## 1. 분석 환경

Linux Kernel 버전: 2.4.13

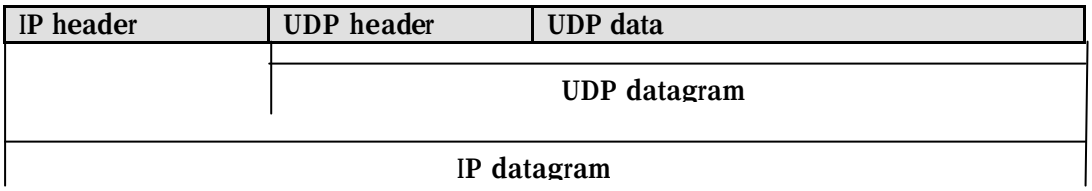
Processor: Intel x86 family

Ethernet Device : 3COM 3c509

## 2. Base Knowledge

### UDP(User Datagram Protocol) 란?

UDP 는 기본적으로 IP layer 상위에 위치하는 프로토콜로 OSI 7 layer에서 비교해본다면 transport layer 에 해당하는 위치에 존재한다.




이 프로토콜은 가장 단순한 형태의 프로토콜 중 하나로 헤더에는 단지 4가지의 필드만이 존재한다. 이들은 아래와 같다.

| 16-bit source port number | 16-bit destination port number |
|---|---|
| 16-bit UDP length | 16-bit UDP checksum |
| data | |

보는 바와 같이 단순한 데이터를 주고 받는데 꼭 필요한 필드들만 갖추어져 있다(물론 기본적인 checksum 필드는 존재). 따라서 할 수 있는 기능도 상당히 제한되어 있어 일반적으로 한 발신 호스트는 끊임없이 데이터를 보내기만 하며 수신 호스트에서는 계속 받기만 한다.

즉, 이 프로토콜로는 네트워크 상태에 따른 흐름 제어(flow control)이나 신뢰성 있는 데이터 전송이 이루어지지 않는다. 따라서 패킷 유실률이 상당히 높은 현재의 무선망 같은 경우에는 이용하기 힘든 프로토콜이라 할 수 있겠다.

반면 데이터 유실에 치명적이지 않은 서비스나 네트워크 망 자체가 에러 발생률이 극히 미비한 경우(ex> Optic cable)에 고속으로 데이터를 전송할 때 등은 유용하게 사용될 수 있다. 같은 layer에서 존재하는 TCP 와 비교해볼 때 다음과 같은 장점들이 존재한다.

- 헤더 길이가 짧기 때문에 실 데이터 전송률이 높다(헤더에 의한 오버헤드가 적다).
- 단방향 전송형태이기 때문에 네트워크 매체를 혼자 점유할 수 있다(동축 케이블과 같이 half duplexing 매체일 경우 특히 유용).
- 양 호스트가 고려해야 할 옵션 사항이 적기 때문에 단위 시간당 더 많은 패킷을 처리할 수 있다.
- 비연결형이기 때문에 다수의 호스트와 통신 작업이 간단하다.

이와 같은 장점들이 존재하기 때문에 나름대로의 활용 영역을 갖고 계속 사용되고 있다.

실제 활용 예를 찾자면 Local Network 에서 즐길 수 있는 네트워크 게임, 품질이 그리 중요하지 않은 실시간 스트리밍 서비스, 인트라넷 내에서의 화상회의 시스템 등을 들 수 있겠다.

## Device ←→ Kernel Area ←→ User Area

현재 널리 사용되고 있는 일반적은 운영체제는 시스템의 메모리를 커널 영역과 사용자 영역을 구분하여 사용하도록 하고 있다. 또한 NIC와 같은 하드웨어 디바이스들과의 인터페이스를 커널이 감싸고 있음으로 해서 사용자의 악의적인 접근 이나 실수를 방지해 주도록 설계되었다.

이와 같은 이유로 사용자는 디바이스들을 직접 제어하지 못하게 되었으므로 네트워크 통신의 경우도 커널에 그 작업을 요청해 커널이 대행하도록 해야 한다(이와 같이 커널에서 지원해주는 서비스를 요청하는 것을 system call 이라 한다). 당연히 사용자는 커널 메모리에 접근할 수 없으므로 사용자가 보내고자 하는 데이터는 사용자 영역의 메모리에 존재하게 되는데, 커널 요청된 데이터를 네트워크 디바이스를 통해 보내려면 해당 데이터가 자신의 영역 메모리에 위치해야 한다. 즉, 사용자 영역의 데이터를 커널 영역의 메모리로 복사하는 작업이 필요함을 알 수 있다.

다음으로 커널이 네트워크 디바이스를 통해 데이터를 보내는 과정도 살펴보자.
디바이스도 자신이 처리해야 할 데이터를 저장하는 메모리(버퍼)가 갖고 있으며 마찬가지로 그 안에 데이터 들어 있어야 전송을 하게 된다. 즉 커널 영역의 데이터를 네트워크 디바이스의 메모리로 복사하는 작업이 필요하다.

반대로 네트워크로부터 데이터를 받아들이는 과정일 경우
네트워크로부터 디바이스가 패킷을 수신했을 경우 커널에게 이를 알려주어야 한다. 패킷을 수신하면 인터럽트가 발생하며 이 처리는 디바이스 드라이버가 맡는다. 드라이버는 지정된 커널 영역 메모리에 패킷을 복사한 후 커널에게 처리를 넘긴다.

드라이버로부터 신호를 건네 받은 커널은 주어진 패킷을 분석하여 적절한 사용자 어플리케이션에게 전송해 주어야 하므로 이 과정에서 다시 커널 영역의 데이터가 사용자 영역의 메모리로 복사될 필요가 생긴다.

마지막으로 사용자 어플리케이션은 필요 시 커널로부터 수신된 데이터를 받아들이게 되는데 데이터를 전송할 때와 마찬가지로 system call을 통해 수행된다. 커널 영역에서 사용자 영역으로의 데이터 복사는 이 단계에서 일어나도 될 것이다.

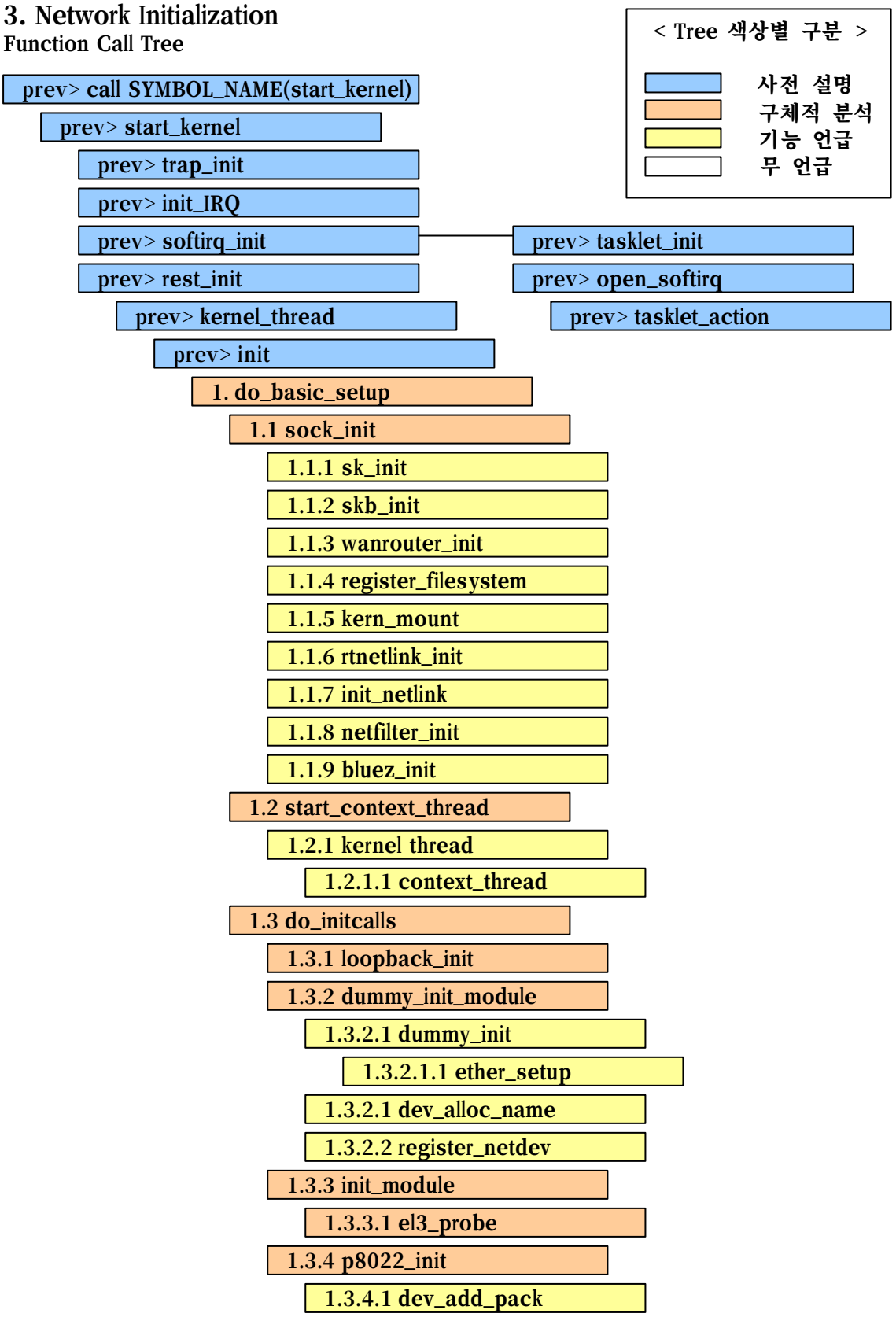
지금까지 살펴본 과정은 상당히 추상적인 것으로 실제 구체적인 처리 메커니즘은 뒤에 이어지는 커널의 소스 분석을 통해 파헤쳐볼 것이다.

**Comment**

| 커널 분석을 위해 필요한 추가적인 정보들은 분석하는 과정에서 이와 같은 별도의 다이얼로그를 통해 필요 시 적절히 첨부하도록 하겠다. |
|---|

## 3. Network Initialization

Function Call Tree



- 1.3.5 snap_init
  - 1.3.5.1 register_8022_client
- 1.3.6 inet_init
  - 1.3.6.1 sock_register
  - 1.3.6.2 inet_add_protocol
  - 1.3.6.3 arp_init
  - 1.3.6.4 ip_init
    - 1.3.6.4.1 dev_add_pack
    - 1.3.6.4.2 ip_rt_init
    - 1.3.6.4.3 proc_net_create
  - 1.3.6.5 tcp_v4_init
    - 1.3.6.5.1 init_waitqueue_head
  - 1.3.6.6 tcp_init
  - 1.3.6.7 icmp_init
    - 1.3.6.7.1 init_waitqueue_head
  - 1.3.6.8 ipip_init
  - 1.3.6.9 ipgre_init
  - 1.3.6.10 ip_mr_init
  - 1.3.6.11 proc_net_create
    - 1.3.6.11.1 create_proc_info_entry
      - 1.3.6.11.1.1 create_proc_entry
      - 1.3.6.11.1.2 ip_mc_procinfo
- 1.3.7 af_unix_init
  - 1.3.7.1 sock_register
  - 1.3.7.2 create_proc_read_entry
  - 1.3.7.3 unix_sysctl_register
- 1.3.8 netlink_proto_init
  - 1.3.8.1 sock_register
  - 1.3.8.2 create_proc_read_entry
- 1.3.9 packet_init
  - 1.3.9.1 sock_register
  - 1.3.9.2 register_netdevice_notifier
  - 1.3.9.3 create_proc_read_entry
- 1.3.10 atalk_init
  - 1.3.10.1 register_netdevice_notifier
  - 1.3.10.2 aarp_proto_init

Tree Analysis

Comment

Kernel 초기화 시작

인텔 32비트 x86 프로세서들은 arch/i386/kernel/**head.S** 에서부터 시작된다.
processor 정보 획득, page table directory/page table 초기화, paging 시작, SMP 여부 조사 등의 일을 처리한다.
arch/i386/kernel/head.S 파일에는 다음과 같은 코드가 포함되어 있다.

```
#ifdef CONFIG_SMP
        movb ready, %cl
        cmpb $1,%cl
        je 1f                   # the first CPU calls start_kernel
                                # all other CPUs call initialize_secondary
        call SYMBOL_NAME(initialize_secondary)
        jmp L6
1:
#endif
        call SYMBOL_NAME(start_kernel)
```

즉 첫 번째 CPU인 경우 무조건 start_kernel을 호출하며, SMP인 경우 그 외의 CPU들은 initialize_secondary 를 호출한다.
initialize_secondary 코드는 arch/i386/kernel/smpboot.c 에 구현되어 있으며 단순히 각 프로세서에 현 스레드의 stack pointer와 instruction pointer를 지정해줄 뿐이다.

다음으로 init/main.c 의 start_kernel [*asmlinkage void __init start_kernel(void) in init/main.c*] 코드가 실행된다. 이곳에서 architecture setup, trap[*trap_init()* in arch/i386/kernel/traps.c], IRQ, scheduler, softirq, time, console, memory, fork, 각종 cache, signal, ipc, smp 등을 초기화한 후 마지막으로 rest_init [*static void rest_init(void) in init/main.c*] 함수를 호출한다.

Help Comment ? trap init

trap_init() 함수는 divide_error, overflow, invalid_op, device_not_available, stack_segment, page_fault 등의 여러 에러들과 함께 system_call 을 interrupt vector에 등록한다. system_call은 index 80 (SYSCALL_VECTOR)에 위치하게 된다.

softirq_init() 함수는 32개의 tasklet들을 초기화하고 softirq vector의 TASKLET_SOFTIRQ와 HI_SOFTIRQ에 해당하는 softirq에 action 함수 포인터로 tasklet_action 함수를 설정한다.

Help Comment ? softirq & tasklet 메커니즘

softirq와 tasklet은 Linux Kernel이 2.4 버전으로 향상되면서 추가된 중요한 메커니즘으로 이전부터 존재하던 Bottom Half 메커니즘의 기능을 상당부분 대체한다. 특히 network stack 관리 부분을 기존의 Bottom Half에서 softirq가 담당하게 되었다.
Bottom Half는 여러 개의 CPU가 장착된 시스템이라도 한 순간에 하나의 CPU에서만 처리될 수 있었던 것에 반해 softirq 메커니즘은 각 CPU당 하나씩 처리될 수 있으므로 SMP환경에서 성능을 극대화시킬 수 있게 된다. 하지만 초기화 코드에서 보여지는 바와 같이 32개의 tasklet 만을 초기화고 있으므로 32개가 넘는 CPU를 장착한 시스템에서라도 32개 이상의 tasklet을 사용하는 것은 불가능하다.

rest_init 함수는 kernel_thread를 통해 init [*static int init(void * unused) in init/main.c*] 함수를 호출토록 하며, init 함수는 do_basic_setup [*static void __init do_basic_setup(void) in*

*init/main.c*]을 이곳에서 다시 sock_init [*void __init sock_init(void) in net/socket.c*] 함수를 호출해 네트워크 초기화를 시작한다.

