Chapitre 13

Gestion des tampons de socket

Nous avons vu au chapitre 12 la structure d'un descripteur de tampon de socket. Nous ne nous nous sommes pas occupés de l'aspect dynamique jusqu'à maintenant. C'est ce que nous allons faire maintenant.

Des fonctions de gestion des descripteurs de tampon de socket sont à disposition du noyau. Elles se classent en quatre catégories :

- *Génération, libération et duplication des descripteurs* : ces fonctions se chargent de l'ensemble de la gestion mémoire du tampon de socket et de son optimisation.
- *Manipulation des paramètres* au sein de la structure `sk_buff`, en particulier les opérations visant la modification de la zone de données des paquets.
- *Gestion des files d'attente* de descripteurs de tampon de socket.
- *Copie* des descripteurs de tampon et des tampons eux-mêmes.

Nous regroupons l'étude de ces fonctions dans ce chapitre car c'est leur place naturelle et que nous allons en utiliser certaines très rapidement. Mais on ne peut pas dire que leur étude soit passionnante. Cela n'a presque rien à voir avec les réseaux, si ce n'est leur utilisation, mais beaucoup plus avec la gestion de la mémoire. On pourra se contenter de parcourir rapidement ce chapitre et y revenir au fur et à mesure des besoins.

13.1 Génération et libération des descripteurs de tampon

Les opérations sur un descripteur de tampon de socket sont celles à lesquelles on peut s'attendre: allocation et libération.

13.1.1 Sources

Code Linux 2.6.10

L'aspect dynamique de l'antémémoire des sockets se trouve dans les fichiers `linux/include/linux/skbuff.h` (que nous avons déjà rencontré) et `linux/net/core/skbuff.c`:

```
1  /*
2   *      Routines ayant a voir avec les manipulations memoire de 'struct sk_buff'.
3   *
4   *      Auteurs :       Alan Cox <iiitac@pyr.swan.ac.uk>
5   *                      Florian La Roche <rzsfl@rz.uni-sb.de>
6   *
7   *      Version :       $Id: skbuff.c,v 1.90 2001/11/07 05:56:19 davem Exp $
8   *
9   *      Rectifications :
10  *              Alan Cox        :       A rectifie le pire des bogues de mise
11  *                                      d'aplomb.
12  *              Dave Platt      :       A rectifie la pile d'interruption.
13  *      Richard Kooijman        :       A rectifie les estampilles temporelles.
14  *              Alan Cox        :       A change le format des tampons.
15  *              Alan Cox        :       ancre de destruction pour AF_UNIX etc.
16  *              Linus Torvalds  :       Un meilleur skb_clone.
17  *              Alan Cox        :       A ajoute skb_copy.
18  *              Alan Cox        :       A ajoute toutes les routines changees que Linus
19  *                                      avait seulement placees dans les en-tetes
20  *              Ray VanTassle   :       A rectifie --skb->lock en free
21  *              Alan Cox        :       champ arp de copie skb_copy
22  *              Andi Kleen      :       A slabifie.
23  *              Robert Olsson   :       A retire skb_head_pool
24  *
25  *      NOTE :
26  *              Les routines __skb_ doivent etre appelees avec les interruptions
27  *      inhibees, sinon vous devez *reellement* vous assurer que l'operation est atomique
28  *      a l'egard de la liste en train d'etre traitee (par exemple via lock_sock()
29  *      ou en inhibant les routines des moities basses, etc).
[...]
37 /*
38  *      Les fonctions de ce fichier ne compileront pas correctement avec gcc 2.4.x
39  */
```

13.1.2 Allocation d'un tampon de socket

13.1.2.1 Description

La fonction:

```
struct sk_buff * alloc_skb(unsigned int size, int gfp_mask);
```

alloue un nouveau tampon de socket sans espace de tête, sans données et dont l'espace de queue a une taille de `size` octets (plus alignement jusqu'à la prochaine adresse 16 bits). Elle renvoie l'adresse du descripteur associé (ou la valeur `NULL` s'il y a un problème).

Les valeurs des drapeaux de `gfp_mask` sont `GFP_USER`, `GFP_KERNEL`, `GFP_ATOMIC` ou `GFP_DMA`.

13.1.2.2 Emplacement des descripteurs de tampon

Les descripteurs de tampon de socket forment une liste chaînée, comme nous l'avons vu au chapitre 12. Le début de cette liste est située à un emplacement de la mémoire vive réservée au noyau, repéré par la variable `skbuff_head_cache`. Celle-ci est définie au début du fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
71 static kmem_cache_t *skbuff_head_cache;
```

13.1.2.3 Définition de la fonction

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

Code Linux 2.6.10

```
110 /*      Alloue un nouveau skbuff. Nous faisons ceci nous-memes pour que nous puissions
111  *      renseigner quelques champs 'prives' et aussi faire des statistiques memoire pour
112  *      trouver toutes les fuites [BEEP].
113  *
114  */
115
116 /**
117  *      alloc_skb       -       alloue un tampon de reseau
118  *      @size : taille a allouer
119  *      @gfp_mask : masque d'allocation
120  *
121  *      Alloue un nouveau &sk_buff. Le tampon renvoye n'a pas d'espace de tete et un
122  *      espace de queue de size octets. L'objet a un compteur de reference de un.
123  *      La valeur renvoyee est le tampon. En cas d'echec, la valeur renvoyee est %NULL.
124  *
125  *      Les tampons peuvent etre alloues dans des interruptions seulement en utilisant
126  *      un @gfp_mask de %GFP_ATOMIC.
127  */
128 struct sk_buff *alloc_skb(unsigned int size, int gfp_mask)
129 {
130         struct sk_buff *skb;
131         u8 *data;
132
133         /* Obtenir la HEAD */
134         skb = kmem_cache_alloc(skbuff_head_cache,
135                                gfp_mask & ~__GFP_DMA);
136         if (!skb)
137                 goto out;
138
139         /* Obtenir les DATA. La taille doit concorder avec skb_add_mtu(). */
140         size = SKB_DATA_ALIGN(size);
141         data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
142         if (!data)
143                 goto nodata;
144
145         memset(skb, 0, offsetof(struct sk_buff, truesize));
146         skb->truesize = size + sizeof(struct sk_buff);
147         atomic_set(&skb->users, 1);
148         skb->head = data;
149         skb->data = data;
150         skb->tail = data;
151         skb->end  = data + size;
152
153         atomic_set(&(skb_shinfo(skb)->dataref), 1);
154         skb_shinfo(skb)->nr_frags  = 0;
155         skb_shinfo(skb)->tso_size = 0;
156         skb_shinfo(skb)->tso_segs = 0;
157         skb_shinfo(skb)->frag_list = NULL;
158 out:
```

```
159         return skb;
160 nodata:
161         kmem_cache_free(skbuff_head_cache, skb);
162         skb = NULL;
163         goto out;
164 }
```

Autrement dit :

- L'adresse `skb` à renvoyer est déclarée ainsi que celle `data` du tampon proprement dit.
- On essaie d'obtenir un emplacement en mémoire vive pouvant contenir le descripteur de tampon, au-delà du début indiqué. Si on n'y parvient pas, on renvoie `NULL`.
- On essaie d'obtenir de la place en mémoire vive pour le tampon proprement dit.
  - La taille devant être un multiple de 16, on ajoute un complément à `size` si nécessaire, en utilisant la macro auxiliaire `SKB_DATA_ALIGN()`.

    Code Linux 2.6.10

    Celle-ci est définie dans le fichier `linux/include/linux/skbuff.h` :

    ```
    40 #define SKB_DATA_ALIGN(X)       (((X) + (SMP_CACHE_BYTES - 1)) & \
    41                                   ~(SMP_CACHE_BYTES - 1))
    ```

  - On essaie d'obtenir de l'emplacement mémoire pour le tampon. Si on n'y parvient pas, on libère le descripteur de tampon obtenu ci-dessus et on renvoie `NULL`.
- On initialise l'emplacement mémoire du descripteur de tampon avec des zéros.
- On renseigne les champs indispensables du descripteur de tampon de socket à ce niveau, c'est-à-dire la taille effective, le nombre d'utilisateurs (égal à 1) et les quatre adresses du tampon.
- On renseigne les champs concernant la fragmentation : référencé une fois, taille effective du tableau des fragments nulle, de taille zéro, liste de fragmentation vide.

  La macro `skb_shinfo()` permet de se référer au champ `end` du descripteur (qui détient les informations sur la fragmentation) :

  Code Linux 2.6.10

  ```
  313 /* Interne */
  314 #define skb_shinfo(SKB)         ((struct skb_shared_info *)((SKB)->end))
  ```

- On renvoie l'adresse du descripteur de tampon.

13.1.3 Libération rapide d'un tampon de socket

13.1.3.1 Description

La fonction :

```
void kfree_skbmem(struct sk_buff *skb);
```

libère le tampon spécifié par la descripteur `skb` ainsi que le descripteur lui-même.

13.1.3.2 Définition

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

Code Linux 2.6.10

```
205 /*
206  *      Libere un skbuff de la memoire sans nettoyer l'etat.
207  */
208 void kfree_skbmem(struct sk_buff *skb)
209 {
210         skb_release_data(skb);
211         kmem_cache_free(skbuff_head_cache, skb);
212 }
```

Autrement dit :

- le tampon est libéré, en utilisant la fonction auxiliaire `skb_release_data()` étudiée ci-après ;
- la partie de mémoire vive allouée au descripteur de tampon est également libérée.

13.1.3.3 Libération du tampon proprement dit

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

```
188 void skb_release_data(struct sk_buff *skb)
189 {
190         if (!skb->cloned ||
191             atomic_dec_and_test(&(skb_shinfo(skb)->dataref))) {
192                 if (skb_shinfo(skb)->nr_frags) {
193                         int i;
194                         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
195                                 put_page(skb_shinfo(skb)->frags[i].page);
196                 }
197
198                 if (skb_shinfo(skb)->frag_list)
199                         skb_drop_fraglist(skb);
200
201                 kfree(skb->head);
202         }
203 }
```

Autrement dit si le descripteur n'est pas cloné ou si le nombre de références est nul :

- si le nombre de fragments n'est pas nul, on libère chacune des pages du tableau de fragments ;
- si la liste de fragmentation est non nulle, on la libère, en utilisant la fonction auxiliaire `skb_drop_fraglist()` étudiée ci-après ;
- on libère la tête de liste, en utilisant la fonction générale `kfree()`.

13.1.3.4 Libération de la liste de fragmentation

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

```
167 static void skb_drop_fraglist(struct sk_buff *skb)
168 {
169         struct sk_buff *list = skb_shinfo(skb)->frag_list;
170
171         skb_shinfo(skb)->frag_list = NULL;
172
173         do {
174                 struct sk_buff *this = list;
175                 list = list->next;
176                 kfree_skb(this);
177         } while (list);
178 }
```

autrement dit on libère chaque descripteur de la liste grâce à la fonction `kfree_skb()` décrite ci-après et on indique que la liste est vide.

13.1.4 Libération propre d'un tampon de socket

13.1.4.1 Description

La fonction :

