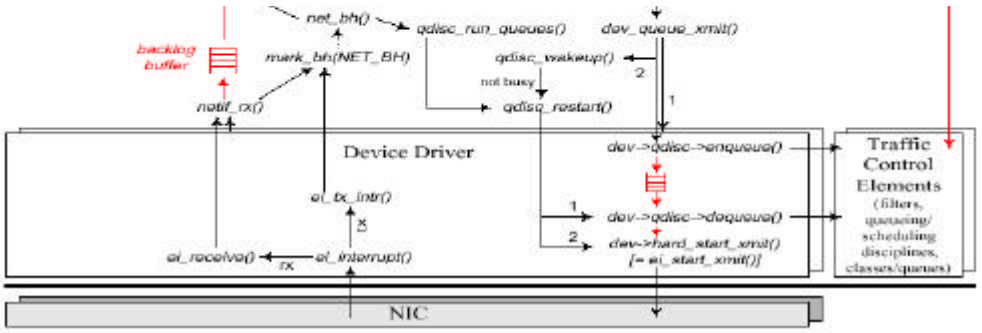
Linux Networking Kernel

Part II

- device driver -



1. Linux Networking Overview

1.1 Introduction

리눅스 네트워킹 레이어는 아래 그림과 같은 구조로 되어 있다. 각 네트워크 카드 별로 Device driver가 있으며, 네트워크 카드에 독립적인, 즉 공통부분이 되는 device Independence Routine이 있다. 그 위쪽으로 네트워크 프로토콜 계층인 IP계층이 있다. 그 위에는 UDP, TCP와 같은 Transport layer가 있다. 계속해서 INET socket layer가 있으며, BSD socket layer가 그 위에 올라가 있다. 이 BSD socket layer를 통하여 Application process와 커널 내의 네트워킹 코드 간에 데이터 송수신 및 제어를 하게 된다.

리눅스 네트워크 소스 코드는 보통 /usr/src/linux/net이하에 위치하며, 네트워크 드라이버는 /usr/src/linux/drivers/net이하에 위치한다. 네트워크 소스 코드의 위치는 대략 다음과 같다. 세부적인 내용에 대하여는 나중에 각 부분을 다룰 때 자세히 알아보도록 하자.

```
silmaril:/usr/src/linux/net# ls
802/          appletalk/   ethernet/    netlink/     protocols.c     socket.o        x25/
Changes       ax25/        ipv4/        netrom/      protocols.o     sunrpc/
Config.in     bridge/      ipv6/        netsyms.c    rose/           sysctl_net.c
Makefile      core/        ipx/         netsyms.o    sched/          sysctl_net.o
README        decnet/      irda/        network.a    sock_n_syms.o   unix/
TUNABLE       econet/      lapb/        packet/      socket.c        wanrouter/

silmaril:/usr/src/linux/net/core# ls
Makefile      dev.o          filter.c       neighbour.o    scm.o        sysctl_net_core.c
core.o        dev_mcast.c    firewall.c     profile.c      skbuff.c     sysctl_net_core.o
datagram.c    dev_mcast.o    iovec.c        rtnetlink.c    skbuff.o     utils.c
datagram.o    dst.c          iovec.o        rtnetlink.o    sock.c       utils.o
dev.c         dst.o          neighbour.c    scm.c          sock.o
```

표 1.1

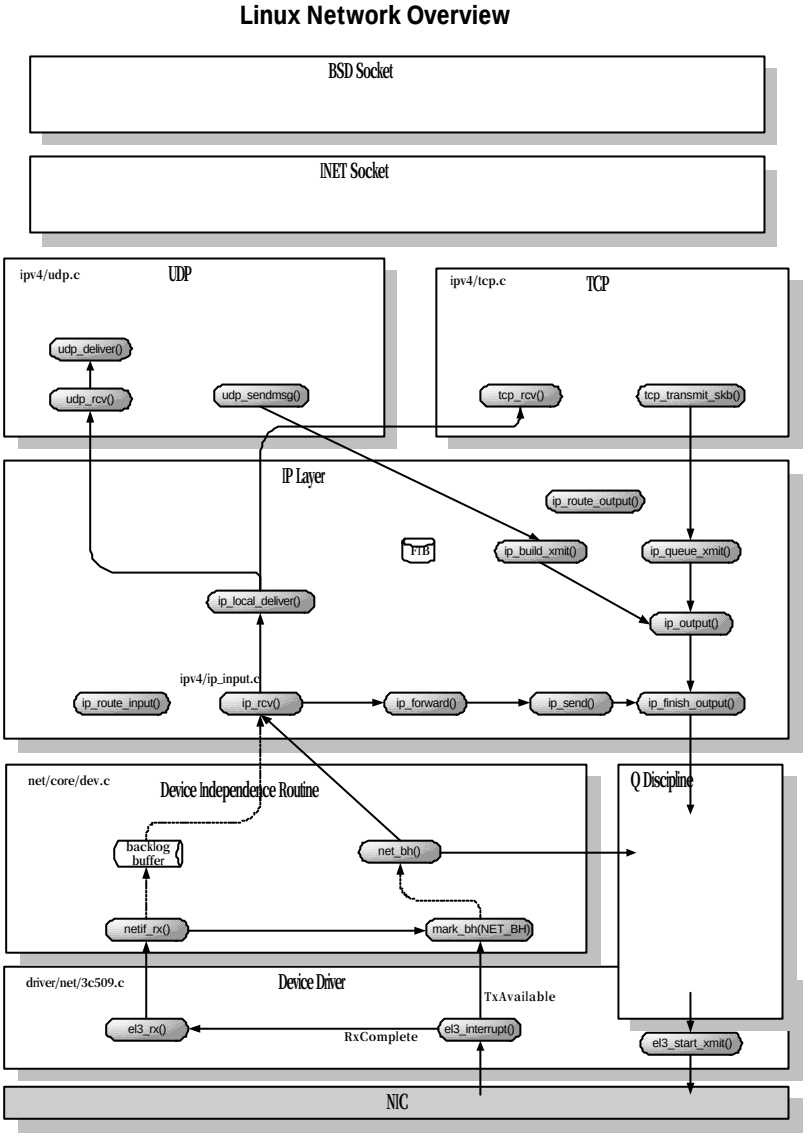


그림 1.1

자, 그럼 앞의 그림을 보며, 네트워크로부터 패킷을 수신하였을 때, 어떻게 동작하는지 그 흐름을 따라가 본 후, 패킷을 송신할 경우 어떻게 동작하는지 살펴보도록 하자.

1.2 데이터 수신 : Data Link Layer

이 보고서에서는 LAN카드로 3c509를 사용하였다. 네트워크 카드마다 driver가 있으

며, 이 driver의 소스코드는 linux/drivers/net/3c509.c 이다.
네트워크 카드는 데이터를 수신하면 인터럽트를 발생시킨다. 그러면 가장 먼저 el3_interrupt()함수가 수행된다. 그림 2.1의 하단부를 보면 el3_interrupt()함수가 그려져 있다. 이 함수에서는 어떤 인터럽트가 걸렸는지 확인하여, 만약에 데이터를 수신한 인터럽트(RxComplete)라면 el3_rx()함수를 호출한다. 또는 만약 데이터 송신이 가능한 경우(TxAvailable)에 의한 인터럽트라면 mark_bh(NET_BH)를 호출한다. mark_bh(NET_BH)함수는 나중에 설명하도록 하겠다.
아무튼 데이타수신시 el3_interrupt()는 el3_rx()함수를 호출한다. el3_rx()함수에서는 네트워크 카드로부터 데이터를 skbuff로 읽어온다. 그리고, netif_rx()함수를 호출하게 된다.
netif_rx()함수는 net/core/dev.c내에 있다. 이 함수는 받은 skbuff를 backlog라는 큐에 넣은후, mark_bh(NET_BH)를 호출한 후, 하드웨어 인터럽트는 종료된다.
그럼 mark_bh(NET_BH)함수가 어떤 기능을 하는지 잠깐 살펴보자. 리눅스에는 Bottom Half Handler라는 루틴이 있다. 이는 하드웨어 인터럽트가 걸렸을 때, 오랜 시간동안 이를 처리하고 있으면 안 되기 때문에, Bottom Half Handler라는 별도의 루틴을 두어 네트워킹의 나머지 부분을 처리하도록 한 것이다. 즉, 하드웨어 인터럽트는 아주 중요한 부분만 처리한 후 종료된다. 대신에 mark_bh(NET_BH)를 통하여 BH Handler가 나머지 부분을 처리하도록 하고 있는 것이다. mark_bh(NET_BH)로 마킹을 해두면, 스케쥴러는 net_bh()함수를 수행하도록 한다.
net_bh()함수는 처음에 보낼 패킷이 있는지 확인한후, 있으면 qdisc_run_queues()를 호출하여 전송하도록 한다. 다음에 backlog버퍼를 확인한 후, dequeue를 수행하여 skbuff로 가져 온다. skbuff에 나타나 있는 type값을 보고 상위계층의 수신 함수를 호출한다. 만약 IP라면, ip_rcv()를 호출한다.

1.3 데이터 수신 : IP Layer

ip_rcv()함수는 net/ipv4/ip_input.c에 있다. ip_rcv()함수는 먼저 수신된 패킷의 헤더를 검사한다. 다음에 패킷이 자신에게 온 것이라면, ip_local_deliver()를 호출한다. 만약 자기것이 아니라면, ip_forward()를 호출하게 된다.
ip_local_deliver()함수는 port넘버를 보고 상위 계층 프로토콜로 데이터를 넘겨준다. 만약 udp라면 udp_rcv()함수가 호출되고, tcp라면 tcp_rcv()함수가 호출된다.

1.4 데이터 수신 : UDP Layer

udp_rcv()함수는 net/ipv4/udp.c에 위치한다. udp_rcv()함수는 해당되는 소켓을 찾는

작업과 udp 체크섬을 수행한 후, udp_deliver()함수를 호출한다.
udp_deliver()함수는 소켓으로 데이터를 넘겨주는 역할을 한다.
Application은 이 데이터를 소켓을 통하여 받게 된다.

1.5 데이터 송신

이제 데이터 송신을 보자. UDP에서는 udp_sendmsg()함수가 socket구조체에 보내고자 하는 데이터를 담아 ip_build_xmit()함수를 호출한다.
 있다. 이 ip_build_xmit()함수는 net/ipv4/ip_output.c파일에 있다. 여기서 skbuff를 만들어 데이터를 넣는다. 그 다음, ip_output()를 호출한다.
ip_output()함수는 다시 ip_finish_out()함수를 호출한다. ip_finish_out()함수는 inline함수로 include/net/ip.h에 정의되어 있다. 이 함수는 해당 device에 맞는 output함수를 호출한다. 우리의 경우 dev_queue_xmit()이 호출된다.
dev_queue_xmit()함수는 net/core/dev.c에 정의되어 있다. 이 함수는 Queueing discipline에 따라 전송을 수행하여, 결국 el3_start_xmit()함수가 호출된다.
el3_start_xmit()함수는 drivers/net/3c509.c에 정의되어 있다. el3_start_xmit()함수는 skbuff구조체의 내용을 네트워크 카드로 전송한다.

2. skbuff Buffer

2.1 skbuff structure

리눅스 네트워크 커널에서는 데이터를 송수신하기 위하여 skbuff(소켓버퍼)라는 buffer를 사용한다. 이 skbuff는 여러 계층의 프로토콜에서 사용할 수 있도록 유연성을 가지고 있다. 즉, 각 프로토콜 계층에서 사용되는 헤더를 쉽게 붙이고 떼어낼수 있도록 설계되어 있다. skbuff는 하나 이상의 skbuff로 연결된 list형태를 가지게 된다. 이러한 skbuffer구조체는 linux/skbuff.h에 정의되어 있으며, 다음 그림과 같다.

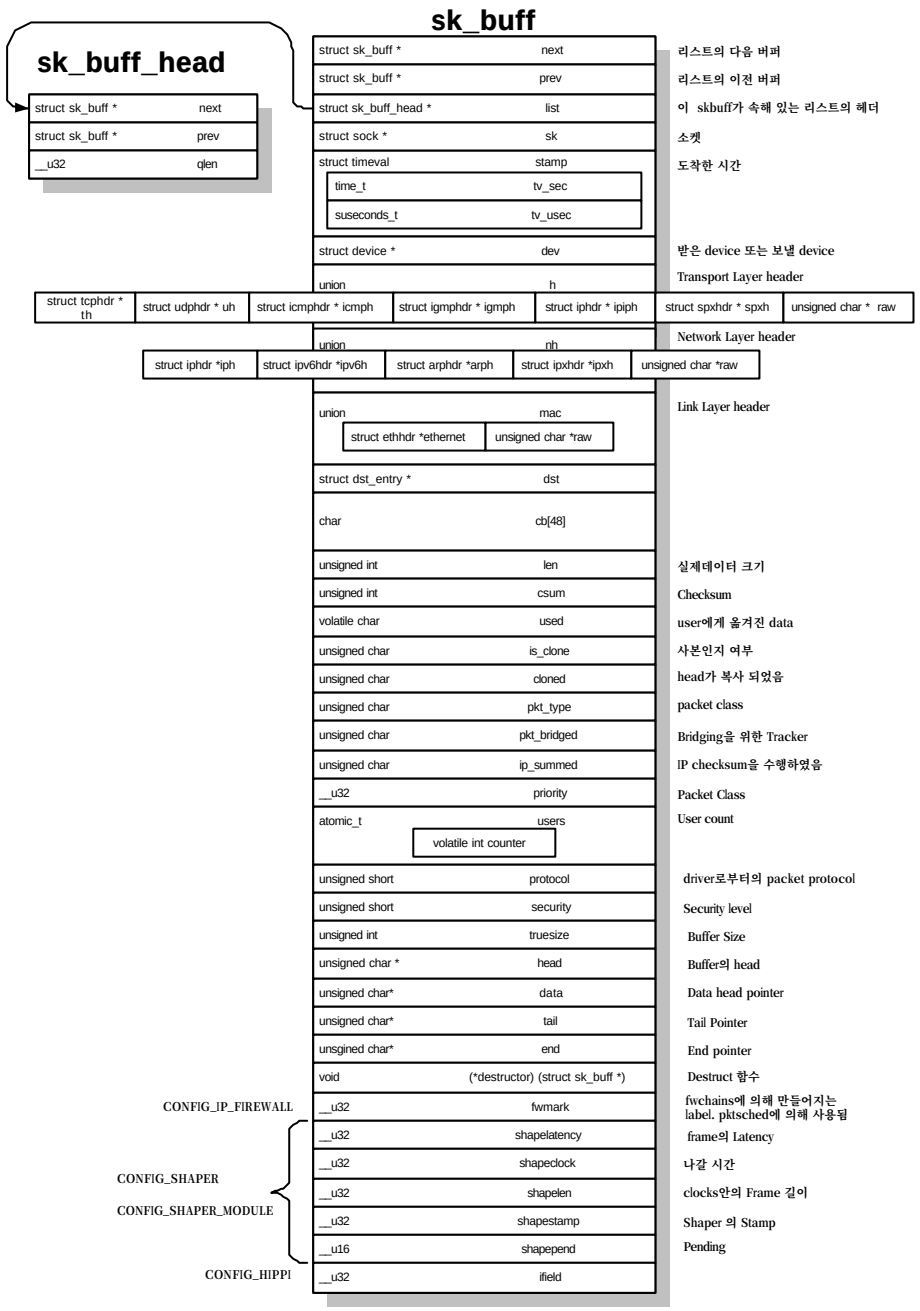


그림 2.1 skbuff structure

그림에서 보는 바와 같이 skbuff 구조체는 여러 가지 멤버 변수를 가지고 있다. 각 변수가 어떤 의미를 갖는지는 그림에 간단히 설명해 놓았지만, 몇가지는 정확하게 이해하기가 어려울 것이다. 그러한 것들은 skbuff를 사용하기 위한 함수 및 inline매크로를

살펴보면서 그 변수들의 의미를 이해하도록 하자. 여기서는 일단 몇 가지 변수만을 설명하겠다.

next와 prev는 각각 다음 skbuff와 이전 skbuff를 가리키는 포인터이다.
list는 이 skbuff의 header를 가리킨다.
len은 실제 데이터의 크기를 말한다.
truesize는 버퍼의 전체 크기를 말한다.
head는 버퍼의 처음을 가리키는 포인터이다.
data는 버퍼 안에 실제 데이터가 있는 처음 위치를 가리키는 포인터이다.
tail은 버퍼의 마지막을 가리키는 포인터이다.
end포인터는 실제 데이터의 끝을 가리킨다.

skbuff구조체의 예를 그림 2.2에서 보여준다.

