Kernel Düzeyi TCP/IP Uygulaması

1. Bölüm

Cihan Kömeçoğlu

Enderunix Yazılım Geliştiricisi

cihan [at] enderunix.org

Bu bölüm BSD işletim sisteminin TCP/IP uygulamasına genel bir bakış niteliğindedir.

İçindekiler

Bu dokumanda kullanılan şekiller ve konuların büyük bir kısmı TCP/IP illustrated Volume 2 (Richard Stevens) kitabından alınmıştır. Bazı bölümler bu kitaptan çeviri yapılmıştır.

1.1 Network uygulamasına Genel Bakış

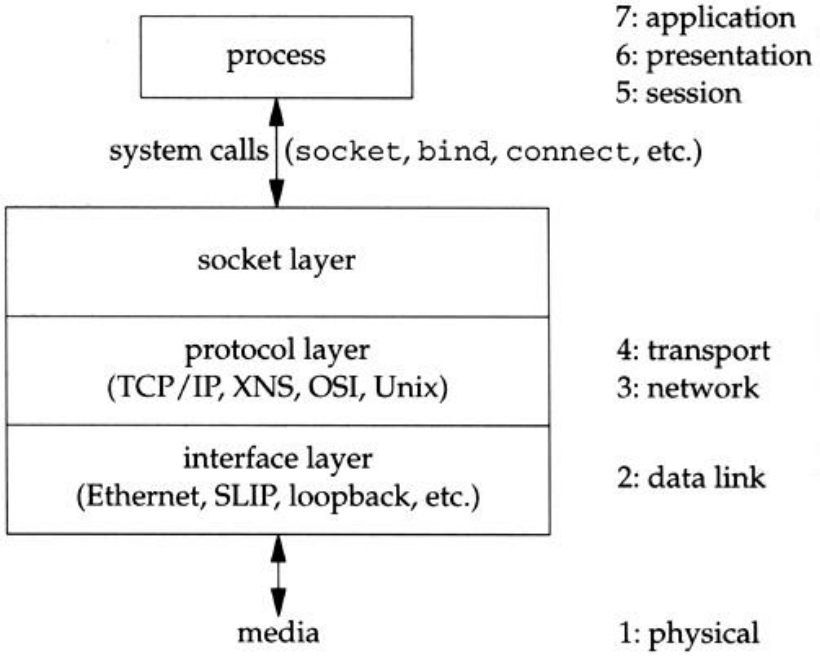
4.4BSD bilinen dört haberleşme protokol ailesini destekler.

1. TCP/IP (İnternet iletişim kural paketi)
2. XNS (Xerox network Sistemi), İletişim kuralı TCP/IP'e benzer ve 1980'lerin ortalarında çok kullanılırdı. XNS kodları NET/3 ile dağıtılmasına rağmen, Berkeley TCP/IP kodu kullanan bazı satıcılar XNS kodlarını silmiştir.
3. OSI Protokolleri, 1980'lerde diğer haberleşme protokollerin yerini almak için tasarlanan bir protokoldür.
4. Unix domain protokolleri, farklı sistemler arasında veri iletimi için kullanılan protokol şekli değildir. Unix domain protokolü process'ler arası haberleşme (IPC) için kullanılan bir protokoldür.

Unix domain protkolünün diğer IPC türlerine göre avantajı ise diğer 3 haberleşme protokollerinin kullandığı aynı arayüzü (soket) kullanmasıdır. Unix domain protokollerinde iki farklı protokol bulunmaktadır. Bunlardan biri TCP'e benzeyen güvenilir, bağlantı yönelimli ve byte-stream bir protokoldür. Diğeri ise UDP şeklinde bağlantısız, datagram ve güvenilir olmayan bir protokoldür.

Kernel'daki network kodu şekil 1.1'de gösterildiği gibi 3 katman olarak tasarlanmıştır. Şeklin sağ tarafında ise OSI referans modeline karşılık gelen katmanlar yazılmıştır.

1. *Soket katmanı* protokol bağımlı katman için protokol bağımsız bir arayüzdür.Tüm sistem çağrıları (system call) protokol bağımsız soket katmanından başlamaktadır. Örneğin , socket, bind, accept gibi sistem çağrıları soket tanımlayıcısını (descriptor) parametre olarak alır. Daha sonra bu parametrenin değerine göre protokol bağımlı kodlar kernel tarafından işletilir.
2. *Protokol katmanı* daha önceden belirtiğimiz dört haberleşme protokolün (TCP/IP, XNS, OSI ve Unix Domain) uygulamasını içerir.
3. *Arayüz katmanı (interface layer)* network aygıtları ile haberleşecek aygıt sürücülerini (device drivers) içerir.