```
void kfree_skb(struct sk_buff *skb);
```

libère le tampon et le descripteur de tampon, spécifiés par l'adresse `skb` du descripteur, proprement, c'est-à-dire en mettant à zéro les champs du descripteur de tampon. Elle vérifie en outre que le descripteur ne se trouve pas dans une file d'attente, sinon un message d'erreur est affiché.

13.1.4.2 Définition

Code Linux 2.6.10

La fonction `kfree_skb()` est définie dans le fichier en-tête `linux/include/linux/skbuff.h`:

```
340 /*
341  * Si users == 1, nous sommes le seul proprietaire et nous pouvons eviter les
342  * changements atomiques redondants.
343  */
344
345 /**
346  *      kfree_skb - libere un sk_buff
347  *      @skb : tampon a liberer
348  *
349  *      Ecarte une reference au tampon et le libere si le compteur d'utilisation a
350  *      atteint zero.
351  */
352 static inline void kfree_skb(struct sk_buff *skb)
353 {
354         if (atomic_read(&skb->users) == 1 || atomic_dec_and_test(&skb->users))
355                 __kfree_skb(skb);
356 }
```

Code Linux 2.6.10

qui renvoie essentiellement à la fonction `__kfree_skb()`, définie dans le fichier `linux/net/core/skbuff.c`:

```
214 /**
215  *      __kfree_skb - fonction privee
216  *      @skb : tampon
217  *
218  *      Libere un sk_buff. Relache toute chose attachee au tampon.
219  *      Nettoie l'etat. Ceci est une fonction d'aide interne. Les utilisateurs
220  *      doivent toujours appeler kfree_skb
221  */
222
223 void __kfree_skb(struct sk_buff *skb)
224 {
225         if (skb->list) {
226                 printk(KERN_WARNING "Warning: kfree_skb passed an skb still "
227                        "on a list (from %p).\n", NET_CALLER(skb));
228                 BUG();
229         }
230
231         dst_release(skb->dst);
232 #ifdef CONFIG_XFRM
233         secpath_put(skb->sp);
234 #endif
235         if(skb->destructor) {
236                 if (in_irq())
237                         printk(KERN_WARNING "Warning: kfree_skb on "
238                                             "hard IRQ %p\n", NET_CALLER(skb));
```

```
239                 skb->destructor(skb);
240         }
241 #ifdef CONFIG_NETFILTER
242         nf_conntrack_put(skb->nfct);
243 #ifdef CONFIG_BRIDGE_NETFILTER
244         nf_bridge_put(skb->nf_bridge);
245 #endif
246 #endif
247 /* XXX : Ceci EST-IL encore necessaire ? - JHS */
248 #ifdef CONFIG_NET_SCHED
249         skb->tc_index = 0;
250 #ifdef CONFIG_NET_CLS_ACT
251         skb->tc_verd = 0;
252         skb->tc_classid = 0;
253 #endif
254 #endif
255
256         kfree_skbmem(skb);
257 }
```

Autrement dit :

- Si le descripteur se trouve dans une file d'attente, on ne devrait pas libérer le descripteur de tampon. Le noyau affiche donc un message d'avertissement.

  La macro générale `BUG()` dépend de l'architecture. Pour les microprocesseurs Intel, elle est définie dans le fichier `linux/include/asm-i386/bug.h`: Code Linux 2.6.10

```
 6 /*
 7  * Dit a l'utilisateur qu'il y a un probleme.
 8  * Le fichier responsable ainsi que la ligne sont codes d'apres l'opcode "officiallement
 9  * indefini" pour l'analyse dans la routine.
10  */
11
12 #if 1   /* Positionne a zero pour un noyau legerement plus petit */
13 #define BUG()                           \
14  __asm__ __volatile__(  "ud2\n"         \
15                         "\t.word %c0\n" \
16                         "\t.long %c1\n" \
17                          : : "i" (__LINE__), "i" (__FILE__))
18 #else
19 #define BUG() __asm__ __volatile__("ud2\n")
20 #endif
```

- On libère l'entrée du cache de destination, grâce à la fonction `dst_release()` étudiée ci-après.
- Si une fonction de destruction est associée au descripteur de tampon, on y fait appel.
- On libère enfin le tampon et le descripteur de tampon grâce à la fonction `kfree_skbmem()` étudiée ci-dessus.

13.1.4.3 Libération d'une entrée de cache de destination

La fonction `dst_release()` est définie dans le fichier en-tête `linux/include/net/dst.h`: Code Linux 2.6.10

```
145 static inline
146 void dst_release(struct dst_entry * dst)
147 {
148         if (dst) {
149                 WARN_ON(atomic_read(&dst->__refcnt) < 1);
150                 atomic_dec(&dst->__refcnt);
151         }
152 }
```

Elle se contente de décrémenter le nombre d'utilisateurs de cette entrée.

La macro générale `WARN_ON()` dépend de l'architecture. Pour les microprocesseurs Intel, elle est définie dans le fichier `linux/include/asm-i386/bug.h`:

Code Linux 2.6.10

```
25 #ifndef HAVE_ARCH_WARN_ON
26 #define WARN_ON(condition) do { \
27         if (unlikely((condition)!=0)) { \
28                 printk("Badness in %s at %s:%d\n", __FUNCTION__, __FILE__, __LINE__); \
29                 dump_stack(); \
30         } \
31 } while (0)
32 #endif
```

Les macros générales `likely()` et `unlikely()` sont définies dans le fichier `linux/include/linux/compiler.h`:

Code Linux 2.6.10

```
54 /*
55  * Des macros generiques dependant du compilateur requises pour la construction du
56  * noyau en-dessous de ce commentaire. Les implementations specifiques a la version du
57  * compilateur/compilateur reel specifique proviennent des fichiers en-tete ci-dessus
58  */
59
60 #define likely(x)       __builtin_expect(!!(x), 1)
61 #define unlikely(x)     __builtin_expect(!!(x), 0)
```

13.1.4.4 Macro de libération

La macro `dev_kfree_skb()` se contente de renvoyer à la fonction étudiée ci-dessus.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
307 #define dev_kfree_skb(a)        kfree_skb(a)
```

13.2 Manipulation du tampon

13.2.1 Espaces de tête et de queue

13.2.1.1 Taille de l'espace de queue

La fonction `skb_tailroom()` indique la taille de l'espace de queue. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
804 /**
805  *      skb_tailroom - octets a la fin du tampon
806  *      @skb : tampon a verifier
807  *
808  *      Renvoie le nombre d'octets de l'espace libre a la fin d'un sk_buff
809  */
810 static inline int skb_tailroom(const struct sk_buff *skb)
811 {
812         return skb_is_nonlinear(skb) ? 0 : skb->end - skb->tail;
813 }
```

La fonction en ligne `skb_is_nonlinear()` est définie un peu plus haut dans le même fichier :

Code Linux 2.6.10

```
655 static inline int skb_is_nonlinear(const struct sk_buff *skb)
656 {
657         return skb->data_len;
658 }
```

13.2.1.2 Taille de l'espace de tête

La fonction `skb_headroom()` indique la taille de l'espace de tête. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
793 /**
794  *      skb_headroom - octets en tete du tampon
795  *      @skb : tampon a verifier
796  *
797  *      Renvoie le nombre d'octets de l'espace libre en tete d'un &sk_buff.
798  */
799 static inline int skb_headroom(const struct sk_buff *skb)
800 {
801         return skb->data - skb->head;
802 }
```

13.2.2 Ajustement de la taille d'un tampon vide

13.2.2.1 Ajustement de la taille de l'espace de tête

La fonction `skb_reserve(len)` ajoute `len` octets à l'espace de tête, pris sur l'espace de queue. Cette fonction n'est bien sûr appropriée que s'il n'existe pas de données dans la zone actuelle, mais seulement si sa taille initiale doit être corrigée.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
815 /**
816  *      skb_reserve - ajuste l'espace de tete
817  *      @skb : tampon a alterer
818  *      @len : octets a deplacer
819  *
820  *      Accroit l'espace de tete d'un &sk_buff vide en reduisant l'espace
821  *      de queue. Ceci est seulement permis pour un tampon vide.
822  */
823 static inline void skb_reserve(struct sk_buff *skb, unsigned int len)
824 {
825         skb->data += len;
826         skb->tail += len;
827 }
```

13.2.2.2 Ajout à la fin de la zone des données

La fonction `skb_put(skb, len)` permet d'augmenter la zone des données à partir de la fin de celle-ci au dépend de l'espace de queue. Ceci se produit rarement car la majorité des protocoles utilisent un en-tête mais rarement un en-queue. Seuls les pointeurs sont positionnés en conséquence; l'utilisateur est responsable de la copie des données elles-mêmes. Avant l'appel de `skb_put()`, il faut vérifier que la taille de l'espace de queue est encore suffisant, sinon le noyau affiche un message d'erreur.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
701 /**
702  *      skb_put - ajoute des donnees a un tampon
703  *      @skb : tampon a utiliser
704  *      @len : quantite de donnees a ajouter
705  *
706  *      Cette fonction etend la zone des donnees du tampon. Si ceci conduit a
707  *      depasser la taille totale du tampon, le noyau panique. Un pointeur sur le
708  *      premier octet de la zone supplementaire est renvoye.
709  */
710 static inline unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
```

```
711 {
712         unsigned char *tmp = skb->tail;
713         SKB_LINEAR_ASSERT(skb);
714         skb->tail += len;
715         skb->len  += len;
716         if (unlikely(skb->tail>skb->end))
717                 skb_over_panic(skb, len, current_text_addr());
718         return tmp;
719 }
```

Code Linux 2.6.10

La macro `SKB_LINEAR_ASSERT()` est définie un peu plus haut dans le même fichier :

```
687 #define SKB_LINEAR_ASSERT(skb)  BUG_ON(skb_is_nonlinear(skb))
```

Code Linux 2.6.10

La définition de la macro générale `BUG_ON()` est définie dans le fichier `linux/include/asm-generic/bug.h`:

```
21 #ifndef HAVE_ARCH_BUG_ON
22 #define BUG_ON(condition) do { if (unlikely((condition)!=0)) BUG(); } while(0)
23 #endif
```

Code Linux 2.6.10

La fonction `skb_over_panic()`, qui gèle le système et affiche un message d'erreur à l'écran, est définie dans le fichier `linux/net/core/skbuff.c`:

```
79 /**
80  *      skb_over_panic  -       fonction privee
81  *      @skb : tampon
82  *      @sz : taille
83  *      @here : adresse
84  *
85  *      Code de support non en ligne pour skb_put(). Non appelable par un utilisateur.
86  */
87 void skb_over_panic(struct sk_buff *skb, int sz, void *here)
88 {
89         printk(KERN_INFO "skput:over: %p:%d put:%d dev:%s",
90                here, skb->len, sz, skb->dev ? skb->dev->name : "<NULL>");
91         BUG();
92 }
```

Code Linux 2.6.10

La fonction générale `current_text_addr()` dépend de l'architecture. Elle est définie dans le fichier `linux/include/asm-i386/processor.h` pour les microprocesseurs Intel :

```
36 /*
37  * Implementation par defaut de la macro qui renvoie le pointeur d'instruction
38  * en cours ("compteur de programmer").
39  */
40 #define current_text_addr() ({ void *pc; __asm__("movl $1f,%0\n1:":"=g" (pc)); pc; })
```