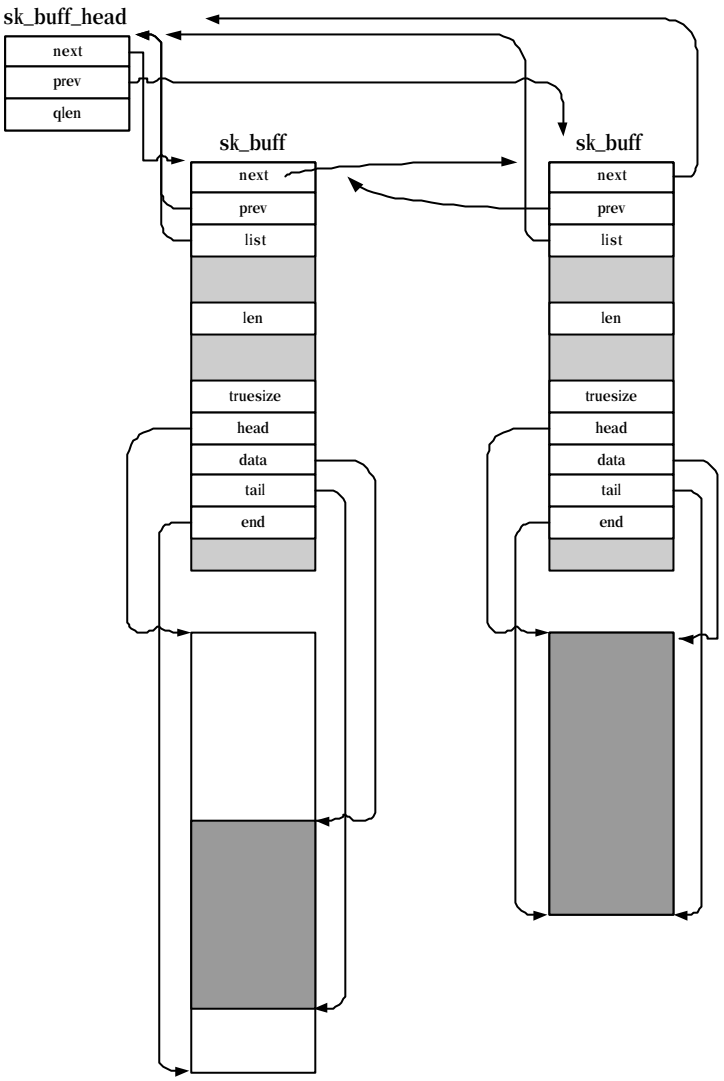


그림 2.2 skbuff example

2.2 alloc_skb 함수

alloc_skb()함수는 새로운 skbuff를 할당한다. 인수로서 버퍼의 크기 size와 마스크 gfp_mask를 사용한다. 이 함수는 net/core/skbuff.c에 정의되어 있으며, 그 흐름은 그림과 같다. 소스 코드를 보며 분석해 보도록 하자.

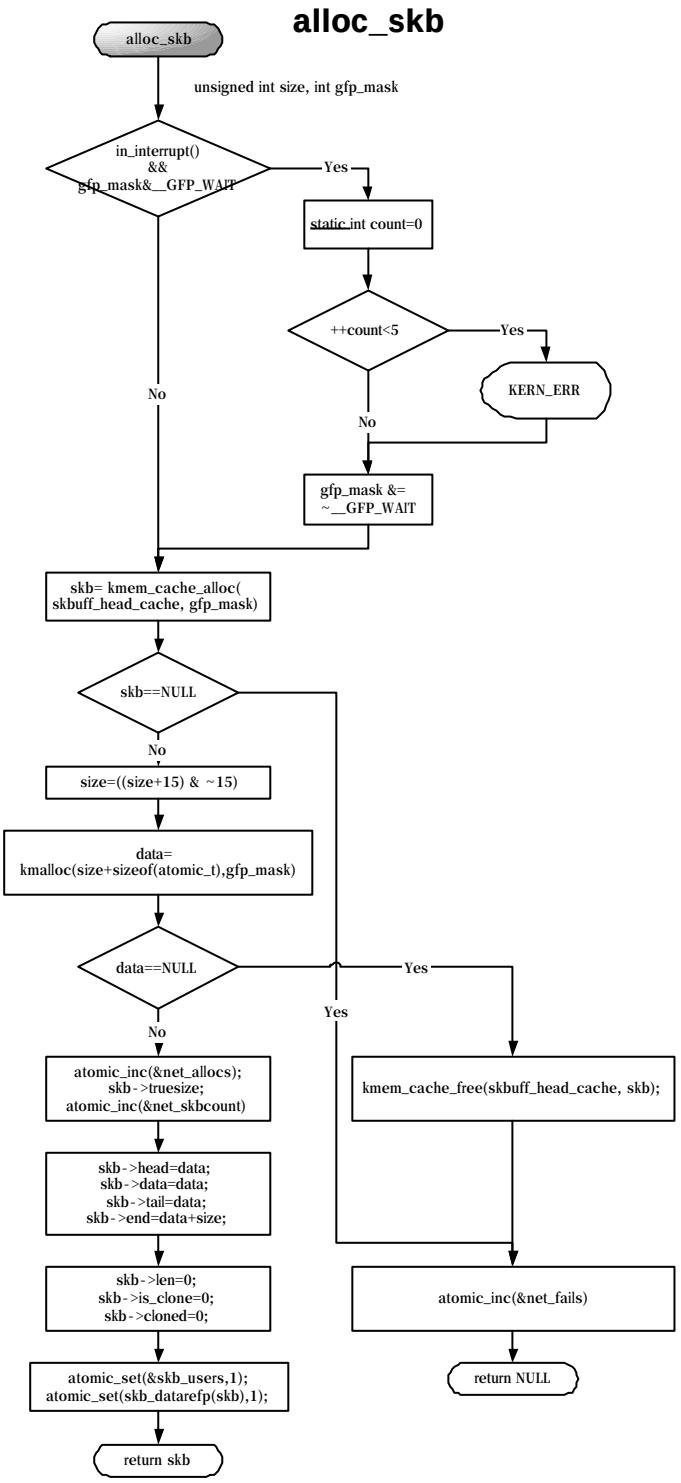

alloc_skb
alloc_skb
unsigned int size, int gfp_mask
in_interrupt()
&&
gfp_mask&__GFP_WAIT
Yes
static int count=0
++count<5
Yes
KERN_ERR
No
No
gfp_mask &=
~__GFP_WAIT
skb= kmem_cache_alloc(
skbuff_head_cache, gfp_mask)
skb==NULL
No
size=((size+15) & ~15)
data=
kmalloc(size+sizeof(atomic_t),gfp_mask)
data==NULL
Yes
Yes
No
atomic_inc(&net_allocs);
skb->truesize;
atomic_inc(&net_skbcount)
kmem_cache_free(skbuff_head_cache, skb);
skb->head=data;
skb->data=data;
skb->tail=data;
skb->end=data+size;
skb->len=0;
skb->is_clone=0;
skb->cloned=0;
atomic_inc(&net_fails)
atomic_set(&skb_users,1);
atomic_set(skb_datarefp(skb),1);
return NULL
return skb

그림 2.3

그러면 소스코드를 살펴보도록 하자.

```
119 struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)
120 {
121     struct sk_buff *skb;
122     u8 *data;
123
124     if (in_interrupt() && (gfp_mask & __GFP_WAIT)) {
125             static int count = 0;
126             if (++count < 5) {
127                     printk(KERN_ERR "alloc_skb called nonatomically "
128                             "from interrupt %p \n", __builtin_return_address(0));
129             }
130             gfp_mask &= ~__GFP_WAIT;
131     }
132
133     /* Get the HEAD */
134     skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
135     if (skb == NULL)
136             goto nohead;
```

124행부터 131줄까지는 인터럽트 상태에서 이 함수가 호출되었을 때, 커널에러를 표시해주는 부분이다. 처음 4회까지는 이 커널 에러를 표시해 준다. 그 이상의 경우는 표시하지 않는다. in_interrupt()매크로함수는 include/asm-i386/hardirq.h에 정의 되어 있으며 다음과 같다.

```
#define in_interrupt() ({ int __cpu = smp_processor_id(); \
        (local_irq_count[__cpu] + local_bh_count[__cpu] != 0); })
```

결국 이 코드가 인터럽트 상태에 있으면 in_interrupt()가 true가 된다. 이때, gfp_mask에 __GFP_WAIT이 세트되어 있으면 커널에러를 표시해준다.

134행을 주목하라. skbuff_head_cache라는 변수는 skbuff.c에 선언된 전역변수로 kmem_cache_t 포인터이다. kmem_cache_t 자료형은 include/linux/slab.h에 struct kmem_cache_s형이고 이 구조체는 mm/slab.c에 정의되어 있고, 커널캐시를 관리하는데 사용되는 구조체이다. 이 skbuff_head_cache는 skb_init()함수에서 초기화되어 있다. 다음은 skb_init()함수이다. 379행을 보면 캐시공간을 struct sk_buff의 크기로 잡고 있다.

```
376 void __init skb_init(void)
377 {
378             skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
```

```
379                                             sizeof(struct sk_buff),
380                                             0,
381                                             SLAB_HWCACHE_ALIGN,
382                                             skb_headerinit, NULL);
383         if (!skbuff_head_cache)
384                 panic("cannot create skbuff cache");
385 }
```

kmem_cache_alloc은 커널내에 캐시공간을 할당하는 함수로, mm/slab.c에 정의되어 있다. 결국 커널캐시공간을 잡아 sk_buff포인터인 skb에 넘겨준다.

```
137
138   /* Get the DATA. Size must match skb_add_mtu(). */
139   size = ((size + 15) & ~15);
```

데이터의 크기를 조정한다. 139행은 하위 4비트를 0으로 만드는 것으로 offset값을 0으로 조정하는 개념으로 생각하면 된다. 만약 size의 하위 4비트가 0이 아닌 값인 경우 자리올림이 생겨난다. 결국 10진수 개념으로 볼 때 size값은 16의 배수가 된다. 예를 들어 size가 17였다면 32가 될 것이고, 1에서 16사이의 값이었다면 16이 될 것이다.

```
140   data = kmalloc(size + sizeof(atomic_t), gfp_mask);
141   if (data == NULL)
142           goto nodata;
143
```

140행은 data공간을 확보하는 것이다. 이때 atomic_t에 대한 공간을 추가적으로 잡아주고 있다. atomic_t 자료형은 include/asm-i386/atomic.h에 정의되어 있으며, 다음과 같다.

```
22 #ifdef __SMP__
23 typedef struct { volatile int counter; } atomic_t;
24 #else
25 typedef struct { int counter; } atomic_t;
26 #endif
```

이 atomic은 메모리를 독립적으로 사용하고자 하는데 사용된다. 즉 여러 프로세스가 이 메모리에 접근할 수 없도록 하는 방법중에 하나이다. 아무튼 원하는 데이터공간에 추가적으로 레퍼런스를 위한 공간을 확보하고 있다.

```
144   /* Note that this counter is useless now - you can just look in the
145    * skbuff_head entry in /proc/slabinfo. We keep it only for emergency
```

```
146      * cases.
147      */
148     atomic_inc(&net_allocs);
149
```

atomic_inc는 독립적으로 값을 하나 증가 시키는 역할을 한다. net_allocs은 atomic_t 형의 전역변수로 net \ core \ skbuff.c에 정의되어 있다. net_allocs는 일종의 통계변수로 할당시마다 그 값이 증가한다. 즉, alloc_skb()함수나 나중에 나오는 skb_clone()함수호출시마다 증가한다.

```
150     skb->truesize = size;
151
152     atomic_inc(&net_skbcount);
153
154     /* Load the data pointers. */
155     skb->head = data;
156     skb->data = data;
157     skb->tail = data;
158     skb->end = data + size;
159
```

150행은 skb의 멤버인 truesize에 우리가 할당한 데이터공간의 크기를 대입하고 있다. 152행은 net_skbcount를 증가시키는데, 이 net_skbcount는 sk_buff할당시 증가하고, kfree_skb()에서 다시 감소한다. 즉, 현재 할당되어 있는 sk_buff수를 유지하고 있다. 155행부터는 skb구조체의 head,data,tail,end에 알맞는 값을 넣어 주고 있는 것이다. 아직까지 데이터공간만 확보하였고, 실제 데이터는 들어있지 않으므로, tail포인터는 데이터의 처음을 가리키고 있다.

```
160     /* Set up other state */
161     skb->len = 0;
162     skb->is_clone = 0;
163     skb->cloned = 0;
164
165     atomic_set(&skb->users, 1);
166     atomic_set(skb_datarefp(skb), 1);
167     return skb;
```

실제 데이터가 아직 없으므로 len=0으로 한다. is_clone은 이 skbuffer가 사본인지 여

부를 나타내는 것으로 지금은 0으로 세트된다. cloned는 복사된 것인지를 나타내는 것으로 역시 0으로 세트된다. 165행은 skbuffer의 users변수를 1로 초기화 하는 것이다. 이는 skbuffer를 사용할 때 증가하였다가, 사용이 끝나면 감소시키게 된다. skb_datarefp(skb)는 데이터공간 바로 다음을 가리킨다. 이 인라인함수는 skbuff.h에 선언되어 있으며 다음과 같다.

```
extern __inline__ atomic_t *skb_datarefp(struct sk_buff *skb)
{
        return (atomic_t *)(skb->end);
}
```

그래서 skb_datarefp(skb)함수는, 140행에서 atomic_t공간을 더 확보한 그 공간을 가리키게 되고, 여기를 1로 초기화 하는 것이다. 167행에서 skb를 반환하며 이 함수를 종료한다.

```
168
169 nodata:
170     kmem_cache_free(skbuff_head_cache, skb);
```

이 부분은 140행에서 메모리 할당에 실패하였을 때, 앞에서 확보한 캐시공간을 해제하는 부분이다.

```
171 nohead:
172     atomic_inc(&net_fails);
173     return NULL;
174 }
```

실패한 수를 기록하고 있는 net_fails 전역변수를 증가시킨 후, 함수 수행이 종료된다. 이상으로 alloc_skb함수에 대해 살펴보았다. 계속해서 sk_buffer관련함수들을 살펴보도록 하자.

2.3 skb_clone 함수

계속해서 살펴볼 함수는 skb_clone함수이다. 이 함수는 실제 데이터 공간은 그대로 두고, skbuff구조체만 복사하는 기능을 한다. 그 흐름은 다음 그림과 같다.

이 함수의 소스코드는 net/core/skbuff.c에 있다. 그러면 소스코드를 분석해 보도록 하자.

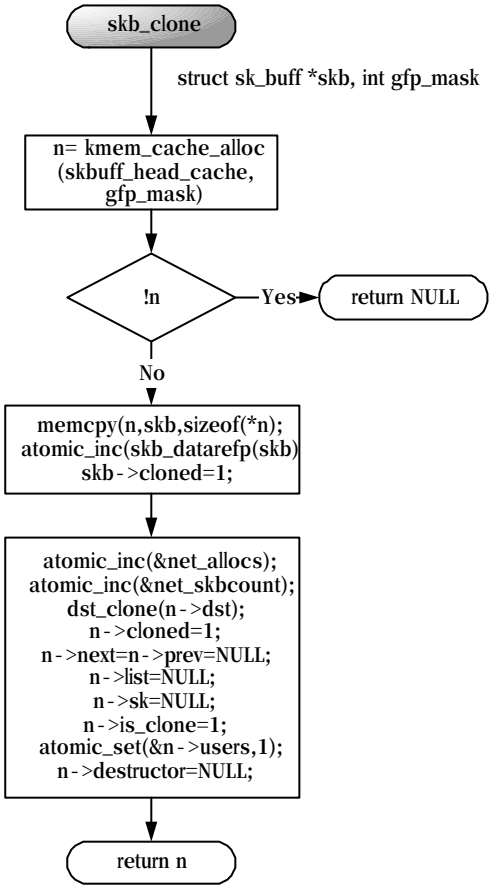


그림 2.4 skb_clone

```
235 struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask)
236 {
237     struct sk_buff *n;
238
239     n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
240     if (!n)
241             return NULL;
242
243     memcpy(n, skb, sizeof(*n));
244     atomic_inc(skb_datarefp(skb));
245     skb->cloned = 1;
246
247     atomic_inc(&net_allocs);
248     atomic_inc(&net_skbcount);
```

239행에서는 새로운 skbuffer를 위한 캐시메모리 공간을 할당하고 있다. 243행에서는 sizeof(*n)은 sizeof(sk_buff)가 되므로, 새로운 sk_buff n에 원래의 sk_buff를 복사하고 있다. 244행에서는 해당버퍼에 reference를 증가시키고 있다.

```
249     dst_clone(n->dst);
```

249행에서는 include/net/dst.h에 보면 dst_clone이 정의되어 있다.

```
83 extern __inline__
84 struct dst_entry * dst_clone(struct dst_entry * dst)
85 {
86         if (dst)
87                 atomic_inc(&dst->use);
88         return dst;
89 }
```

결국 dst엔트리의 use라는 변수값을 증가시키고 있다. dst_entry는 나중에 살펴보도록 하자.

```
250     n->cloned = 1;
251     n->next = n->prev = NULL;
252     n->list = NULL;
253     n->sk = NULL;
254     n->is_clone = 1;
255     atomic_set(&n->users, 1);
256     n->destructor = NULL;
257     return n;
258 }
259
```

skbuffer가 clone되었으므로 새로운 skbuffer n과 skb 모두 cloned가 1로 된다. 251행과 252행에서 기존의 list로부터 떨어져 나온다. 254행에서 n은 사본이므로 is_clone변수를 1로 세트해준다. 257행에서 새로운 skbuffer n을 반환한다. 그림에서는 clone함수의 수행 결과를 간략히 보여주고 있다.

