /**
819  * el1_get_stats :
820  * @dev : La carte a partir de laquelle obtenir les statistiques
821  *
822  * Pour des peripheriques plus elegants, cette fonction est necessaire pour rapatrier les
823  * statistiques de la carte elle-meme. La 3c501 n'a pas de statistiques dans le materiel.
824  * Nous les maintenons tous, aussi sont-elles toujours par definition a jour.
825  *
826  * Renvoie les statistiques pour la carte a partir des donnes privees de la carte
827  */
828
829 static struct net_device_stats *el1_get_stats(struct net_device *dev)
830 {
831         struct net_local *lp = netdev_priv(dev);
832         return &lp->stats;
833 }
```

Comme la carte 3Com 501 ne conserve pas de données statistiques, cette fonction renvoie la partie privée du descripteur de périphérique passé en argument.