13.2.2.3 Ajout en début de la zone des données

La fonction `skb_push()` opère comme la fonction précédente mais en déplaçant le début de la zone des données, au dépend de l'espace de tête. La taille de l'espace de tête est vérifiée au préalable.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
728 /**
729  *      skb_push - ajoute des donnees au debut d'un tampon
730  *      @skb : tampon a utiliser
731  *      @len : quantite de donnees a ajouter
732  *
733  *      Cette fonction etend la zone des donnees du tampon au debut du
734  *      tampon. Si ceci conduit a depasser la taille totale de l'espace de tete du tampon,
```

```
735  *      le noyau panique. Un pointeur sur le premier octet des donnees supplementaires
             est renvoye.
736  */
737 static inline unsigned char *skb_push(struct sk_buff *skb, unsigned int len)
738 {
739         skb->data -= len;
740         skb->len  += len;
741         if (unlikely(skb->data<skb->head))
742                 skb_under_panic(skb, len, current_text_addr());
743         return skb->data;
744 }
```

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

```
94  /**
95   *      skb_under_panic -       fonction privee
96   *      @skb : tampon
97   *      @sz : taille
98   *      @here : adresse
99   *
100  *      Code de support hors ligne pour skb_push(). Non appelable par un utilisateur.
101  */
102
103 void skb_under_panic(struct sk_buff *skb, int sz, void *here)
104 {
105         printk(KERN_INFO "skput:under: %p:%d put:%d dev:%s",
106                here, skb->len, sz, skb->dev ? skb->dev->name : "<NULL>");
107         BUG();
108 }
```

13.2.2.4 Retrait de données au début

La fonction `skb_pull(skb, len)` tronque de `len` octets le début de la zone des données d'un tampon au profit de l'espace de tête, après avoir vérifié qu'on demeurera après ceci dans la zone de mémoire affectée au tampon. Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
746 static inline unsigned char *__skb_pull(struct sk_buff *skb, unsigned int len)
747 {
748         skb->len -= len;
749         BUG_ON(skb->len < skb->data_len);
750         return skb->data += len;
751 }
752
753 /**
754  *      skb_pull - retire des donnees au debut d'un tampon
883  *      @skb : tampon a utiliser
884  *      @len : quantite de donnees a retirer
757  *
758  *      Cette fonction efface le debut des donnees d'un tampon, les renvoyant a
759  *      l'espace de tete. Un pointeur a la donnee suivante dans le tampon
760  *      est renvoye. Une fois que les donnees ont ete retirees, les placements suivants
761  *      surchargeront les anciennes donnees.
762  */
763 static inline unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)
764 {
765         return unlikely(len > skb->len) ? NULL : __skb_pull(skb, len);
766 }
```

13.2.3 Manipulation du compteur de références

13.2.3.1 Incrémentation du nombre de références

Code Linux 2.6.10

La fonction `skb_get()` augmente de un le nombre de références au descripteur de tampon. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
327 /**
328  *      skb_get - reference un tampon
329  *      @skb : tampon a referencer
330  *
331  *      Ajoute une autre reference a un tampon de socket et renvoie un pointeur
332  *      au tampon.
333  */
334 static inline struct sk_buff *skb_get(struct sk_buff *skb)
335 {
336         atomic_inc(&skb->users);
337         return skb;
338 }
```

13.2.3.2 Détection de la multiplicité des références

Code Linux 2.6.10

La fonction `skb_shared()` vérifie si le descripteur de tampon possède plusieurs références ou une seule. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
378 /**
379  *      skb_shared - le tampon est-il partage ?
380  *      @skb : tampon a verifier
381  *
382  *      Renvoie vrai si plus d'une personne fait reference a ce
383  *      tampon.
384  */
385 static inline int skb_shared(const struct sk_buff *skb)
386 {
387         return atomic_read(&skb->users) != 1;
388 }
```

Remarquons que le code n'est correct que si le compteur est non nul, mais ceci est le cas puisque celui qui pose la question fait référence au descripteur de tampon.

13.2.4 Détection de clonage

13.2.4.1 Disponibilité du clone

Code Linux 2.6.10

La fonction `skb_cloned()` indique si le descripteur de tampon est un clone et si le tampon associé est disponible. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
365 /**
366  *      skb_cloned - le tampon est-il un clone ?
367  *      @skb : tampon a verifier
368  *
369  *      Renvoie vrai si le tampon a ete genere avec skb_clone() et est
370  *      une des copies partagees du tampon. Les tampons clones sont
371  *      des donnes partagees, aussi ne doit-on pas ecrire dessus en circonstances normales.
372  */
373 static inline int skb_cloned(const struct sk_buff *skb)
374 {
375         return skb->cloned && atomic_read(&skb_shinfo(skb)->dataref) != 1;
376 }
```

13.2.4.2 Déclonage

La fonction `skb_unshare()` vérifie si le descripteur de tampon est un clone. Si c'est le cas, une copie du tampon est effectuée que l'on associe à ce clone, dont le compteur de renvois prend la valeur un. Le compteur de renvoi de l'ancien tampon est décrémenté. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
414 /*
415  *      Copie des tampons partages dans un nouveau sk_buff. Nous faisons effectivement COW sur
416  *      les paquets pour traiter les cas ou nous avons une ecriture locale et une
417  *      retransmission et d'autres trucs du meme genre. Le cas usuel est le vidage tcp
418  *      d'un paquet en train d'etre retransmis.
419  */
420
421 /**
422  *      skb_unshare - fait une copie d'un tampon partage
423  *      @skb : tampon a verifier
424  *      @pri : priorite pour l'allocation de la memoire
425  *
426  *      Si le tampon de socket est un clone, cette fonction cree une nouvelle
427  *      copie des donnees, decremente le compteur de reference de l'ancienne copie et
428  *      renvoie la nouvelle copie avec un compteur de reference a 1. Si le tampon
429  *      n'est pas un clone, le tampon originel est renvoye. Si elle est appelee avec un verrou
430  *      tournant depuis une interruption, l'etat @pri doit etre %GFP_ATOMIC
431  *
432  *      %NULL est renvoye en cas d'echec d'allocation memoire.
433  */
434 static inline struct sk_buff *skb_unshare(struct sk_buff *skb, int pri)
435 {
436         might_sleep_if(pri & __GFP_WAIT);
437         if (skb_cloned(skb)) {
438                 struct sk_buff *nskb = skb_copy(skb, pri);
439                 kfree_skb(skb); /* Libere notre copie partagee */
440                 skb = nskb;
441         }
442         return skb;
443 }
```

La fonction `skb_copy()` sera étudiée dans une section ultérieure.

13.2.4.3 Clonage

La fonction `skb_share_check()` vérifie si le tampon est partagé. Si c'est le cas, il est cloné. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
390 /**
391  *      skb_share_check - verifie si un tampon est partage et le clone s'il l'est
392  *      @skb : tampon a verifier
393  *      @pri : priorite pour l'allocation memoire
394  *
395  *      Si le tampon est partage, le tampon est clone et la reference a l'ancienne
396  *      copie est ecartee. Un nouveau clone avec une reference de 1 est renvoyee.
397  *      Si le tampon n'est pas partage, le tampon originel est renvoye. Si elle est
398  *      appelee avec un statut d'interruption ou avec un verrou tournant detenu,
399  *      pri doit etre GFP_ATOMIC.
400  *
401  *      NULL est renvoye en cas d'echec d'allocation memoire.
402  */
403 static inline struct sk_buff *skb_share_check(struct sk_buff *skb, int pri)
404 {
405         might_sleep_if(pri & __GFP_WAIT);
406         if (skb_shared(skb)) {
```

```
407                 struct sk_buff *nskb = skb_clone(skb, pri);
408                 kfree_skb(skb);
409                 skb = nskb;
410         }
411         return skb;
412 }
```

La fonction `skb_clone()` sera également étudiée dans une section ultérieure.

13.3 Gestion des files d'attente de paquets

Nous allons décrire les méthodes destinées à la gestion des descripteurs de tampon de socket dans les files d'attente.

13.3.1 Caractères généraux

13.3.1.1 Atomicité des opérations sur les tampons de socket

La plupart des opérations sur les tampons de socket sont mises en œuvre durant les phases critiques ou peuvent être interrompues par des opérations de priorité supérieure (traitement d'interruption, interruption logicielle, etc.). Un chaînage incorrect pourrait se produire par exemple en raison d'opérations successives imbriquées les unes dans les autres de manière complexe sur une file d'attente. Il en résulterait inévitablement une erreur d'accès à la mémoire puis un blocage du système. Cela explique pourquoi le traitement des paquets doit avoir lieu atomiquement.

Il en résulte certes un coût supplémentaire parce qu'il faut implémenter des mécanismes particuliers comme les verrous rotatifs ou les sémaphores pour être en mesure d'obtenir des états mieux sécurisés. C'est toutefois la seule façon de garantir qu'il n'existe aucun état incohérent susceptible de mettre en péril la stabilité du sous-système réseau et, par conséquent, celle du système d'exploitation. Sous Linux, la sécurité et la stabilité de l'ensemble du système d'exploitation prennent toujours le pas sur la performance et les bancs de test.

13.3.1.2 Opérations *inline*

La plupart des opérations sur les files d'attente sont définies en tant que procédure *inline*. La définition *inline* signifie qu'elles ne constituent pas des procédures à proprement parler, mais qu'elles sont insérées comme une macro dans le tronc de la fonction émettrice. On évite ainsi le temps système d'un appel de fonction, moyennant la prise en compte d'un noyau légèrement plus volumineux. Le rôle de fonction préserve malgré tout la clarté et la maintenabilité du code source.

13.3.2 Gestion des structures de files d'attente

13.3.2.1 Initialisation

La fonction `skb_queue_head_init(list)` initialise une instance de la structure `skb_queue_head` de façon à pouvoir l'utiliser en tant que file d'attente. Les pointeurs sont positionnés sur la structure et la longueur définie à zéro : dans une file d'attente vide, `next` et `prev` pointent sur la file d'attente `list` et non sur `NULL`.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
498 static inline void skb_queue_head_init(struct sk_buff_head *list)
499 {
500         spin_lock_init(&list->lock);
```

```
501         list->prev = list->next = (struct sk_buff *)list;
502         list->qlen = 0;
503 }
```

13.3.2.2 Test de vacuité

La fonction `skb_queue_empty(list)` vérifie si la file d'attente `list` est vide ou si elle contient encore un descripteur de tampon. Le champ `list->qlen` est renvoyé par souci de facilité.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
316 /**
317  *      skb_queue_empty - verifie si la file d'attente est vide
318  *      @list : tete de file d'attente
319  *
320  *      Renvoie vrai si la file d'attente est vide, faux sinon.
321  */
322 static inline int skb_queue_empty(const struct sk_buff_head *list)
323 {
324         return list->next == (struct sk_buff *)list;
325 }
```

13.3.2.3 Longueur de la file d'attente

La fonction `skb_queue_len()` renvoie la longueur de la file d'attente passée en paramètre.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
487 /**
488  *      skb_queue_len   - obtient la longueur de la file d'attente
489  *      @list_ : liste a mesurer
490  *
491  *      Renvoie la longueur d'une file d'attente &sk_buff.
492  */
493 static inline __u32 skb_queue_len(const struct sk_buff_head *list_)
494 {
495         return list_->qlen;
496 }
```

13.3.3 Gestion des tampons d'une file d'attente

Pour les fonctions suivantes, il existe deux versions différentes : lorsqu'une fonction a déjà mis une interruption hors circuit, ceci n'a plus à être effectué dans la fonction concernant les files d'attente et peut être évité. Les fonctions sans interruption verrouillée se reconnaissent grâce à deux caractères de soulignement en préfixe, par exemple `__skb_dequeue()`.