2.4 skb_copy 함수

skb_copy 함수는 skbuff 구조체뿐만 아니라 실제 데이터까지 복사하는 함수이다. 이 함수의 수행결과를 보면 그림과 같다.

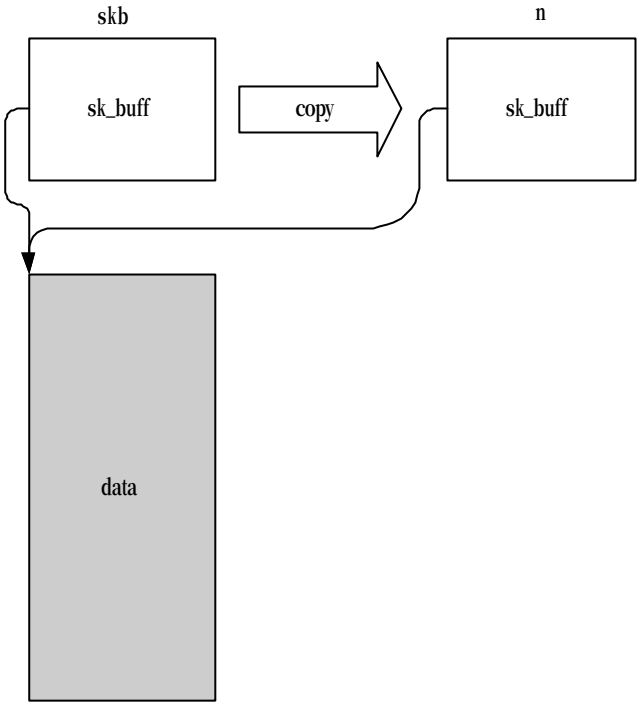


그림 3.7 clone의 결과

```
264 struct sk_buff *skb_copy(struct sk_buff *skb, int gfp_mask)
265 {
266     struct sk_buff *n;
267     unsigned long offset;
272
273     n=alloc_skb(skb->end - skb->head, gfp_mask);
274     if(n==NULL)
275             return NULL;
276
281     offset=n->head-skb->head;
```

273행에서 sk_buff 공간을 확보하고, 281행에서 새로운 데이터 공간과 원래 데이터 공간의 주소값의 차이값을 계산하고 있다.

```
282
283     /* Set the data pointer */
284     skb_reserve(n,skb->data-skb->head);
```

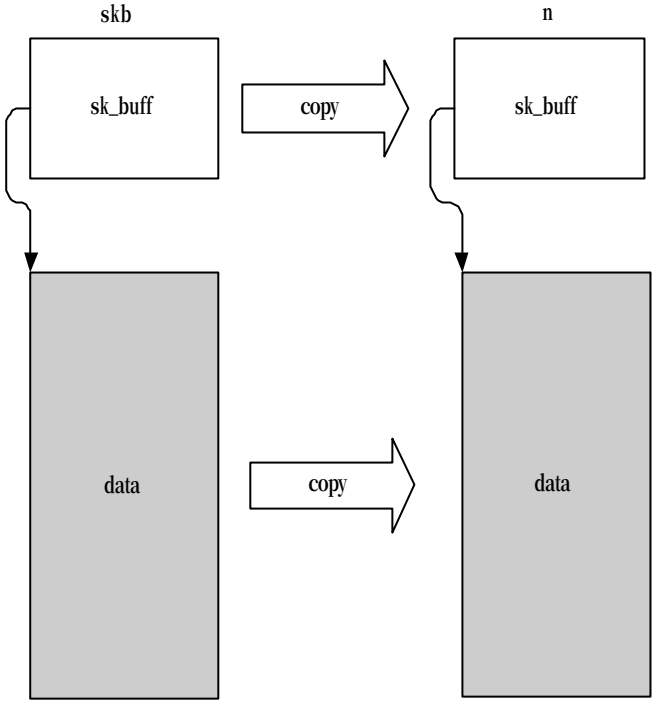


그림 3.8 copy의 결과

skb_reserve함수는 /include/linux/skbuff.h에 정의되어 있으며 data와 tail포인터의 값을 len값 만큼 늘려주는 기능을 한다.

```
517 extern __inline__ void skb_reserve(struct sk_buff *skb, unsigned int len)
518 {
519             skb->data+=len;
520             skb->tail+=len;
521 }
```

이 경우 len은 원래 skb의 데이터공간중 실제데이터의 앞부분 공간의 크기이므로, n에서도 그러한 공간을 잡아준다.

```
285     /* Set the tail pointer and length */
286     skb_put(n,skb->len);
```

skb_put함수는 skbuff.h에 정의되어 있으며 다음과 같다.

```
extern __inline__ unsigned char *__skb_put(struct sk_buff *skb, unsigned int len)
{
        unsigned char *tmp=skb->tail;
        skb->tail+=len;
        skb->len+=len;
```

```
        return tmp;
}
```

결국, 원래 skb의 실제 데이터 길이 만큼 새로운 skbuff n에서도 조정을 해준다.

```
287     /* Copy the bytes */
288     memcpy(n->head,skb->head,skb->end-skb->head);
```

원래의 skb의 데이터 버퍼 전체를 새로운 n의 데이터 버퍼로 복사한다.

```
289     n->csum = skb->csum;
290     n->list=NULL;
291     n->sk=NULL;
292     n->dev=skb->dev;
293     n->priority=skb->priority;
294     n->protocol=skb->protocol;
295     n->dst=dst_clone(skb->dst);
296     n->h.raw=skb->h.raw+offset;
297     n->nh.raw=skb->nh.raw+offset;
298     n->mac.raw=skb->mac.raw+offset;
299     memcpy(n->cb, skb->cb, sizeof(skb->cb));
300     n->used=skb->used;
301     n->is_clone=0;
302     atomic_set(&n->users, 1);
303     n->pkt_type=skb->pkt_type;
304     n->stamp=skb->stamp;
305     n->destructor = NULL;
306     n->security=skb->security;
307 #ifdef CONFIG_IP_FIREWALL
308         n->fwmark = skb->fwmark;
309 #endif
310     return n;
311 }
```

나머지 skbuffer의 각종 변수들의 내용을 복사하고 있다. 296행부터 298행은 각 계층 헤더의 포인터인데, 새로운 데이터공간의 주소가 원래 데이터공간의 주소와 틀리므로, 이에 대한 차이값, 즉 281행에서 구한 offset을 주어 그 값을 조정한다.

2.5 skb_headerinit 함수

skb_headerinit함수는 skbuff의 멤버변수를 초기화 하는 함수이다.

```
180 static inline void skb_headerinit(void *p, kmem_cache_t *cache,
181                                     unsigned long flags)
182 {
183     struct sk_buff *skb = p;
184
185     skb->destructor = NULL;
186     skb->pkt_type = PACKET_HOST;        /* Default type */
187     skb->pkt_bridged = 0;               /* Not bridged */
188     skb->prev = skb->next = NULL;
189     skb->list = NULL;
190     skb->sk = NULL;
191     skb->stamp.tv_sec=0;        /* No idea about time */
192     skb->ip_summed = 0;
193     skb->security = 0;  /* By default packets are insecure */
194     skb->dst = NULL;
195 #ifdef CONFIG_IP_FIREWALL
196         skb->fwmark = 0;
197 #endif
198     memset(skb->cb, 0, sizeof(skb->cb));
199     skb->priority = 0;
200 }
```

186행의 pkt_type을 디폴트로서 PACKET_HOST를 치환하고 있다. PACKET_HOST는 include/linux/if_packet.h에 정의되어 있다. packet의 type은 아래와 같이 정의되어 있다.

```
#define PACKET_HOST             0       /* To us                */
#define PACKET_BROADCAST        1       /* To all               */
#define PACKET_MULTICAST        2       /* To group             */
#define PACKET_OTHERHOST        3       /* To someone else      */
#define PACKET_OUTGOING         4       /* Outgoing of any type */
/* These ones are invisible by user level */
#define PACKET_LOOPBACK         5       /* MC/BRD frame looped back */
#define PACKET_FASTROUTE        6       /* Fastrouted frame     */
```

2.6 kfree_skbmem 함수

kfree_skbmem 함수는 skbuffer를 해제 하는 함수이다. kree_skbmem함수는 상태를 지우지 않고 메모리를 해제하고, __kree_skbmem함수는 상태를 지운다.

```
202 /*
203  * Free an skbuff by memory without cleaning the state.
204  */
205 void kfree_skbmem(struct sk_buff *skb)
206 {
207     if (!skb->cloned || atomic_dec_and_test(skb_datarefp(skb)))
208             kfree(skb->head);
209
210     kmem_cache_free(skbuff_head_cache, skb);
211     atomic_dec(&net_skbcount);
212 }
```

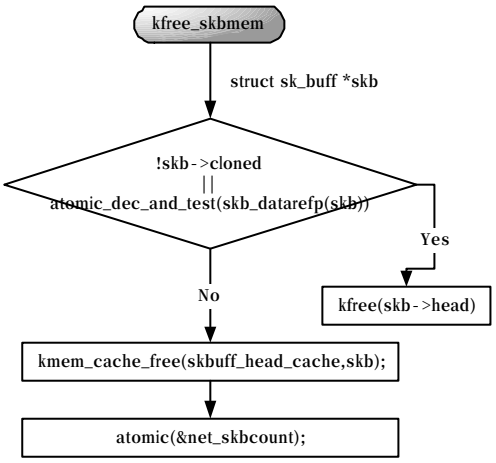


그림 2.9 kfree_skbmem

207행의 조건은, clone되지 않았거나, 데이터공간의 reference를 감소시켜 0이 된다면, 즉, 더 이상의 참조가 없다면, kfree()를 수행하여 데이터공간을 해제한다. skbuffer구조체를 해제하고, net_skbcount값을 감소시킨다.

계속해서 __kfree_skb함수를 보도록 하자

```
214 /*
215  * Free an sk_buff. Release anything attached to the buffer. Clean the state.
216  */
217
```

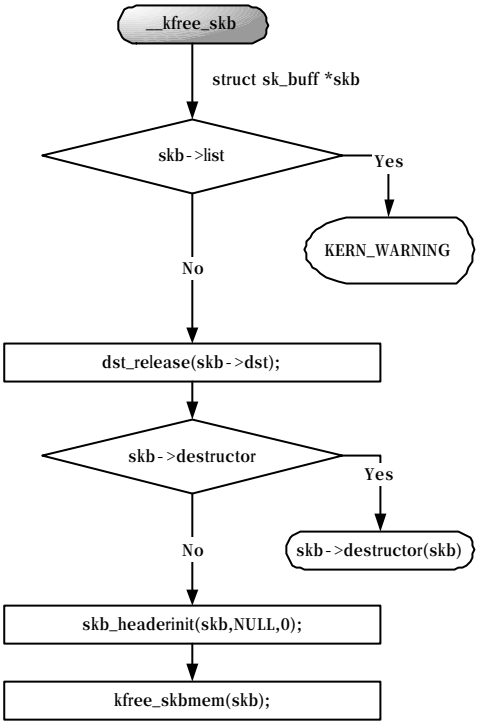


그림 2.10 __kfree_skb

```
218 void __kfree_skb(struct sk_buff *skb)
219 {
220     if (skb->list)
221             printk(KERN_WARNING "Warning: kfree_skb passed an skb still "
222                     "on a list (from %p).\n", __builtin_return_address(0));
223
224     dst_release(skb->dst);
225     if(skb->destructor)
226             skb->destructor(skb);
227     skb_headerinit(skb, NULL, 0);  /* clean state */
228     kfree_skbmem(skb);
229 }
```

__kfree_skb는 list가 해제 상태에서 호출되어야 한다. 그렇지 않을 경우, 커널 경고 메시지가 출력될 것이다. 224행의 dst_release()함수는 dst->use를 감소시키는 기능을 한다. 만약 특정한 destructor가 있으면 이를 호출한다. skb_headerinit()을 호출하여 skbuff의 멤버변수값을 초기화한후, kfree_skbmem()을 호출한다.

2.7 skb_realloc_headroom 함수

skb_realloc_headroom함수는 데이터공간 앞에 newheadroom이라는 빈공간을 잡아 주는 함수이다.

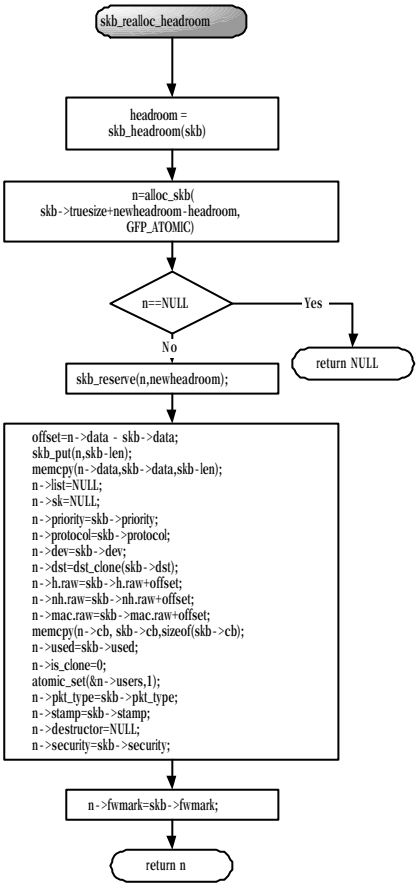


그림 2.11 skb_realloc_headroom

```
313 struct sk_buff *skb_realloc_headroom(struct sk_buff *skb, int newheadroom)
314 {
315     struct sk_buff *n;
316     unsigned long offset;
317     int headroom = skb_headroom(skb);
318
```

317행을 보면 skb_headroom()이 나온다. 이 인라인 함수는 skbuff.h에 정의되어 있

으며, 다음과 같다.

```
extern __inline__ int skb_headroom(struct sk_buff *skb)
{
    return skb->data-skb->head;
}
```

결국 head와 data사이의 공간이 headroom이 된다.

```
319     /*
320      *      Allocate the copy buffer
321      */
322
323     n=alloc_skb(skb->truesize+newheadroom-headroom, GFP_ATOMIC);
324     if(n==NULL)
325             return NULL;
```

newheadroom을 갖는 skbuffer를 할당한다.

```
326
327     skb_reserve(n,newheadroom);
333     offset=n->data-skb->data;
336     skb_put(n,skb->len);
338     memcpy(n->data,skb->data,skb->len);
339     n->list=NULL;
340     n->sk=NULL;
341     n->priority=skb->priority;
342     n->protocol=skb->protocol;
343     n->dev=skb->dev;
344     n->dst=dst_clone(skb->dst);
345     n->h.raw=skb->h.raw+offset;
346     n->nh.raw=skb->nh.raw+offset;
347     n->mac.raw=skb->mac.raw+offset;
348     memcpy(n->cb, skb->cb, sizeof(skb->cb));
349     n->used=skb->used;
350     n->is_clone=0;
351     atomic_set(&n->users, 1);
352     n->pkt_type=skb->pkt_type;
353     n->stamp=skb->stamp;
354     n->destructor = NULL;
355     n->security=skb->security;
```

```
356 #ifdef CONFIG_IP_FIREWALL
357         n->fwmark = skb->fwmark;
358 #endif
359
360     return n;
361 }
```

새로 할당한 skbuffer n에 원래 데이터를 복사한다.