1. do_basic_setup in init/main.c

→ architecture dependant 한 CPU 설정을 모두 마치고 PCI, SBUS, ISAPNP 등의 device 초기화를 수행한다.

→ 네트워크 초기화를 수행한다.

sock_init(), start_context_thread(), do_initcalls()

이 세 함수는 하나의 단위로 묶어 생각할 수 있다. 네트워크를 초기화하기 위해 process context가 필요하여 중간에 start_context_thread() 함수가 추가되었다. 실제 protocol 초기화도 do_initcalls() 에 의해 수행된다.

→ IRDA, PCMCIA 초기화를 마치고 함수를 복귀한다.

Comment

Network 초기화 시작

1.1 sock_init in net/socket.c

네트워크 초기화의 시작 부분이다.

```
for (I = 0; I < NPROTO; i++)
        net_families[i] = NULL;        ← protocol family table NULL로 초기화
                                         (kernel에 등록된 protocol이 없게된다)
sk_init();                             ← sock SLAB cache 초기화
skb_init();                            ← skbuff SLAB cache 초기화
wanrouter_init();                      ← WAN router 초기화
register_filesystem(&sock_fs_type);    ← protocol module 초기화
sock_mnt = kern_mount(&sock_fs_type);
rtnetlink_init();                      ← network link 초기화
init_netlink();
retfilter_init();                      ← network filter 초기화
bluez_init();                          ← Bluetooth 초기화
```

1.2 start_context_thread in kernel/context.c

context_thread를 실행시킨다. (context_thread를 kernel_thread의 첫번째 인자로 넘겨 실행한다)

스레드 수행이 완전히 종료된 후에 반환된다(*wait_for_completion(&startup)*).

1.3 do_initcalls in init/main.c

이제껏 수행이 미뤄졌던 각종 초기화 루틴들이 호출되는 곳이다.

. 아키텍처 종속적인 CPU 부분과 UID cache, Memory(CPU cache/swap/shared), File System, block/character device 등을 초기화한다.

. 다음으로 Network Device를 초기화한 후

. IDE, CD-ROM, PCI 초기화를 거쳐

. Network을 초기화한다.

본 정보는 커널이 컴파일 될 시 debug용으로 만들어내는 System.map 파일을 분석해 봄으로써 얻을 수 있다.

1.3.1 loopback_init in drivers/net/loopback.c

loopback device는 device list의 가장 처음에 위치한다.

Help Comment ? loopback device

loopback이란 자기 자신과의 통신을 의미하므로 이 경우 굳이 network device를 거칠 필요

가 없으며 그렇게 하면 오히려 시스템에 오버헤드만 발생시킬 뿐이다. 따라서 일반적인 네트워크 통신에서 loopback를 예외로 분리하여 처리한다.

### 1.3.2 dummy_init_module in drivers/net/dummy.c

각 dummy device의 초기화는 같은 파일의 dummy_init 함수를 통해 이루어진다.

```
dev_dummy.init = dummy_init;          ← device 초기화 함수 포인터 설정
SET_MODULE_OWNER(&dev_dummy);         ← device의 소유자 설정
err=dev_alloc_name(&dev_dummy,        ← dummy%d 라는 이름으로 device를 할
                   "dummy%d ");       당
err = register_netdev(&dev_dummy);    ← dev_dummy 를 net_device list에 등록
```

**Help Comment – dummy device**

dummy device 역시 loopback device와 같이 특수 목적으로 사용되는 가상의 device이다. dummy라는 단어가 의미하듯이 데이터를 보내도 응답이 없는 장치로 예를 들어 디버그 모드로 가동 중일 때는 standard output을 통해 디버그 정보를 보내고 실제 가동 모드일 경우 dummy device로 정보를 뿌리게 함으로써 손쉬운 모드 전환과 데이터 출력을 통한 오버헤드 최소화를 이룰 수 있다.

### 1.3.3 init_module in drivers/net/3c509.c

네트워크 디바이스를 초기화한다.

이는 장착된 디바이스에 적합한 코드가 수행되며 본 분석의 경우 3c509 라 가정하고 있기 때문에 drivers/net/3c509.c의 init_module 함수가 이에 해당된다.

```
while (el3_probe(0) == 0) {
  el3_root_dev->irq = irq[el3_cards];          ← device 정보 검사
  el3_root_dev->if_port = xcvr[el3_cards];     ← IRQ 설정
}                                              ← Interface port number 설정
```

#### 1.3.3.1 el3_probe in drivers/net/3c509.c

device가 어떤 bus(EISA/MCA 등)의 몇 번째 슬롯에 장착되어 있는가를 알아내고 device 정보를 설정하는 등의 일을 수행한다.

### 1.3.4 p8022_init in net/802/p8022.c

ETH_P_802_2 타입의 패킷 핸들러를 등록(dev_add_pack)한다.

### 1.3.5 snap_init in net/802/psnap.c

SNAP layer를 초기화한다.

소스 주석에 따르면 아직 IP 용으로는 SNAP layer를 사용하지 않는다고 한다.

### 1.3.6 inet_init in net/ipv4/af_inet.c

TCP/IP 초기화

```
(void) sock_register(&inet_family_ops);   ← SOCKET module에 inet_family_ops를
                                            link 시킨다.
for (p=inet_protocol_base; p != NULL; ) {
  inet_add_protocol(p);                   ← 지원되는 protocol 들을 등록한다.
}
arp_init();                               ← ARP 초기화 (2차 보고서 참조)
ip_init();                                ← IP 초기화
tcp_v4_init();                            ← TCP v4 초기화
tcp_init();                               ← TCP 초기화
icmp_init();                              ← ICMP 초기화
```

```
ipip_init();                                ← IPIP 초기화
ipgre_init();                               ← IPGRE 초기화
ip_mr_init();                               ← IP multicast router 초기화
proc_net_create ( "..", 0, ..._get_info);   ← raw, netstat, smp, sockstat, tcp, udp
proc_net_create ( "..", 0, ..._get_info);   에 해당하는 /proc entry를 생성
```

**1.3.6.6. ip_init** in net/ipv4/ip_output.c

```
dev_add_pack(&ip_packet_type);   ← ip packet에 핸들러 등록
ip_rt_init();                    ← ip routing table 초기화
proc_net_create( "igmp", 0,      ← IP multicast가 지원될 경우 igmp
ip_mc_procinfo);                 /proc entry를 생성
```

**1.3.7 af_unix_init** in net/unix/af_unix.c

BSD Unix socket을 초기화한다.
unix_family_ops를 SOCKET module에 등록하고 "net/unix" proc entry를 생성한다.

**1.3.8 netlink_proto_init** in net/netlink/af_netlink.c

netlink의 protocol을 초기화한다.
netlink_family_ops를 SOCKET module에 등록하고 "net/netlink" proc entry를 생성한다.

**1.3.9 packet_init** in net/packet/af_packet.c

raw packet socket를 초기화한다.
packet_family_ops를 SOCKET module에 등록.
"net/packet" proc entry를 생성.
packet_netdev_notifier를 notifier_chain에 등록한다.

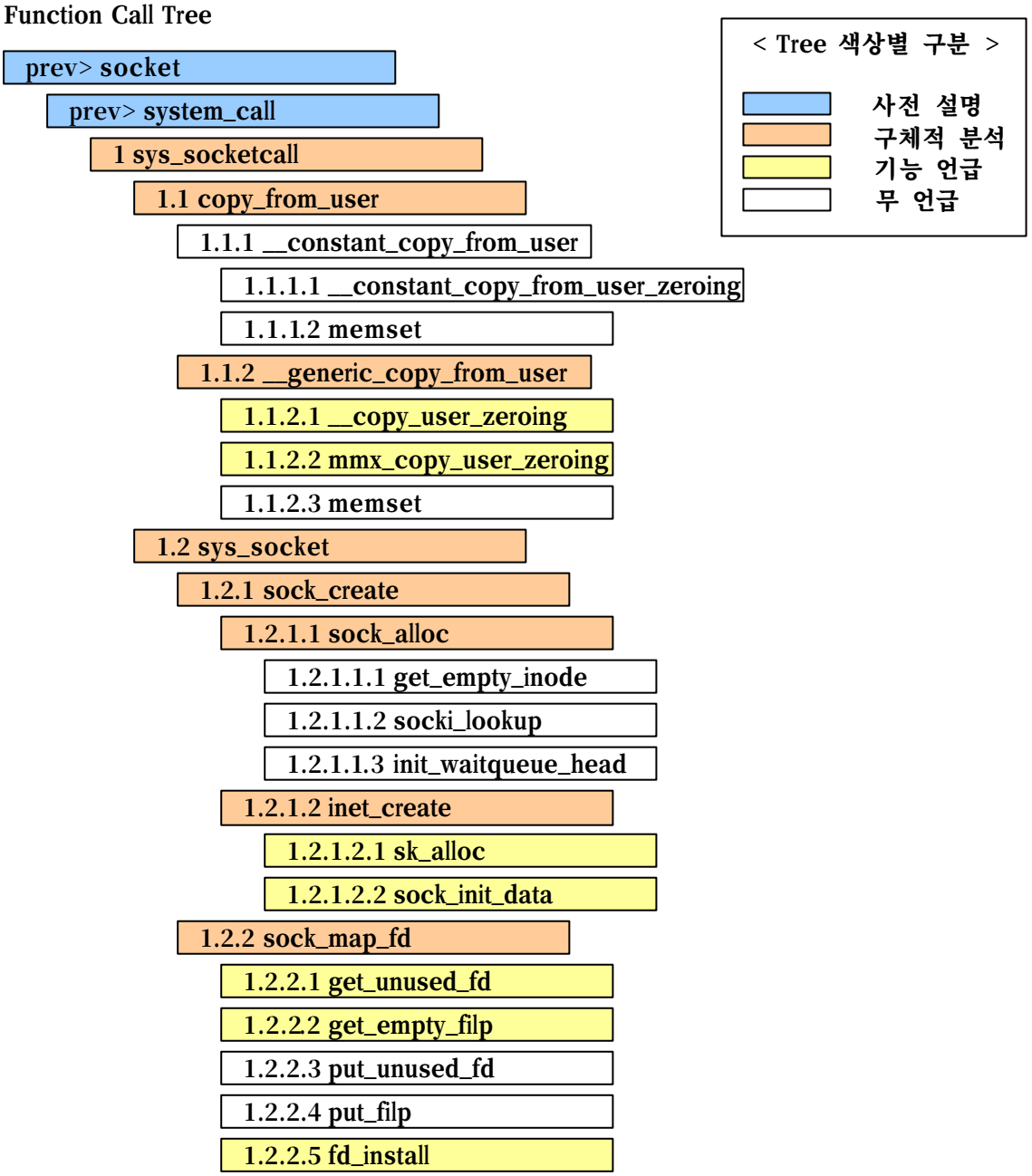
**1.3.10 atalk_init** in net/appletalk/ddp.c

Appletalk DDP protocol for Ethernet을 초기화한다.

```
sock_register(&atalk_family_ops);          ← atalk_family_ops를 SOCKET module
                                           에 등록.
dev_add_pack(ltalk_packet_type);           ← ltalk/ppptalk_packet_type 패킷 핸들
dev_add_pack(ppptalk_packet_type);         러를 등록.
register_netdevice_notifier(&ddp_notifier); ddp_notifier를 notifier_chain에 등록.
aarp_proto_init();                         ← appletalk용 ARP 초기화.
proc_net_create( "..", 0, ..._get_info);   ← "appletalk", "atalk_route",
                                           "atalk_iface" proc entry 생성
```

## 4. Socket Creation

Function Call Tree



Tree Analysis
소켓 생성을 위해 socket 함수[*socket(int domain, int type, int protocol)*]를 호출하게 되면 system call 이 발생(interrupt vector에서 index 0x80에 해당, "#define SYSCALL_VECTOR 0x80 "in include/asm-i386/hw_irq.h) 하게 된다. (현재 UDP에 초점을 맞추고 있으므로 이 경우 domain 에 AF_INET을, type 에 SOCK_DGRAM을, protocol 에 0(default 선택)을 대응) system_call 처리에 의해 sys_socketcall 함수가 호출되면서 소켓 생성 작업이 시작된다.

1. sys_socketcall in net/socket.c
→ *asmlinkage lon sys_socketcall(int call, unsigned long *args)*

```
if (copy_from_user(a, args, nargs[call]))
    return ?EFAULT;
switch(call)
{
    case SYS_SOCKET:
        err = sys_socket(a0, a1, a[2]);
        break;
   case SYS_BIND:
        …
   case SYS_CONNECT:
        …
    …
}
return err;
```

← User 영역(args)에서 Kernel 영역(a)로 nargs[call] 만큼의 데이터를 복사해온다.
← call의 값에 따라 action이 분기된다.
← 소켓 생성
← 소켓 바인딩
← 소켓 연결
← 그 외의 action (LISTEN, ACCEPT, SEND, SENDTO, RECV, RECVFROM 등)

소켓 생성의 경우 call 값으로 SYS_SOCKET 이 넘어오며 따라서 sys_socket 함수가 호출된다. 이 때 long array 타입 변수 a에는 처음부터 차례로 family, type, protocol 값이 들어가게 되며 이 값들이 그대로 sys_socket 함수의 입력인자로 넘겨진다.

Help Comment ? system call table
System call table은 arch/i386/kernel/entry.S 파일을 참조할 것

1.1 copy_from_user in include/asm-i386/uaccess.h
User Memory Area으로부터 Kernel Memory Area로 데이터를 복사해온다.
매크로 원형은 copy_from_user(to, from, n)으로, from에서 to로 n(size) 만큼의 데이터를 복사해 온다. 이중 n을 살펴 적절한 built-in 처리 루틴이 있다면 __constant_copy_from_user를, 그렇지 않다면 __generic_copy_from_user 함수를 호출한다. 이 둘의 주요 차이점은 __constant_copy_from_user는 inline 함수이기 때문에 수행 성능이 뛰어나다는 것이다.

1.1.2 __generic_copy_from_user in arch/i386/lib/usercoy.c
→ *unsigned long __generic_copy_from_user(void *to, const void *from, unsigned long n)*
inline 되지 않는 함수로 특이한 점은 AMD사의 3DNow 지원여부에 따라 다른 함수를 실행시킨다는 것이다. 3DNow 지원 시는 __copy_user_zeroing 대신 mmx_copy_user_zeroing 라는 3DNow에 최적화된 함수를 호출한다.