13.3.3.1 Ajout d'un élément au début

La fonction `skb_queue_head(list, skb)` permet d'ajouter le descripteur de tampon d'adresse `skb` au début de la file d'attente `list`.

Elle est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
1235 /**
1236  *      skb_queue_head -  met un tampon en file d'attente en debut de liste
1237  *      @list : liste a utiliser
1238  *      @newsk : tampon a mettre en file d'attente
1239  *
1240  *      Met un tampon en file d'attente au debut de la liste. Cette fonction tire le
1241  *      verrou de la liste et peut donc etre utilisee de facon sure avec d'autres
1242  *      fonctions de de &sk_buff verrouillant de facon sure.
1243  *
```

```
1244  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
1245  */
1246 void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)
1247 {
1248         unsigned long flags;
1249
1250         spin_lock_irqsave(&list->lock, flags);
1251         __skb_queue_head(list, newsk);
1252         spin_unlock_irqrestore(&list->lock, flags);
1253 }
```

Elle fait appel à la fonction `__skb_queue_head()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
505 /*
506  *      Insere un sk_buff au debut d'une liste.
507  *
508  *      Les fonctions "__skb_xxxx()" sont celles qui sont non atomiques et qui
509  *      peuvent seulement etre appelees avec les interruptions inhibees.
510  */
511
512 /**
513  *      __skb_queue_head - met en tete d'une file d'attente un tampon
514  *      @list : liste a utiliser
515  *      @newsk : tampon a mettre en file d'attente
516  *
517  *      Met en file d'attente un tampon au debut d'une liste. Cette fonction ne tire aucun
518  *      verrou, vous devez donc detenir les verrous requis avant de l'appeler.
519  *
520  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
521  */
522 extern void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk);
523 static inline void __skb_queue_head(struct sk_buff_head *list,
524                                     struct sk_buff *newsk)
525 {
526         struct sk_buff *prev, *next;
527
528         newsk->list = list;
529         list->qlen++;
530         prev = (struct sk_buff *)list;
531         next = prev->next;
532         newsk->next = next;
533         newsk->prev = prev;
534         next->prev  = prev->next = newsk;
535 }
```

On utilise l'algorithme standard d'insertion dans une liste.

13.3.3.2 Ajout d'un élément à la fin

La fonction `skb_queue_tail(list, skb)` permet d'ajouter le descripteur de tampon d'adresse `skb` à la fin de la file d'attente `list`.

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

Code Linux 2.6.10

```
1255 /**
1256  *      skb_queue_tail - met un tampon en file d'attente a la fin d'une liste
1257  *      @list : liste a utiliser
1258  *      @newsk : tampon a mettre en file d'attente
1259  *
1260  *      Met un tampon en file d'attente a la fin de la liste. Cette fonction tire le
1261  *      verrou de la liste et peut donc etre utilisee de facon sure avec d'autres
1262  *      fonctions de &sk_buff verrouillant de facon sure.
```

```
1263  *
1264  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
1265  */
1266 void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
1267 {
1268         unsigned long flags;
1269
1270         spin_lock_irqsave(&list->lock, flags);
1271         __skb_queue_tail(list, newsk);
1272         spin_unlock_irqrestore(&list->lock, flags);
1273 }
```

Elle fait appel à la fonction `__skb_queue_tail()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
537 /**
538  *      __skb_queue_tail - met un tampon en file d'attente a la fin de la liste
539  *      @list : liste a utiliser
540  *      @newsk : tampon a placer dans la file d'attente
541  *
542  *      Met un tampon en file d'attente a la fin d'une liste. Cette fonction ne tire aucun
543  *      verrou, vous devez donc detenir les verrous requis avant de l'appeler.
544  *
545  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
546  */
547 extern void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk);
548 static inline void __skb_queue_tail(struct sk_buff_head *list,
549                                     struct sk_buff *newsk)
550 {
551         struct sk_buff *prev, *next;
552
553         newsk->list = list;
554         list->qlen++;
555         next = (struct sk_buff *)list;
556         prev = next->prev;
557         newsk->next = next;
558         newsk->prev = prev;
559         next->prev  = prev->next = newsk;
560 }
```

13.3.3.3 Retrait d'un élément au début

La fonction `skb_dequeue(list)` permet de récupérer le premier descripteur de tampon élément de la file d'attente. Elle renvoie l'adresse de celui-ci ou, si aucun élément ne se trouve dans la file d'attente, le pointeur `NULL`.

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

Code Linux 2.6.10

```
1181 /**
1182  *      skb_dequeue - retire de la tete d'une file d'attente
1183  *      @list : liste de laquelle il faut retirer
1184  *
1185  *      Retire la tete de la liste. Le verrou de la liste est tire, donc la fonction
1186  *      peut etre utilisee de facon sure avec d'autres fonctions de liste qui verrouillent.
1187  *      L'item de tete est renvoye, ou %NULL si la liste est vide.
1188  */
1189
1190 struct sk_buff *skb_dequeue(struct sk_buff_head *list)
1191 {
1192         unsigned long flags;
1193         struct sk_buff *result;
1194
1195         spin_lock_irqsave(&list->lock, flags);
```

```
1196         result = __skb_dequeue(list);
1197         spin_unlock_irqrestore(&list->lock, flags);
1198         return result;
1199 }
```

Elle fait appel à la fonction `__skb_dequeue()` définie dans le fichier `linux/include/linux/-skbuff.h`:

Code Linux 2.6.10

```
563 /**
564  *      __skb_dequeue - retire de la tete de la file d'attente
565  *      @list : liste de laquelle il faut retirer
566  *
567  *      Retire la tete de la liste. Cette fonction ne tire aucun verrou,
568  *      elle doit donc etre utilisee uniquement avec les verrous appropries detenus.
569  *      L'item de tete est renvoye, ou %NULL si la liste est vide.
570  */
571 extern struct sk_buff *skb_dequeue(struct sk_buff_head *list);
572 static inline struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
573 {
574         struct sk_buff *next, *prev, *result;
575
576         prev = (struct sk_buff *) list;
577         next = prev->next;
578         result = NULL;
579         if (next != prev) {
580                 result       = next;
581                 next         = next->next;
582                 list->qlen--;
583                 next->prev   = prev;
584                 prev->next   = next;
585                 result->next = result->prev = NULL;
586                 result->list = NULL;
587         }
588         return result;
589 }
```

13.3.3.4 Retrait d'un élément à la fin

La fonction `skb_dequeue_tail(list)` permet de retirer le dernier descripteur de tampon élément de la file d'attente passée en argument. Elle renvoie l'adresse de celui-ci ou, si aucun élément ne se trouve dans la file d'attente, le pointeur `NULL`.

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

Code Linux 2.6.10

```
1201 /**
1202  *      skb_dequeue_tail - retire a la fin de la file d'attente
1203  *      @list : liste de laquelle retirer
1204  *
1205  *      Retire la queue de la liste. Le verrou de la liste est tire, la fonction
1206  *      peut donc etre utilisee de facon sure avec d'autres fonctions de liste qui
1207  *      verrouillent. L'item de queue est renvoye, ou %NULL si la liste est vide.
1208  */
1209 struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list)
1210 {
1211         unsigned long flags;
1212         struct sk_buff *result;
1213
1214         spin_lock_irqsave(&list->lock, flags);
1215         result = __skb_dequeue_tail(list);
1216         spin_unlock_irqrestore(&list->lock, flags);
1217         return result;
1218 }
```

Elle fait appel à la fonction `__skb_dequeue_tail()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
637 /**
638  *      __skb_dequeue_tail - retire a la fin de la file d'attente
639  *      @list : liste de laquelle retirer
640  *
641  *      Retire la fin de la liste. Cette fonction ne tire aucun verrou,
642  *      elle ne doit donc etre utilisee qu'avec les verrous appropries detenus.
643  *      L'item de queue est renvoye ou %NULL si la liste est vide.
644  */
645 extern struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list);
646 static inline struct sk_buff *__skb_dequeue_tail(struct sk_buff_head *list)
647 {
648         struct sk_buff *skb = skb_peek_tail(list);
649         if (skb)
650                 __skb_unlink(skb, list);
651         return skb;
652 }
```

qui renvoie à la fonction `__skb_unlink()` définie un peu plus haut dans le même fichier :

Code Linux 2.6.10

```
616 /*
617  * retire un sk_buff d'une liste. _Doit_ etre appelee atomiquement, et avec
618  * la liste connue...
619  */
620 extern void        skb_unlink(struct sk_buff *skb);
621 static inline void __skb_unlink(struct sk_buff *skb, struct sk_buff_head *list)
622 {
623         struct sk_buff *next, *prev;
624
625         list->qlen--;
626         next       = skb->next;
627         prev       = skb->prev;
628         skb->next  = skb->prev = NULL;
629         skb->list  = NULL;
630         next->prev = prev;
631         prev->next = next;
632 }
```

13.3.3.5 Vidage d'une file d'attente

La fonction `skb_queue_purge(list)` permet de vider la file d'attente passée en argument. Elle est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
1220 /**
1221  *      skb_queue_purge - vide une liste
1222  *      @list : liste a vider
1223  *
1224  *      Enleve tous les tampons d'une liste &sk_buff. Chaque tampon est enleve de
1225  *      la liste en ecartant une reference. Cette fonction tire le verrou de la
1226  *      liste et est donc atomique a l'egard des autres fonctions de liste qui verrouillent.
1227  */
1228 void skb_queue_purge(struct sk_buff_head *list)
1229 {
1230         struct sk_buff *skb;
1231         while ((skb = skb_dequeue(list)) != NULL)
1232                 kfree_skb(skb);
1233 }
```

La fonction `__skb_queue_purge()`, quant à elle, est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
910 /**
911  *      __skb_queue_purge - vide une liste
912  *      @list : liste a vider
913  *
914  *      Enleve tous les tampons d'une liste &sk_buff. Chaque tampon est enleve de
915  *      la liste en ecartant une reference. Cette fonction ne tire pas le verrou de
916  *      la liste, aussi l'appelant doit-il detenir les verrous adequats pour l'utiliser.
917  */
918 extern void skb_queue_purge(struct sk_buff_head *list);
919 static inline void __skb_queue_purge(struct sk_buff_head *list)
920 {
921         struct sk_buff *skb;
922         while ((skb = __skb_dequeue(list)) != NULL)
923                 kfree_skb(skb);
924 }
```

13.3.3.6 Insertion d'un élément avant un autre

La fonction `skb_insert(oldskb, newskb)` permet d'insérer le descripteur de tampon d'adresse `newskb` avant le descripteur de tampon d'adresse `oldskb` dans la file d'attente à laquelle appartient `*oldskb`. Rappelons qu'un descripteur de tampon ne peut se trouver que dans une seule file d'attente à la fois et que celle-ci fait l'objet d'un des champs du descripteur.