2.8 기타 함수

다음 3개의 함수들은 skbuff.c에 정의된 함수이다.

```
 87 void skb_over_panic(struct sk_buff *skb, int sz, void *here)
 88 {
 89     panic("skput:over: %p:%d put:%d dev:%s",
 90             here, skb->len, sz, skb->dev ? skb->dev->name : "<NULL>");
 91 }
 92
 93 void skb_under_panic(struct sk_buff *skb, int sz, void *here)
 94 {
 95         panic("skput:under: %p:%d put:%d dev:%s",
 96                 here, skb->len, sz, skb->dev ? skb->dev->name : "<NULL>");
 97 }
 98
 99 void show_net_buffers(void)
100 {
101     printk("Networking buffers in use          : %u \ n",
102             atomic_read(&net_skbcount));
103     printk("Total network buffer allocations   : %u \ n",
104             atomic_read(&net_allocs));
105     printk("Total failed network buffer allocs : %u \ n",
106             atomic_read(&net_fails));
107 #ifdef CONFIG_INET
108     printk("IP fragment buffer size            : %u \ n",
109             atomic_read(&ip_frag_mem));
110 #endif
111 }
```

2.9 기타 매크로 함수

이번 절에서 설명되는 매크로 함수들은 모두 skbuff.h파일에 정의되어 있다.

```
164 extern __inline__ atomic_t *skb_datarefp(struct sk_buff *skb)
165 {
166     return (atomic_t *)(skb->end);
167 }
```

데이터의 끝부분을 나타낸다. skbuffer에서 데이터공간을 할당할 때, 마지막에 reference를 위한 공간을 함께 확보한다. 이 함수는 그 부분을 가리키게 된다.

```
169 extern __inline__ int skb_queue_empty(struct sk_buff_head *list)
170 {
171     return (list->next == (struct sk_buff *) list);
172 }
```

skb리스트인 sk_buff_head list의 next포인터가 자기자신을 가리키면, empty상태(=1)을 리턴한다.

```
174 extern __inline__ void kfree_skb(struct sk_buff *skb)
175 {
176     if (atomic_dec_and_test(&skb->users))
177                 __kfree_skb(skb);
178 }
```

skb의 users를 감소시킨후 검사하여 0이 되면 __kfree_skb()를 호출한다. __kfree_skb()함수는 앞에서 설명한 대로 상태를 지운후 메모리 해제를 수행한다.

```
180 /* Use this if you didn't touch the skb state [for fast switching] */
181 extern __inline__ void kfree_skb_fast(struct sk_buff *skb)
182 {
183     if (atomic_dec_and_test(&skb->users))
184                 kfree_skbmem(skb);
185 }
```

skb의 users를 감소시킨후 검사하여 0이 되면, 상태를 지우지 않고 메모리를 해제하는 kfree_skbmem()함수를 호출한다.

```
187 extern __inline__ int skb_cloned(struct sk_buff *skb)
188 {
189     return skb->cloned && atomic_read(skb_datarefp(skb)) != 1;
190 }
```

skb가 clone되었는지 검사하는 함수

```
192 extern __inline__ int skb_shared(struct sk_buff *skb)
193 {
194     return (atomic_read(&skb->users) != 1);
195 }
```

skb버퍼의 데이터공간이 공유되고 있는지 여부를 검사하는 함수.

```
204 extern __inline__ struct sk_buff *skb_unshare(struct sk_buff *skb, int pri)
205 {
206     struct sk_buff *nskb;
207     if(!skb_cloned(skb))
208             return skb;
209     nskb=skb_copy(skb, pri);
210     kfree_skb(skb);             /* Free our shared copy */
211     return nskb;
212 }
```

clone된 skb를 copy한 것으로 바꾸어, 공유하지 않도록 하는 함수. tcpdump나 forward 등등에서 유효하다.

```
221 extern __inline__ struct sk_buff *skb_peek(struct sk_buff_head *list_)
222 {
223     struct sk_buff *list = ((struct sk_buff *)list_)->next;
224     if (list == (struct sk_buff *)list_)
225             list = NULL;
226     return list;
227 }
```

list의 next가 가리키는 sk_buff를 리턴한다. 즉 리스트의 맨처음 sk_buff를 리턴한다.

```
229 extern __inline__ struct sk_buff *skb_peek_tail(struct sk_buff_head *list_)
230 {
231     struct sk_buff *list = ((struct sk_buff *)list_)->prev;
232     if (list == (struct sk_buff *)list_)
233             list = NULL;
234     return list;
235 }
```

list의 맨 마지막 sk_buff를 리턴한다.

```
241 extern __inline__ __u32 skb_queue_len(struct sk_buff_head *list_)
242 {
243     return(list_->qlen);
```

```
244 }
```

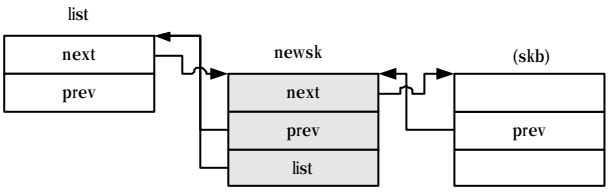
list의 길이를 리턴한다.

```
246 extern __inline__ void skb_queue_head_init(struct sk_buff_head *list)
247 {
248     list->prev = (struct sk_buff *)list;
249     list->next = (struct sk_buff *)list;
250     list->qlen = 0;
251 }
```

skbuffer list를 초기화 한다. list가 비어 있을 때, prev나 next값은 list자체를 가리킨다.

```
260 extern __inline__ void __skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)
261 {
262     struct sk_buff *prev, *next;
263
264     newsk->list = list;
265     list->qlen++;
266     prev = (struct sk_buff *)list;
267     next = prev->next;
268     newsk->next = next;
269     newsk->prev = prev;
270     next->prev = newsk;
271     prev->next = newsk;
272 }
```

skbuff list의 머리에 새로운 skbuff를 넣는 함수이다.



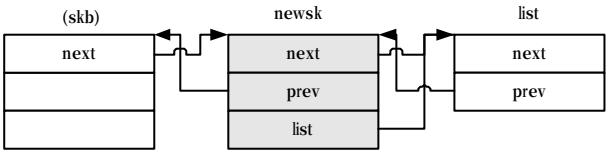

```
274 extern spinlock_t skb_queue_lock;
276 extern __inline__ void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)
277 {
278     unsigned long flags;
279
```

```
280     spin_lock_irqsave(&skb_queue_lock, flags);
281     __skb_queue_head(list, newsk);
282     spin_unlock_irqrestore(&skb_queue_lock, flags);
283 }
```

skb_queue_head는 __skb_queue_head와 같으나, lock을 걸어 atomic한 동작을 하도록 한다

```
289 extern __inline__ void __skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
290 {
291     struct sk_buff *prev, *next;
292
293     newsk->list = list;
294     list->qlen++;
295     next = (struct sk_buff *)list;
296     prev = next->prev;
297     newsk->next = next;
298     newsk->prev = prev;
299     next->prev = newsk;
300     prev->next = newsk;
301 }
```

리스트의 tail에 skbuff를 삽입하는 함수이다.




```
303 extern __inline__ void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
304 {
305     unsigned long flags;
306
307     spin_lock_irqsave(&skb_queue_lock, flags);
308     __skb_queue_tail(list, newsk);
309     spin_unlock_irqrestore(&skb_queue_lock, flags);
310 }
```

__skb_queue_tail함수의 atomic버전이다.

```
316 extern __inline__ struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
317 {
```

```
318     struct sk_buff *next, *prev, *result;
319
320     prev = (struct sk_buff *) list;
321     next = prev->next;
322     result = NULL;
323     if (next != prev) {
324             result = next;
325             next = next->next;
326             list->qlen--;
327             next->prev = prev;
328             prev->next = next;
329             result->next = NULL;
330             result->prev = NULL;
331             result->list = NULL;
332     }
333     return result;
334 }
```

list큐로부터 head부분의 skbuff를 꺼낸다.

```
336 extern __inline__ struct sk_buff *skb_dequeue(struct sk_buff_head *list)
337 {
338     long flags;
339     struct sk_buff *result;
340
341     spin_lock_irqsave(&skb_queue_lock, flags);
342     result = __skb_dequeue(list);
343     spin_unlock_irqrestore(&skb_queue_lock, flags);
344     return result;
345 }
```

__skb_dequeue()의 atomic버젼이다.

```
351 extern __inline__ void __skb_insert(struct sk_buff *newsk,
352     struct sk_buff * prev, struct sk_buff *next,
353     struct sk_buff_head * list)
354 {
355     newsk->next = next;
356     newsk->prev = prev;
357     next->prev = newsk;
358     prev->next = newsk;
359     newsk->list = list;
360     list->qlen++;
361 }
```

prev와 next사이에 newsk를 삽입하는 함수이다.

```
366 extern __inline__ void skb_insert(struct sk_buff *old, struct sk_buff *newsk)
367 {
368     unsigned long flags;
369
370     spin_lock_irqsave(&skb_queue_lock, flags);
371     __skb_insert(newsk, old->prev, old, old->list);
372     spin_unlock_irqrestore(&skb_queue_lock, flags);
373 }
```

old바로 앞에 newsk를 삽입하는 함수로 atomic버젼이다.

```
379 extern __inline__ void __skb_append(struct sk_buff *old, struct sk_buff *newsk)
380 {
381     __skb_insert(newsk, old, old->next, old->list);
382 }
```

old 뒤에 newsk를 추가하는 함수이다.

```
384 extern __inline__ void skb_append(struct sk_buff *old, struct sk_buff *newsk)
385 {
386     unsigned long flags;
387
388     spin_lock_irqsave(&skb_queue_lock, flags);
389     __skb_append(old, newsk);
390     spin_unlock_irqrestore(&skb_queue_lock, flags);
391 }
```

old 뒤에 newsk를 추가하는 함수로 atomic버젼이다.

```
397 extern __inline__ void __skb_unlink(struct sk_buff *skb, struct sk_buff_head *list)
398 {
399     struct sk_buff * next, * prev;
400
401     list->qlen--;
402     next = skb->next;
403     prev = skb->prev;
404     skb->next = NULL;
405     skb->prev = NULL;
406     skb->list = NULL;
407     next->prev = prev;
408     prev->next = next;
409 }
```

list로부터 skb를 끊어 버린다.

```
418 extern __inline__ void skb_unlink(struct sk_buff *skb)
```

```
419 {
420     unsigned long flags;
421
422     spin_lock_irqsave(&skb_queue_lock, flags);
423     if(skb->list)
424             __skb_unlink(skb, skb->list);
425     spin_unlock_irqrestore(&skb_queue_lock, flags);
426 }
```

__skb_unlink의 atomic버젼이다.

```
428 /* XXX: more streamlined implementation */
429 extern __inline__ struct sk_buff *__skb_dequeue_tail(struct sk_buff_head *list)
430 {
431     struct sk_buff *skb = skb_peek_tail(list);
432     if (skb)
433             __skb_unlink(skb, list);
434     return skb;
435 }
```

list의 tail로부터 skb를 dequeue하는 함수이다.

```
437 extern __inline__ struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list)
438 {
439     long flags;
440     struct sk_buff *result;
441
442     spin_lock_irqsave(&skb_queue_lock, flags);
443     result = __skb_dequeue_tail(list);
444     spin_unlock_irqrestore(&skb_queue_lock, flags);
445     return result;
446 }
```

__skb_dequeue_tail의 atomic버젼이다.

```
452 extern __inline__ unsigned char *__skb_put(struct sk_buff *skb, unsigned int len)
453 {
454     unsigned char *tmp=skb->tail;
455     skb->tail+=len;
456     skb->len+=len;
457     return tmp;
458 }
```

실제데이터크기를 len만큼 늘린다.

```
460 extern __inline__ unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
461 {
462     unsigned char *tmp=skb->tail;
```

```
463     skb->tail+=len;
464     skb->len+=len;
465     if(skb->tail>skb->end)
466     {
467             __label__ here;
468             skb_over_panic(skb, len, &&here);
469 here:               ;
470     }
471     return tmp;
472 }
```

__skb_put수행후 tail이 end를 벗어나는 경우 skb_over_panic이 발생하도록 한다.

```
474 extern __inline__ unsigned char *__skb_push(struct sk_buff *skb, unsigned int len)
475 {
476     skb->data-=len;
477     skb->len+=len;
478     return skb->data;
479 }
```

데이터공간의 앞부분을 len만큼 확장한다.

```
481 extern __inline__ unsigned char *skb_push(struct sk_buff *skb, unsigned int len)
482 {
483     skb->data-=len;
484     skb->len+=len;
485     if(skb->data<skb->head)
486     {
487             __label__ here;
488             skb_under_panic(skb, len, &&here);
489 here:               ;
490     }
491     return skb->data;
492 }
```

__skb_push 수행 후, skb_under_panic이 발생하도록 한다.

```
494 extern __inline__ char *__skb_pull(struct sk_buff *skb, unsigned int len)
495 {
496     skb->len-=len;
497     return      skb->data+=len;
498 }
```

데이터 공간 앞부분 len만큼을 잘라버린다.

```
500 extern __inline__ unsigned char * skb_pull(struct sk_buff *skb, unsigned int len)
501 {
502     if (len > skb->len)
```

```
503                     return NULL;
504         return __skb_pull(skb,len);
505 }
```
__skb_pull을 수행하기전에 len이 데이터 공간보다 큰지를 검사한다

```
507 extern __inline__ int skb_headroom(struct sk_buff *skb)
508 {
509         return skb->data-skb->head;
510 }
```
데이터 공간의 앞부분의 크기를 리턴한다.

```
512 extern __inline__ int skb_tailroom(struct sk_buff *skb)
513 {
514         return skb->end-skb->tail;
515 }
```
데이터 공간의 뒷부분의 크기를 리턴한다.

```
517 extern __inline__ void skb_reserve(struct sk_buff *skb, unsigned int len)
518 {
519         skb->data+=len;
520         skb->tail+=len;
521 }
```
데이터 공간 전체를 len만큼 뒤로 옮긴다

```
523 extern __inline__ void __skb_trim(struct sk_buff *skb, unsigned int len)
524 {
525         skb->len = len;
526         skb->tail = skb->data+len;
527 }
```
데이터 공간의 크기를 len으로 한다

```
529 extern __inline__ void skb_trim(struct sk_buff *skb, unsigned int len)
530 {
531         if (skb->len > len) {
532                 __skb_trim(skb, len);
533         }
534 }
```
__skb_trim에 앞서 skb->len과 len를 비교한다.

```
536 extern __inline__ void skb_orphan(struct sk_buff *skb)
537 {
538         if (skb->destructor)
539                 skb->destructor(skb);
```

```
540     skb->destructor = NULL;
541     skb->sk = NULL;
542 }
```
skb의 소켓변수 sk를 NULL로 만든다

```
544 extern __inline__ void skb_queue_purge(struct sk_buff_head *list)
545 {
546     struct sk_buff *skb;
547     while ((skb=skb_dequeue(list))!=NULL)
548             kfree_skb(skb);
549 }
```
list내를 전부 지운다

```
551 extern __inline__ struct sk_buff *dev_alloc_skb(unsigned int length)
552 {
553     struct sk_buff *skb;
554
555     skb = alloc_skb(length+16, GFP_ATOMIC);
556     if (skb)
557             skb_reserve(skb,16);
558     return skb;
559 }
```
주어진 길이에 16을 더한 공간으로 sk_buff를 할당하고, 데이터 공간을 16만큼 뒤로 밀어버린다. 결국 데이터 공간 앞에 16바이트 공간을 비워둔다.

```
561 extern __inline__ struct sk_buff *
562 skb_cow(struct sk_buff *skb, unsigned int headroom)
563 {
564     headroom = (headroom+15)&~15;
565
566     if ((unsigned)skb_headroom(skb) < headroom || skb_cloned(skb)) {
567             struct sk_buff *skb2 = skb_realloc_headroom(skb, headroom);
568             kfree_skb(skb);
569             skb = skb2;
570     }
571     return skb;
572 }
```
데이터공간의 앞부분에 16바이트 단위의 headroom을 확보한다.