Şekil 1.1 Network kodunun genel tasarımı

1.2 Tanımlayıcılar (Descriptors)

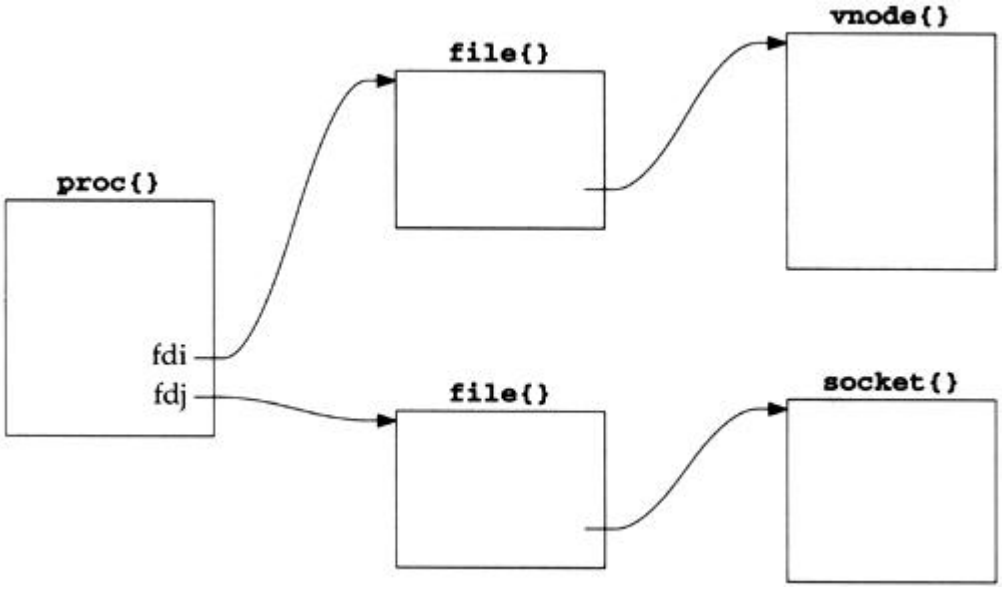
Socket fonksiyonu oluşturulacak soket tipini belirtir. Fonksiyondan oluşturduğumuz soketi belirten bir tanımlayıcı (descriptor) döner. Aşağıda örnek bir soket çağrısı bulunmaktadır.

```
fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Fonksiyonun ilk parametresi soketin hangi network protokolü ailesi için kullanılacağını belirtmektedir. Burada AF_INET Ipv4 network protokol ailesini belirtmektedir. SOCK_DGRAM parametresi de Ipv4 network ailesinden hangi tip protokolün kullanılacağını belirtmektedir. Bu örnekte UDP datagramları oluşturulmaktadır.

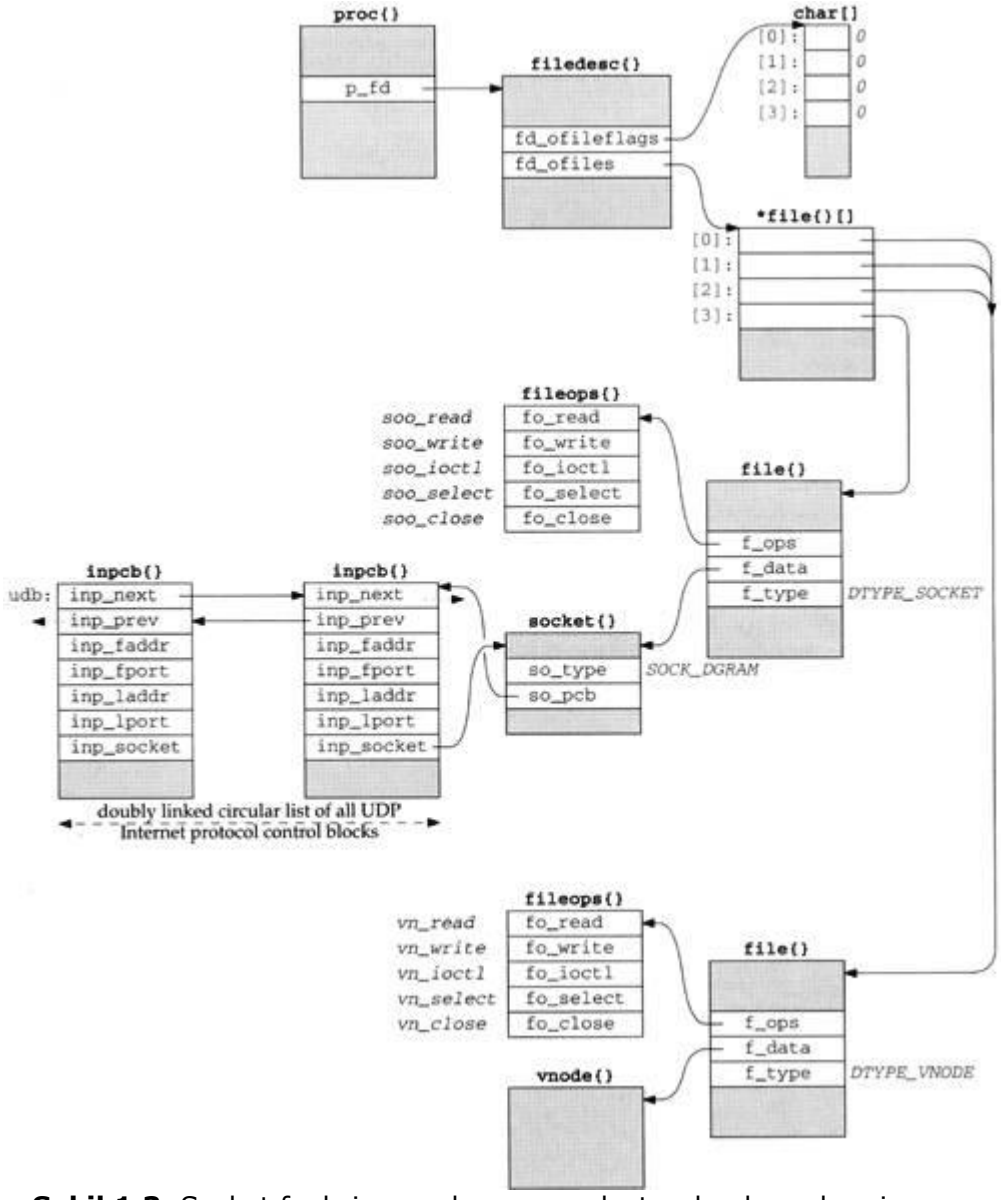
Bu fonksiyondan dönen tanımlayıcı diğer tanımlayıcılarda kullanılan tüm özellikler kullanılabilir.Yani, *read* ve *write* fonksiyonlarında, tanımlayıcıyı *dup* fonksiyonu ile kopyalanıp *fork* işleminden sonra parent-child proseslerinde, tanımlayıcının özelliklerinin *fcntl* fonksiyonu ile değiştirilmesinde kullanılabilir.UDP datagramların gönderilmesi için kullanılan fonksiyonlardan *sendto* ve *recvfrom* fonksiyonların ilk parametresi, socket fonksiyonundan dönen tanımlayıcıdır.

Şimdi socket fonksiyonu çağrıldıktan sonra kernel tarafından oluşturulan veri yapılarını göreceğiz. Her prosesin kendine ait bir tanımlayıcı tablosu bulunmaktadır.Bu tanımlayıcı tablosunda işaretçiler bulunmaktadır.Bir tanımlayıcı bu işaretçilerin tabloadki konumunu belirtmektedir.Proses tablosunda tutulan bu işaretçiler open-file tablosundaki belli bir konumu göstermektedir. Open-file tablosunda ise açılan soket ve dosyalar ile ilgili bilgilerin tutulduğu bir yapıyı gösteren işaretçiler bulunmaktadır. Şekil 1.2 bu veri yapısını göstermektedir.



Şekil 1.2 Tanımlayıcı ve kernel veri yapısı arasındaki ilişki