Code Linux 2.6.10

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

```
1321 /**
1322  *      skb_insert      -       insere un tampon
1323  *      @old : tampon devant lequel inserer
1324  *      @newsk : tampon a inserer
1325  *
1326  *      Place un paquet avant un paquet donne dans une liste. Les verrous de la liste sont
1327  *      tires, donc cette fonction est atomique a l'egard des autres appels verrouilles a la
1328  *      liste. Un tampon ne peut pas etre place dans deux listes en meme temps.
1329  */
1330
1331 void skb_insert(struct sk_buff *old, struct sk_buff *newsk)
1332 {
1333         unsigned long flags;
1334
1335         spin_lock_irqsave(&old->list->lock, flags);
1336         __skb_insert(newsk, old->prev, old, old->list);
1337         spin_unlock_irqrestore(&old->list->lock, flags);
1338 }
```

Elle fait appel à la fonction `__skb_insert()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
592 /*
593  *      Insere un paquet dans une liste.
594  */
595 extern void        skb_insert(struct sk_buff *old, struct sk_buff *newsk);
596 static inline void __skb_insert(struct sk_buff *newsk,
597                                 struct sk_buff *prev, struct sk_buff *next,
598                                 struct sk_buff_head *list)
599 {
600         newsk->next = next;
601         newsk->prev = prev;
602         next->prev  = prev->next = newsk;
603         newsk->list = list;
```

```
604         list->qlen++;
605 }
```

13.3.3.7 Insertion d'un élément après un autre

La fonction `skb_append(oldskb, newskb)` permet d'insérer le descripteur de tampon d'adresse `newskb` après le descripteur de tampon d'adresse `oldskb` dans la file d'attente à laquelle appartient `*oldskb`.

Elle est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
1301 /**
1302  *      skb_append      -       place un tampon apres un autre
1303  *      @old : tampon apres lequel inserer
1304  *      @newsk : tampon a inserer
1305  *
1306  *      Place un paquet apres un paquet donne dans une liste. Les verrous de la liste sont
1307  *      tires, donc cette fonction est atomique a l'egard des autres appels verrouilles de
1308  *      liste. Un tampon ne peut pas etre place dans deux listes en meme temps.
1309  */
1310
1311 void skb_append(struct sk_buff *old, struct sk_buff *newsk)
1312 {
1313         unsigned long flags;
1314
1315         spin_lock_irqsave(&old->list->lock, flags);
1316         __skb_append(old, newsk);
1317         spin_unlock_irqrestore(&old->list->lock, flags);
1318 }
```

Elle fait appel à la fonction `__skb_append()` définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
607 /*
608  *      Place un paquet apres un paquet donne dans une liste.
609  */
610 extern void        skb_append(struct sk_buff *old, struct sk_buff *newsk);
611 static inline void __skb_append(struct sk_buff *old, struct sk_buff *newsk)
612 {
613         __skb_insert(newsk, old, old->next, old->list);
614 }
```

13.3.3.8 Retrait dans une file d'attente

La fonction `skb_unlink(skb)` permet de retirer le descripteur de tampon dont l'adresse est passée en paramètre de la file d'attente à laquelle il appartient.

Elle est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
1274 /**
1275  *      skb_unlink      -       retire un tampon d'une liste
1276  *      @skb : tampon a retirer
1277  *
1278  *      [Retire un paquet d'une liste]. Les verrous de la liste sont tires,
1279  *      donc cette fonction est atomique a l'egard d'autres appels verrouilles de liste.
1280  *
1281  *      Marche meme sans connaitre la liste dans laquelle il se trouve, qui doit etre
1282  *      accessible. Ceci signifie aussi que LA LISTE DOIT EXISTER lorsque vous
1283  *      retirez. Ainsi une liste doit avoir son contenu non lie avant qu'il ne soit
1284  *      detruit.
1285  */
1286 void skb_unlink(struct sk_buff *skb)
1287 {
```

```
1288         struct sk_buff_head *list = skb->list;
1289
1290         if (list) {
1291                 unsigned long flags;
1292
1293                 spin_lock_irqsave(&list->lock, flags);
1294                 if (skb->list == list)
1295                         __skb_unlink(skb, skb->list);
1296                 spin_unlock_irqrestore(&list->lock, flags);
1297         }
1298 }
```

13.3.3.9 Pointage sur le premier élément d'une liste

La fonction `skb_peek(list)` renvoie un pointeur sur le premier élement de la file d'attente passée en paramètre, si celle-ci n'est pas vide. Sinon `NULL` est renvoyé. L'élément n'est pas retiré de la file d'attente.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
445 /**
446  *      skb_peek
447  *      @list_ :  liste sur laquelle pointer
448  *
449  *      Pointe sur un &sk_buff. Contrairement a beaucoup d'autres operations, vous _DEVEZ_
450  *      etre soigneux avec celle-ci. Pointer laisse le tampon dans la
451  *      liste et quelqu'un d'autre peut se l'approprier. Vous devez detenir
452  *      les verrous appropries ou avoir une file d'attente privee pour faire ca.
453  *
454  *      Renvoie %NULL pour une liste vide ou un pointeur sur l'element de tete.
455  *      Le compteur de reference n'est pas incremente et la reference est ainsi
456  *      volatile. A utiliser avec precaution.
457  */
458 static inline struct sk_buff *skb_peek(struct sk_buff_head *list_)
459 {
460         struct sk_buff *list = ((struct sk_buff *)list_)->next;
461         if (list == (struct sk_buff *)list_)
462                 list = NULL;
463         return list;
464 }
```

13.3.3.10 Pointage sur le dernier élément d'une liste

La fonction `skb_peek_tail(list)` renvoie un pointeur sur le dernier élement de la file d'attente passée en paramètre, si celle-ci n'est pas vide. Sinon `NULL` est renvoyé. L'élément n'est pas retiré de la file d'attente.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
466 /**
467  *      skb_peek_tail
468  *      @list_ : liste sur laquelle pointer
469  *
470  *      Pointe sur un &sk_buff. Contrairement a beaucoup d'autres operations vous _DEVEZ_
471  *      etre soigneux avec celle-ci. Pointer laisse le tampon dans la
472  *      liste et quelqu'un d'autre peut se l'approprier. Vous devez detenir
473  *      les verrous appropries ou avoir une file d'attente privee pour ca.
474  *
475  *      Renvoie %NULL pour une liste vide ou un pointeur sur l'element de queue.
476  *      Le compteur de reference n'est pas incremente et la reference est ainsi
477  *      volatile. A utiliser avec precaution.
478  */
```

```
479 static inline struct sk_buff *skb_peek_tail(struct sk_buff_head *list_)
480 {
481         struct sk_buff *list = ((struct sk_buff *)list_)->prev;
482         if (list == (struct sk_buff *)list_)
483                 list = NULL;
484         return list;
485 }
```

13.4 Initialisation de l'antémémoire des tampons de socket

L'antémémoire de tampons de socket est initialisée grâce à la fonction `skb_init()`, définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
1428 void __init skb_init(void)
1429 {
1430         skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
1431                                               sizeof(struct sk_buff),
1432                                               0,
1433                                               SLAB_HWCACHE_ALIGN,
1434                                               NULL, NULL);
1435         if (!skbuff_head_cache)
1436                 panic("cannot create skbuff cache");
1437 }
```

On se contente d'essayer d'instantier la variable `skbuff_head_cache` pour qu'elle pointe sur un emplacement mémoire de la taille d'un descripteur de socket. Si on n'y parvient pas, on affiche un message noyau.

13.5 Fonctions de copie

13.5.1 Duplication d'un descripteur de tampon

13.5.1.1 Description

La fonction:

```
struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask);
```

duplique le descripteur de tampon de socket `skb` et renvoie l'adresse du nouveau (ou la valeur `NULL` s'il y a un problème). On spécifie que les deux descripteurs de tampon en présence possèdent un clone. Les pointeurs du descripteur originel et ceux du nouveau descripteur désignent le même tampon. Puisqu'il n'existe aucune référence inverse dans le tampon à son ou ses descripteurs de tampon, la zone mémoire du tampon doit être traitée en lecture seule.

Cette fonction est requise entre autres au cours de l'implémentation de la multidiffusion. On évite ainsi la fastidieuse copie du tampon lors de l'émission d'un paquet sur plusieurs périphériques réseau.

13.5.1.2 Définition

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

```
259 /**
260  *      skb_clone       -       duplique un sk_buff
261  *      @skb : tampon a cloner
262  *      @gfp_mask : priorite d'allocation
263  *
```

```
264  *      Duplique un &sk_buff. Le nouveau n'est pas propriete d'une socket. Les
265  *      copies partagent les memes donnees de paquet mais pas la meme structure. Le nouveau
266  *      tampon a un compteur de reference de 1. Si l'allocation echoue, la
257  *      fonction renvoie %NULL, sinon le nouveau tampon est renvoye.
268  *
269  *      Si cette fonction est appelee depuis une interruption, gfp_mask() doit etre
270  *      %GFP_ATOMIC.
271  */
272
273 struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask)
274 {
275         struct sk_buff *n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
276
277         if (!n)
278                 return NULL;
279
280 #define C(x) n->x = skb->x
281
282         n->next = n->prev = NULL;
283         n->list = NULL;
284         n->sk = NULL;
285         C(stamp);
286         C(dev);
287         C(real_dev);
288         C(h);
289         C(nh);
290         C(mac);
291         C(dst);
292         dst_clone(skb->dst);
293         C(sp);
294 #ifdef CONFIG_INET
295         secpath_get(skb->sp);
296 #endif
297         memcpy(n->cb, skb->cb, sizeof(skb->cb));
298         C(len);
299         C(data_len);
300         C(csum);
301         C(local_df);
302         n->cloned = 1;
303         C(pkt_type);
304         C(ip_summed);
305         C(priority);
306         C(protocol);
307         C(security);
308         n->destructor = NULL;
309 #ifdef CONFIG_NETFILTER
310         C(nfmark);
311         C(nfcache);
312         C(nfct);
313         nf_conntrack_get(skb->nfct);
314         C(nfctinfo);
315 #ifdef CONFIG_NETFILTER_DEBUG
316         C(nf_debug);
317 #endif
318 #ifdef CONFIG_BRIDGE_NETFILTER
319         C(nf_bridge);
320         nf_bridge_get(skb->nf_bridge);
321 #endif
322 #endif /*CONFIG_NETFILTER*/
323 #if defined(CONFIG_HIPPI)
324         C(private);
325 #endif
```