3. Device driver

3.1 device driver

이번 장에서는 3c509네트워크 카드용 device driver에 대해 알아보겠다. driver전체를 보는 것은 의미가 없으므로 프레임 송수신을 위한 함수만 살펴볼 것이다. 3c509 driver 소스코드는 drivers/net/3c509.c에 있다.

3.2 el3_interrupt 함수

이 함수는 네트워크 인터페이스 카드 (보통 NIC라고 한다)에서 인터럽트가 발생하면, 가장 먼저 수행되는 함수이다. 이 함수에서는 인터럽트 상태를 확인하여, RxComplete이면 수신 루틴을, TxAvailable이면 송신루틴을 수행한다. 그러면 소스코드와 흐름을 보자.

```
614 /* The EL3 interrupt handler. */
615 static void
616 el3_interrupt(int irq, void *dev_id, struct pt_regs *regs)
617 {
618     struct device *dev = (struct device *)dev_id;
619     struct el3_private *lp;
620     int ioaddr, status;
621     int i = max_interrupt_work;
622
623     if (dev == NULL) {
624                 printk ("el3_interrupt(): irq %d for unknown device.\n", irq);
625                 return;
626     }
627
628     lp = (struct el3_private *)dev->priv;
629     spin_lock(&lp->lock);
```

이 함수의 인수로서 device구조체 포인터를 넘겨 받고 있다. struct device구조체는 include/linux/netdevice.h에 정의되어 있으며, 각 인터페이스 드라이버에 대한 정보를 가지고 있다. 이 구조체에 대하여는 다음 장에서 살펴보겠다. 621행의

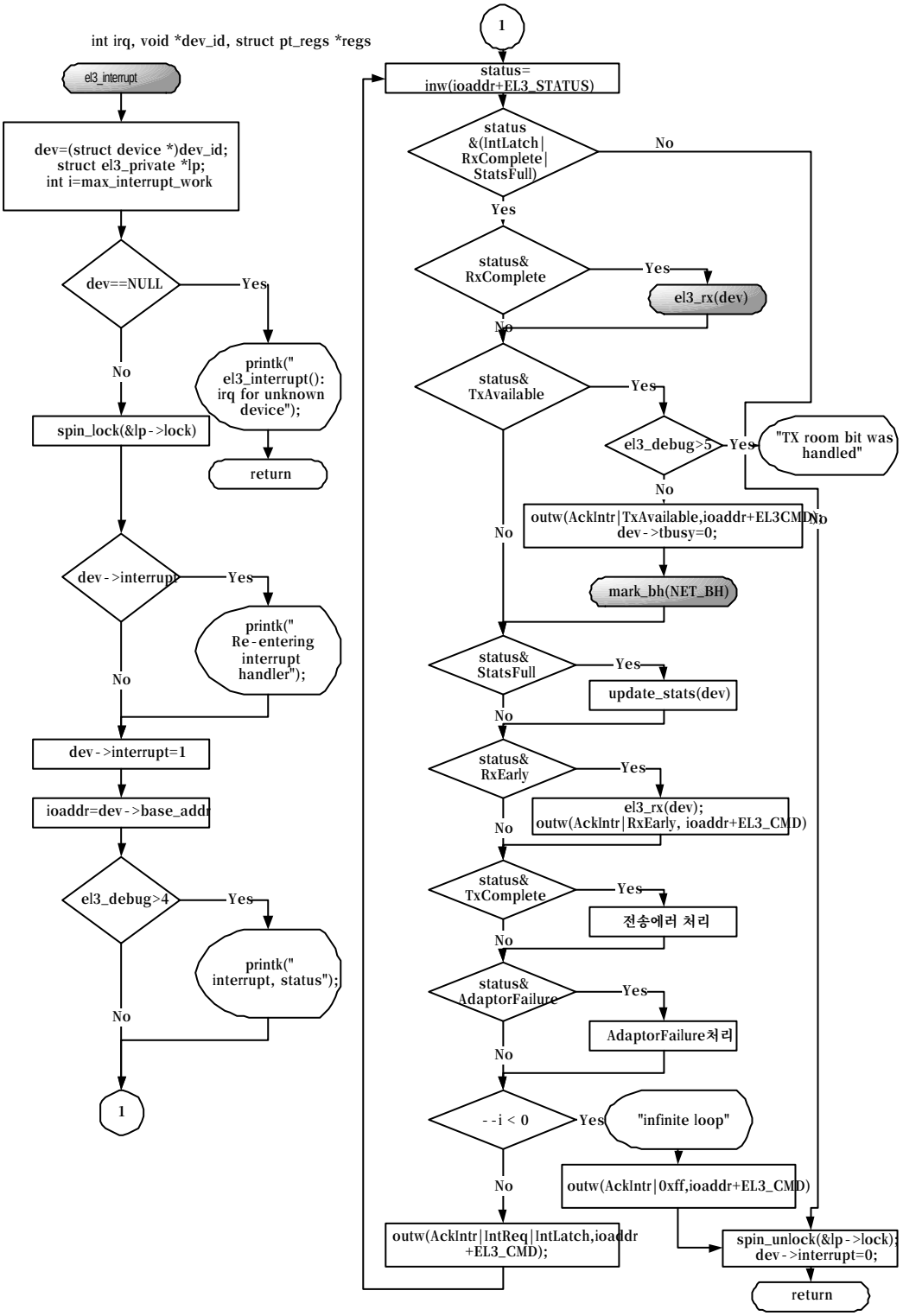


그림 3.1 el3_interrupt()

max_interrupt_work값은 10으로 설정되어 있다. 628행에서 우리의 3c509카드가 사용할 정보가 Ip로 넘어온다. 629행에 spin_lock함수는 include/asm-i386/spinlock.h에 다음과 같이 정의되어 있다.

```
#define spin_lock_init(lock)      do { } while(0)
#define spin_lock(lock)           (void)(lock) /* Not "unused variable". */
#define spin_trylock(lock)        (1)
#define spin_unlock_wait(lock)    do { } while(0)
#define spin_unlock(lock)         do { } while(0)
#define spin_lock_irq(lock)       cli()
#define spin_unlock_irq(lock)     sti()
```

결국 단일 CPU에서 spin_lock은 특별한 기능을 하지 않는다.

```
631   if (dev->interrupt)
632           printk("%s: Re-entering the interrupt handler.\n", dev->name);
633   dev->interrupt = 1;
634
635   ioaddr = dev->base_addr;
636
```

dev->interrupt=1로 하여 현재 이 장치가 인터럽트 처리중임을 표시한다. 635행에서는 이 장치의 io base 어드레스를 가져온다.

```
637   if (el3_debug > 4) {
638           status = inw(ioaddr + EL3_STATUS);
639           printk("%s: interrupt, status %4.4x.\n", dev->name, status);
640   }
641
642   while ((status = inw(ioaddr + EL3_STATUS)) &
643              (IntLatch | RxComplete | StatsFull)) {
644
```

이 while문에서는 장치의 상태를 읽어 IntLatch, RxComplete, 또는 StatsFull상태이면 계속 처리를 해준다.

```
645           if (status & RxComplete)
646                   el3_rx(dev);
647
```

만약 RxComplete상태이면, 즉 데이터를 수신하였으면 el3_rx()함수를 수행한다. 이 함수는 다음 절에서 살펴 볼 것이다.

```
 648            if (status & TxAvailable) {
 649                    if (el3_debug > 5)
 650                            printk("    TX room bit was handled.\n");
 651                    /* There's room in the FIFO for a full-sized packet. */
 652                    outw(AckIntr | TxAvailable, ioaddr + EL3_CMD);
 653                    dev->tbusy = 0;
 654                    mark_bh(NET_BH);
 655            }
```

648행부터 655행은 전송가능한 상태가 된 것으로 mark_bh(NET_BH)로 Bottom Half Handler를 세팅하고 있다. mark_bh() 및 Bottom Half Handler에 대하여는 다음 장에서 다룰 것이다.

```
 656            if (status & (AdapterFailure | RxEarly | StatsFull | TxComplete)) {
 657                    /* Handle all uncommon interrupts. */
 658                    if (status & StatsFull)                         /* Empt
y statistics. */
 659                            update_stats(dev);
 660                    if (status & RxEarly) {                         /* Rx ea
rly is unused. */
 661                            el3_rx(dev);
 662                            outw(AckIntr | RxEarly, ioaddr + EL3_CMD);
 663                    }
 664                    if (status & TxComplete) {                      /* Really Tx error.
*/
 665                            struct el3_private *lp = (struct el3_private *)dev->priv;
 666                            short tx_status;
 667                            int i = 4;
 668
 669                            while (--i>0 && (tx_status = inb(ioaddr + TX_STATUS)) >
0) {
 670                                    if (tx_status & 0x38) lp->stats.tx_aborted_errors+
+;
 671                                    if (tx_status & 0x30) outw(TxReset, ioaddr + EL3_
CMD);
 672                                    if (tx_status & 0x3C) outw(TxEnable, ioaddr + EL
```

```
3_CMD);
  673                                     outb(0x00, ioaddr + TX_STATUS); /* Pop the stat
us stack. */
  674                             }
  675                     }
  676                     if (status & AdapterFailure) {
  677                             /* Adapter failure requires Rx reset and reinit. */
  678                             outw(RxReset, ioaddr + EL3_CMD);
  679                             /* Set the Rx filter to the current state. */
  680                             outw(SetRxFilter | RxStation | RxBroadcast
  681                                     | (dev->flags & IFF_ALLMULTI ? RxMulticast :
0)
  682                                     | (dev->flags & IFF_PROMISC ? RxProm : 0),
  683                                     ioaddr + EL3_CMD);
  684                             outw(RxEnable, ioaddr + EL3_CMD); /* Re-enable the recei
ver. */
  685                             outw(AckIntr | AdapterFailure, ioaddr + EL3_CMD);
  686                     }
  687             }
  688
```

656행부터 688행은 각각 StatsFull, RxEarly,TxComplete,AdaptorFailure상태에 대한 처리를 하고 있다. 이 부분 설명은 하드웨어 의존적이고 에러처리를 해주는 부분이므로 생략한다.

```
  689             if (--i < 0) {
  690                     printk("%s: Infinite loop in interrupt, status %4.4x. \n",
  691                             dev->name, status);
  692                     /* Clear all interrupts. */
  693                     outw(AckIntr | 0xFF, ioaddr + EL3_CMD);
  694                     break;
  695             }
```

처음에 621행에서 i=max_interrupt_work로 치환하였고, while문을 한번 돌 때마다 i값이 감소한다. 결국 최대 10회까지 인터럽트처리를 해준다.

```
  696             /* Acknowledge the IRQ. */
  697             outw(AckIntr | IntReq | IntLatch, ioaddr + EL3_CMD); /* Ack IRQ */
  698     }
```

697행에서는 하드웨어 인터럽트 처리가 끝났음을 네트워크 장치에게 알려준다. 698행의 brace는 while문의 끝부분이다.

```
699
700    if (el3_debug > 4) {
701                printk("%s: exiting interrupt, status %4.4x.\n", dev->name,
702                                inw(ioaddr + EL3_STATUS));
703    }
704    spin_unlock(&lp->lock);
705    dev->interrupt = 0;
706    return;
707 }
```

704행에서는 629행에서와 쌍을 이루어 lock한 것을 풀어주는 부분이다. 그렇지만, 현재 아무기능도 하지 않는다. 705행에서는 interrupt=0을 하여 인터럽트 처리가 끝났음을 표시한다.

3.3 el3_rx 함수

이 el3_rx()함수는 네트워크 인터페이스 카드가 데이터를 수신하였을때, el3_interrupt() 함수에 의해 호출된다.

```
761 static int
762 el3_rx(struct device *dev)
763 {
764    struct el3_private *lp = (struct el3_private *)dev->priv;
765    int ioaddr = dev->base_addr;
766    short rx_status;
767
768    if (el3_debug > 5)
769                printk("    In rx_packet(), status %4.4x, rx_status %4.4x.\n",
770                                inw(ioaddr+EL3_STATUS), inw(ioaddr+RX_STATUS));
771    while ((rx_status = inw(ioaddr + RX_STATUS)) > 0) {
```

네트워크 인터페이스 카드의 RX상태를 읽어, 처리할 내용이 있으면 반복하여 수행하도록 한다.

```
772                if (rx_status & 0x4000) { /* Error, update stats. */
773                        short error = rx_status & 0x3800;
774
```

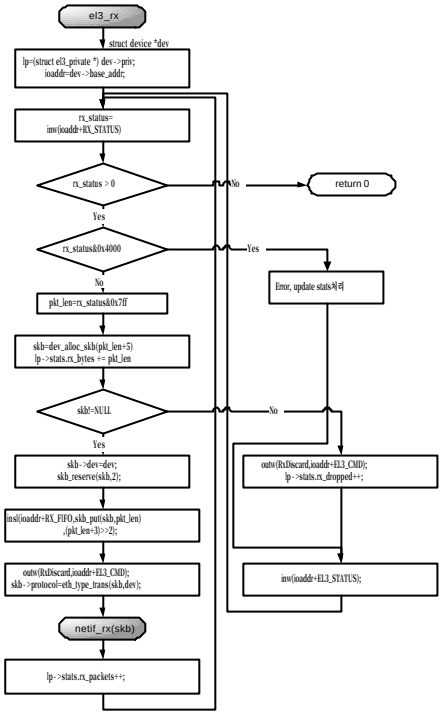


그림 3.2 el3_rx

```
775                 outw(RxDiscard, ioaddr + EL3_CMD);
776                 lp->stats.rx_errors++;
777                 switch (error) {
778                 case 0x0000:            lp->stats.rx_over_errors++; break;
779                 case 0x0800:            lp->stats.rx_length_errors++; break;
780                 case 0x1000:            lp->stats.rx_frame_errors++; break;
781                 case 0x1800:            lp->stats.rx_length_errors++; break;
782                 case 0x2000:            lp->stats.rx_frame_errors++; break;
783                 case 0x2800:            lp->stats.rx_crc_errors++; break;
784                 }
```

772행부터 784행까지는 수신 에러 처리를 해주고 있다.

```
785             } else {
786                 short pkt_len = rx_status & 0x7ff;
787                 struct sk_buff *skb;
788
789                 skb = dev_alloc_skb(pkt_len+5);
790                 lp->stats.rx_bytes += pkt_len;
```

786행에서 rx_status의 하위11비트는 수신된 데이타의 길이를 의미하고 있다. 수신된 데이타의 길이는 pkt_len에 들어간다. 789행에서 skbuffer공간을 할당한다. 이때 크기에 5를 더하고 있다. 790행은 총 수신한 데이터 크기를 가지고 있는 통계변수를 갱신한다.

```
791                     if (el3_debug > 4)
792                             printk("Receiving packet size %d status %4.4x. \n",
793                                     pkt_len, rx_status);
794                     if (skb != NULL) {
795                             skb->dev = dev;
796                             skb_reserve(skb, 2);     /* Align IP on 16 byte */
797
798                             /* 'skb->data' points to the start of sk_buff data area. */
799 #ifdef  __powerpc__
800                             insl_unswapped(ioaddr+RX_FIFO, skb_put(skb,pkt_len),
801                                                                    (pkt_len + 3) >> 2);
802 #else
803                             insl(ioaddr + RX_FIFO, skb_put(skb,pkt_len),
804                                     (pkt_len + 3) >> 2);
805 #endif
806
807                             outw(RxDiscard, ioaddr + EL3_CMD); /* Pop top Rx packe
 t. */
808                             skb->protocol = eth_type_trans(skb,dev);
809                             netif_rx(skb);
810                             lp->stats.rx_packets++;
811                             continue;
812                     }
```

795행에서 수신한 인터페이스를 skb의 dev에 기록하고 있다. 796행에서 데이터 공간에 2바이트를 추가적으로 더 비워둔다. 결국 789행에서의 dev_alloc_skb와 796행의 skb_reserve를 수행하면 데이터 공간 앞에 16+2, 즉 18바이트를 비워두게 된다. 803행에서 네트워크 카드로 부터 데이터를 skb의 데이터공간에 읽어오고 있다. insl함수는 in()의 string타입으로 해당크기만큼 연속적으로 long(4바이트)형으로 읽어오는 기능을 한다. 크기에서 >>2를 한 이유가 바로 한번에 4바이트씩 읽기 때문인 것이다. 808행에서 eth_type_trans()함수는 net/ethernet/eth.c에 정의되어 있으며 주요부분은 다음과 같다.

```
179 unsigned short eth_type_trans(struct sk_buff *skb, struct device *dev)
180 {
181         struct ethhdr *eth;
182         unsigned char *rawp;
183
184         skb->mac.raw=skb->data;
185         skb_pull(skb,dev->hard_header_len);
186         eth= skb->mac.ethernet;
210         if (ntohs(eth->h_proto) >= 1536)   /* 1536 == 0x0600 */
211                 return eth->h_proto;
228 }
```

결국 데이터 공간으로 부터 ethernet헤더를 떼어내고, skb.mac포인터를 세팅한후, Ethernet의 프로토콜 타입(ip의 경우 0x0800)을 반환한다. 그래서 skb->protocol값에는 이 반환값이 들어 가게 될 것이다. 다음으로 809행에서 netif_rx()함수를 호출하고 있다. 이 함수는 net/core/dev.c에 정의된 함수로 다음 장에서 다룬다.

```
813                     outw(RxDiscard, ioaddr + EL3_CMD);
814                     lp->stats.rx_dropped++;
815                     if (el3_debug)
816                             printk("%s: Couldn't allocate a sk_buff of size %d.\n",
817                                    dev->name, pkt_len);
818             }
```

813행부터 817행 부분은 skbuff할당을 못받은 경우를 처리하고 있다.

```
819             inw(ioaddr + EL3_STATUS);                       /* Delay. */
820             while (inw(ioaddr + EL3_STATUS) & 0x1000)
821                     printk(KERN_DEBUG "      Waiting for 3c509 to discard packet, st
atus %x.\n",
822                            inw(ioaddr + EL3_STATUS) );
823     }
825     return 0;
826 }
```

819행은 약간의 delay를 갖기 위한 부분이고 820행부터 822행은 에러처리를 하는 부분이다. 825행에서 모든 수신처리가 끝나서 함수가 종료된다.