Help Comment ? 3DNow
3DNow 는 AMD 사가 선보인 SIMD 기술로 floating point number 처리 시에 한 번의 명령어로 여러 가지 기능을 처리할 수 있도록 해준다. 이 명령어 셋을 사용할 경우 일반적인

floating point 명령어로 처리하는 것 보다 향상된 수행능력을 보여 3D 게임이나 과학용 어플리케이션에 응용할 수 있다. 이에 대응되는 Intel 사의 기술로는 SSE와 SSE2 가 있는데, 3DNow의 경우 글을 쓰는 시점에서 가장 최신의 AMD 프로세서(Athlon XP)에서 지원되는 버전은 Intel 사의 SSE 와 호환되며 AMD의 차세대 프로세서인 코드명 Hammer 시리즈에서는 SSE2 버전까지 호환되는 3DNow를 집적시킬 것이라 말하고 있다.

### 1.2 sys_socket in net/socket.c

→ *asmlinkage long sys_socket(int family, int type, int protocol)*

입력값으로 넘어온 family, type, protocol 에 해당하는 socket struct를 생성하며 이를 유효한 file descriptor 과 연결한다. 가정상

### 1.2.1 sock_create in net/socket.c

→ *int sock_create(int family, int type, int protocol, struct socket **res)*

```
struct socket *sock;                                  ← 초기화되지 않은 socket 생성
if (family < 0 || family >= NPROTO)                   ← family와 type이 유효한 범위에 속하는
   return ?EAFNOSUPOORT;                                지 확인
if (type < 0 || type >= SOCK_MAX)
   return ?EINVAL;
if (net_families[family] == NULL) {                   ← 요청한 family가 시스템에서 지원되는
    I = -EAFNOSUPPORT;                                  지 확인
   goto out;
}
if (!(sock = sock_alloc())) {                         ← 소켓을 할당한다.
    …; goto out;
}
if ((I = net_families[family] ->create(sock,          ← net_families에서 family에 해당하는
protocol)) < 0) {                                     net_proto_family를 선택해 create 함수
    …; goto out;                                      포인터를 호출한다( 설정상
}                                                     inet_family_ops가 선택되며, inet_create
                                                      함수가 호출된다).
```

설정상 각각의 인자는 AF_INET, SOCK_DGRAM, default 값이 넘어온다.

#### 1.2.1.1 sock_alloc in net/socket.c

inode와 socket을 하나씩 생성한 후 이들을 bind 및 초기화시킨 후 socket을 반환한다. 만약 유효한 inode가 남아있지 않다면 NULL 값을 반환한다.

#### 1.2.1.2 inet_create in net/ipv4/af_inet.c

→ *static int inet_create(struct socket *sock, int protocol)*

INET socket을 생성한다.

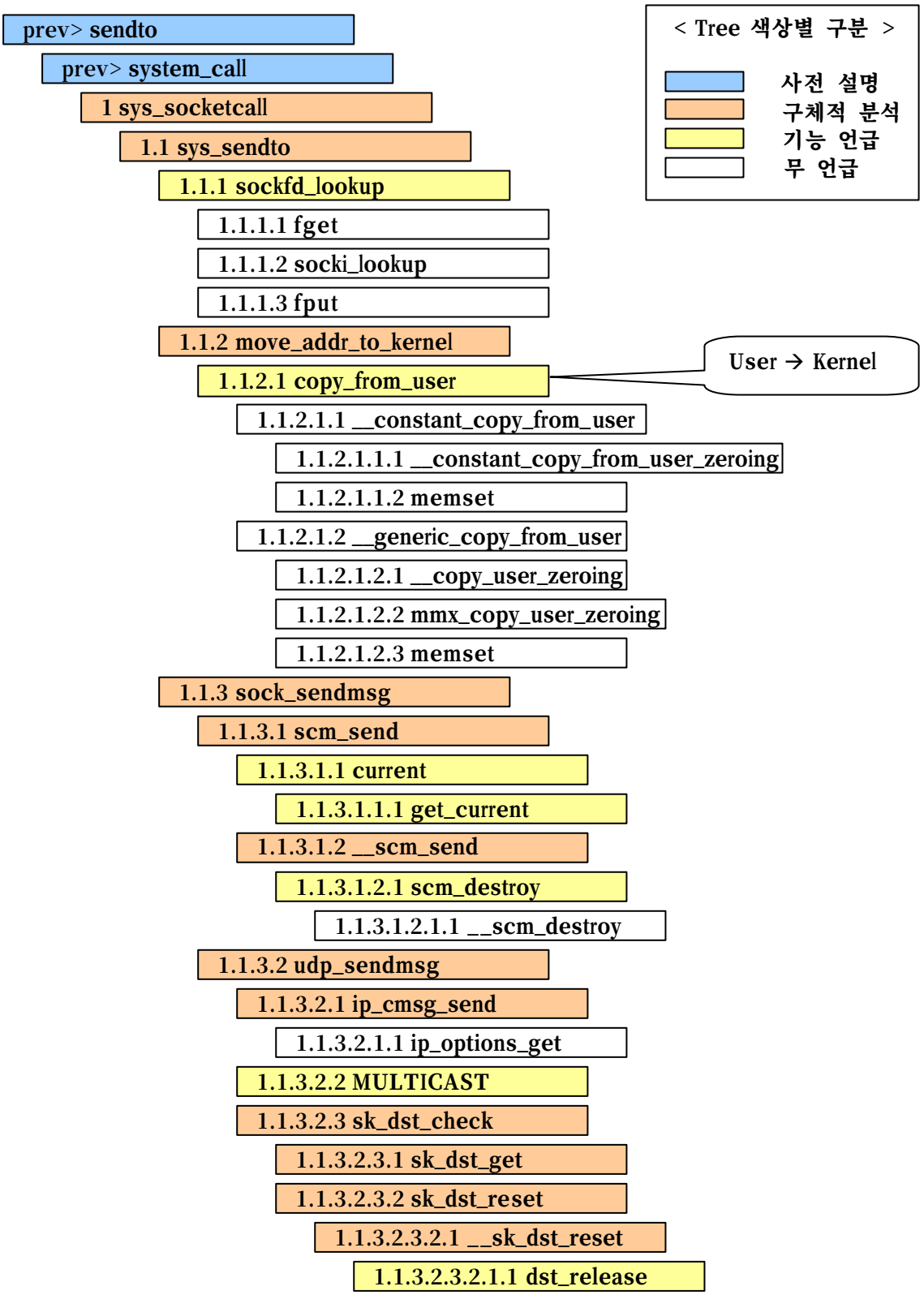
sock 구조체를 할당 받고[*struct sock *sk_alloc(int family, int priority, int zero_it) in net/core/sock.c*] 요청된 type/protocol pair에 맞도록 초기화한다.

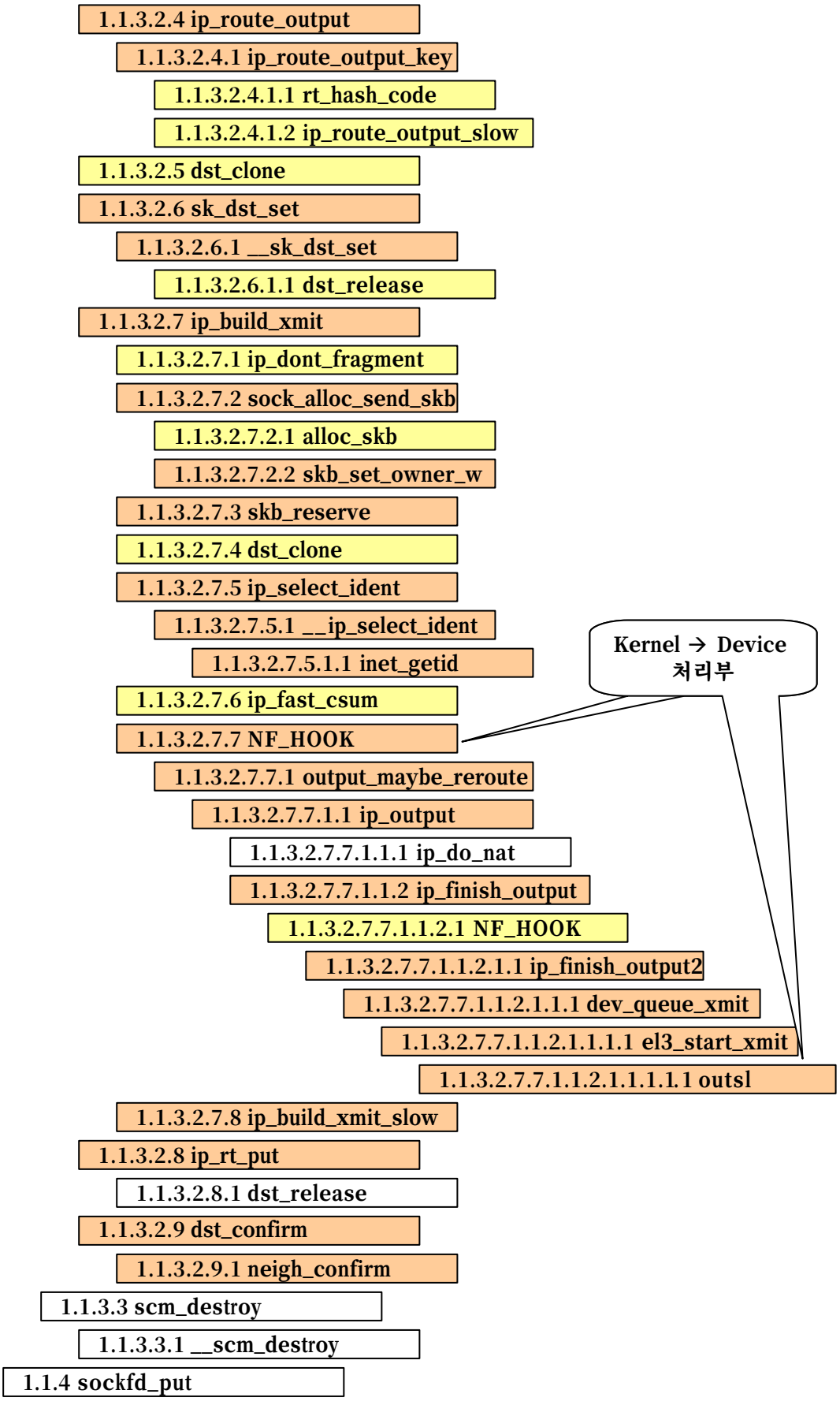
### 1.2.2 sock_map_fd in net/socket.c

```
fd = get_unused_fd ();                                ← 유효한 file descriptor를 얻는다.
struct file *file = get_empty_filp();                 ← 유효한 file pointer를 얻는다.
…                                                     ← 여기에 초기화 코드들이 들어간다.
…
fd_install(fd, file);                                 ← file을 fd array에 install 한다.
return fd;                                            ← 정상적으로 마쳤으면 fd를 반환
```

## 5. Data Sending

Function Call Tree




1.1.3.2.4 ip_route_output
1.1.3.2.4.1 ip_route_output_key
1.1.3.2.4.1.1 rt_hash_code
1.1.3.2.4.1.2 ip_route_output_slow
1.1.3.2.5 dst_clone
1.1.3.2.6 sk_dst_set
1.1.3.2.6.1 __sk_dst_set
1.1.3.2.6.1.1 dst_release
1.1.3.2.7 ip_build_xmit
1.1.3.2.7.1 ip_dont_fragment
1.1.3.2.7.2 sock_alloc_send_skb
1.1.3.2.7.2.1 alloc_skb
1.1.3.2.7.2.2 skb_set_owner_w
1.1.3.2.7.3 skb_reserve
1.1.3.2.7.4 dst_clone
1.1.3.2.7.5 ip_select_ident
1.1.3.2.7.5.1 __ip_select_ident
1.1.3.2.7.5.1.1 inet_getid
1.1.3.2.7.6 ip_fast_csum
1.1.3.2.7.7 NF_HOOK
1.1.3.2.7.7.1 output_maybe_reroute
1.1.3.2.7.7.1.1 ip_output
1.1.3.2.7.7.1.1.1 ip_do_nat
1.1.3.2.7.7.1.1.2 ip_finish_output
1.1.3.2.7.7.1.1.2.1 NF_HOOK
1.1.3.2.7.7.1.1.2.1.1 ip_finish_output2
1.1.3.2.7.7.1.1.2.1.1.1 dev_queue_xmit
1.1.3.2.7.7.1.1.2.1.1.1.1 el3_start_xmit
1.1.3.2.7.7.1.1.2.1.1.1.1.1 outsl
Kernel → Device
처리부
1.1.3.2.7.8 ip_build_xmit_slow
1.1.3.2.8 ip_rt_put
1.1.3.2.8.1 dst_release
1.1.3.2.9 dst_confirm
1.1.3.2.9.1 neigh_confirm
1.1.3.3 scm_destroy
1.1.3.3.1 __scm_destroy
1.1.4 sockfd_put

Tree Analysis

데이터 송신을 위해 sendto(sockfd, buff, len, dst_addr, addrlen) 함수를 호출하면 system call 이 발생(interrupt vector에서 index 0x80에 해당, "#define SYSCALL_VECTOR 0x80 "in include/asm-i386/hw_irq.h)하게 된다. system call interrupt handler에 의해 sys_socketcall 함수가 호출되면서 UDP 데이터 송신 과정이 수행된다.

1. sys_socketcall in net/socket.c

→ *asmlinkage lon sys_socketcall(int call, unsigned long *args)*

```
if (copy_from_user(a, args, nargs[call]))
    return ?EFAULT;
switch(call)
{
    case SYS_SOCKET:
        …
    …
    case SYS_SENDTO:
        err = sys_sendto(a0,(void *)a1,
             a[2], a[3], (struct sockaddr *)
              a[4], a[5]);
       break;
    …
}
return err;
```

← User 영역(args)에서 Kernel 영역(a)로 nargs[call] 만큼의 데이터를 복사해온다.
← call의 값에 따라 action이 분기된다.
← 소켓 생성
← 데이터 송신

UDP 데이터 송신의 경우 call 값으로 SYS_SENDTO 값이 넘어오며 따라서 sys_sendto 함수가 호출된다. 이 때 long array 타입 변수 a에는 처음부터 차례로 fd(file descriptor), *buff(pointer of data buffer), length , flags , addr(sockaddr type struct), addr_len 값이 들어오게 되며 이 값들이 그대로 sys_sendto 함수의 입력인지로 넘겨진다.

1.1 sys_sendto in net/socket.c

→ *asmlinkage long sys_sendto(int fd, void * buff, size_t len, unsigned flags, struct sockaddr *addr, int addr_len)*

Datagram 패킷을 주어진 주소로 전송한다.

```
struct socket *sock;
struct msghdr msg;
sock = sockfd_lookup(fd, &err);

…
if(addr)
{
   err = move_addr_to_kernel(addr,
                        addr_len, address);
}
err = sock_sendmsg(sock, &msg, len);
```

← socket 구조체 생성
← message header 구조체 생성
← file descriptor(file handle)에 해당하는 socket slot을 찾는다.
← msg 초기화
← socket address를 User 영역으로부터 Kernel 영역으로 데이터를 복사해온다.
← 메시지를 전송한다.