```
326 #ifdef CONFIG_NET_SCHED
327         C(tc_index);
328 #ifdef CONFIG_NET_CLS_ACT
329         n->tc_verd = SET_TC_VERD(skb->tc_verd,0);
330         n->tc_verd = CLR_TC_OK2MUNGE(skb->tc_verd);
331         n->tc_verd = CLR_TC_MUNGED(skb->tc_verd);
332         C(input_dev);
333         C(tc_classid);
334 #endif
335
336 #endif
337         C(truesize);
338         atomic_set(&n->users, 1);
339         C(head);
340         C(data);
341         C(tail);
342         C(end);
343
344         atomic_inc(&(skb_shinfo(skb)->dataref));
345         skb->cloned = 1;
346
347         return n;
348 }
```

dont le code est suffisamment parlant (en gros les champs du nouveau descripteur prennent la même valeur que ceux du descripteur à dupliquer).

La fonction en ligne `dst_clone()` est définie dans le fichier `linux/include/net/dst.h`: Code Linux 2.6.10

```
137 static inline
138 struct dst_entry * dst_clone(struct dst_entry * dst)
139 {
140         if (dst)
141                 atomic_inc(&dst->__refcnt);
142         return dst;
143 }
```

13.5.2 Copie d'un tampon et de son descripteur

13.5.2.1 Description

La fonction:

```
struct sk_buff *skb_copy(const struct sk_buff *skb, int gfp_mask);
```

effectue une copie du descripteur de tampon passé en argument, ainsi que du tampon lui-même. Elle renvoie l'adresse du nouveau descripteur de tampon de socket ou la valeur `NULL` s'il y a un problème.

13.5.2.2 Définition

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

```
401 /**
402  *      skb_copy        -       cree une copie privee d'un sk_buff
403  *      @skb : tampon a copier
404  *      @gfp_mask : priorite d'allocation
405  *
406  *      Fait une copie a la fois d'un &sk_buff et de ses donnees. Ceci est utilise
407  *      lorsque l'appelant desire modifier les donnees et a besoin d'une copie privee
408  *      des donnees a alterer. Renvoie %NULL en cas d'echec, le pointeur au tampon
```

```
409  *      en cas de succes. Le tampon renvoye a un compteur de reference de 1.
410  *
411  *      Comme sous-produit, cette fonction transforme un &sk_buff non lineaire en
412  *      un qui est lineaire, de sorte que ce &sk_buff devient completement prive
413  *      et que l'appelant peut modifier toutes les donnees du tampon renvoye. Ceci
414  *      signifie qu'il n'est pas recommande d'utiliser cette fonction dans des
415  *      circonstances ou seul l'en-tete doit etre modifie. Utiliser pskb_copy() alors.
416  */
417
418 struct sk_buff *skb_copy(const struct sk_buff *skb, int gfp_mask)
419 {
420         int headerlen = skb->data - skb->head;
421         /*
422          *      Alloue le tampon de copie
423          */
424         struct sk_buff *n = alloc_skb(skb->end - skb->head + skb->data_len,
425                                       gfp_mask);
426         if (!n)
427                 return NULL;
428
429         /* Positionne le pointeur des donnees */
430         skb_reserve(n, headerlen);
431         /* Positionne le pointeur tail et la longueur */
432         skb_put(n, skb->len);
433         n->csum      = skb->csum;
434         n->ip_summed = skb->ip_summed;
435
436         if (skb_copy_bits(skb, -headerlen, n->head, headerlen + skb->len))
437                 BUG();
438
439         copy_skb_header(n, skb);
440         return n;
441 }
```

Autrement dit :

- On déclare une taille d'en-tête, que l'on initialise avec la différence entre l'adresse des données et celle de l'en-tête, ce qui donne bien ce qui est voulu.
- On déclare l'adresse `n` (comme *new*) du nouveau descripteur de tampon.
- On essaie d'allouer de la place en mémoire vive pour le nouveau tampon. Si on n'y parvient pas, on renvoie `NULL`.
- On ajuste la taille de l'espace de tête du tampon.
- On ajuste la taille effective des données du nouveau tampon.
- On renseigne les champs `csum` et `ip_summed` du nouveau descripteur de tampon, en copiant les valeurs de ceux de l'ancien.
- On copie les données dans le nouveau tampon, grâce à la fonction `skb_copy_bits()`, décrite ci-dessous. Si une erreur intervient lors de la copie, on interrompt tout.
- On copie l'en-tête du descripteur à dupliquer vers le nouveau, grâce à la fonction `copy_skb_header()`, décrite ci-dessous.
- On renvoie l'adresse du nouveau descripteur de tampon.

13.5.2.3 Copie des données du tampon

La fonction :

```
skb_copy_bits(const struct sk_buff *skb, int offset, void *to, int len);
```

permet de copier `len` octets de la partie des données contenues dans le tampon repéré par le descripteur de tampon `skb` à partir du décalage `offset` à l'emplacement mémoire d'adresse `to`.

Elle est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
855 /* Copie les bits de donnees d'un skb vers un tampon du noyau. */
856
857 int skb_copy_bits(const struct sk_buff *skb, int offset, void *to, int len)
858 {
859         int i, copy;
860         int start = skb_headlen(skb);
861
862         if (offset > (int)skb->len - len)
863                 goto fault;
864
865         /* Copie l'en-tete. */
866         if ((copy = start - offset) > 0) {
867                 if (copy > len)
868                         copy = len;
869                 memcpy(to, skb->data + offset, copy);
870                 if ((len -= copy) == 0)
871                         return 0;
872                 offset += copy;
873                 to     += copy;
874         }
875
876         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
877                 int end;
878
879                 BUG_TRAP(start <= offset + len);
880
881                 end = start + skb_shinfo(skb)->frags[i].size;
882                 if ((copy = end - offset) > 0) {
883                         u8 *vaddr;
884
885                         if (copy > len)
886                                 copy = len;
887
888                         vaddr = kmap_skb_frag(&skb_shinfo(skb)->frags[i]);
889                         memcpy(to,
890                                vaddr + skb_shinfo(skb)->frags[i].page_offset+
891                                offset - start, copy);
892                         kunmap_skb_frag(vaddr);
893
894                         if ((len -= copy) == 0)
895                                 return 0;
896                         offset += copy;
897                         to     += copy;
898                 }
899                 start = end;
900         }
901
902         if (skb_shinfo(skb)->frag_list) {
903                 struct sk_buff *list = skb_shinfo(skb)->frag_list;
904
905                 for (; list; list = list->next) {
906                         int end;
907
908                         BUG_TRAP(start <= offset + len);
909
910                         end = start + list->len;
911                         if ((copy = end - offset) > 0) {
912                                 if (copy > len)
913                                         copy = len;
914                                 if (skb_copy_bits(list, offset - start,
915                                                   to, copy))
```

```
916                                     goto fault;
917                             if ((len -= copy) == 0)
918                                     return 0;
919                             offset += copy;
920                             to     += copy;
921                     }
922                     start = end;
923             }
924     }
925     if (!len)
926             return 0;
927
928 fault:
929     return -EFAULT;
930 }
```

Autrement dit :

- On déclare une variable de début, que l'on initialise avec la longueur de l'en-tête du tampon.

Code Linux 2.6.10

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

```
660 static inline unsigned int skb_headlen(const struct sk_buff *skb)
661 {
662         return skb->len - skb->data_len;
663 }
```

- Si la somme du décalage et de la longueur passés en argument est supérieure à la longueur des données du tampon, on renvoie l'opposé du code d'erreur `EFAULT`.
- On détermine le nombre d'octets à copier pour l'en-tête. S'il est non nul, on copie l'en-tête. Si ce qu'on a copié est suffisant, on a terminé, on renvoie donc 0. Sinon on déplace le décalage.
- On copie chacun des fragments du tableau des fragments.

Code Linux 2.6.10

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

```
821 #define BUG_TRAP(x) do { \
822         if (unlikely(!(x))) { \
823                 printk(KERN_ERR "KERNEL: assertion (%s) failed at %s (%d)\n", \
824                         #x,  __FILE__ , __LINE__); \
825         } \
826 } while(0)
```

Code Linux 2.6.10

Les fonctions `kmap_skb_frag()` et `kunmap_skb_frag()` sont définies dans le fichier `linux/include/linux/skbuff.h`:

```
1055 static inline void *kmap_skb_frag(const skb_frag_t *frag)
1056 {
1057 #ifdef CONFIG_HIGHMEM
1058         BUG_ON(in_irq());
1059
1060         local_bh_disable();
1061 #endif
1062         return kmap_atomic(frag->page, KM_SKB_DATA_SOFTIRQ);
1063 }
1064
1065 static inline void kunmap_skb_frag(void *vaddr)
1066 {
1067         kunmap_atomic(vaddr, KM_SKB_DATA_SOFTIRQ);
1068 #ifdef CONFIG_HIGHMEM
1069         local_bh_enable();
1070 #endif
1071 }
```

– On copie les données des tampons associés aux descripteurs de la liste des fragments.
– On renvoie le nombre d'octets copiés.

13.5.2.4 Copie de l'en-tête du descripteur

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

```
350 static void copy_skb_header(struct sk_buff *new, const struct sk_buff *old)
351 {
352         /*
353          *      Decalage entre les deux zones des donnees, en octets
354          */
355         unsigned long offset = new->data - old->data;
356
357         new->list       = NULL;
358         new->sk         = NULL;
359         new->dev        = old->dev;
360         new->real_dev   = old->real_dev;
361         new->priority   = old->priority;
362         new->protocol   = old->protocol;
363         new->dst        = dst_clone(old->dst);
364 #ifdef CONFIG_INET
365         new->sp         = secpath_get(old->sp);
366 #endif
367         new->h.raw      = old->h.raw + offset;
368         new->nh.raw     = old->nh.raw + offset;
369         new->mac.raw    = old->mac.raw + offset;
370         memcpy(new->cb, old->cb, sizeof(old->cb));
371         new->local_df   = old->local_df;
372         new->pkt_type   = old->pkt_type;
373         new->stamp      = old->stamp;
374         new->destructor = NULL;
375         new->security   = old->security;
376 #ifdef CONFIG_NETFILTER
377         new->nfmark     = old->nfmark;
378         new->nfcache    = old->nfcache;
379         new->nfct       = old->nfct;
380         nf_conntrack_get(old->nfct);
381         new->nfctinfo   = old->nfctinfo;
382 #ifdef CONFIG_NETFILTER_DEBUG
383         new->nf_debug   = old->nf_debug;
384 #endif
385 #ifdef CONFIG_BRIDGE_NETFILTER
386         new->nf_bridge  = old->nf_bridge;
387         nf_bridge_get(old->nf_bridge);
388 #endif
389 #endif
390 #ifdef CONFIG_NET_SCHED
391 #ifdef CONFIG_NET_CLS_ACT
392         new->tc_verd = old->tc_verd;
393 #endif
394         new->tc_index   = old->tc_index;
395 #endif
396         atomic_set(&new->users, 1);
397         skb_shinfo(new)->tso_size = skb_shinfo(old)->tso_size;
398         skb_shinfo(new)->tso_segs = skb_shinfo(old)->tso_segs;
399 }
```

Elle se contente tout simplement de recopier un certain nombre de champs de l'ancien descripteur vers le nouveau.