3.4 el3_start_xmit 함수

이 el3_start_xmit()함수는 실제로 네트워크 카드를 통하여 데이터를 전송하는 함수이다.

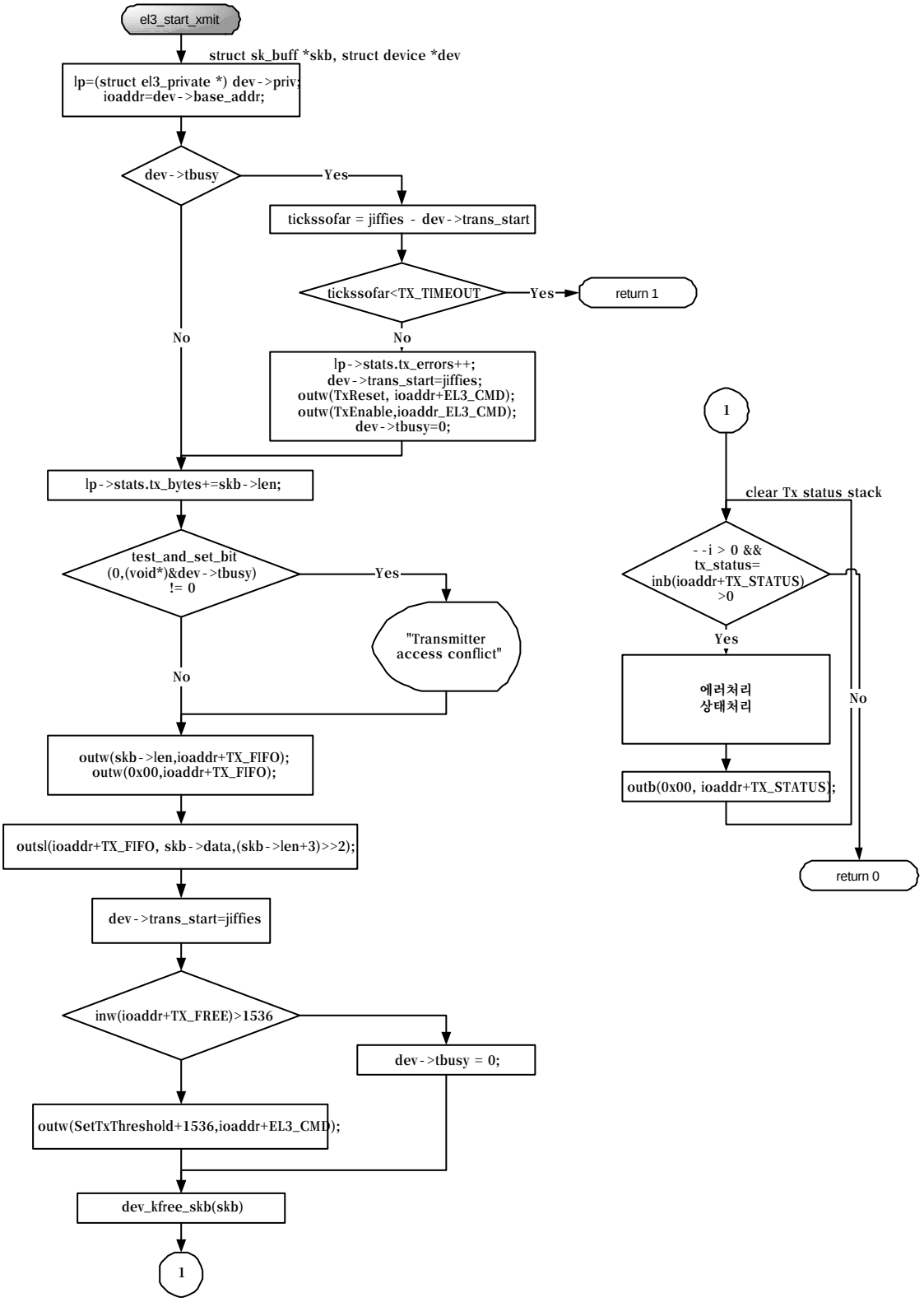

el3_start_xmit
struct sk_buff *skb, struct device *dev
lp=(struct el3_private *) dev->priv;
ioaddr=dev->base_addr;
dev->tbusy
Yes
tickssofar = jiffies - dev->trans_start
tickssofar<TX_TIMEOUT
Yes
return 1
No
No
lp->stats.tx_errors++;
dev->trans_start=jiffies;
outw(TxReset, ioaddr+EL3_CMD);
outw(TxEnable,ioaddr_EL3_CMD);
dev->tbusy=0;
lp->stats.tx_bytes+=skb->len;
test_and_set_bit
(0,(void*)&dev->tbusy)
!= 0
Yes
"Transmitter
access conflict"
No
outw(skb->len,ioaddr+TX_FIFO);
outw(0x00,ioaddr+TX_FIFO);
outsl(ioaddr+TX_FIFO, skb->data,(skb->len+3)>>2);
dev->trans_start=jiffies
inw(ioaddr+TX_FREE)>1536
dev->tbusy = 0;
outw(SetTxThreshold+1536,ioaddr+EL3_CMD);
dev_kfree_skb(skb)
1
1
clear Tx status stack
--i > 0 &&
tx_status=
inb(ioaddr+TX_STATUS)
>0
Yes
에러처리
상태처리
outb(0x00, ioaddr+TX_STATUS);
No
return 0

그림 3.3

```
507 static int
508 el3_start_xmit(struct sk_buff *skb, struct device *dev)
509 {
510     struct el3_private *lp = (struct el3_private *)dev->priv;
511     int ioaddr = dev->base_addr;
512
513     /* Transmitter timeout, serious problems. */
514     if (dev->tbusy) {
515             int tickssofar = jiffies - dev->trans_start;
516             if (tickssofar < TX_TIMEOUT)
517                     return 1;
518             printk("%s: transmit timed out, Tx_status %2.2x status %4.4x "
519                     "Tx FIFO room %d.\n",
520                     dev->name, inb(ioaddr + TX_STATUS), inw(ioaddr + EL3_STATUS),
521                     inw(ioaddr + TX_FREE));
522             lp->stats.tx_errors++;
523             dev->trans_start = jiffies;
524             /* Issue TX_RESET and TX_START commands. */
525             outw(TxReset, ioaddr + EL3_CMD);
526             outw(TxEnable, ioaddr + EL3_CMD);
527             dev->tbusy = 0;
528     }
```

이 함수는 tbusy가 아닌 상태에서 호출되는 함수인데 tbusy가 세트되어있는 상태가 검출되면 에러를 처리하고 tbusy=0상태로 만든다. jiffies라는 변수는 include/linux/sched.h에 정의되어 있 전역변수이다.

```
529
530     lp->stats.tx_bytes += skb->len;
531
532     if (el3_debug > 4) {
533             printk("%s: el3_start_xmit(length = %u) called, status %4.4x.\n",
534                     dev->name, skb->len, inw(ioaddr + EL3_STATUS));
535     }
```

530행에서는 통계변수를 갱신하고 있다.

```
555     /* Avoid timer-based retransmission conflicts. */
```

```
556    if (test_and_set_bit(0, (void*)&dev->tbusy) != 0)
557            printk("%s: Transmitter access conflict. \n", dev->name);
```

556행에서 dev->tbusy상태를 1로 세트한다. test_and_set_bit함수는 include/asm-i386/bitops.h에 정의되어 있으며, 비트를 1로 세트하고, 원래 값을 반환한다.

```
558    else {
575            /* Put out the doubleword header... */
576            outw(skb->len, ioaddr + TX_FIFO);
577            outw(0x00, ioaddr + TX_FIFO);
582            outsl(ioaddr + TX_FIFO, skb->data, (skb->len + 3) >> 2);
```

네트워크 카드에 skb->data를 전송한다. 역시 그 크기는 4바이트씩 수행하므로 (skb->len+3)>>2로 처리하였다.

```
585            dev->trans_start = jiffies;
586            if (inw(ioaddr + TX_FREE) > 1536) {
587                    dev->tbusy = 0;
588            } else
589                    /* Interrupt us when the FIFO has room for max-sized packet. */
590                    outw(SetTxThreshold + 1536, ioaddr + EL3_CMD);
595    }
```

전송이 끝났으면 dev->tbusy를 0으로 클리어 하고, 안 끝났으면, 끝난후 인터럽트가 걸리도록 네트워크카드를 세트한다.

```
597    dev_kfree_skb (skb);
```

skbuffer를 해제한다. dev_kfree_skb()함수는 kfree_skb()함수로 skbuff.h에 정의되어 있다.

```
599    /* Clear the Tx status stack. */
600    {
601            short tx_status;
602            int i = 4;
603
604            while (--i > 0      &&      (tx_status = inb(ioaddr + TX_STATUS)) > 0) {
605                    if (tx_status & 0x38) lp->stats.tx_aborted_errors++;
606                    if (tx_status & 0x30) outw(TxReset, ioaddr + EL3_CMD);
```

```
607                         if (tx_status & 0x3C)  outw(TxEnable, ioaddr + EL3_CMD);
608                         outb(0x00, ioaddr + TX_STATUS); /* Pop the status stack. */
609                 }
610     }
611     return 0;
612 }
```

599행부터 610행까지는 네트워크 카드 내부의 Tx status stack을 비우는 역할을 한다. 611행에서 이 함수가 정상적으로 종료되고 있다.

4. netdevice

4.1 netdevice

이번 장에서는 네트워크 인터페이스 카드에 독립적인 함수들에 대해 알아본다. 또한, 각 장치를 관리하는 device라는 구조체에 대해서 살펴본다. 이러한 함수들은 net/core/dev.c파일에 정의되어 있으며, device구조체는 include/linux/netdevice.h에 정의되어 있다.

4.2 device 구조체

device구조체는 각 장치에 대한 정보를 가지고 있다. 이 구조체는 include/linux/netdevice.h에 정의되어 있다.

char	*name	인터페이스의 이름
unsigned long	rmem_end	공유 수신 메모리 끝
unsigned long	rmem_start	공유 수신 메모리 시작
unsigned long	mem_end	공유메모리 끝
unsigned long	mem_start	공유메모리 시작
unsigned long	base_addr	장치의 IO 주소
unsigned int	irq	장치의 IRQ번호
volatile unsigned char	start	동작을 시작했다
unsigned long	interrupt	인터럽트 처리중
unsigned long	tbusy	전송중
struct device	*next	다음 장치를 가리킨다
int	(*init)(struct device *dev)	초기화 함수
void	(*destructor)(struct device *dev)	해제 함수
int	ifindex	인터페이스 index
int	iflink	인터페이스 link
unsigned char	if_port	인터페이스의 포트 선택
unsigned char	dma	DMA 번호
struct net_device_stats*	(*get_stats)(struct device *dev)	상태를 얻기 위한 함수
struct iw_statistics*	(*get_wireless_stats)(struct device *dev)	wireless상태 얻는 함수
unsigned long	trans_start	최근에 전송한 시각 (단위는 jiffies)
unsigned long	last_rx	최근에 수신한 시각
unsigned short	flags	인터페이스 flag
unsigned short	gflags	gflag
unsigned	mtu	인터페이스 MTU
unsigned short	type	인터페이스 hardware type
unsigned short	hard_header_len	hardware 헤더 길이
void	*priv	인터페이스별 데이터공간을 가리킨다.
unsigned char	broadcast[MAX_ADDR_LEN]	하드웨어 broadcast 주소
unsigned char	pad	dev_addr을 8바이트 단위로 조정하기 위한 pad

unsigned char	dev_addr[MAX_ADDR_LEN]	하드웨어 주소
unsigned char	addr_len	하드웨어 주소의 길이
struct dev_mc_list	*mc_list	Multicast MAC 주소의 리스트
int	mc_count	Multicast 주소의 개수
int	promiscuity	
int	allmulti	
unsigned long	pkt_queue	큐된 packet수
struct device	*slave	slave device
void	*atalk_ptr	AppleTalk link
void	*ip_ptr	IPv4 특정 데이터
void	*dn_ptr	Decnet 특정 데이터
struct Qdisc	*qdisc	Q discipline
struct Qdisc	*qdisc_sleeping	Q discipline sleeping
struct Qdisc	*qdisc_list	Q discipline list
unsigned long	tx_queue_len	최대 전송할 수 있는 큐의 크기
int	bridge_port_id	
int	(*open)(struct device *dev)	장치를 open하는 함수
int	(*stop)(struct device *dev)	장치를 stop하는 함수
int	(*hard_start_xmit) (struct sk_buff *skb, strut device *dev)	장치의 전송 시작 함수
int	(*hard_header) (struct sk_buff *skb,struct deice *dev,unsigned short type,void *daddr,void *saddr,unsigned len)	하드웨어 헤더 구성함수
int	(*rebuild_header)(struct sk_buff *skb)	헤더 재구성 함수
void	(*set_multicast_list)(struct device *dev)	Multicast리스트 구성함수
int	(*set_mac_address)(struct device *dev, void *dr)	MAC주소 설정함수
int	(*do_ioctl)(struct device *dev,struct ifreq *fr, int cmd)	별도의 io제어 함수
int	(*set_config)(struct device *dev, struct ifmap *map)	별도의 설정 세트함수
int	(*hard_header_cache)(struct neighbour *neigh,struct hh_cache *hh)	하드웨어 헤더 캐시 설정함수
void	(*header_cache_update)(struct hh_cache *hh,struct device *dev, unsigned char * haddr)	하드웨어 헤더 캐시 갱신함수
int	(*change_mtu)(struct device *dev, int new_mtu)	MTU변경 함수
int	(*hard_header_parse)(struct sk_buff *skb, unsigned char *haddr)	헤더 분석 함수
int	(*neigh_setup)(struct device *dev, struct neigh_parms *)	
int	(*accept_fastpath)(struct device *, struct dst_entry*)	

4.3 netif_rx 함수

네트워크 인터페이스 카드 드라이버가 데이터를 수신한후 netif_rx함수를 호출하도록 되어 있다. 즉 netif_rx()함수는 데이터 수신을 처리하는 함수이다. 이 함수는 net/core/dev.c에 정의되어 있다.