Şekil1.3 bu veri yapısını daha ayrıntılı bir şekilde göstermektedir. Bu şekilde çalıştırılan programda herhangi bir I/O yönlendirme işlemi yapılmamıştır. Şekildeki tanımlayıcı tablosunda ise 0 değerindeki tanımlayıcı standart output, 1 değerindeki tanımlayıcı standart input, 2 değerindeki tanımlayıcı standart error ve 3 değerindeki tanımlayıcı da *socket* fonksiyonu çağrıldıktan sonra oluşturulan tanımlayıcıyı belirtmektedir.



Şekil 1.3 Socket fonksiyonundan sonra oluşturulan kernel veri yapısı

Bir proses *socket* gibi bir sistem çağrısı yaptığı zaman, kernel proses'in tanımlayıcı tablo yapısına erişir.Bu yapıdaki *p_fd, filedesc* yapısını işaret eder. Bu filedes yapısında iki üye bulunmaktadır. Bunlardan biri karakter dizisi olan *fd_ofileflags* (her bir tanımlayıcı için bayrakları tutar) ve diğeri daha önceden de belirtiğimiz gibi açılan soket ya da dosya ile ilgili dosya yapılarını gösteren *fd_ofiles* işaretçisi bulunmaktadır.Her bir tanımlayıcı için tutulan bayraklar 8 bit genişliğindedir. Close-on-exec bayrağı ve mapped-from-device bayrağı için birer bit olmak üzere toplam 2 bitlik bir alan her tanımlayıcı için belli bir değer atanır. Bu bayraklar şekilde 0 olarak gsterilmiştir.

Şekil 1.3'de görüldüğü gibi *fd_ofiles* tarafından gösterilen şekildeki **file{}[]* dizisi *file* yapısına işaret eden işaretçileri tutmaktadır.Bu tanımlayıcı dizisi ve tanımlayıcı bayrak dizilerin konumu 0'dan başlar ve artan yönde ilerlemektedir.Daha önceden belirtildiği gibi bu dizinin konumları tanımlayıcılar tarafından tutulmaktadır.*file{}[] dizisinde 0,1,2 konumları şeklin altındaki aynı *file* yapısına işaret etmektedir çünkü bu tanımlayıcıların tamamı terminal ile ilgilidir. 3 nolu konum ise farklı bir *file* yapısına işaret etmektedir çünkü bu file yapısı soket için oluşturulmuştur.

Aşağıda, şekil 1.3'de gösterilen yapıların FreeBSD işletim sisteminden tanımlanmış şekli bulunmaktadır.Bu yapıların üyelerinin tamamı burada gösterilmemiştir. Sadece ilgili kısımlar bulunmaktadır.