1.1.2 move_addr_to_kernel in net/socket.c

→ *int move_addr_to_kernel(void *uaddr, int ulen, void *kaddr)*

```
if(ulen<0||ulen>MAX_SOCK_ADD)
   return ?EINVAL;
```

← length 가 유효범위 밖에 있다면 에러값을 반환

```
if(ulen==0)
   return 0;
if(copy_from_user(kaddr,uaddr,ulen))
   return ?EFAULT;
return 0;
```

← length 값이 0 이면 복사할 필요가 없다.
← 그렇지 않다면 User 영역의 address를 Kernel 영역으로 복사한다.

copy_from_user 호출 시 주어진 입력인자 kaddr, uaddr, ulen이 각각 copy_from_user 함수의 to, from, size 인자에 해당된다. 즉 uaddr 에서 ulen 만큼의 데이터가 kaddr 로 복사된다.
copy_from_user 함수에 대해서는 Socket Creation 장을 참조한다.

### 1.1.3 sock_sendmsg in net/socket.c

→ *int sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)*

```
struct scm_cookie scm;
err = scm_send(sock, msg, &scm);
if (err >= 0) {
    err = sock->ops->sendmsg(sock,
       msg, size, &scm);
    scm_destroy(&scm);
}
```

← scm_cookie 구조체 생성
← scm_send 호출(별도 설명)

← sock->ops 의 sendmsg 함수 포인터 실행, 설정상 sock는 UDP 타입이므로 sendmsg는 udp_sendmsg 함수를 가리킨다.

설정상 sock는 UDP 타입이므로 sendmsg는 udp_sendmsg 함수를 가리킨다.

#### 1.1.3.1 scm_send in include/net/scm.h

→ *static __inline__ int scm_send(struct socket *sock, struct msghdr *msg, struct scm_cookie *scm)*

```
scm->creds.uid = current->uid;
scm->creds.gid = current->gid;
scm->creds.pid = current->pid;
return __scm_send(sock, msg, scm);
```

← scm의 uid(user ID), gid(group ID), pid(process ID)를 task의 uid, gid, pid로 설정
← __scm_send 함수로 나머지 작업 위임

변수처럼 보이는 current는 사실 include/asm_i386/curreht.h 에 정의되어 있는 매크로로 같은 파일의 get_current(void) 함수를 가리킨다. 이 함수는 현재 task의 task_struct 를 반환한다.

#### 1.1.3.1.2 __scm_send in net/core/scm.c

→ *int __scm_send(struct socket *sock, struct msghdr *msg, struct scm_cookie *p)*

```
for (cmsg = CMSG_FIRSTHDR(msg);
     cmsg;
     cmsg=CMSG_NXTHDR(msg, cmsg))
{
    if (cmsg->cmsg_len < … )
       goto error;
   if (cmsg->cmsg_level!=SOL_SOCKET)
       continue;
   switch (cmsg->cmsg_type) {
        case SCM_RIGHTS:
            …
        case SCM_CREDENTIALS:
            …
        default:
            goto error;
    }
```

← 주어진 message header(msg)의 각각의 control message header들에 대해

← message length가 유효한 지 검사

← message level이 유효한 지 검사

← message type에 따라
← rw: access rights (array of int)

← rw: struct ucred

← 위 둘 이외에는 error 로 처리한다.

```
    return 0;
error:
    scm_destroy(p);
}
```

← 에러가 발생했을 경우
← 데이터를 초기화하고 자원을 회수한다. __scm_destory 함수를 호출한다.

socket level control message type에 대해서는 include/linux/socket.h에 정의되어 있다. 위 두 값 외에 SCM_CONNECT 가 더 정의되어 있다.

### 1.1.3.2 udp_sendmsg in net/ipv4/udp.c

→ *int udp_sendmsg(struct sock *sk, struct msghdr *msg, int len)*

```
struct ipcm_cookie ipc;
struct udpfakehdr ufh;
struct rtable *rt = NULL;
…

if (msg->msg_controllen) {
    err = ip_cmsg_send(msg, &ipc);
     …
}
…
if (MULTICAST(daddr))
     …

if (connected)
     rt = (struct rtable*)sk_dst_check(sk,0);
…
if (rt == NULL)
    err = ip_route_output(&rt, daddr,
            ufh.saddr, tos, ipc.oif);
     …
    if (rt->rt_flags&RTCF_BROADCAST &&
                !sk->broadcast)
        goto out;
    if (connected)
         sk_dst_set(sk,
                  dst_clone(&rt->u.dst));




if (msg->msg_flags&MSG_CONFIRM)
    goto do_confirm;
back_from_confirm:
…
err = ip_build_xmit(sk,
     (sk->no_check==UDP_CSUM_NOXMIT
      ? udp_getfrag_nosum
      : udp_getfrag),
     &ufh,ulen, &ipc, rt, msg->msg_flags);

out:
```

← 주어진 len, msg의 유효성 검사 및 데이터 설정
← ip control message가 유효하다면 전송

← 목적지 주소가 multicast 주소일 경우 ipc.oif와 ufh.saddr 필드 설정, 비연결형으로 설정
← 연결형 UDP 일 경우 처리
← sk의 destination entry를 취해 rtable 형태로 cast하여 rt에 할당한다.
← rtable의 아직 없다면
← rt를 새로 추가한다.

← 방송(broadcast)용 패킷일 경우 마무리 처리 단계로 건너뛴다.

← 연결형 UDP일 경우 routing table에 있는 목적지 주소를 현 socket에 그대로 복사한다. 이 부분에서 목적지 주소를 정상으로 수정해 주기 때문에 간혹 잘못된 주소값이 주어진 패킷도 원래 연결되어 있는 상대방에게 재대로 전송될 수 있게 된다.

← Network path 확인용 패킷일 경우 path 확인 루틴으로 건너뛴다.

← UDP header(ufh) 설정
← 패킷을 전송한다.
← socket의 no_check 필드가 UDP_CSUM_NOXMIT 로 설정되어 있다면 check sum을 요구하지 않는다는 의미이므로 check sum을 피하는 루틴이 대신 실행된다.

```
ip_rt_put(rt);
…
do_confirm:
dst_confirm(&rt->u.dst);
…
```

← routing table rt를 제거한다.
← 자원 회수 및 반환
← path confirm용 패킷 처리
← rt의 목적지 주소의 path을 확인한다.
← msg->msg_flags에 따라 out 이나 back_from_confirm 으로 goto된다.

**1.1.3.2.1 ip_cmsg_send** in net/ipv4/ip_sockglue.c

→ *int ip_cmsg_send(struct msghdr *msg, struct ipcm_cookie *ipc)*

이 함수의 동작방식은 __scm_send 함수(1.1.3.1.2)와 유사한다.
주어진 msg의 각 control message에 대해 length와 level 의 유효성을 검사하고 type에 따른 데이터 초기화 작업을 수행한다.
control message type으로는 include/linux/in.h 에 정의되어 있는 IP_RETOPTS와 IP_PKTINFO 만이 유효하며 나머지는 모두 에러로 처리된다.

**1.1.3.2.3 sk_dst_check** in include/net/sock.h

→ *static inline struct dst_entry * sk_dst_check9struct sock *sk, u32 cookie)*

주어진 sk의 destination 필드가 유효한지 검사하여 sk의 dst_cache 의 값을 갖는 dst_entry를 만들어 반환한다.
값이 유효하지 않을 경우 sk의 의 destination field를 초기화하고 NULL을 반환한다.

**1.1.3.2.3.1 sk_dst_get** in include/net/sock.h

→ *static inline struct dst_entry * sk_dst_get(struct sock *sk)*

주어진 sk의 dst_cache 의 값을 갖는 새로운 dst_entry를 만들어 반환한다.

**1.1.3.2.3.2 sk_dst_reset** in include/net/sock.h

→ *static inline void sk_dst_reset(struct sock *sk)*

동기화하여 __sk_dst_reset(sk)를 호출한다.

**1.1.3.2.3.2.1 __sk_dst_reset** in include/net/sock.h

→ *static inline void __sk_dst_reset(struct sock *sk)*

주어진 sk의 dst_cache 필드를 NULL로 초기화하고 이전의 dst_cache 자원을 회수한다(dst_release).

**1.1.3.2.4 ip_route_output** in include/net/route.h

→ *static inline int ip_route_output(struct rtable **rp, u32 daddr, u32 saddr, u32 tos, int oif)*

{ dst :daddr, src:saddr, oif:oif, tos:tos } 로 rt_key를 만들어 routing table에 추가한다(ip_route_output_key 함수 호출).

**1.1.3.2.4.1 ip_route_output_ key** in net/ipv4/route.c

→ *int ip_route_output_key(struct rtbale **rp, const struct rt_key *key)*

주어진 key값으로 hash code를 만들어[*rt_hash_code() in net/ipv4/route.c*] rt_hash_table의 해당 슬롯의 chain에 새로운 entry를 추가한다.
상태에 따라 ip_route_output_slow 함수로 넘겨져 나중에 처리될 수도 있다.

**1.1.3.2.6 sk_dst_set** in include/net/sock.h

→ *static inline void sk_dst_set(struct sock *sk, struct dst_entry *dst)*

동기화하여 __sk_dst_set(sk, dst)를 호출한다.

**1.1.3.2.6.1 __sk_dst_set** **in include/net/sock.h**

→ *static inline void __sk_dst_set(struct sock *sk, struct dst_entry *dst)*

__sk_dst_reset 과 유사한 구조의 함수이다.

sk의 이전 dst_cache의 자원을 해제(dst_release)한 후, 이를 초기화하는 대신 주어진 dst 값을 할당한다.

**1.1.3.2.7 ip_build_xmit** **in net/ipv4/ip_output.c**

→ *int ip_build_xmit(struct sock *sk,*
*int getfrag( const void *, char *, unsigned int, unsigned int),*
*const void *frag, unsigned length, struct ipcm_cookie *ipc,*
*struct rtable *rt, int flags )*

단편화(fragmentation)되지 않은 패킷을 전송한다.

```
struct sk_buff *skb;
struct iphdr *iph;
if (!sk->protoinfo.af_inet.hdrincl) {
    …
   if (length > rt->u.dst.pmtu
      || ipc->opt != NULL)
      return ip_build_xmit_slow(…);




} else {
   if (length > rt->u.dst.dev->mtu)
       …
}
if (ip_dont_fragment(sk, &rt->u.dst))
   df = htons(IP_DF);
…
…
…
…
skb = sock_alloc_send_skb(…);
skb_reserve(skb, hh_len);

skb->priority = sk->priority;
skb->dst = dst_clone(&rt->u.dst);
skb->nh.iph = iph = (struct iphdr
                  *)skb_put(skb, length);
if(!sk->protinfo.af_inet.hdrincl) {
    iph->version=4;
    iph->ihl=5;
    ...
    iph->check = ip_fast_csum(…);
}
…
err = NF_HOOK(PF_INET,
   NF_IP_LOCAL_OUT, skb, NULL,
    rt->u.dst.dev, output_maybe_reroute );
```

← ip header
← socket에서 af_inet의 header가 포함되어 있지 않을 경우
← 패킷 전체 길이(헤더 포함)가 HOST의 mtu 보다 크거나 control message에 option이 지정돼 있지 않을 경우 ip_build_xmit_slow(…) 함수를 호출한다. 이 함수는 패킷을 단편화시켜 전송하는 기능을 수행하며 자세한 설명은 별도로 하겠다.

← 패킷 길이(헤더 제외) 길이가 상대 device의 mtu 보다 클 경우 에러 처리

← 목적지까지의 path에서 이 패킷의 단편화되지 않고 전송될 수 있는지 확인한 후 가능하다면 그렇게 설정한다. IP_DF는 include/net/ip.h에 정의되어 있는 IP flags로 "Don't Fragment"를 의미한다. df 변수값은 iph->frag_off에 할당된다.
← 패킷 전송용 buffer를 할당받는다.
← skb의 data, tail 필드값을 헤더 길이만큼 증가시킨다.
← 패킷의 우선순위 설정
← 패킷의 목적지 설정
← 패킷의 ip header 설정

← 패킷이 헤더를 포함할 경우
← iph의 각 필드들을 설정한다. IP header의 구조를 참조하면 이해하기 쉽다.

← hooking, output_maybe_reroute 함에 skb를 넘긴다. 이 과정에서 Kernel의 데이터가 Device로 넘겨진다.

```
…
```

1.1.3.2.7.2 sock_alloc_send_skb in net/core/sock.c

→ *struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size, int noblock, int *errcode)*

이 함수는 범용적인 send/receive buffer 처리 함수이다.

sk_buffer를 할당받아(alloc_skb) 쓰기 버퍼 해제 함수의 포인터를 설정(skb_set_owner_w)하여 반환한다.

1.1.3.2.7.2.2 skb_set_owner_w in include/net/sock.h

→ *static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)*

skb의 sk 필드에 sk를 할당하고 destructor

```
skb->sk = sk;                          ← sk 할당
skb->destructor = sock_wfree;          ← 쓰기 버퍼 해제 함수의 포인터 설정
atomic_add(skb->truesize,              ← sk의 wmem_alloc에 skb->truesize를
           &sk->wmem_alloc);             원자적으로 더한다.
```

쓰기 버퍼 해제 함수인 sock_wfree는 net/core/sock.c 에 정의되어 있으며 kfree_skb로부터 자동으로 호출된다.

1.1.3.2.7.3 skb_reserve in include/linux/skbuff.h

→ *static inline int skb_skb_reserve(struct sk_buff *skb, unsigned int len)*

패킷의 headroom을 조정하는 기능을 수행한다.

주어진 skb의 data와 tail의 값을 len 만큼 증가시킨다.

1.1.3.2.7.5 ip_select_ident in include/net/ip.h

→ *static inline void ip_select_ident(struct iphdr *iph, struct dst_entry *dst, struct sock *sk)*

ip header의 id 필드(identification)의 값을 결정하는 함수로 __ip_select_ident 함수를 호출한다.

1.1.3.2.7.5.1 __ip_select_ident in net/ipv4/route.c

→ *void __ip_select_ident(struct iphdr *iph, struct dst_entry *dst)*

routing table(주어진 dst를 rtable 형으로 캐스팅)에서 peer를 얻어[extern inline __u16 inet_getid(struct inet_peer *p) in include/net/inetpeer.h] 그 id를 해더의 id로 할당한다.

1.1.3.2.7.5.1.1 inet_getid in include/net/inetpeer.h

→ *extern inline __u16 inet_getid(struct inet_peer *p)*

주어진 p의 ip_id_count 필드값을 반환한다.

이 때 자신은 1 증가된 ip_id_count 값을 유지하게 된다.

Help Comment

여기부터 Kernel의 데이터가 어떻게 Device로 넘겨지는지를 보여준다.

1.1.3.2.7.7 NF_HOOK in include/linux/netfileter.h

→ *#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) (okfn)(skb)*

넘겨진 output_maybe_reroute 함수를 skb를 인자로 넘겨 호출한다.