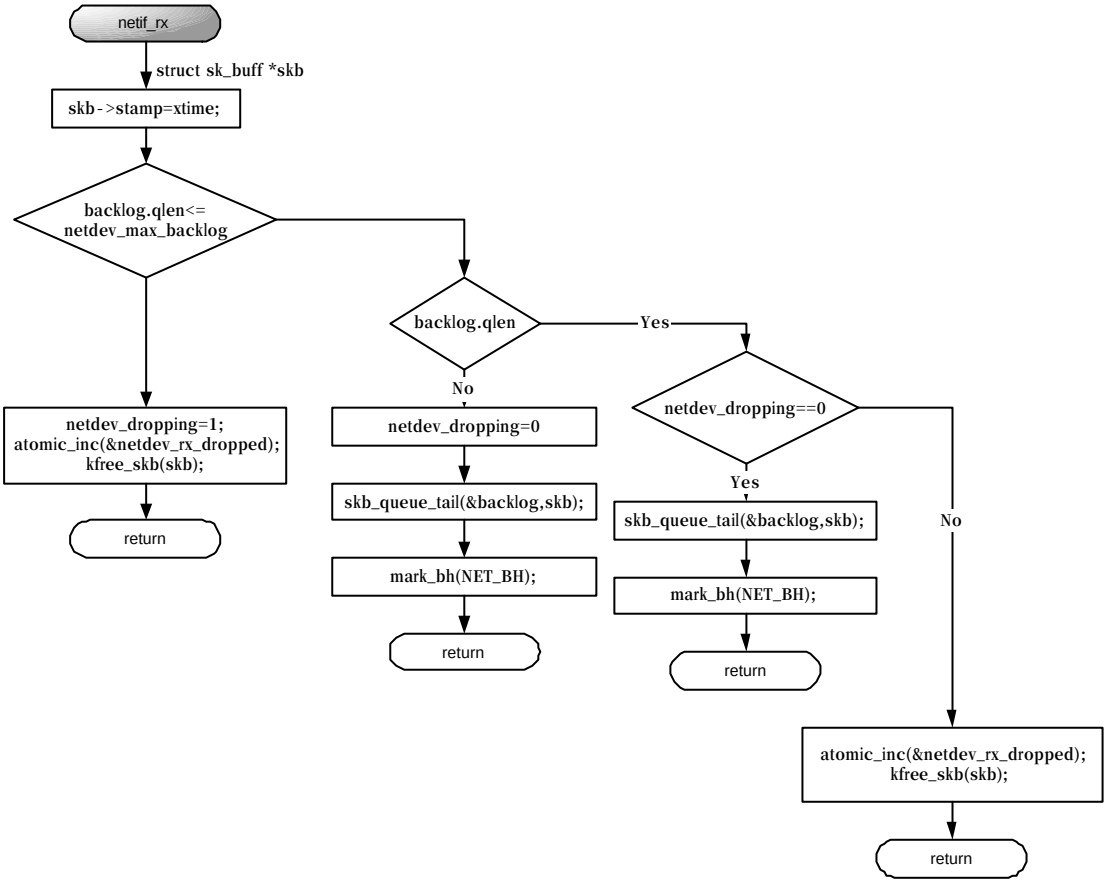


그림 4.1

```
757 void netif_rx(struct sk_buff *skb)
758 {
763     skb->stamp = xtime;
770     if (backlog.qlen <= netdev_max_backlog) {
771                 if (backlog.qlen) {
```

```
772                     if (netdev_dropping == 0) {
773                             skb_queue_tail(&backlog,skb);
774                             mark_bh(NET_BH);
775                             return;
776                     }
777                     atomic_inc(&netdev_rx_dropped);
778                     kfree_skb(skb);
779                     return;
780             }
```

backlog버퍼에 공간이 남아 있으면 tail에 skb를 enqueue하고 mark_bh(NET_BH)을 수행한 후 함수를 종료한다. 777행부터 779행은 만약 netdev_dropping이 0이 아니면 skb를 해제하여 수신된 패킷을 drop시킨다. netdev_dropping은 전역변수로 791행을 보면 backlog가 꽉 차면 1로 세트된다. backlog버퍼는 dev.c파일 156행에 struct sk_buff_head형으로 정의되어 있다. backlog는 전역변수이다.

```
785             netdev_dropping = 0;
787             skb_queue_tail(&backlog,skb);
788             mark_bh(NET_BH);
789             return;
790     }
```

backlog.qlen이 비어 있는 경우, 785행부터 789행이 수행되며, 마찬가지로 backlog버퍼리스트에 skb를 삽입하고 mark_bh(NET_BH)를 수행한후, 함수를 종료한다.

```
791     netdev_dropping = 1;
792     atomic_inc(&netdev_rx_dropped);
793     kfree_skb(skb);
794 }
```

backlog버퍼가 꽉찬 경우 netdev_dropping을 1로 세트하고, skb를 해제한다.

4.4 mark_bh() 함수

이 mark_bh()함수는 Bottom Half Handler 수행을 위해 세팅을 해주는 함수이다. 이 함수는 include/asm-i386/softirq.h에 정의되어 있으며 NET_BH값은 include/linux/interrupt.h에 9로 정의되어 있다. mark_bh()함수를 잠깐 살펴보면 다음과 같다. 결국 bh_active라는 변수의 9번째 비트를 세트함으로써 BH handler가

net_bh()를 수행하도록 한다.

```
100 extern inline void mark_bh(int nr)
101 {
102             set_bit(nr, &bh_active);
103 }
```

bh_active변수는 kernel/softirq.c에 전역변수로 정의되어 있다.

```
 23 atomic_t bh_mask_count[32];
 24 unsigned long bh_active = 0;
 25 unsigned long bh_mask = 0;
 26 void (*bh_base[32])(void);
```

여기서 26행을 보면 active상태시 수행해야 하는 함수가 들어가도록 되어 있다. 이는 kernel/softirq.c에 있는 초기화 함수__initfunc()를 보면 mark_bh(NET_BH)로 세트되었을 때, net_bh()가 호출되도록 설정하고 있다. 아래에 _initfunc() 소스코드의 일부를 보였다.

```
1893 __initfunc(int net_dev_init(void))
1894 {
1895       struct device *dev, **dp;
1905       skb_queue_head_init(&backlog);
2027       init_bh(NET_BH, net_bh);
2031       dev_mcast_init();
2044       return 0;
2045 }
```

2027행을 보면 NET_BH에 대해 net_bh함수를 세트하고 있다. 계속해서 init_bh()함수를 살펴보자. init_bh()함수는 kernel/softirq.c에 있다.

```
 76 extern inline void init_bh(int nr, void (*routine)(void))
 77 {
 78       unsigned long flags;
 79
 80       bh_base[nr] = routine;
 81       atomic_set(&bh_mask_count[nr], 0);
 82
```

```
83      spin_lock_irqsave(&i386_bh_lock, flags);
84      bh_mask |= 1 << nr;
85      spin_unlock_irqrestore(&i386_bh_lock, flags);
86 }
```

80행을 보면 NET_BH에 해당하는 인덱스로 bh_base를 net_bh함수로 설정하고 있다.

4.5 net_bh() 함수

```
835 void net_bh(void)
836 {
837     struct packet_type *ptype;
838     struct packet_type *pt_prev;
839     unsigned short type;
840     unsigned long start_time = jiffies;
849
850     NET_PROFILE_ENTER(net_bh);
857
```

850행의 NET_PROFILE_ENTER라는 함수는 include/net/profile.h에 정의되어 있으며 profile을 만들기 위한 것이다.

```
858     if (qdisc_head.forw != &qdisc_head)
859             qdisc_run_queues();
```

858행을 보면, 전송을 위한 큐 discipine의 헤드를 나타내는 qdisc_head 전역변수가 있다. (net/sched/sch_generic.c). 이 변수는 처음에 qdisc_head.forw = qdisc_head 로 초기화 되어 있다. 여기에 큐 discipline를 추가하게 되는 것이다. 아무튼 어떤 큐 discipline이 있으면, qdisc_run_queues()를 호출한다. 이 함수는 말 그대로 큐를 수행하는 함수이다.

```
875     while (!skb_queue_empty(&backlog))
876     {
877             struct sk_buff * skb;
878
880             if (jiffies - start_time > 1)
881                     goto net_bh_break;
882
886             skb = skb_dequeue(&backlog);
887
```

backlog q에 수신된 데이터가 있으면 처리하는 부분이다 886행을 보면 backlog큐로부터 sk데이터를 꺼내고 있다.

```
916             skb->h.raw = skb->nh.raw = skb->data;
```

```
 918             if (skb->mac.raw < skb->head || skb->mac.raw > skb->data) {
 919                     printk(KERN_CRIT "%s: wrong mac.raw ptr, proto=%04x \n", skb->de
v->name, skb->protocol);
 920                     kfree_skb(skb);
 921                     continue;
 922             }
 928             type = skb->protocol;
```

Transport Header와 Network Header를 초기화 하고 있다. 이는 아직까지 완전한 값은 아니다. 918-922행은 에러처리 부분이다. 928행에서 type값을 얻어온다. skb->protocol은 디바이스 드라이버에서 이미 수신한 프레임을 이용하여 치환하였다.

```
 945             pt_prev = NULL;
 946             for (ptype = ptype_all; ptype!=NULL; ptype=ptype->next)
 947             {
 948                     if (!ptype->dev || ptype->dev == skb->dev) {
 949                             if(pt_prev)
 950                             {
 951                                     struct sk_buff *skb2=skb_clone(skb, GFP_ATOMI
C);
 952                                     if(skb2)
 953                                             pt_prev->func(skb2,skb->dev, pt_prev);
 954                             }
 955                             pt_prev=ptype;
 956                     }
 957             }
```

ptype_all이란 ETH_P_ALL타입의 모든 type구조체를 연결해놓은 리스트의 처음을 가리킨다. struct packet_type구조체는 include/net/netdevice.h에 정의되어 있다. 아무튼 946-957행까지는 pt_prev에 수신한 type에 맞는 type구조체를 얻어온다. 948의 dev가 NULL이면 wildcard기능을 한다. 즉, 디바이스에 상관없이 수행되는 타입이다. 같은 타입에 대하여 ptype리스트에 2개 이상이 설정되어 있을수 있는데, 이 경우 clone되어 처리된다.

```
 959             for (ptype = ptype_base[ntohs(type)&15]; ptype != NULL; ptype = ptype->next)
 960             {
 961                     if (ptype->type == type && (!ptype->dev || ptype->dev==skb->dev))
 962                     {
 967                             if(pt_prev)
 968                             {
 969                                     struct sk_buff *skb2;
 971                                     skb2=skb_clone(skb, GFP_ATOMIC);
 978                                     if(skb2)
 979                                             pt_prev->func(skb2, skb->dev, pt_pre
```

```
v);
   980                                  }
   982                                  pt_prev=ptype;
   983                          }
   984                 } /* End of protocol list loop */
```

여기서는 ptype_base[]로부터 해당 type구조체 리스트를 끝까지 검사하여 같은 type이 있으면 clone하여 처리한다. 이 블록은 type이 ETH_P_ALL형태가 아닌 경우에 해당된다.

```
   990                 if(pt_prev)
   991                         pt_prev->func(skb, skb->dev, pt_prev);
   996                 else {
   997                         kfree_skb(skb);
   998                 }
   999         }
```

해당 type에 맞는 수신처리함수를 호출하고 있다.

```
  1009     if (qdisc_head.forw != &qdisc_head)
  1010             qdisc_run_queues();
```

다시한번 송신루틴을 호출하고 있다.

```
  1025     NET_PROFILE_LEAVE(net_bh);
  1026     return;
  1027
  1032 }
```

PROFILE에 기록한후 종료한다.

5. Linux Traffic Control

1. Linux의 트래픽 제어부 개요

Linux커널(버젼 2.2.X)의 Traffic Control Code는 다음과 같은 4개의 주요부분으로 구성되어 있다.

큐 구조 (Queue Discipline)
클래스 (Class)
필터 (Filter)
정책제어 (Policing)

각 네트워크 장치는 그것에 할당된 큐 구조를 가지고 있다. 큐 구조는 그 장치로 출력되어야 할 패킷을 어떻게 처리할 것인가를 결정하고 수행하게 된다. 가장 단순한 큐 구조는 하나의 FIFO큐를 가지고 있는 형태가 될 것이다. 더 복잡한 형태의 큐 구조는 필터와 클래스가 있는 형태로, 필터에 맞는 패킷이 해당 클래스의 큐에 삽입된다. 여러 필터가 하나의 클래스로 연결될 수도 있다. 여러 개의 클래스를 가진 큐의 예를 그림 5.1에 보였다.

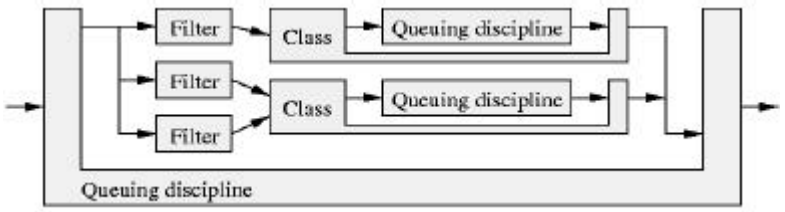


그림 5.1 단순한 큐 구조의 예

큐 구조의 enqueue 함수가 호출되면, 필터 중에 이 패킷이 맞는 것이 있는지 확인한다. 그래서 이 패킷은 필터에 맞는 클래스에 들어가게 된다. 만약

필터 중에 맞는 것이 없다면, 디폴트 클래스에 들어가게 된다. 일반적으로 하나의 클래스는 하나의 큐를 갖지만, 여러 개의 클래스가 하나의 큐를 사용할 수도 있다. 이 때 패킷에는 어떤 클래스로부터 들어 왔는지에 관한 정보를 갖지는 않는다. 보통, 패킷을 큐에 넣을 때 특정 플로우에 대해 정책제어(policing)를 할 수 있다. 다시 말하자면, 어떤 비율을 초과했을 때, 그 패킷을 무시할 수 있다.

2. Linux 트래픽 제어 관련 Resource

리눅스 커널은 /usr/src/linux 디렉토리 이하에 위치하고 있다. 또한 트래픽 제어를 위한 프로그램은 별도로 제공되며 iproute2/tc라는 디렉토리에 있다. tc라는 프로그램은 사용자 응용 프로그램으로, 리눅스 트래픽 제어부를 관리하는 데 사용되는 프로그램이다[44]. 이 소스는 iproute2-version-tar.gz 형태로 배포되며, ftp://ftp.inr.ac.ru/ip-routing이나 Mirror사이트 (ftp://ftp.gts.cz/pub/MIRRORS/ftp.inr.ac.ru/ip-routing/ 또는 ftp://ftp.sunsite/icm/edu/pl 등) 에서 다운로드 받을 수 있다. 커널소스 중에서 트래픽 제어에 관한 부분은 주로 /usr/src/linux/net/sched 디렉토리에 위치하며, 커널레벨과 유저레벨 사이의 인터페이스 부분은 /usr/src/linux/include/linux/pkt_cls.h와 /usr/src/linux/pkt_sched.h에 선언되어 있다. 또한 커널레벨과 유저레벨 사이의 통신 메커니즘을 제공하는 rtnetlink는 /usr/src/linux/net/core/rtnetlink.c와 /usr/src/linux/rtnetlink.h에 구현 되어 있다. 이 rtnetlink는 netlink에 기초하고 있으며, netlink 관련된 구현부분은 /usr/src/linux/net/netlink.c와 /usr/src/linux/include/linux/netlink.h 에 위치하고 있다.

리눅스 커널소스는 ftp://ftp.kernel.org/pub/linux/kernel에서 구할 수 있다.

3. Linux의 큐 구조(Queueing Disciplines)

리눅스 트래픽 제어에 관련된 키워드 및 파일명은 표. 5.1과 같이 요약된다

표 5.1 트래픽 제어부에 사용되는 키워드와 파일명

요소	tc 키워드	파일명 prefix	
		Kernel	tc
Queueing discipline	qdisc	sch_	q_
Class	class	(sch_)	(q_)
Filter	filter	cls_	f_

각각의 큐 구조는 다음과 같은 함수들을 제공한다. (include/net/pkt_sched.h의 struct Qdisc_ops를 참조)

enqueue 패킷을 큐 구조에 삽입하는 함수. 만약 큐 구조가 클래스를 가지고 있다면, enqueue 함수는 먼저 클래스를 선택한 후, 해당 큐 구조의 enqueue 함수를 호출한다.

dequeue 패킷을 실제로 전송하기 위하여 큐에서 패킷을 꺼낸다. 만약 보낼 패킷이 없거나 보낼 필요가 없을 경우에는 NULL를 리턴한다.

requeue 패킷을 다시 큐에 넣는 함수이다. dequeue에 의해 꺼내졌으나, 전송에 실패하였거나 어떤 문제로 다시 큐에 넣을 때 사용된다. 보통 큐의 맨 앞에 다시 넣어진다.

drop 큐에서 하나의 패킷을 꺼내어 없애버린다.

init 큐 구조를 초기화하고 설정한다.

reset 큐 구조를 초기 상태로 되돌린다. 모든 큐의 내용을 지우고, 타이머를 멈춘다. 클래스를 가지고 있으면 클래스에 연결된 모든 큐를 reset시킨다.

destroy 큐 구조를 제거한다. 모든 클래스와 필터를 없애고, 할당받은 모

든 자원들을 해제한다. 다만, 큐 구조 그 자체를 나타내는 구조체 만을 남긴다.

dump 관리에 사용되는 통계자료 등을 반환한다. 설정상태 및 변수상태를 알아보는 데 사용된다.