```
sys/proc.h
struct proc{
        .
        .
    struct filedesc *p_fd;
        .
}


sys/filedesc.h
struct filedesc{
        .
        .
    struct file **fdofiles;
        .
}


sys/file.h
struct file{
        .
        .
    short f_type;
    void *f_data;
    struct fileops *f_ops
        .
}

sys/socketvar.h
struct socket{
        .
        .
    void so_pcb;
    short   so_type;
        .
}

sys/queue.h
#define LIST_ENTRY(type)                                    \
struct {                                                \
      struct type *le_next;   /* bir sonraki yapı*/             \
      struct type **le_prev;  /* bir önceki yapının adresi */       \
}

netinet/in_pcb.h
struct inpcb{
        .
        .
    LIST_ENTRY(inpcb) inp_list;
    struct  socket *inp_socket
/* Yerel ve yabancı port ve ip adresi. */
struct  in_conninfo inp_inc;
#define inp_fport      inp_inc.inc_fport
#define inp_lport      inp_inc.inc_lport
#define inp_faddr      inp_inc.inc_faddr
#define inp_laddr      inp_inc.inc_laddr
        .
}
```

file yapısının *f_type* üyesi *DTYPE_SOCKET* ya da *DTYPE_VNODE* olarak tanımlayıcının tipini belirlemektedir. V-node'lar disk dosya sistemi, network dosya sistemi, cdrom dosya sistemi gibi farklı dosya sistemleri için genel bir mekanizmadır. TCP/IP soketleri her zaman *DTYPE_SOCKET* tipini kullanır.

Daha önceden örnek olarak gösterilen socket fonksiyonunda, soket tipi olarak SOCK_DGRAM belirlenmişti. Bu tip socket yapısının so_type üyesinde saklanmaktadır.Internet protokol kontrol blok yapısı ise inpcb olarak adlandırılır. socket yapısının so_pcb üyesi inpcb yapısına işaret eder ve inpcb yapısının inp_socket üyesi de socket yapısına işaret eder.Şekilde görüldüğü gibi yapılar arasındaki ilişki çift yönlüdür. Çünkü sokette çift yönlü bir etkileşim vardır.

1. *sendto* gibi bir sistem çağrısı yapıldığı zaman, kernel tanımlayıcının değeri olan *fd_ofiles*'ı kullanarak *file* yapısındaki tanımlayıcının belirttiği konuma gelir ve *file* yapısından *socket* yapısına ve oradan da *inpcb* yapısına erişir.
2. Eğer bir UDP datagram ulaştığında ise kernel uygun bir eşleşme bulana kadar (UDP port ve ip adres bilgileri) UDP protokol kontrol blok veri yapısını araştırır ve uygun bir protokol kontrol blok yapısı bulunduğunda *inp_socket* işaretçisi ile ilgili sokete erişir.

inp_faddr ve inp_laddr yabancı ve yerel ip adresini ve inp_fport ve inp_lport yabancı ve yerel port adresini içerir.

Şeklin sol tarafında udb ile başlayan başka bir inpcb yapısı gösterilmiştir. Bu yapı UDP PCB bağlı listesinin başıdır.

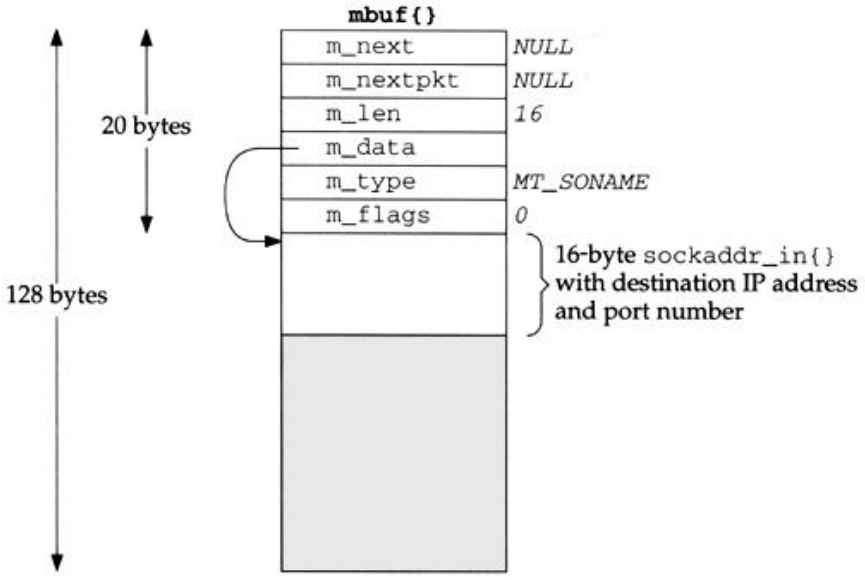
1.3 Mbuf (Memory Buffers) ve Output İşleme

Berkeley network ile ilgili kodların büyük bir kısmını, çeşitli bilgilerin tutulması için memory buffer yani *mbuf* yapıları yer alır.

Soket Yapısını İçeren Mbuf Yapısı

sendto(fd, buffer, BUFFSIZE, 0, (struct sockaddr*) &serv, sizeof(serv))