나머지 인자들은 debug 시에 쓰이는데 debug 모드에서는 NF_HOOK가 nf_hook_slow 함수[in net/core/netfilter.c]로 선언된다.

1.1.3.2.7.7.1 output_maybe_reroute in net/ipv4/ip_output.c

→ *static inline int output_maybe_reroute(struct sk_buff *skb)*
단순히 넘겨진 skb의 목적지로 데이터를 전송하는 함수 포인터를 호출[*skb->dst->outout(skb)*]한다. 이 함수 포인터는 ip_output 포인터를 가리킨다.

**1.1.3.2.7.7.1.1 ip_output** in net/ipv4/ip_output.c
→ *int ip_output(struct *skb)*
ip_finish_output(skb) 를 호출

**1.1.3.2.7.7.1.1.2 ip_finish_output** in net/ipv4/ip_output.c
→ *__inline__ int ip_finish_output(struct sk_buff *skb)*
skb에 dev와 protocol 필드 설정 후 다시 network filter hooking(즉, NF_HOOK)을 실행한다. 이 때 실행 함수 포인터로 ip_finish_output2 를 지정한다.

**1.1.3.2.7.7.1.1.2.1.1 ip_finish_output2** in net/ipv4/ip_output.c
→ *static inline int ip_finish_output2(struct sk_buff *skb)*
skb의 dst(destination)의 hh(hh_cache 즉 버퍼)가 유효할 경우 hh의 output 함수[이 경우 dev_queue_xmit()]를 호출해 skb를 내보낸다.

**1.1.3.2.7.7.1.1.2.1.1.1 dev_queue_xmit** in net/core/dev.c
→ *int dev_queue_xmit(struct sk_buff *skb)*
주어진 버퍼를 목적 디바이스(skb->dev)로 보낸다.

```
struct Qdisc *q;
…
if (skb->ip_summed == CHECKSUM_HW
    && … ) {
  if ((skb = skb_checksum_help(skb))
                                == NULL )
    return ?ENOMEM;
}
q = dev->qdisc;
if (q->enqueue) {
  int ret = q->enqueue(skb, q);
  …
  return …;
}
…
```

← 패킷의 checksum 처리가 되어 있지 않고 디바이스가 이 패킷의 프로토톨에 맞는 하드웨어적 checksum 메커니즘을 지원하지 않는다면 이 곳에서 checksum을 수행한다.

← 디바이스의 큐를 얻어
← 디바이스의 큐에 패킷을 넣는 함수가 설정되어 있을 경우 이를 통해 패킷을 큐에 넣는다. (설정상 3COM 3c509 디바이스를 사용하므로 el3_start_xmit 함수가 호출된다.)
← 디바이스에 큐가 없는 경우 처리(일반적으로 loopback 과 같은 software device가 이런 경우에 해당된다).

설정상 3COM 3c509 디바이스의 드라이버에 정의되어 있는 el3_start_xmit 함수를 호출하게 된다.

**1.1.3.2.7.7.1.1.2.1.1.1.1 el3_start_xmit** in drivers/net/3c509.c
→ *start int el3_start_xmit(struct sk_buff *skb, struct net_device *dev)*

```
struct el3_private *lp
          = (struct el3_private *)dev->priv;
…
netif_stop_queue (dev);

lp->stats.tx_bytes += skb->len;
…
```

← driver specific 한 자료구조

← device의 queue의 사용을 잠시 멈춤 (race condition 을 피하기 위해)
← 전송할 바이트 수(패킷 길이)를 설정

```
outw(skb->len, ioaddr + TX_FIFO);
…
outsl(ioaddr + TX_FIFO, skb->data,
                        (skb->len + 3) >> 2);
```

← 디바이스에 전송할 데이터의 주소의 길이를 설정
← 커널 영역에 있는 패킷을 디바이스로 실제로 복사하는 루틴이다. 각 인자는 차례대로 port, src, count 이다.

**1.1.3.2.7.7.1.1.2.1.1.1.1.1 outsl** in include/asm_sh/io.h

→ *void outsl(unsigned long port, void *src, unsigned long count)*

Kernel 영역으로부터 Device로 주어진 길이 만큼 데이터를 복사해 넣는다.
호출 시 코드는 다음과 같다.
outsl(ioaddr + TX_FIFO, skb->data, (skb->len + 3) >> 2);
즉, "skb->data"로부터 "(skb->len+3) >> 2"만큼의 데이터를 "ioaddr + TX_FIFO"로 복사해 넣는다.

**1.1.3.2.7.8 ip_build_xmit_slow** in net/ipv4/ip_output.c

→ *int ip_build_xmit(struct sock *sk,*
*int getfrag( const void *, char *, unsigned int, unsigned int),*
*const void *frag, unsigned length, struct ipcm_cookie *ipc,*
*struct rtable *rt, int flags )*

이 함수의 동작 방식이나 기능은 ip_build_xmit 함수와 상당히 유사하다. 차이점은 ip_build_xmit 함수가 단편화되지 않은 패킷을 전송하는데 비해 이 함수는 단편화된 패킷을 전송한다는 것이다.

```
…
fraglen = length - offset + fragheaderlen;
…
id = sk->protinfo.af_inet.id++;

do {
   char *data;
   struct sk_buff *skb;
   skb = sock(alloc_send_skb( … ) );
    …
   data = skb_put(skb, fraglen);
    …
} while (offs et >= 0);
…
```

← 단편화된 패킷의 길이를 정한다.

← 단편화된 패킷들은 모두 원래 패킷의 id를 갖는다.
← 단편화된 패킷들을 전송한다.

← buffer를 할당 받는다.

← 정해진 길이만큼만 데이터를 복사한다.

← 데이터를 모두 보낼 때까지 반복

**1.1.3.2.8 ip_rt_put** in include/net/route.h

→ *static inline void ip_rt_put(struct rtable * rt)*

주어진 rt의 dst 필드(dst_entry) 자원을 해제(dst_release(&rt->dst))한다.

**1.1.3.2.9 dst_confirm** in include/net/dst.h

→ *static inline void dst_confirm(struct dst_entry *dst)*

목적지 entry가 올바른 값이 유효할 경우 dst의 neighbour 필드를 neigh_confirm 함수에 넘겨 확인한다.

**1.1.3.2.9.1 neigh_confirm** in include/net/neighbour.h

→ *static inline int neigh_confirm(struct neighbour *neigh)*

넘겨진 neigh가 유효할 경우 confirmed 필드에 jiffies를 할당해 확인 작업이 완료되었음을 표시한다.

## 6. Data Receiving

**Comment**

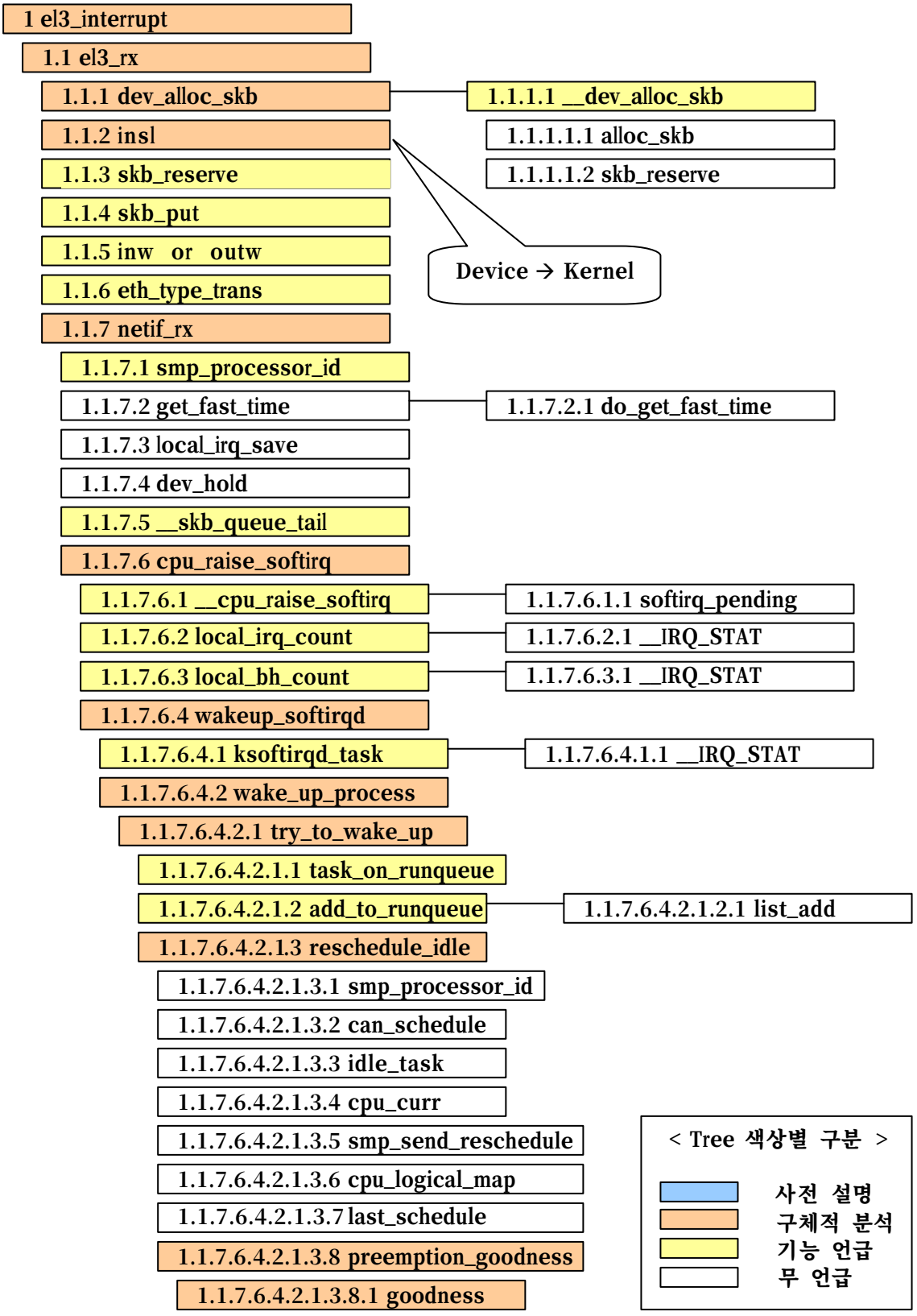
UDP 데이터 수신 과정은 세 단계의 서로 독립된 과정에 의해 수행된다.

1. 먼저 NIC가 네트워크로부터 데이터를 수신해 CPU의 큐에 집어넣고 softirq 메커니즘을 활성화시키게 되며,
2. softirq Sub System이 작동하면서 NET_RX_SOFTIRQ 값이 raise 되어 있을 경우 net_rx_action 함수를 호출하면서 CPU에 큐에 저장되어 있던 패킷을 상위 layer로 올려주게 된다. 이 때 IP layer와 UDP layer에서의 작업이 진행되고, 최종 패킷인 sock 구조체의 수신 큐에 저장되게 된다.
3. 마지막으로 사용자(어플리케이션)가 직접 데이터 수신 system call(recvfrom 함수)을 수행하면 요청된 sock의 수신 큐로부터 데이터를 가져가게 된다.

위의 세 과정을 한 번에 분석하기에는 범위가 너무 넓어 오히려 일관성이 떨어지게 될 뿐더러, 각 단계의 시작과 끝이 명확하기 때문에 Data Receiving 장은 Part I, II, III 로 나누어 진행하였다. 당연히 각 Part는 위의 세 단계 순으로 진행된다.

## Data Receiving - Part I

Function Call Tree



Tree Analysis

Help Comment - interrupt

NIC가 네트워크를 통해 자신의 local MAC address에 해당하는 이더넷 프레임이나 link layer용 broadcast 프레임을 받게 되면 인터럽트가 발생된다. NIC는 DMA나 PIO 등의 메커니즘을 통해 패킷 데이터를 RAM으로 옮겨놓고 sk buffer를 할당 받은 후 커널로 처리과정을 넘기게 된다. 이 과정은 NIC 하드웨어와 디바이스 드라이버에 의해 진행되며 위 Tree에서 '1 el3_interrupt'에서 '1.1 el3_probe'의 과정이다.
디바이스 드라이버는 마지막으로 '1.1.7 netif_rx(skb)'를 호출함으로써 커널로 작업을 이행시킨다.

1. el3_interrupt in drivers/net/3c509.c
→ *static void el3_interrupt(int irq, void *dev_id, struct pt_regs *regs)*

1.1 el3_rx in drivers/net/3c509.c
→ *static int el3_rx(struct net_device *dev)*

```
struct el3_private *lp = (struct el3_private *)dev->priv;
int ioaddr = dev->base_addr;

short rx_status;
while ((rx_status
            = inw(ioaddr+RX_STATUS)) > 0)
{
  if (rx_status & 0x4000) {
     outw(RxDiscard,ioaddr+EL3_CMD);

    lp->stats.rx_errors++;
    switch (error) {
    case 0x0000:
       lp->stats.rx_over_error++; break;
    case 0x0800:
      lp->stats.rx_length_error++; break;
     …
     }
  } else {
    short pkt_len = rx_status & 0x7ff;
    struct sk_buff *skb;
    skb = dev_alloc_skb(pkt_len+5);

    lp->stats.rx_bytes += pkt_len;
    if (skb != NULL) {
       insl(ioaddr+RX_FIFO, skb_put(skb,
         pkt_len), (pkt_len+3) >> 2);
      outw(RxDiscard,ioaddr+EL3_CMD);

       skb->protocol =
                  eth_type_trans(skb, dev);
       netif_rx(skb);
```

← device specific 한 자료구조, device의 각종 정보를 담는다.
← device에서 읽어 들일 데이터의 시작 주소
← 수신 상태 저장 변수
← device의 수신 상태를 읽어 들여 처리해야 할 사항이 있는 지 점검

← error 발생 시
← 디바이스에 수신 패킷을 버리라는 정보를 보낸다.
← 수신 총 에러 카운트를 증가시킨다.
← 에러 종류에 따라 해당 에러 카운트를 증가시킨다.

← error 처리 종료
← 정상적인 데이터 수신
← 패킷 길이(rx_status의 하위 12bit)

← 패킷 버퍼 할당(수신된 패킷 길이 + 5, 5는 붙여질 헤더 길이)
← 읽어 들인 총 바이트 수 증가
← 할당된 버퍼가 유효하다면
← device로부터 패킷 길이만큼의 데이터를 읽어온다.
← Kernel이 읽어온 패킷은 device에게 필요가 없으므로 버리도록 한다.
← 패킷의 프로토콜을 확인/할당

← device로부터 패킷을 받아 상위 layer

```
        lp->stats.rx_packets++;
        continue;
      }


    outw(RxDiscard, ioaddr + EL3_CMD);
     lp->stats.rx_dropped++;
   }
   ...
}
```

에서 처리할 수 있도록 큐에 저장
← 수신된 패킷 카운트를 증가시킨다.
← 다음 패킷 처리
← 유효한 소캣 버퍼를 할당받지 못했을 경우
← device에 패킷을 버리라 명하고
← 버려진 패킷 카운트를 증가시킨다.