모든 함수들은 해당되는 struct Qdisc의 포인터를 이용하여 큐 구조를 참조한다. 패킷이 인터페이스에 들어갈 때 (net/core/dev/c의 dev_queue_xmit 함수), 디바이스의 큐 구조(include/linux/netdevice.c의 struct device의 qdisc필드)에 있는 enqueue함수가 호출된다. 그 다음에 dev_queue_xmit을 include/net/pkt_sched.h의 qdisc_wakeup을 호출하여, 방금 넣은 패킷을 전송하기 위한 시도를 한다. qdisc_wakeup함수는 즉시 net/sched/sch_generic.c의 qdisc_restart를 호출한다. qdisc_restart 함수는 큐 구조를 살펴보고, 패킷을 보내는 주 함수이다. 이 함수는 먼저 디바이스의 큐 구조로부터 패킷을 얻기 위한 시도를 한다. 만약 성공하면, 실제로 패킷을 보내기 위해 디바이스의 hard_start_xmit 함수를 호출한다. 어떤 이유로 보내는데 실패하게 되면, requeue함수로 패킷을 되돌려 놓는다. 또한 큐 구조체는 필요에 따라 (예를 들어 타이머의 시간 경과) qdisc_wakeup을 호출할 수도 있다. TBF(Token Bucket Filter)가 그 예이다. 또한, net/core/dev.c의 net_bh의 run_queues는 qdisc_restart를 호출할 수도 있다. net_bh는 리눅스 네트워킹 스택의 'bottom-half"핸들러로, 패킷이 큐 될 때 항상 동작한다. 그림 5.2에서 이를 간단하게 정리하여 보이고 있다.

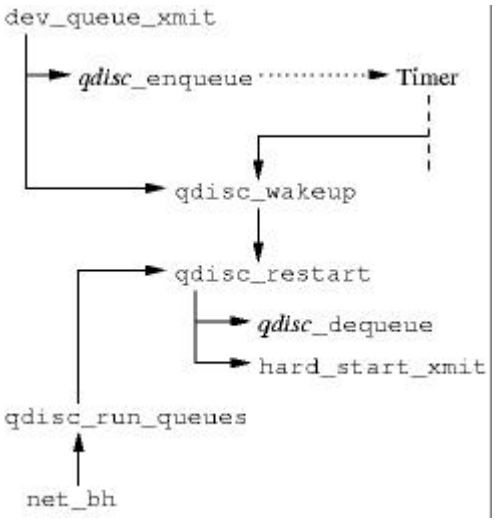


그림 5.2 패킷전송시 호출되는 함수

리눅스는 어떤 드라이버를 커널내부에 넣어 함께 컴파일 할 수 있고, 모듈형태로 따로 컴파일 하여 실행 시 필요할 때 붙여서 사용할 수 있다. 큐 구조가 모듈형태가 아닌 커널내부에 컴파일 된다면, net/sched/sch_api.c의 pkt_sched_init에 의해 등록된다. 또한 모듈로 컴파일 되었다면 init_module함수로부터 register_qdisc를 통해 등록된다. 큐 구조의 인스턴스가 만들어질 때, 옵션의 벡터가 init함수로 전달된다. 옵션의 벡터는 include/linux/rtnetlink.h에 선언된 struct rtattr *형태이다. 각 옵션은 그 타입과 값의 길이, 그리고 값으로 구성된다. 옵션 타입과 그 값에 사용되는 구조체는 include/linux/pkt_sched.h에 선언되어 있다. 옵션 벡터는 rtattr_parse를 호출하여 다시 나뉘어진다. rtattr_parse 함수는 옵션의 타입에 따라 색인된 각각의 요소의 포인터의 배열을 반환한다. 옵션의 길이는 RTA_PAYLOAD라는 매크로를 통해서 접근할 수 있으며, 옵션의 내용은 RTA_DATA를 통하여 접근할 수 있다. 옵션 벡터는 rtnetlink메커니즘을 사용하여 사용자 레벨과 커널 레벨 간에 전달된다.

큐의 인스턴스는 32비트 숫자 ID로 식별된다. 이 32비트숫자는 major number와 minor number로 나뉘어지며, 큐 구조에서 minor number는 항상 0이다. 이러한 숫자 표현은 보통 'major:minor'와 같은 형태로 표시한다.

4. 클래스 (Class)

클래스는 사용자에 의해 설정되는 class ID 또는 큐 구조에 의해 할당되는 internal ID로 식별된다. internal ID는 큐 구조에서 인덱스나 포인터 등으로 사용될 수 있으며 보통 포인터로 이용된다. 여기서 internal ID 0의 값은 "없음" 또는 "hot found"라는 특별한 의미를 갖는다. class ID는 u32형이며 internal ID는 unsigned long형태가 되며, 커널 내부에서는 internal ID를 이용하여 클래스를 참조한다. get 함수와 change 함수만이 class ID를 사용한다. 현재 구현에서는 하나의 Class ID는 하나의 클래스에 맞도록 되어 있다. Class ID 역시 큐 구조의 ID 형태로 표현되며, major number는 그 클래스가 속해 있는 큐 구조의 ID가 되며, minor number가 그 클래스를 식별하는 데 사용된다.

클래스를 지원하는 큐 구조는 다음과 같은 함수들을 제공한다. 이 함수들은 include/net/pkt_sched.h의 struct Qdisc_class_ops에 선언되어 있다. 참고로 FIFO와 같은 큐 구조는 클래스를 지원하지 않는 큐이므로 이 항목에 해당사항이 없다.

graft 새로운 큐를 클래스에 연결한다. 이 함수는 과거에 클래스에 연결되어 있던 큐 구조를 리턴한다.

get class ID를 이용해 클래스를 찾고, internal ID를 반환한다. 만약 클래스가 usage count를 사용한다면, 이를 증가시킨다.

put get에 의해 참조된 클래스가 역참조 될 경우, 이 함수가 호출된다. 만약 클래스가 usage count를 사용한다면 이를 감소시킨다. 만약 이 count가 0이 되면 put함수는 이 클래스를 제거한다.

change 클래스의 속성을 바꾸거나, 새로운 클래스를 만든다. 어떤 큐 구조는 초기화할 때 만들어지는 고정된 개수의 클래스를 갖는다. change함수로 전달된 옵션 벡터는 큐 구조의 init함수에 전달되는 벡터와 동일하며, include/

linux/pkt_sched.h에 그 선언이 되어 있다.

delete 클래스 삭제요구를 처리할 때 사용된다. 만약 클래스가 사용되고 있지 않으면, 클래스를 삭제한다.

walk 큐 구조체의 모든 클래스에 반복 수행하며, 각각의 callback함수를 호출한다. 이 함수는 큐 구조체의 모든 클래스의 분석 자료를 얻을 때 사용된다.

tcf_chain 클래스에 연결되어 있는 필터 리스트의 포인터를 반환 하는 함수이다.

bind_tcf 필터의 인스턴스와 클래스를 연결한다. bind_tcf는 큐 구조가 클래스 삭제를 명확히 거부할 필요가 있을 때를 제외하곤, get과 거의 동일하다. (예를 들어 sch_cbq는 필터에 의해 참조되고 있는 클래스의 삭제를 못하게 한다.)

unbind_tcf 클래스로부터 필터의 인스턴스를 제거한다. unbind_tcf함수는 put함수와 거의 유사하다.

dump_class 큐 구조의 dump함수와 마찬가지로 클래스의 분석자료를 리턴한다.

큐 구조의 enqueue함수는 보통 include/net/pkt_cls.h의 tc_classify를 호출하여 클래스를 선택한다. tc_classify함수는 include/net/pkt_cls.h에 선언된 struct tcf_result형태를 리턴한다. tcf_result는 class ID(classid), internal ID(class)를 포함한다. tc_classify함수의 리턴값은 -1이거나 정책결정(policing decision)값이 된다. 이는 include/linux/pkt_cls.h에 선언되어 있으며, 정책결정 결과 수락이 되면 정책결정값은 TC_POLICE_OK값인 0으로 반환한다.

리눅스에는 해당 클래스로 분류하는 간단한 방법이 있는데, 만약 skb->priority가 현재 큐 구조의 클래스의 ID와 같다면 다른 시도 없이 이를 사용하는 것이다. skb->priority는 패킷이 사용자레벨에서 만들어 질 때 sk->priority로 세트할 수 있다. skb변수는 include/linux/skbuff.h의 struct sk_buff구조체이다. 또한 이는 소켓에서 sk->priority로 표현되며, 이는

include/net/sock.h의 struct sock구조체이다. sk->priority는 소켓 옵션 SO_PRIORITY (net/core/sock.c)로 세트될 수 있다. 이러한 방법은 패킷 스케쥴러를 테스트하거나, 호스트에서 다른 패킷에 우선 순위를 줄 대 사용될수 있다. 그러나 이 skb->priority는 IPv4헤더의 TOS를 의미하기도 하므로, Diffserv에서는 사용할 수 없다.

enqueue함수가 클래스를 선택한 후에, 큐 구조 내부의 enqueue함수가 호출된다. 어떤 식으로 패킷을 저장할 것인가는 그 큐 구조의 구현에 따라 다르게 된다. 예를 들어 FIFO는 하나의 단일 버퍼에 패킷을 저장하게 되고, WF2Q+ 스케쥴러라면, 여러개의 버퍼중의 하나에 패킷이 저장될 것이다.

5. 필터(Filter)

필터는 큐 구조가 들어온 패킷을 어떤 클래스에 넣을 것인지 결정하는데 사용된다. 이것은 큐 구조가 enqueue 동작을 할 때 사용되며. 필터들은 각 큐 구조별 또는 클래스별로 관리되는 필터리스트에 위치하며. 우선순위에 의해 정렬되어 있다. 각 필터는 어떤 프로토콜이 사용되는지를 가지고 있다. 프로토콜 번호는 include/linux/if_ether.h에 정의되어 있으며, skb->protocol에도 사용되는 것이다. 같은 필터리스트의 같은 프로토콜의 필터들은 다른 우선순위를 가져야 한다.

필터는 내부구조를 가질 수 있다. 이는 32비트 핸들로 참조되어 제어할 수 있는 요소가 될 수 있다. 이러한 핸들은 classID와 유사하지만, major number와 minor number로 나뉘어지지는 않는다. 핸들 0은 항상 필터자체를 의미한다. 클래스와 마찬가지로 필터는 internal ID를 가지며, get 함수에 의해 얻어진다. 필터의 내부구조는 여러 가지 형태가 될 수 있다.

필터는 다음 함수로 제어된다. 이 함수들은 include/net/pkt_cls.h의 struct tcf_proto_ops에 정의된다.

classify 클래스화(패킷이 들어갈 클래스를 결정)를 수행하며, 정책결정의 결과 즉 TC_POLICE_.. 중의 하나를 리턴한다. 만약 결과가

TC_POLICE_UNSPEC이 아니면, struct tcf_result 안의 res에 선택된 class ID와 추가적으로 internal class ID가 리턴된다. 만약 internal class ID 없을 경우 res->class는 0의 값을 갖는다.

init 필터를 초기화한다.

destory 필터를 제거할 때 호출된다. 또한, sch_cbq 큐 구조체는 클래스를 제거할 때, 불필요하게 된 필터를 제거할 때 destroy함수가 사용된다.

get 핸들 값을 이용하여 필터를 찾는다. 내부 필터 ID를 반환한다.

put get에 의해 참조된 필터요소가 더 이상 사용되지 않을 때 호출된다.

change 새로운 필터를 구성하거나, 현재 있는 필터의 속성을 변경한다. 구성 파라메터는 큐 구조나 클래스의 경우와 마찬가지 메커니즘으로 전달된다.

delete 필터의 한 요소를 삭제한다. 전체 필터를 지우기 위해서는 destroy를 사용한다.

walk 필터의 모든 요소를 반복하여 callback함수를 호출한다. 이는 분석자료를 얻을 때 사용된다.

dump 필터나 필터의 한 요소로부터 분석자료를 얻을 때 사용된다.

리눅스에서는 RSVP필터를 제공하고 있는데, 실제코드는 cls_rsvp.h에 있으며, cls_rsvp.c와 cls_rsvp6.c는 단지 include만 하고 있거나, 몇몇 파라메터를 설정해 주는 부분 만을 포함하고 있다.

필터의 종류에 따라 그 인스턴스가 패킷을 어떤 클래스에 넣을 것인지 분류하는 범위가 틀리다. 예를 들어, cls_fw와 cls_route필터를 사용할 때, 큐 내부의 하나의 인스턴스는 패킷을 모든 클래스에 넣을 수 있다. 이러한 필터는 프로토콜 스택의 다른 요소에 의해 저장된 패킷 서술자로부터 class ID를 취한다. 예를 들면, cls_fw은 firewall코드의 marking기능을 사용한다. 이러한 필터를 일반(generic) 필터라고 한다. (그림 5.3)

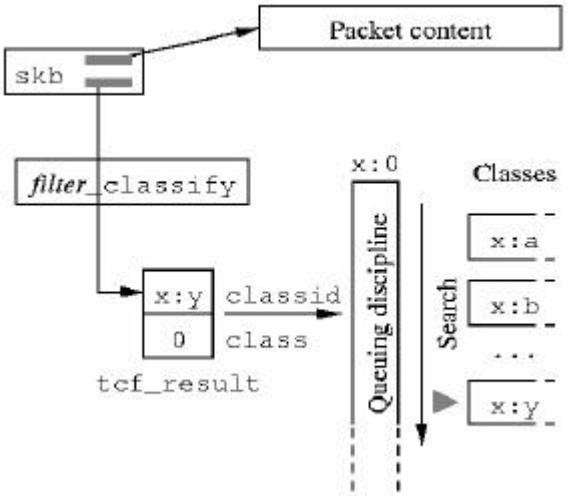


그림 5.3 Generic Filter

다른 종류의 필터(cls_rsvp와 cls_u32)는 클래스 내부에 필터리스트를 가지고 있다. 이러한 필터를 특정(specific) 필터라고 한다. 어떤 필터의 인스턴스는 internal filter ID로 구별되며 클래스에서의 internal ID와 유사하다. 그러나 클래스에서와는 달리 필터들은 internal filter ID만 갖고 'filter ID'를 갖지 않는다. 대신에, 어떤 것들이 등록되어 있고 어떤 우선순위를 갖는지는 큐 구조 또는 클래스에 의해 식별된다.

비록 특정필터(specific filter)가 클래스당 적어도 하나의 인스턴스나 요소를 갖기 때문에, 클래스 내부에 필터의 internal ID를 저장하고, 클래스화의 결과로 그것을 사용할 수 있다. 그림 5.4에 특정필터의 사용 예를 보이고 있다. 클래스 구조의 포인터가 internal ID로 사용되고 있다.

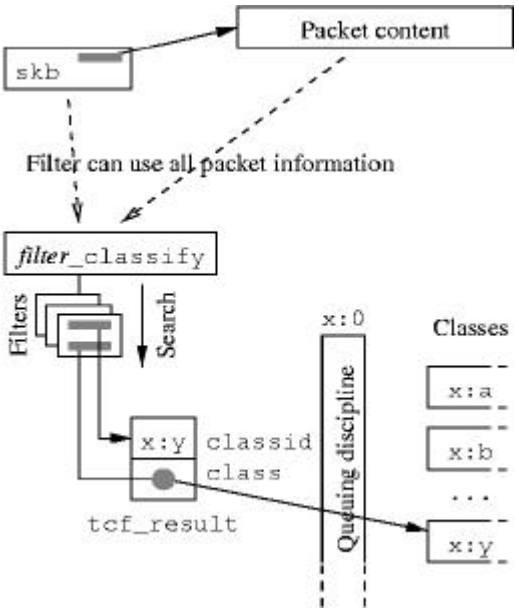


그림 5.4 Specific Filter