Yukarıda görüldüğü gibi sendto sistem çağrısının beşinci parametresi İnternet soket adres yapısını ve altıncı parametresi bu yapının boyutunu belirtmektedir. Bu sistem çağrısı yapıldığında soket katmanın yapacağı ilk iş bu argumanların doğruluğunu kontrol etmektir. Eğer herhangi bir hata bulunmazsa bu yapı mbuf yapılarına kopyalanır. Aşağıdaki şekil bu işlemin sonucunda oluşan mbuf yapısını göstermektedir.



Şekil 1.4 *sendto* sistem çağrısındaki hedef adresini içeren Mbuf yapısı

Mbuf yapısının ilk 20 byte'ı mbuf hakkında bilgiler içeren bir başlıktır. 20-byte'lık başlık 2 byte ve 4 byte'lık alanlar içermektedir.Mbuf yapısının toplam boyutu 128 byte'tır.

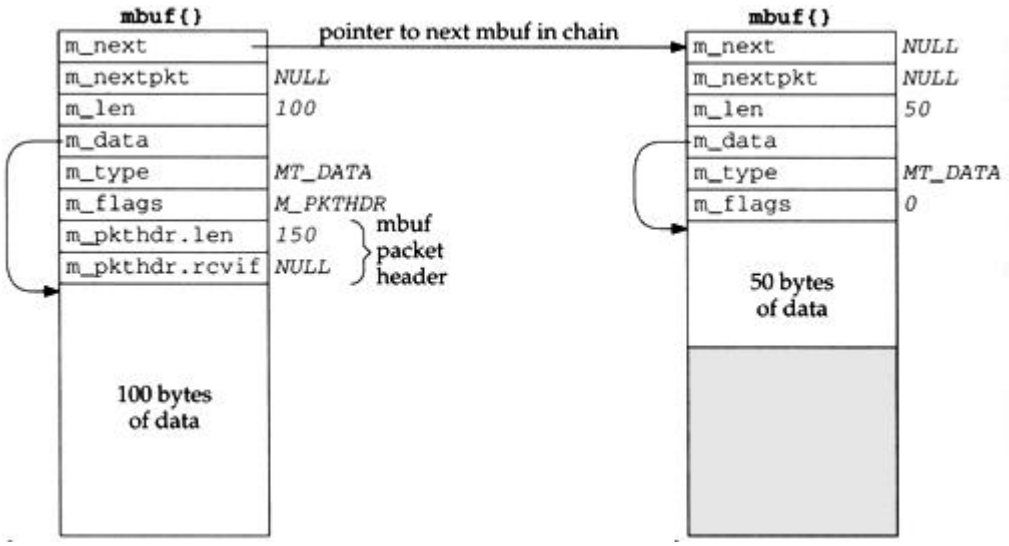
Mbuf yapıları *m_next* ve *m_nextpkt* işaretçileri ile birbirleriyle bağlanmaktadır.Fakat şekildeki mbuf tek olduğundan bu işaretçilere null atanmıştır.

Mbuf yapısının *m_data* üyesi mbuf yapısındaki veriye işaret eder ve *m_len* üyesi de bu verinin büyüklüğünü belirtir. Şekil 1.4'de *m_data* işaretçisi mbuf başlığından sonra başlayan verinin ilk byte'ına işaret etmektedir.Son 92 (108-16)byte'ı ise kullanılmamaktadır(Şekil 1.4'de karalanan bölge).

Bu yapının *m_type* üyesi de mbuf yapısında taşınan verinin tipini belirtmektedir ve şekilde MT_SONAME (socket name)değerini almıştır.Başlıktaki son üye ise *m_flags* olup şekilde sıfır değerini alıştır.

Veriyi İçeren Mbuf Yapısı

Soket katmanı sendto çağrısında belirtilen veriyi, bir yada daha fazla mbuf yapısına kopyalar.*sendto* çağrısının ikinci argumanı verinin (*buffer*) başlangıcını gösterir ve üçüncü argümanı da verinin boyutunu belirtir. Şekil 1.5 150 byte'lık bir verinin nasıl iki tane mbuf'da tutulduğunu gösterir.



Şekil 1.5 150 byte'lık veriyi tutan iki mbuf yapısı

İki veya daha fazla mbuf yapılarının birbirleri ile bağlanmasına mbuf zinciri (mbuf chain) denir.Zincirdeki mbuf yapıları birbirlerine m_next işaretçisi ile bağlanmaktadır.

Bu tür yapılarda m_pkthdr.len ve m_pkthdr.rcvif olmak üzere mbuf başlığına iki üye daha eklenmektedir. Bu üyeler mbuf zincirinin ilk halkasında bulunur. Bu üyeler paket başlığını içerir ve sadece mbuf zincirinin ilk halkasında kullanılır. *m_flags* üyesi bu mbuf yapısının paket başlığını taşıdığını belirtmek için *M_PKTHDR* değerini almıştır.Paket başlığındaki *len* üyesi mbuf zincirindeki verilerin toplam boyutunu belirtir (şekildeki örnek için 150 byte'tır). *rcvif* üyesi de paketlerin alındığı arayüze işaret eden bir işaretçi içermektedir. Fakat şekil 1.5'de output işlemi için oluşturulan bir mbuf yapısı olduğu için bu işaretçi null değerini almıştır.

Bir mbuf yapısı her zaman 128 byte'tır. Mbuf zincirinin ilk halkası 100 byte veri ve diğer halkaları da 108 byte veri taşır. Eğer bir veri 208 byte'ı aşıyorsa, üç ya da daha fazla mbuf kullanmak yerine, cluster olarak adlandırılan bir yapı kullanılır.