### 1.1.1 dev_alloc_skb in include/linux/skbuff.h

→ *static inline struct sk_buff *dev_alloc_skb(unsigned int length)*

sk_buff를 새로 하나 할당 받아 반환한다.

__dev_alloc_skb 함수를 호출하며 GFP_ATOMIC 를 gfp_mask 인자로 넘긴다. 이 옵션은 인터럽트에 의해서만 사용되며 또한 인터럽트는 이 옵션으로만 이 함수를 호출해야 한다.

### 1.1.2 insl in include/asm_sh/io.h

→ *void insl(unsigned long port, void *dst, unsigned long count)*

Device로부터 Kernel 영역의 메모리로 주어진 길이 만큼의 데이터를 복사해온다.

호출 시 코드는 다음과 같다.

insl(ioaddr+RX_FIFO, skb_put(skb, pkt_len), (pkt_len+3) >> 2);

즉, "ioaddr+RX_FIFO " 로부터 "(pkt_len+3) >> 2 " 만큼의 데이터를 "skb_put(skb, pkt_len) " 로 복사해온다.

### 1.1.7 netif_rx in /net/core/dev.c

→ *int netif_rx(struct sk_buff *skb)*

device로부터 패킷을 읽어들여 상위 layer에서 처리할 수 있도록 큐에 저장한다.

```
int this_cpu = smp_processor_id();
struct softnet_data *queue;
queue = &softnet_data[this_cpu];


netdev_rx_stat[this_cpu].total++;
if (queue->input_pkt_queue.qlen
   <= netdev_max_backlog)
{
  if (queue->input_pkt_queue.qlen)
  {
    if (queue->throttle)
      goto drop;
enqueue:
    __skb_queue_tail(
        &queue->input_pkt_queue, skb);
    cpu_raise_softirq(this_cpu,
        NET_RX_SOFTIRQ);


   return
```

← 현 CPU의 ID를 얻어낸다.
← queue를 구조체
← 본 softnet_data 구조체는 수신되는 패킷들을 저장해 두는 큐로 사용되며 각 CPU에 하나씩 부착되어 있다.
← 현 CPU에 수신된 총 패킷 수를 증가
← 큐의 크기가 backlog 한계치를 넘어서지 않았을 경우

← 수신 큐의 길이가 0 이 아닐 경우, 즉 큐에 저장되어 있는 패킷이 있을 경우
← 수신 큐가 꽉 차 있을 경우 패킷을 버린다.

← 큐의 끝에 패킷을 저장한다.

← softirq나 Bottom Half 메커니즘에 의해 처리될 수 있도록 현 CPU의 softirq를 NET_RX_SOFTIRQ만큼 raise 시킨다. (별도 설명)
← 현 CPU의 congestion level을 반환

```
            softnet_data[this_cpu].cng_level;
    }
    if (queue->throttle) {

      queue->throttle = 0;
#ifdef CONFIG_NET_HW_FLOWCONTROL
      if (atomic_dec_and_test(
                          &netdev_dropping))
          netdev_wakeup();
#endif
      }
      goto enqueue;
    }


if (queue->throttle == 0) {

    queue->throttle = 1;
    netdev_rx_stat[this_cpu].throttled++;

#ifdef CONFIG_NET_HW_FLOWCONTROL
   if (atomic_inc(&netdev_dropping))
#endif
}
drop:
netdev_rx_stat[this_cpu].dropped++;

…
```

← 수신된 패킷이 없으나 현재 수신 큐가 꽉 차 있을 경우
← 꽉 차 있지 않다고 설정 변경
← Network Flow Control 이 지원될 경우
← netdev_dropping 값을 낮추고

← net device를 활성화시킨다.

← 수신 큐에 있는 패킷 추가

← 수신 큐의 길이가 backlog 의 한계치를 넘어선 경우이다.
← 수신 큐가 꽉 차지 않은 상태로 설정되어 있다면
← 큐가 꽉 찼음을 표시
← 현 CPU의 패킷 수신 상태 저장를 구조체에 큐가 꽉 찬 횟수(count)를 증가
← Network Flow Control 이 지원될 경우
← netdev_dropping 값을 증가

← 패킷이 버려졌을 경우
← 현 CPU의 network device의 수신 상태 보관 구조체에 버려진 패킷 카운트 증가
← 상태복구 및 자원 해제

Important Data Structure 의 softnet_data 를 참조하자.
netdev_dropping 값을 기준으로 Network Flow Control 이 수행됨을 알 수 있다.

Help Comment - softirq

Kernel 버전이 2.4로 향상되면서 softirq라는 메커니즘이 추가되면서 network stack 관리를 Bottom Half에서 softirq가 담당하게 되었다.
Bottom Half는 여러 개의 CPU가 장착된 시스템이라도 한 순간에 하나의 CPU에서만 처리될 수 있었던 것에 반해 softirq 메커니즘은 각 CPU당 하나씩 처리될 수 있으므로 SMP환경에서 성능을 극대화시킬 수 있게 된다.

#### 1.1.7.6 cpu_raise_softirq in kernel/softirq.c

→ *inline void cpu_raise_softirq(unsigned int cpu, unsigned int nr)*

넘겨진 CPU의 softirq를 nr 만큼 raise(pending) 시킨[*__cpu_raise_softirq(cpu, nr) in include/linux/interrupt.h*] 후, 현 CPU의 local irq 나 local bottom half 의 카운트가 모두 0 일 경우 softirq daemon 을 활성화[*wakeup_softirqd() in kernel/softirq.c*]시킨다. 이 때 NET_RX_SOFTIRQ 값이 인자로 주어지는데 이는 후에 softriq 서브 시스템에 의해 do_softirq() 함수가 호출되었을 때 이 값에 의해 net_rx_action()함수가 호출되어 네트워크로부터 수신된 데이터를 처리하도록 해준다.

do_softirq(), net_rx_action() 함수에 대해서는 Data Receiving Part II에서 다루도록 하겠다.

**1.1.7.6.4 wakeup_softirqd** in kernel/softirq.c

→ *static inline void wakeup_softirqd(unsigned cpu)*

넘겨진 CPU의 커널 softirq 데몬의 프로세스(태스크)를 얻어[*ksoftirqd_task(cpu)*] 실행[*wakeup_up_process(tast_struct tsk) in kernel/sched.c*]시킨다.

Comment

이하 부분은 scheduling 관련 부분이다. UDP 처리 메커니즘 분석과는 직접적으로 연관되지 않은 부분이므로 건너뛰어도 좋다.

**1.1.7.6.4.2 wake_up_process** in kernel/sched.c

→ *inline int wake_up_process(struct task_struct * p)*

넘겨진 프로세스를 **비동기적으로** 수행시킨다(수행을 시도[*try_to_wake_up(struct task_struct * p, int synchronous) in kernel/sched.c*]한다).

**1.1.7.6.4.2.1 try_to_wake_up** in kernel/sched.c

→ *static inline int try_to_wake_up(struct task_struct * p, int synchronous)*

```
p->state = TASK_RUNNING;                    ← 넘겨진 프로세스를 RUNNING으로 설정

if (task_on_runqueue(p))                    ← 프로세스가 현재 run queue에 있다면
  goto out;                                 추가작업 필요 없음
add_to_runqueue(p);                         ← 그렇지 않다면(run queue에 없다면)
                                            run_queue에 프로세스 추가
if (!synchronous || !(p->cpus_allowed       ← 비동기 활성화 옵션이 주어졌거나, 현
            & (1 << smp_processor_id()))    재 수행중인 CPU가 프로세스가 수행될
                                            (할당된) CPU가 아닌 경우
  reschedule_idle(p);                       ← 프로세스 재스케줄링 작업을 이후로
…                                           미룬다. (즉, 비동기 처리를 말한다)
```

**1.1.7.6.4.2.1.3 reschedule_idle** in kernel/sched.c

→ *static void reschedule_idle(struct task_struct * p)*

SMP를 지원하느냐 그렇지 않느냐에 의해 처리과정이 달라진다.

단일 CPU 시스템일 경우 단순히 넘겨진 프로세스가 CPU를 선점해야 할 지를 결정[preemption_goodness(…) in kernel/sched.c]한다.

SMP 시스템이라면 조금 복잡해진다.

넘겨진 프로세스가 마지막으로 수행된 CPU가 현재 idle 상태라면 그 CPU에서 수행시키도록 하며 그렇지 않다면 다른 idle CPU를 찾아 프로세스를 넘기는 등의 방식으로 작업이 진행된다.

**1.1.7.6.4.2.1.3.8 preemption_goodness** in kernel/sched.c

→ *static inline int preemption_goodness(struct task_struct * prev, struct_task_struct * p, int cpu)*

새로운 프로세스를 선점시킬 것인지를 나타내는 값을 결정해 반환한다.

**1.1.7.6.4.2.1.3.8.1 goodness** in kernel/sched.c

→ *static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)*

주어진 프로세스의 스케줄링 가중치 값을 반환한다. 우선순위와 프로세스의 정책, 남아있는 clock tick, 실시간 프로세스 여부 등에 의해 결정된다.

```
int weight;                                 ← 가중치
```

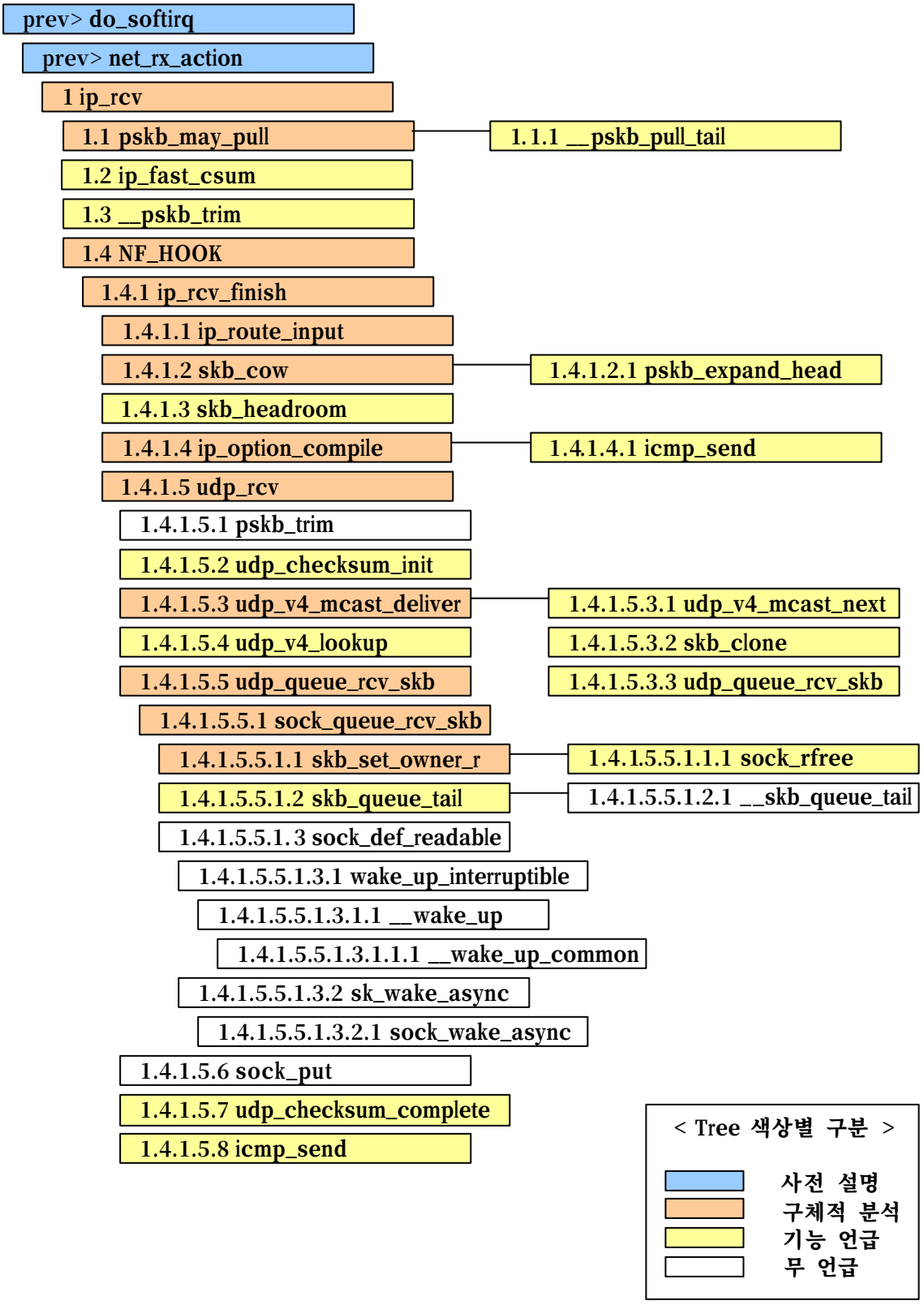
```
weight = -1;
if (p->policy & SCHED_YIELD)
  goto out;
if (p->policy == SCHED_OTHER) {

  weight = p->counter;
  if (!weight)
    goto out;
#ifdef CONFIG_SMP
  if (p->processor == this_cpu)
    weight += PROC_CHANGE_PENALTY;
#endif

  if (p->mm == this_mm || !p->mm)
    weight += 1;
  ...
}
weight = 1000 + p->rt_priority;

out:
return weight;
```

← 프로세스의 수행정책이 선점을 허용하지 않을 경우 중단
← Real-Time을 요구하지 않는 일반적인 프로세스의 경우
← 가중치에 남아있는 clock tick을 할당
← clock tick이 0일 경우(즉, 할당된 시간만큼 수행을 다 했을 경우) 중단
← SMP 모드일 경우
← 주어진 프로세스가 현재 프로세서에서 수행될 것이라면 프로세서 전환에 따른 패널티(소요시간)가 적으므로 그만큼 가중치를 더한다.
← 프로세스에 지정된 메모리 관리자가 현재의 메모리 관리자이거나 특정 메모리 관리자를 지정하지 않았다면 약간의 가중치 부여
← Real-Time 프로세스의 경우 최고의 가중치 부여(일반 프로세스와 확연히 구분)

← 계산된 가중치 반환

## Data Receiving - Part II

Function Call Tree



Tree Analysis

커널의 스케줄러가 수행[schedule() in kernel/sched.c]되면서 softirq 데몬 프로세스가 수행중일 경우 이 데몬이 작동하면서 do_softirq함수[asmlinkage void do_softirq() in kernel/softirq.c]가 호출된다.
설정상 UDP 데이터 패킷이 네트웍으로부터 전송되었으므로 Data Receiving Part I 장의 처리에 의해 패킷이 CPU의 큐에 저장되게 되며 '1.1.7.6.4 wakeup_softirqd '함수처리에 의해 softirq 데몬 프로세스가 스케줄러의 Running Queue에 들어가 있다. 따라서 이 데몬이 실행될 때 do_softirq 함수의 호출이 일어난다.
do_softriq 함수는 CPU의 softriq pending 값을 확인해 적절한 처리 함수를 호출하는데 '1.1.7.6 cpu_raise_softirq ' 함수에서 NET_RX_SOFTIRQ 값을 설정했으므로 이 경우 net_rx_action 함수[static void net_rx_action(struct softirq_action *h) in net/core/dev.c]가 호출된다.
마지막으로 net_rx_action 함수에서는 CPU의 큐로부터 패킷을 꺼내어 패킷 타입(struct packet_type)을 살펴 해당 타입 패킷의 처리 함수[packet_type.func(struct sk_buff *, struct net_device *, struct packet_type*)에 패킷과 함께 처리를 넘긴다.
이 경우 UDP 패킷, 즉 이 단계에서는 아직 IP 패킷이므로 ip_rcv 함수가 호출된다.