13.5.3 Copie du tampon sans linéarisation

Code Linux 2.6.10

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

```
444 /**
445  *      pskb_copy       -       cree une copie d'un sk_buff avec une tete privee.
446  *      @skb : tampon a copier
447  *      @gfp_mask : priorite d'allocation
448  *
449  *      Fait une copie a la fois d'un &sk_buff et d'une partie de ses donnees, celles
450  *      situees dans l'en-tete. Les donnees fragmentees restent partagees. Ceci est
451  *      utilise lorsque l'appelant ne desire modifier que l'en-tete du &sk_buff et a besoin
452  *      d'une copie privee de l'en-tete a alterer. Renvoie %NULL en cas d'echec
453  *      ou l'adresse du tampon en cas de succes.
454  *      Le tampon renvoye a un compteur de reference de 1.
455  */
456
457 struct sk_buff *pskb_copy(struct sk_buff *skb, int gfp_mask)
458 {
459         /*
460          *      Alloue le tampon de copie
461          */
462         struct sk_buff *n = alloc_skb(skb->end - skb->head, gfp_mask);
463
464         if (!n)
465                 goto out;
466
467         /* Positionne le pointeur des donnees */
468         skb_reserve(n, skb->data - skb->head);
469         /* Positionne le pointeur tail et la longueur */
470         skb_put(n, skb_headlen(skb));
471         /* Copie les octets */
472         memcpy(n->data, skb->data, n->len);
473         n->csum      = skb->csum;
474         n->ip_summed = skb->ip_summed;
475
476         n->data_len  = skb->data_len;
477         n->len       = skb->len;
478
479         if (skb_shinfo(skb)->nr_frags) {
480                 int i;
481
482                 for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
483                         skb_shinfo(n)->frags[i] = skb_shinfo(skb)->frags[i];
484                         get_page(skb_shinfo(n)->frags[i].page);
485                 }
486                 skb_shinfo(n)->nr_frags = i;
487         }
488
489         if (skb_shinfo(skb)->frag_list) {
490                 skb_shinfo(n)->frag_list = skb_shinfo(skb)->frag_list;
491                 skb_clone_fraglist(n);
492         }
493
494         copy_skb_header(n, skb);
495 out:
496         return n;
497 }
```

Le code se comprend aisément.

13.5.4 Copie d'un tampon et de son descripteur avec modification

13.5.4.1 Description

La fonction :

```
struct sk_buff *skb_copy_expand(const struct sk_buff *skb,
                                int newheadroom,
                                int newtailroom,
                                int gfp_mask);
```

effectue une copie du tampon et du descripteur du tampon désigné par le descripteur `skb`, en en profitant pour modifier la taille de l'espace de tête (passant à `newheadroom` octets) et celle de l'espace d'en-queue (passant à `newtailroom` octets). On renvoie l'adresse du nouveau descripteur de tampon de socket ou la valeur `NULL` s'il y a un problème.

13.5.4.2 Définition

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

```
582 /**
583  *      skb_copy_expand -       copie et etend un sk_buff
584  *      @skb : tampon a copier
585  *      @newheadroom : nouveaux octets libres en tete
586  *      @newtailroom : nouveaux octets libres en queue
587  *      @gfp_mask : priorite d'allocation
588  *
589  *      Fait une copie a la fois d'un &sk_buff et de ses donnees tout en
590  *      allouant un espace supplementaire.
591  *
592  *      Ceci est utilise lorsque l'appelant desire modifier les donnees et a besoin d'une
593  *      copie privee des donnees a alterer ainsi que de plus d'espace pour de nouveaux champs.
594  *      Renvoie %NULL en cas d'echec ou l'adresse du tampon en cas
595  *      de succes. Le tampon renvoye a un compteur de reference de 1.
596  *
597  *      Vous devez passer %GFP_ATOMIC comme priorite d'allocation si cette fonction
598  *      est appelee depuis une interruption.
599  *
600  *      ALERTE DE BOGUE : ip_summed n'est pas copie. Pourquoi cela marche-t-il ? Est-elle
601  *      seulement utilisee par netfilter dans les cas ou checksum est recalculee ? --ANK
602  */
603 struct sk_buff *skb_copy_expand(const struct sk_buff *skb,
604                                 int newheadroom, int newtailroom, int gfp_mask)
605 {
606         /*
607          *      Alloue le tampon de copie
608          */
609         struct sk_buff *n = alloc_skb(newheadroom + skb->len + newtailroom,
610                                       gfp_mask);
611         int head_copy_len, head_copy_off;
612
613         if (!n)
614                 return NULL;
615
616         skb_reserve(n, newheadroom);
617
618         /* Positionne le pointeur tail et la longueur */
619         skb_put(n, skb->len);
620
621         head_copy_len = skb_headroom(skb);
622         head_copy_off = 0;
```

```
623         if (newheadroom <= head_copy_len)
624                 head_copy_len = newheadroom;
625         else
626                 head_copy_off = newheadroom - head_copy_len;
627
628         /* Copie l'en-tete et les donnees lineairement. */
629         if (skb_copy_bits(skb, -head_copy_len, n->head + head_copy_off,
630                           skb->len + head_copy_len))
631                 BUG();
632
633         copy_skb_header(n, skb);
634
635         return n;
636 }
```

Par rapport à la fonction de copie simple, on voit surtout la différence au niveau de l'instantiation du nouveau descripteur de tampon.

13.6 Ajustement d'une zone des données occupée

Nous avons vu ci-dessus comment ajuster la taille d'une zone des données vides. C'est plus difficile lorsque celle-ci contient des données. Nous allons voir comment faire dans ce cas ici.

13.6.1 Retrait de données au début

13.6.1.1 Fonction d'appel

La fonction `pskb_pull(skb, len)` tronque de `len` octets le début de la zone des données d'un tampon, après avoir vérifié qu'on demeurera après ceci dans la zone de mémoire affectée au tampon. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
768 extern unsigned char *__pskb_pull_tail(struct sk_buff *skb, int delta);
769
770 static inline unsigned char *__pskb_pull(struct sk_buff *skb, unsigned int len)
771 {
772         if (len > skb_headlen(skb) &&
773             !__pskb_pull_tail(skb, len-skb_headlen(skb)))
774                 return NULL;
775         skb->len -= len;
776         return skb->data += len;
777 }
778
779 static inline unsigned char *pskb_pull(struct sk_buff *skb, unsigned int len)
780 {
781         return unlikely(len > skb->len) ? NULL : __pskb_pull(skb, len);
782 }
783
784 static inline int pskb_may_pull(struct sk_buff *skb, unsigned int len)
785 {
786         if (likely(len <= skb_headlen(skb)))
787                 return 1;
788         if (unlikely(len > skb->len))
789                 return 0;
790         return __pskb_pull_tail(skb, len-skb_headlen(skb)) != NULL;
791 }
```

Elle fait essentiellement appel à la fonction interne `__pskb_pull_tail()`.

13.6.1.2 Fonction interne

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

```
716 /**
717  *      __pskb_pull_tail - avance la fin d'un en-tete de skb
718  *      @skb : tampon a reallouer
719  *      @delta : nombre d'octets a avancer a la fin
720  *
721  *      Cette fonction n'a de sens que pour un &sk_buff fragmente,
722  *      elle etend l'en-tete en deplacant la fin et en copiant les donnees
723  *      necessaires de la partie fragmentee.
724  *
725  *      &sk_buff DOIT avoir un compteur de reference de 1.
726  *
727  *      Renvoie %NULL (et &sk_buff ne change pas) si l'etirement echoue
728  *      ou la valeur de la nouvelle fin du skb en cas de succes.
729  *
730  *      Tous les pointeurs pointant sur l'en-tete de skb peuvent changer et doivent
731  *      donc etre charges a nouveau apres appel de cette fonction.
732  */
733
734 /* Deplace la fin de la tete du skb, en copiant les donnees de la partie fragmentee
735  * lorsque c'est necessaire.
736  * 1. On peut echouer a cause d'un echec memoire.
737  * 2. On peut changer les pointeurs de skb.
738  *
739  * C'est gentiment complique. Heureusement, elle est appelee seulement dans des cas
       exceptionels.
740  */
741 unsigned char *__pskb_pull_tail(struct sk_buff *skb, int delta)
742 {
743         /* Si skb n'a pas suffisamment d'espace libre a la fin, en prendre plus
744          * ainsi que 128 octets pour les futures extensions. Si nous n'avons pas assez
745          * de place a la fin, reallouer sans extension seulement si skb est clone.
746          */
747         int i, k, eat = (skb->tail + delta) - skb->end;
748
749         if (eat > 0 || skb_cloned(skb)) {
750                 if (pskb_expand_head(skb, 0, eat > 0 ? eat + 128 : 0,
751                                      GFP_ATOMIC))
752                         return NULL;
753         }
754
755         if (skb_copy_bits(skb, skb_headlen(skb), skb->tail, delta))
756                 BUG();
757
758         /* Optimisation : pas de fragments, pas de raisons de preestimer
759          * la taille des pages a placer. Superbe.
760          */
761         if (!skb_shinfo(skb)->frag_list)
762                 goto pull_pages;
763
764         /* Estimer la taille des pages a placer. */
765         eat = delta;
766         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
767                 if (skb_shinfo(skb)->frags[i].size >= eat)
768                         goto pull_pages;
769                 eat -= skb_shinfo(skb)->frags[i].size;
770         }
771
772         /* Si nous avons besoin de mettre a jour la frag_list, nous sommes ennuyes.
773          * Il est certainement possible d'ajouter un decalage aux donnees de skb,
```

```
774          * mais on s'attend a ce que prendre en compte ce depilage soit
775          * une operation tres rare, il est pire de combattre contre les tetes de
776          * skb gonflees et de nous crucifier ici a la place.
777          * Pur masochisme, en effet. 8)8)
778          */
779         if (eat) {
780                 struct sk_buff *list = skb_shinfo(skb)->frag_list;
781                 struct sk_buff *clone = NULL;
782                 struct sk_buff *insp = NULL;
783 
784                 do {
785                         if (!list)
786                                 BUG();
787 
788                         if (list->len <= eat) {
789                                 /* Mange en entier. */
790                                 eat -= list->len;
791                                 list = list->next;
792                                 insp = list;
793                         } else {
794                                 /* Mange partiellement. */
795 
796                                 if (skb_shared(list)) {
797                                         /* Tres mauvais ! Nous avons besoin de
                                            bifurquer la liste. :-( */
798                                         clone = skb_clone(list, GFP_ATOMIC);
799                                         if (!clone)
800                                                 return NULL;
801                                         insp = list->next;
802                                         list = clone;
803                                 } else {
804                                         /* Ceci peut etre place sans
805                                          * probleme. */
806                                         insp = list;
807                                 }
808                                 if (!pskb_pull(list, eat)) {
809                                         if (clone)
810                                                 kfree_skb(clone);
811                                         return NULL;
812                                 }
813                                 break;
814                         }
815                 } while (eat);
816 
817                 /* Liberer les fragments. */
818                 while ((list = skb_shinfo(skb)->frag_list) != insp) {
819                         skb_shinfo(skb)->frag_list = list->next;
820                         kfree_skb(list);
821                 }
822                 /* Et inserer un nouveau clone en tete. */
823                 if (clone) {
824                         clone->next = list;
825                         skb_shinfo(skb)->frag_list = clone;
826                 }
827         }
828         /* Succes ! Maintenant nous pouvons faire subir des changements aux donnees de skb. */
829 
830 pull_pages:
831         eat = delta;
832         k = 0;
833         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
834                 if (skb_shinfo(skb)->frags[i].size <= eat) {
```