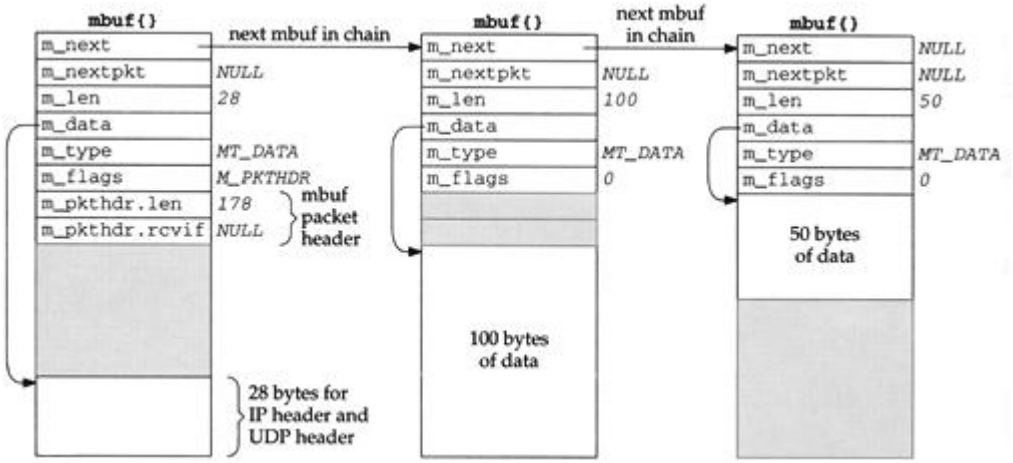
Zincirin ilk mbuf yapısında verinin toplam boyutu tutulur. Bunun nedeni ise, zincirdeki her bir mbuf taşıdığı verinin miktarının kendi *m_len* üyesinde saklasaydı, verinin toplam

boyutunu bilmek için zincirdeki her bir mbuf'ın *m_len* üyesini toplamak gerekirdi. Bu da gereksiz bir işlemdir.

IP ve UDP Başlıklarının Mbuf'a Eklenmesi

Soket katmanı soket adres yapısını mbuf'a (Şekil 1.4) ve veriyide mbuf zincirine (Şekil 1.5) kopyaladıktan sonra soket tanımlayıcısına bağlı olarak (Udp soket) ilgili protokol katmanı çağrılır.Belli bir şekilde, UDP output rutini çağrılır ve mbuf'ı gösteren bir işaretçi arguman olarak geçirilir.

Şekil 1.5'deki mbuf zincirinin başına veriyi eklemenin yolu, başka bir mbuf tahsis edip, yeni mbuf yapısının zincirin başı yapılıp ve 100 byte'lık veri taşıyan mbuf halkasındaki paket başlığını yeni mbuf halkasına kopyalanmasıdır. Şekil 1.6 birbiri ile bağlanmış üç mbuf yapısını göstermektedir.



Şekil 1.8 IP ve UDP başlıklarının eklendiği bir Mbuf zinciri

IP ve UDP başlığı zincirin başı olan yeni mbuf'ın sonuna yerleştirilir. Bu IP ve UDP başlıklarının kopyalanmasına gerek kalmadan, daha alt seviye protokollerin IP başlığın üstüne eklenmesini sağlar. Zincirin ilk mbuf yapısındaki *m_data* işaretçisi bu iki başlığın başlangıç adresini gösterir ve *m_len*'in değeri 28'dir (20+8). Daha sonraki başlıklar, paket başlığı ve IP başlığı arasındaki 72 byte'lık kısma yerleşir.*m_data* ve *m_len* alanların değerleri yapılan değişikliğe göre yeni değerler atanır.

Şekilde görüldüğü gibi paket başlığı 100 byte'lık veri taşıyan mbuf'dan yeni mbuf'a taşınmıştır. Paket başlığı her zaman zincirin ilk mbuf yapısında olmak zorundadır.Bu taşınma için ikinci mbuf'dan *M_PKTHDR* bayrağı silinmiş ve yeni mbuf'a atanmıştır. İkinci mbuf'da paket başlığı için ayrılmış alan artık kullanım dışıdır.Son olarak paket başlığındaki len üyesi (m_pkthdr.len) 28 byte daha artırılarak 178 byte olmuştur.

UDP output rutini UDP başlığını oluşturur ve IP başlığı ile ilgili bazı kısımlarını doldurur. Mesela hedef IP adresini IP başlığına kaydeder fakat başlık cheksum değerini IP output rutinine bırakır.

UDP checksum değeri hesaplanır ve UDP başlığına kaydedilir. UDP checksum değerinin hesaplanması için mbuf zincirinde tutulan tüm veri (örnekte 150 byte'lık veri) UDP output rutinine geçirilmesi gerekir. Kullanıcının tampon belleğindeki veri, kernel'ın mbuf yapısına birkez kopyalanır.

Son olarak UDP output rutini mbuf zincirini gösteren bir işaretçiyi IP output rutinine geçirerek IP output rutinini çağırır. Aşağıdaki kodda *udp_output* fonksiyonun kodlarının bir kısmı gösterilmektedir. Kodun ilk bloğunda IP başlığına IP adres bilgileri ve bir üst protokol bilgisi atanmaktadır. UDP başlığına ise port bilgileri atanmaktadır. Son olarakta *ip_output* fonksiyonu çağrılıyor.