1. ip_rcv in net/ipv4/ip_input.c
→ *int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)*
IP 패킷 처리가 시작되는 뿌리이다.
이 곳에서 패킷 정보를 바탕으로 패킷에 오류가 있는 지 여부를 검사하고

```
if( skb->pkt_type
    == PACKET_OTHERHOST)
  goto drop;

if ((skb = skb_share_check(skb,
                  GFP_ATOMIC)) == NULL)
  goto out;
if (!pskb_may_pull(skb,
                   sizeof(struct iphdr)))
  goto inhdr_error;
…
if (!pskb_may_pull(skb, iph->ipl*4))
  goto inhdr_error;
if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
  goto inhdr_error;
…
if (skb->len > len) {
  __pskb_trim(skb, len);
  …
}
return NF_HOOK(…, ip_rcv_finish);
…
```

← 목적지 호스트가 다른 패킷은 분석하지 않고 바로 버린다.

← 버퍼가 다른 프로세스에 의해 공유되고 있다면 패킷을 버린다.

← 패킷의 길이가 IP 헤더의 길이보다도 작다면 패킷을 버린다. 필요시 단편화된 패킷들을 조합하는 기능을 수행한다.

← 패킷의 길이가 IP 헤더의 length 필드에 명시된 길이보다 작다면 패킷을 버림.
← check sum 을 계산하여 오류가 있다면 패킷을 버린다.
← 모든 오류 검사 통과
← 버퍼의 크기가 패킷 길이보다 클 경우 버퍼를 잘라낸다(자원 낭비 방지).

← network filter를 hooking 한다.
(별도 설명)

1.1 pskb_may_pull in include/linux/skbuff.h
→ *static inline int pskb_may_pull(static sk_buff *skb, unsigned int len)*
넘겨진 skb를 pull(한 쪽 방향으로 데이터를 밀어 넣음) 시킬 수 있는가를 확인
skb의 해더 길이가 len 이상일 경우 true, skb의 전체 길이가 len 보다 작을 경우

false를 반환한다.
위 두 경우에 모두 해당되지 않을 경우 skb를 len-skb의 해더길이 '만큼 앞쪽으로 밀어넣는다[__pskb_pull_tail(struct sk_buff *skb, int delta) in net/core/skbuff.c].

### 1.4 NF_HOOK in include/linux/netfileter.h

→ *#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) (okfn)(skb)*
넘겨진 ip_rcv_finish 함수를 skb를 인자로 넘겨 호출한다.
나머지 인자들은 debug 시에 쓰이는데 debug 모드에서는 NF_HOOK가 nf_hook_slow 함수[in net/core/netfilter.c]로 선언된다.

### 1.4.1 ip_rcv_finish in net/ipv4/ip_input.c

→ *static inline int ip_rcv_finish(struct sk_buff *skb)*
이름 그래도 IP 패킷 수신 작업의 마무리 처리를 담당하는 함수이다.
처리과정 후

```
…
if (iph->ihl > 5 ) {
  struct ip_options *opt;
  if (skb_cow(skb, skb_headroom(skb))
    goto drop;

  ip (ip_options_compile(NULL, skb))
    goto inhdr_error;
  opt = &(IPCB(skb)->opt);
  …
}
return skb->dst->input(skb);
…
```

← 헤더 공간이 부족할 경우 충분한 공간을 새로 할당 받은 후 복사를 수행한다.

← IP option의 유효성을 검사하고 옵션 필드들을 채운다.
← compile된 IP 옵션을 적용한다.

← 설정상 input 함수 포인터는 udp_rcv를 가리킨다.

#### 1.4.1.2 skb_cow in include/linux/skbuff.h

→ *static inline int skb_cow(struct sk_buff *skb, unsigned int headroom)*
넘겨진 skb의 헤더 공간이 부족할 경우 충분한 공간을 원자적으로 할당 받아[*int pskb_expand_head(struct sk_buff *skb, int nhead, int ntial, int gfp_mask) in net/core/skbuff.c*] 헤더를 복사한다.

#### 1.4.1.4 ip_option_compile in net/ipv4/ip_options.c

→ *int ip_options_compile(struct ip_options * opt, struct sk_buff *skb)*
옵션의 유효성 검사를 수행하고 skb의 옵션 필드들에 값을 채운다. opt가 NULL이 넘어왔으므로 skb->data가는 IP 헤더를 가리켜야 한다.
skb의 값이 유효하나 처리 중 에러가 발생했을 경우 패킷 송신자에게 parameter 에 error가 존재함을 알리는 ICMP 패킷을 전송[*void icmp_send(struct sk_buff *skb_in, int type, int code, u32 info) in net/ipv4/icmp.c*]한다.

Comment
이상으로 IP layer 에서의 처리가 모두 끝났고, TCP layer에서 처리가 시작된다.

#### 1.4.1.5 udp_rcv in net/ipv4/udp.c

→ *int udp_rcv(struct sk_buff *skb)*

```
struct sock *sk;
struct udphdr *uh;
struct rtable *rt =(struct rtable*)skb->dst;
```

← UDP 헤더 구조체
← routing table 구조체

```
…
uh = skb->h.uh;                          ← 패킷 길이 검사 후 trim
if (udp_checksum_init(skb, uh, ulen,     ← UDP 헤더 추출
                      saddr, daddr) < 0)
  goto csum_error;                       ← UDP checksum을 초기화한다.
if(rt->rt_flags & (RTCF_BROADCAST
                   | RTCF_MULTICAST))    ← broadcast/multicast 용 패킷일 경우
  return udp_v4_mcast_deliver(             multicast 처리용 함수로 작업 이전
                   skb, uh, saddr, daddr);
sk = udp_v4_lookup(saddr, uh->source,    ← udp hash table에서 현 패킷에 해당하
    daddr, uh->dest, skb->dev->ifindex);   는 sock을 찾아낸다.
if (sk != NULL) {                        ← 찾아냈을 경우 sk의 수신 큐의 마지막
  udp_queue_rcv_skb(sk, skb);              에 skb를 추가한다.
  sock_put(sk);
  return 0;
}
if (udp_checksum_complete(skb))          ← checksum 이 잘못됐을 경우 에러처리
  goto csum_error;
…

icmp_send(skb,  ICMP_DEST_UNREACH,       ← 이곳까지 넘어왔다는 것은 원치 않는
ICMP_PORT_UNREACH, 0);                     패킷을 받았다는 것을 의미한다.
…                                        ← 송신자에게 Destination Unreachable /
                                           Port Unreachable ICMP 패킷을 전송한다.
```

Help Comment ? struct sock & sk_buff

sock 구조체는 수신 큐와 송신 큐를 각각 하나씩 갖고 있으며 이 큐에는 sk_buffer 들이 linked list 형태로 들어가게 된다. sk_buff 구조체는 자신과 같은 타입(sk_buff)의 next, prev 포인터 필드를 갖고 있으므로 곧바로 list를 이룰 수 있는 구조이다.

**1.4.1.5.3 udp_v4_mcast_deliver** in net/ipv4/udp.c

→ *static int udp_v4_mcast_deliver(struct sk_buff *skb, struct udphdr *uh, u32 saddr, u32 daddr)*

```
struct sock *sk;
sk = udp_hash[ntohs(uh->dest)            ← 패킷의 목적지 주소에 해당하는 수신
              & (UDP_HTABLE_SIZE - 1 )];   큐의 리스트를 hash table로부터 얻어낸
                                           다.
do {                                     ← 모든 수신 큐에 대해
  sknext = udp_v4_mcast_next(…);
  if (sknext)
    skb1= skb_clone(skb, GFP_ATOMIC);    ← 패킷을 복사하여
  if (skb1)
    udp_queue_rcv_skb(sk, skb1);         ← 수신 큐의 끝에 집어 넣는다.
} while(sknext);
…
```

**1.4.1.5.5 udp_queue_rcv_skb** in net/ipv4/udp.c

→ *static int udp_queue_rcv_skb(struct sock * sk, struct sk_buff *skb)*

주어진 sk의 수신 큐에 skb를 추가[*sock_queue_rcv_skb(sk, skb)를 호출*]한다.
만약 큐가 꽉 차 있다면 패킷(skb)을 버린다.

**1.4.1.5.5.1 sock_queue_rcv_skb** **in include/net/sock.h**

→ *static inline void skb_set_rcv_skb(struct sock *sk, struct sk_buff *skb)*

skb의 소유자를 명시[*skb_set_owner_r in include/net/sock.h*]하고 sk의 수신 큐에 skb를 집어넣는다.

**1.4.1.5.5.1.1 skb_set_owner_r** **in include/net/sock.h**

→ *static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk)*

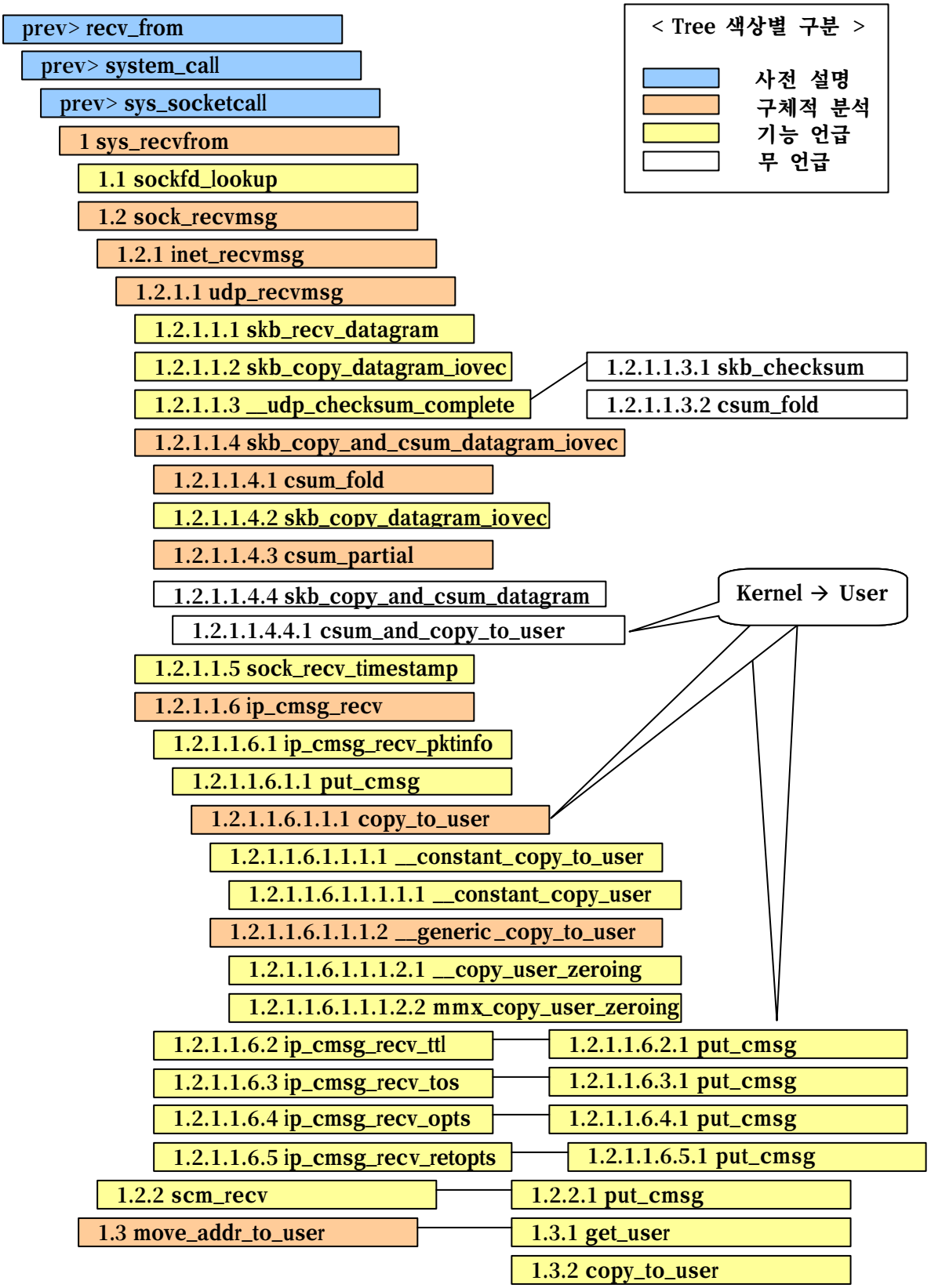
주어진 skb의 소유자가 sk임을 명시하고

skb의 파괴자[*void sock_rfree(struct sk_buff *skb) in net/core/sock.c*]를 지정한다.

파괴자는 소켓 버퍼가 더 이상 필요가 없어지게 되면 커널에 의해 호출돼 자원(버퍼)을 회수해간다.