```
835                         put_page(skb_shinfo(skb)->frags[i].page);
836                         eat -= skb_shinfo(skb)->frags[i].size;
837                 } else {
838                         skb_shinfo(skb)->frags[k] = skb_shinfo(skb)->frags[i];
839                         if (eat) {
840                                 skb_shinfo(skb)->frags[k].page_offset += eat;
841                                 skb_shinfo(skb)->frags[k].size -= eat;
842                                 eat = 0;
843                         }
844                         k++;
845                 }
846         }
847         skb_shinfo(skb)->nr_frags = k;
848
849         skb->tail     += delta;
850         skb->data_len -= delta;
851
852         return skb->tail;
853 }
```

13.6.1.3 Extension de l'espace de tête

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

```
499 /**
500  *      pskb_expand_head - realloue l'en-tete d'un &sk_buff
501  *      @skb : tampon a reallouer
502  *      @nhead : place a ajouter en tete
503  *      @ntail : place a ajouter a la fin
504  *      @gfp_mask : priorite d'allocation
505  *
506  *      Etend (ou cree une copie identique, si &nhead et &ntail sont nuls)
507  *      un en-tete de skb. &sk_buff lui-meme n'est pas change. &sk_buff DOIT avoir
508  *      son compteur de reference a 1. Renvoie zero en cas de succes, ou une erreur
509  *      si l'extension echoue. Dans ce dernier cas, &sk_buff n'est pas change.
510  *
511  *      Tous les pointeurs pointant sur l'en-tete de skb peuvent changer et doivent
512  *      donc etre charges a nouveau apres appel de cette fonction.
513  */
514
515 int pskb_expand_head(struct sk_buff *skb, int nhead, int ntail, int gfp_mask)
516 {
517         int i;
518         u8 *data;
519         int size = nhead + (skb->end - skb->head) + ntail;
520         long off;
521
522         if (skb_shared(skb))
523                 BUG();
524
525         size = SKB_DATA_ALIGN(size);
526
527         data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
528         if (!data)
529                 goto nodata;
530
531         /* Copier seulement les donnees effectives... et, helas, l'en-tete. Ceci devrait
532          * etre optimise dans les cas ou l'en-tete est vide. */
533         memcpy(data + nhead, skb->head, skb->tail - skb->head);
534         memcpy(data + size, skb->end, sizeof(struct skb_shared_info));
535
536         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
```

```
537                 get_page(skb_shinfo(skb)->frags[i].page);
538
539         if (skb_shinfo(skb)->frag_list)
540                 skb_clone_fraglist(skb);
541
542         skb_release_data(skb);
543
544         off = (data + nhead) - skb->head;
545
546         skb->head     = data;
547         skb->end      = data + size;
548         skb->data    += off;
549         skb->tail    += off;
550         skb->mac.raw += off;
551         skb->h.raw   += off;
552         skb->nh.raw  += off;
553         skb->cloned   = 0;
554         atomic_set(&skb_shinfo(skb)->dataref, 1);
555         return 0;
556
557 nodata:
558         return -ENOMEM;
559 }
```

Code Linux 2.6.10

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

```
180 static void skb_clone_fraglist(struct sk_buff *skb)
181 {
182         struct sk_buff *list;
183
184         for (list = skb_shinfo(skb)->frag_list; list; list = list->next)
185                 skb_get(list);
186 }
```

13.6.1.4 Réallocation

La fonction `skb_realloc_headroom(skb, newheadroom)` permet de générer un nouveau tampon de socket dont l'espace de tête a maintenant la taille `newheadroom`. La portion de données de l'ancien tampon est copiée dans le nouveau et la plupart des champs du descripteur sont pris en compte. Seuls les champs `sk` et `list` sont définis à `NULL`.

Code Linux 2.6.10

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

```
561 /* Fait une copie privee de skb avec une tete sur laquelle on peut ecrire
       et un peu d'espace de queue */
562
563 struct sk_buff *skb_realloc_headroom(struct sk_buff *skb, unsigned int headroom)
564 {
565         struct sk_buff *skb2;
566         int delta = headroom - skb_headroom(skb);
567
568         if (delta <= 0)
569                 skb2 = pskb_copy(skb, GFP_ATOMIC);
570         else {
571                 skb2 = skb_clone(skb, GFP_ATOMIC);
572                 if (skb2 && pskb_expand_head(skb2, SKB_DATA_ALIGN(delta), 0,
573                                              GFP_ATOMIC)) {
574                         kfree_skb(skb2);
575                         skb2 = NULL;
576                 }
577         }
578         return skb2;
579 }
```

13.6.2 Retrait à la fin

13.6.2.1 Fonction d'appel

La fonction `skb_trim(skb, len)` taille (*to trim* en anglais) la fin de la zone des données d'un tampon pour que celle-ci fasse `len` octets. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
853 extern int ___pskb_trim(struct sk_buff *skb, unsigned int len, int realloc);
854
855 static inline void __skb_trim(struct sk_buff *skb, unsigned int len)
856 {
857         if (!skb->data_len) {
858                 skb->len  = len;
859                 skb->tail = skb->data + len;
860         } else
861                 ___pskb_trim(skb, len, 0);
862 }
863
864 /**
865  *      skb_trim - efface la fin d'un tampon
866  *      @skb : tampon a alterer
867  *      @len : nouvelle longueur
868  *
869  *      Coupe la longueur d'un tampon en effacant la fin des donnees. Si
870  *      le tampon est deja en-dessous de la longueur specifiee, il n'est pas modifie.
871  */
872 static inline void skb_trim(struct sk_buff *skb, unsigned int len)
873 {
874         if (skb->len > len)
875                 __skb_trim(skb, len);
876 }
877
878
879 static inline int __pskb_trim(struct sk_buff *skb, unsigned int len)
880 {
881         if (!skb->data_len) {
882                 skb->len  = len;
883                 skb->tail = skb->data+len;
884                 return 0;
885         }
886         return ___pskb_trim(skb, len, 1);
887 }
888
889 static inline int pskb_trim(struct sk_buff *skb, unsigned int len)
890 {
891         return (len < skb->len) ? __pskb_trim(skb, len) : 0;
892 }
```

13.6.2.2 Fonction interne

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

Code Linux 2.6.10

```
667 /* Taille skb a la longueur len. On peut changer les pointeurs de skb si "realloc" est 1.
668  * Si realloc==0 et que la taille est impossible sans changer les donnees,
669  * c'est BUG().
670  */
671
672 int ___pskb_trim(struct sk_buff *skb, unsigned int len, int realloc)
673 {
674         int offset = skb_headlen(skb);
675         int nfrags = skb_shinfo(skb)->nr_frags;
```

```
676         int i;
677
678         for (i = 0; i < nfrags; i++) {
679                 int end = offset + skb_shinfo(skb)->frags[i].size;
680                 if (end > len) {
681                         if (skb_cloned(skb)) {
682                                 if (!realloc)
683                                         BUG();
684                                 if (pskb_expand_head(skb, 0, 0, GFP_ATOMIC))
685                                         return -ENOMEM;
686                         }
687                         if (len <= offset) {
688                                 put_page(skb_shinfo(skb)->frags[i].page);
689                                 skb_shinfo(skb)->nr_frags--;
690                         } else {
691                                 skb_shinfo(skb)->frags[i].size = len - offset;
692                         }
693                 }
694                 offset = end;
695         }
696
697         if (offset < len) {
698                 skb->data_len -= skb->len - len;
699                 skb->len       = len;
700         } else {
701                 if (len <= skb_headlen(skb)) {
702                         skb->len      = len;
703                         skb->data_len = 0;
704                         skb->tail     = skb->data + len;
705                         if (skb_shinfo(skb)->frag_list && !skb_cloned(skb))
706                                 skb_drop_fraglist(skb);
707                 } else {
708                         skb->data_len -= skb->len - len;
709                         skb->len       = len;
710                 }
711         }
712
713         return 0;
714 }
```

13.6.3 Extension de l'espace de tête

La fonction `skb_cow(skb, headroom)` vérifie si le tampon possède au moins `headroom` octets dans l'espace de tête et s'il s'agit d'un clone. Si une des deux situations est réalisée, un nouveau tampon est généré. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
964 /**
965  *      skb_cow - copie l'en-tete de skb lorsque c'est necessaire
966  *      @skb : tampon a intimider
967  *      @headroom : headroom necessaire
968  *
969  *      Si le skb passe manque de suffisamment de headroom ou si sa partie donnees
970  *      est partagee, les donnees sont reallouees. Si la reallocation echoue, une erreur
971  *      est renvoyee et le skb originel n'est pas change.
972  *
973  *      Le resultat est un skb avec une zone skb->head...skb->tail sur laquelle on
974  *      peut ecrire et au moins @headroom d'espace de tete.
975  */
976 static inline int skb_cow(struct sk_buff *skb, unsigned int headroom)
977 {
978         int delta = (headroom > 16 ? headroom : 16) - skb_headroom(skb);
979
```

```
980         if (delta < 0)
981                 delta = 0;
982
983         if (delta || skb_cloned(skb))
984                 return pskb_expand_head(skb, (delta + 15) & ~15, 0, GFP_ATOMIC);
985         return 0;
986 }
```

13.7 Allocation d'un tampon de socket pour l'émission

La fonction en ligne `dev_alloc_skb(length)` génère un tampon de socket dont l'espace de tête a une taille de 16 octets et l'espace de queue une taille de `length` octets.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
926 /**
927  *      __dev_alloc_skb - alloue un skbuff pour l'emission
928  *      @length : longueur a allouer
929  *      @gfp_mask : masque get_free_pages, passe a alloc_skb
930  *
931  *      Alloue un nouveau &sk_buff et lui assigne un compteur d'utilisation de un. Le
932  *      tampon a un espace de tete non specifie. Les utilisateurs doivent allouer
933  *      l'espace de tete qu'ils pensent necessaire sans comptabiliser l'espace de
934  *      construction. L'espace de construction est utilise pour des optimisations.
935  *
936  *      %NULL est renvoye s'il n'y a pas d'emplacement memoire libre.
937  */
938 static inline struct sk_buff *__dev_alloc_skb(unsigned int length,
939                                               int gfp_mask)
940 {
941         struct sk_buff *skb = alloc_skb(length + 16, gfp_mask);
942         if (likely(skb))
943                 skb_reserve(skb, 16);
944         return skb;
945 }
946
947 /**
948  *      dev_alloc_skb - alloue un skbuff pour l'emission
949  *      @length : longueur a allouer
950  *
951  *      Alloue un nouveau &sk_buff et lui assigne un compteur d'utilisation de un. Le
952  *      tampon a un headroom non specifie. Les utilisateurs doivent allouer
953  *      le headroom qu'ils pensent necessaire sans comptabiliser l'espace de
954  *      construction. L'espace de construction est utilise pour des optimisations.
955  *
956  *      %NULL est renvoye s'il n'y a pas d'emplacement memoire libre. Bien que cette
957  *      fonction alloue de la memoire, elle peut etre appelee depuis une interruption.
958  */
959 static inline struct sk_buff *dev_alloc_skb(unsigned int length)
960 {
961         return __dev_alloc_skb(length, GFP_ATOMIC);
962 }
```