```
netinet/udp_usrreq.c
static int
udp_output(inp, m, addr, control, td)
            register struct inpcb *inp;
            struct mbuf *m;
            struct sockaddr *addr;
            struct mbuf *control;
            struct thread *td;
{
                        .

.
  ui = mtod(m, struct udpiphdr *);
  bzero(ui->ui_x1, sizeof(ui->ui_x1));
  ui->ui_pr = IPPROTO_UDP;
  ui->ui_src = laddr;
  ui->ui_dst = faddr;
  ui->ui_sport = lport;
  ui->ui_dport = fport;
  ui->ui_ulen = htons((u_short)len + sizeof(struct udphdr));
                        .
                        .
  error = ip_output(m, inp->inp_options, NULL, ipflags,inp->inp_moptions, inp);
                        .
                        .
  }
```

IP Output

IP output rutini IP başlığı ile ilgili geri kalan işlemleri yapar. IP checksum değerinin hesaplanması, paketin gideceği arayüze karar verilmesi, IP datagramının parçalanıp parçalanmayacağına (fragmentation) karar verilmesi gibi işlemleri yapar. Bu işlemleri yaptıktan sonra ilgili arayüzün output fonksiyonunu çağırır.

Arayüzün Ethernet olduğu varsayılırsa, ethernet output fonksiyonu çağrılır ve parametre olarak da mbuf zincirine işaret eden bir işaretçi geçirilir.

Ethernet Output

Ethernet output fonksiyonun ilk işlemi 32 bit Ip adresini 48 bit Ethernet adresine dönüşümünü sağlamaktır. Bunun için ARP kullanılır. ARP request paketi gönderilir ve ARP reply paketi beklenir.

Ethernet output rutini 14 byte'lık ethernet başlığını zincirdeki ilk mbuf'a ip başlığından önce yerleştirir. Bu başlıkta 6 byte hedef ethernet adresi, 6 byte ethernet kaynak adresi, 2 byte'da ethernet frame tipi yer alır.

Mbuf zinciri daha sonra arayüzün output kuyruğunun sonuna yerleştirilir. Arayüz meşgul değilse, arayüzün "start output" rutini çağrılır. Eğer arayüz meşgul ise, işlemini bitirdiği zaman output kuyruğunda bulunan yeni bir mbuf'ı işleme girecektir.

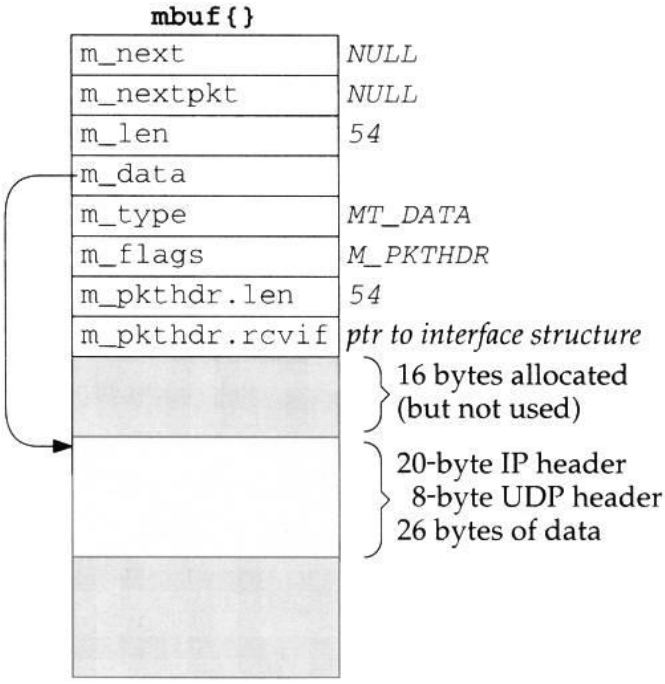
Arayüz output kuyruğundaki mbuf işleme girdiği zaman, mbuf zincirindeki verinin tamamı iletim tampon belleğine (transmit buffer) kopyalanır ve outputu başlatır. Örnek olarak, 14 byte ethernet başlığı, 20 byte IP başlığı, 8 byte UDP başlığı ve 150 byte toplam olarak 192 byte veri iletim tampon belleğine kopyalanacaktır. Mbuf zincirindeki veri aygıtın iletim tampon belleğine kopyalandıktan sonra, mbuf zinciri ethernet aygıt sürücüsü tarafından yok edilir.

1.4 Input İşleme

Input asenkron (asynchronous) olduğundan output işlemeden daha farklı bir işlem yapılır.Yani input paketlerin elde edilmesi, proses tarafından bir sistem çağrısı yapılması yerine, ethernet aygıt sürücüsüne kesme(interrupt) gönderilir.

Ethernet Input

Ethernet aygıt sürücüsü kesmeleri işleme alır ve aygıttan okunan veriler mbuf zincirine taşınır.Örnek olarak, 54 byte'lık verinin okunup mbuf zincirine kopyalanır. 54 byte'lık verinin, 20 byte'ı IP başlığı, 8 byte'ı UDP başlığı ve 26 byte'ı veridir.



Şekil 1.7 Ethernet'in input'tan aldığı veriyi içeren tekli mbuf yapısı

Bu mbuf veriyi içeren ilk mbuf olduğundan paket başlığı taşımaktadır (*M_PKTHDR* bayrağı *m_flags* üyesinde belirtilmiştir). Paket başlığındaki *len* üyesi verinin toplam boyutunu içerir ve *rcvif* üyesi de paketlerin alındığı arayüz ile ilgili olan arayüz yapısına (interface structure) işaret eden bir işaretçidir. rcvif üyesi sadece alınan paketler için kullanılır, gönderilen paketler için kullanılmaz.