## Data Receiving - Part III

Function Call Tree



Tree Analysis

사용자(어플리케이션)가 데이터 수신을 위해 recvfrom 함수를 호출하면 system call이 발생(interrupt vector에서 index 0x80에 해당, "#define SYSCALL_VECTOR 0x80" in include/asm-i386/hw_irq.h)하게 된다. system call interrupt handler에 의해 sys_socketcall 함수가 호출되며 이 경우 UDP 데이터 수신 과정이므로 call 인자로 SYS_RECVFROM 이 넘겨지므로 sys_recvfrom 함수가 호출된다. (sys_socketcall 함수는 Socket Creation과 Data Sending 과정에서 다루었으므로 이번 장에서는 생략하였다)

**1. sys_recvfrom** in net/socket.c

→ asmlinkage long sys_recvfrom(int fd, void * ubuf, size_t size, unsigned flags, struct sockaddr *addr, int *addr_len)

UDP 데이터 수신 처리의 뿌리가 되는 함수이다. 대략의 진행과정을 살펴보자.

```
struct socket *sock;
struct msghdr msg;
…
sock = sockfd_lookup(fd, &err);          ← fd에 해당하는 socket을 얻어온다.
                                           (fd : file handle)
…                                        ← msg 필드 설정
err=sock_recvmsg(sock, &msg, size,       ← 패킷을 읽어들인다.
                                 flags);
if ( err >= 0 && addr != NULL            ← 에러가 없다면
    && msg.msg_namelen)
{
  err2=move_addr_to_user(address,        ← User 영역으로 address를 넘긴다.
     msg.msg_namelen, addr, addr_len);     (Kernel -> User)
  …
}
…
```

**1.2 sockfd_recvmsg** in net/socket.c

→ *int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)*

```
struct scm_cookie scm;                   ← socket control message cookie
size = sock->ops->recvmsg(sock, msg,     ← sock의 operation 함수 포인터에서 데
                          size, flags, &scm);   이터 수신용 함수를 호출한다. 설정상
                                                inet 패킷이기 때문에 inet_recvmsg 함수
                                                가 호출된다.
if (size >= 0)
  scm_recv(sock, msg, &scm, flags);      ← msg의 control message를 읽어 들임
return size;                             ← 읽어 들인 데이터의 길이를 반환
```

**1.2.1 inet_recvmsg** in net/ipv4/af_inet.c

→ *int inet_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags, struct scm_cookie *scm)*

sk->prot->recvmsg(수신된 패킷의 상위 layer의 메시지 수신 처리 함수)를 호출하는데 설정상 UDP 패킷이기 때문에 udp_recvmsg 함수가 호출된다.

**1.2.1.1 udp_recvmsg** in net/ipv4/udp.c

→ *static __inline__ int udp_checksum_complete(struct sk_buff *skb)*

```
…
skb = skb_recv_datagram(sk, flags,
            noblock, &err);
…
copied=skb ->len ? sizeof(struct udphdr);
…
// option 에 따라
  skb_copy_datagram_iovec(…)

  or ( __udp_checksum_complete (skb)
     and skb_copy_datagram_iovec(…)
  or skb_copy_and_csum_datagram_iovec(…)
// 중 선택 실행
…
sock_recv_timestamp(msg, sk, skb);
…
if (sk->protoinfo.af_inet.cmsg_flags)
  ip_cmsg_recv(msg, skb);
…
```

← sk의 수신 큐로부터 패킷을 하나 읽어 온다.

← 복사할 데이터의 길이(헤더 길이 제외)

← checksum이 불필요: User IO Vector에 버퍼를 복사
← MSG_TRUNC flag 설정: checksum 확인 후 복사
← 그 외의 경우 checksum을 수행하고 skb를 io vector에 복사한다.

← 수신된 시간 기록
← address 복사
← control message flag가 설정되어 있다면 메시지 수신(별도 설명)

**1.2.1.1.4 skb_copy_and_csum_datagram_iovec** in net/core/datagram.c
→ *int skb_copy_and_csum_datagram_iovec(const struct sk_buff *skb, int hlen, struct iovec *iov)*
주어진 skb의 checksum을 수행하고 User의 iovec(IO vector)에 skb를 복사한다.

**1.2.1.1.4.1 csum_fold** in include/ams -i386/checksum.h
→ *static inline unsigned int csum_fold(unsigned int sum)*
부분적인 checksum을 포갠다(합한다).

**1.2.1.1.4.3 csum_partial** in include/ams -i386/checksum.h
→ *asmlinkage unsigned int csum_partial(const unsigned char *buff, int len, unsigned int sum)*
버퍼(buff)의 주어진 길이(len) 만큼의 블록들의 각각의 값을 sum에 합한다

**1.2.1.1.6 ip_cmsg_recv** in net/ipv4/ip_sockglue.c
→ *void ip_cmsg_recv(struct msghdr *msg, struct sk_buff *skb)*
IP layer control message 수신 함수이다.
skb->sk->protinfo.af_inet.cmsg_flags를 확인해 적절한 함수를 호출한다.
사용 빈도수에 따라 나열하면 pktinfo > ttl > tos > opts > retopts 순이며 각각 ip_cmsg_recv_xxx(struct msghdr *msg, struct sk_buff *skb) 함수에 의해 처리된다.
각각 처리를 마친 후 공통적으로 put_cmsg 함수[*in net/core/scm.c*]를 호출해 control message의 헤더를 User 영역에 복사[*copy_to_user in include/asm-i386/uaccess.h*]한다.

**1.2.1.1.6.1.1.1 copy_to_user** in include/asm-i386/uaccess.h
Kernel Memory Area로부터 User Memory Area로 데이터를 복사한다.
Socket Creation 장의 copy_from_user와 동일한 메커니즘으로 수행된다.
매크로 원형은 copy_to_user(to, from, n)으로, from에서 to로 n(size) 만큼의 데이터를 복사해 넣는다. 이중 n을 살펴 적절한 built-in 처리 루틴이 있다면 __constant_copy_to_user를, 그렇지 않다면 __generic_copy_to_user 함수를 호출한다.

이 둘의 주요 차이점은 __constant_copy_to_user는 inline 함수이기 때문에 수행 성능이 뛰어나다는 것이다.

Help Comment ? delayed copy to user

분석 내용을 차근히 되 집어 보면 알 수 있듯이 IP layer 에서는 패킷 처리가 끝나고 오류가 없다 해도 User 영역으로 데이터를 복사하지 않는다. 이는 비록 수신지 주소가 호스트 자신이라 해도 이를 필요로 하는 실제 어플리케이션이 자신에게는 없을 수도 있기 때문에 아직 이 패킷이 자신에게 필요한 것인지 최종 결정을 내릴 수 없기 때문이다. 만약 IP layer 에서 패킷을 User 영역으로 복사를 한 상황에서 TCP나 UDP 처리 시 목적지 port 번호(어플리케이션 주소)에 해당하는 어플리케이션이 존재하지 않는다면 패킷을 User 영역으로 옮긴 것은 쓸모 없는 작업이 되어버리고 수신한 패킷은 버려져야 한다. 더구나 메모리를 액세스하는 것은 상당한 오버헤드를 유발하기 때문에 이 작업은 되도록 피해야 하는 것이다.
따라서 위와 같이 TCP나 UDP 등 IP의 상위 layer에서 port 번호까지 최종 확인 후 User 영역에 데이터를 복사하게 된다.

1.2.1.1.6.1.1.1.2 __generic_copy_to_user in arch/i386/lib/usercopy.c

→ *unsigned long __generic_copy_to_user(void *to, const void *from, unsigned long n)*

inline 되지 않는 함수로 특이한 점은 AMD사의 3DNow 지원여부에 따라 다른 함수를 실행시킨다는 것이다. 3DNow 지원 시는 __copy__user 대신 mmx_copy_user 라는 보다 최적화된 함수를 호출한다.

1.3 move_addr_to_user in net/socket.c

→ *int move_addr_to_user(void *kaddr, int klen, void *uaddr, int *ulen)*

Kernel 영역의 address를 User 영역으로 복사한다.

kaddr: 커널 영역의 주소, klen: 커널 영역의 주소 길이, uaddr: User 영역의 주소, ulen: User 영역의 주소 길이가 된다.

```
if ((err=get_user(len, ulen)))          ← User 영역에 ulen 만큼의 공간을 확보.
  return err;                           ← 여유 공간이 부족하면 error
if (len>klen)                           ← 확보된 공간이 필요 공간보다 크면 필요한 만큼만 설정
  len = klen;
if (len<0 || len> MAX_SOCK_ADDR)        ← 확보된 공간의 크기가 유효한 범위가 아니면 error
  return ?EINVAL;
if (len) {                              ← 유효한 사용자 공간이 확보됐다면
  if (copy_to_user(uaddr, kaddr, len))  ← Kernel 영역의 address를 User로 복사
    return ?EFAULT;
}
return __put_user(klen, ulen);          ← 자원 해제
```

copy_to_user 함수의 처리 과정은 이미 설명했으므로 생략한다.

## Appendix

### Important Data Structures

struct net_proto_family — in include/linux/net.h

```
int     family;
int     (* create)(struct socket *sock, int protocol);
short   authentication;
short   encryption;
short   encrypt_net;
```

이 struct는 inet_family_ops, unix_family_ops 등 xxx_family_ops struct의 기본형이 된다.
예를 들어 inet_family_ops는 다음과 같다.

```
struct net_proto_family inet_family_ops = {
        family: PF_INET,
        create: inet_create
};
```

즉, inet_family_ops는 PF_INET에 속하며 생성함수는 inet_create이다.

struct proto_ops — in include/linux/net.h

```
int family;
int (*release)     (struct socket *sock);
int (*bind)        (struct socket *sock, struct sockaddr *umyaddr, int sockaddr_len);
int (*connect)     (struct socket *sock, struct sockaddr *uservaddr, int sockaddr_len, int flags);
int (*socketpair) (struct socket *sock1, struct socket *sock2);
int (*accept)      (struct socket *sock, struct socket *newsock, int flags);
int (*getname)     (struct socket *sock, struct sockaddr *uaddr, int *usockaddr_len, int peer);
unsigned int (*poll)        (struct file *file, struct socket *sock, struct poll_table_struct *wait);
int (*ioctl)       (struct socket *sock, unsigned int cmd, unsigned long arg);
int (*listen)      (struct socket *sock, int len);
int (*shutdown)  (struct socket *sock, int flags);
int (*setsockopt)(struct socket *sock, int level, int optname, char *optval, int optlen);
int (*getsockopt)(struct socket *sock, int level, int optname, char *optval, int *optlen);
int (*sendmsg)   (struct socket *sock, struct msghdr *m, int total_len, struct scm_cookie *scm);
int (*recvmsg)   (struct socket *sock, struct msghdr *m, int total_len, int flags, struct scm_cookie *scm);
int (*mmap)      (struct file *file, struct socket *sock, struct vm_area_struct * vma);
ssize_t (*sendpage)       (struct socket *sock, struct page *page, int offset, size_t size, int flags);
```

프로토콜 수행에 필요한 각 함수들을 함수 포인터로 갖고 있어 인덱스에 따라 적절한 함수를 호출한다. 이 구조체는 socket 구조체의 한 원소로 들어간다.

struct socket — in include/linux/net.h

```
socket_state            state;

unsigned long           flags;
struct proto_ops        *ops;
struct inode            *inode;
struct fasync_struct    *fasync_list;     /* Asynchronous wake up list        */
struct file             *file;            /* File back pointer for gc    */
struct sock             *sk;
wait_queue_head_t       wait;
```

```
short                   type;
unsigned char           passcred;
```

static struct net_proto_family *net_families[NPROTO]; in net/socket.c
시스템에서 지원하는 protocol 들을 등록해 놓은 list 이다. net_proto_family 는 Network Initialization 장의 Important Data Struct에서 다루었다.

struct msghdr in include/net/route.h

```
        union
        {
                struct dst_entry  dst;
                struct rtable             *rt_next;
        } u;
        unsigned                  rt_flags;
        unsigned                  rt_type;
        __u32                     rt_dst;   /* Path destination       */
        __u32                     rt_src;   /* Path source            */
        int                       rt_iif;
        /* Info on neighbour */
        __u32                     rt_gateway;
        /* Cache lookup keys */
        struct rt_key             key;
        /* Miscellaneous cached information */
        __u32                     rt_spec_dst; /* RFC1122 specific destination */
        struct inet_peer  *peer; /* long-living peer info */
#ifdef CONFIG_IP_ROUTE_NAT
        __u32                     rt_src_map;
        __u32                     rt_dst_map;
#endif
```

struct iphdr in include/linux/ip.h

```
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8    ihl:4,
                version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
        __u8    version:4,
                ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
        __u8    tos;
        __u16   tot_len;
        __u16   id;
        __u16   frag_off;
        __u8    ttl;
        __u8    protocol;
        __u16   check;
        __u32   saddr;
        __u32   daddr;
        /*The options start here. */
```

struct **udpfakehdr** in **net/ipv4/udp.c**

```
        struct udphdr uh;
        u32 saddr;
        u32 daddr;
        struct iovec *iov;
        u32 wcheck;
```

struct **udphdr** in **include/linux/udp.h**

```
        __u16   source;
        __u16   dest;
        __u16   len;
        __u16   check;
```

struct **scm_cookie** in **include/net/scm.h**

```
        struct ucred        creds;      /* Skb credentials      */
        struct scm_fp_list  *fp;        /* Passed files         */
        unsigned long       seq;        /* Connection seqno     */
```

→ socket control message

struct **ipcm_cookie** in **include/net/ip.h**

```
        u32                 addr;
        int                 oif;
        struct ip_options *opt;
```

→ ip control message

struct **ip_options** in **include/linux/ip.h**

```
__u32           faddr;                  /* Saved first hop address */
  unsigned char optlen;
  unsigned char srr;
  unsigned char rr;
  unsigned char ts;
  unsigned char is_setbyuser:1,                 /* Set by setsockopt?
        */
                is_data:1,                      /* Options in __data, rather than skb
        */
               is_strictroute:1,        /* Strict source route                  */
                srr_is_hit:1,                   /* Packet destination addr was our one
        */
                is_changed:1,                   /* IP checksum more not valid
        */
                rr_needaddr:1,                  /* Need to record addr of outgoing
dev     */
                  ts_needtime:1,                /* Need to record timestamp
        */
                ts_needaddr:1;                  /* Need to record addr of outgoing
dev  */
```

```
unsigned char router_alert;
unsigned char __pad1;
unsigned char __pad2;
unsigned char __data[0];
```

struct **softnet_data** in include/linux/netdevice.h

```
int                     throttle;
int                     cng_level;
int                     avg_blog;
struct sk_buff_head     input_pkt_queue;
struct net_device*output_queue;
struct sk_buff          *completion_queue;
```

extern struct netif_rx_stats **netdev_rx_stat[]** in include/linux/netdevice.h

struct **netif_rx_stats** in include/linux/netdevice.h

```
unsigned total;
unsigned dropped;
unsigned time_squeeze;
unsigned throttled;
unsigned fastroute_hit;
unsigned fastroute_success;
unsigned fastroute_defer;
unsigned fastroute_deferred_out;
unsigned fastroute_latency_reduction;
unsigned cpu_collision;
```

**Comment**

소스 길이가 너무 긴 것은 지면 상에 정의가 위치한 소스파일의 경로만을 명시하였다.

struct **rtable** in include/net/route.h
→ source path 참조

struct **dst_entry** in include/net/dst.h
→ source path 참조

struct **sk_buff** in include/linux/skbuff.h
→ source path 참조

struct **sock** in include/net/sock.h
→ source path 참조

struct **net_device** in include/linux/netdevice.h or netfilter.h
→ source path 참조

## References

**[Book or Document]**
The Linux Kernel, by David A Rusling
Linux Kernel Internals, by Michael Beck, and so on : ADDISON-WESLEY
TCP/IP Illustrated, by Wright Stevens

**[Web Site]**
http://lxr.linux.no/
http://www.gnumonks.org/ftp/pub/doc/packet-journey-2.4.html
http://linuxkernel.to/module/port-2.4/eng/lkp.html#toc

**[Source Code]**
Linux Kernel version 2.4.13