Mbuf'ın verilerin tutulduğu yerin ilk 16 byte'ı arayüz katmanı için ayrılmıştır fakat kullanılmaz. Bu örnekte veri (54 byte) geriye kalan 84 byte'lık kısma sığabildiğinden, veri tek bir mbuf'da yerleştirilmiştir.

Aygıt sürücüsü alınan paketin hangi protokol tarafından işleneceğine karar vermek için, mbuf'ı ethernet input rutinine aktarır ve ethernet input rutini ethernet başlığın tip (type) alanına bakarak karar verir. Şekildeki örneği dikkate alarak, tip alanı IP protokolünü belirttiğinden, mbuf IP input kuyruğuna eklenecektir. IP input rutinin çalıştırılaması için bir yazılımsal kesme (software interrupt) gönderilir.

IP Input

IP input asenkrondur ve yazılımsal kesme tarafından çalışması için ayarlanır. Sistemin arayüzlerinin birinden paket alındığı zaman, arayüz katmanı tarafından yazılımsal kesme oluşturulur. IP rutini döngüyü çalıştırma başladığı zaman, input kuyruğundaki her bir datagramı işlemeye başlar ve kuyruğun tamamını işledikten sonra rutinden döner.

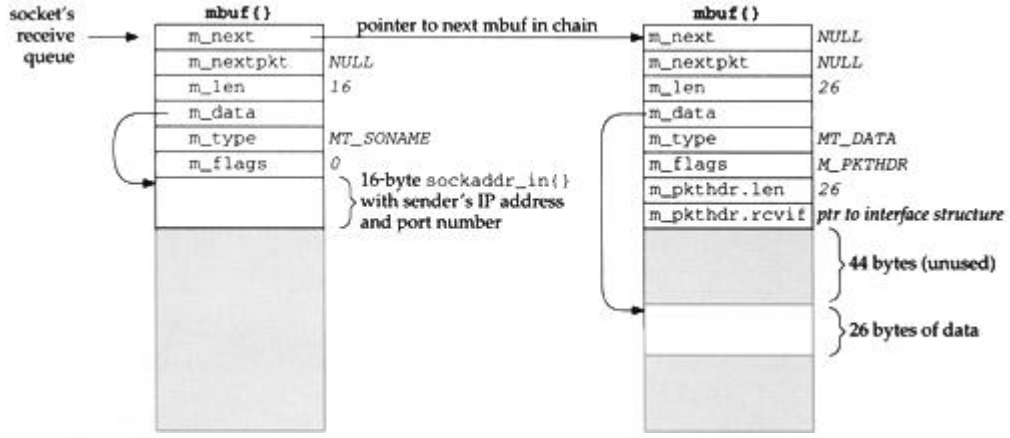
IP rutini aldığı her bir datagramı işler. IP başlığının checksum değerini kontrol eder, datagramın doğru hosta gelip gelmediğini kontrol eder ve sistem yönlendirici olarak ayarlandı ise datagramı istenen IP adresine iletir. IP datagramı son hedefe ulaştı ise, IP başlığındaki protokol alanı hangi protokolün input rutini çağırılacağını belirtir (ICMP, IGMP, TCP ya da UDP gibi). Şekil 1.7'de IP başlığından sonra UDP başlığı geldiğinden UDP input rutini çağrılır.

UDP Input

UDP input rutini UDP başlığındaki alanları (uzunluk ve checksum) kontrol eder ve alınan datagramın işlenip işlenmeyeceğine karar verir. Bir proses belli bir UDP portundaki tüm datagramları alabilir ya da kernel'dan hedef ip, kaynak ip , hedef port ve kaynak port'a göre datagramların kısıtlanmasını isteyebilir.

UDP rutini global bir değişken olan udb'den (Şekil 1.3) başlar ve UDP kontrol blok bağlı listesi boyunca, bağlı listedeki yerel port numarasını (inp_local) gelen UDP datagramındaki hedef port numarasına eşleşen bir kontrol blok arar. Protokol kontrol bloğu soket tarafından oluşturulduğundan ve eşleşen bir protokol kontrol blok bulunduğu zaman, protokol kontrol bloktaki *inp_socket* işaretçisine bakılarak istenen sokete ulaşılır ve gelen veri soketin kuyruğuna bırakılır.

Prosese UDP datagramı ulaştığından, göndericinin IP adresi ve UDP port numarası mbuf'a yerleştirilir ve bu mbuf ile veri soketin gelen kuyruğunun sonuna eklenir. Şekil 1.8 soketin gelen kuyruğuna eklenen iki mbuf'ı göstermektedir.



Şekil 1.8 Göndericinin adresi ve verisi

Bu zincirdeki ikinci mbuf ile şekil 1.7'deki mbuf karşılaştırıldığında, *m_len* ve *m_pkthdr.len* üyeleri 28 (20 bytes IP başlığı ve 8 byte UDP başlığı) azalmıştır ve *m_data* işaretçisinin değeri 28 artmıştır. Soketin gelen kuyruğuna ekleneceği zaman UDP ve UP başlıkları silinir ve sadece veri bırakılır.

Zincirdeki ilk mbuf 16 byte'lık İnternet soket adres yapısını içerir. Bu adres yapısında göndericinin IP adresi ve UDP port numarası bulunmaktadır. Bu mbuf yapısının tipi MT_SONAME'dir. *recfrom* ya da *recvmsg* gibi sistem çağrıları kullanan prosese bu bilgileri ulaştırmak için, bu mbuf soket katmanı tarafından oluşturulmuştur. Zincirdeki ikinci mbuf'da soket adres yapısı için 16 byte'lık yer olsa bile, bu bilgilerin iletilmesi için başka bir mbuf oluşturulmak zorundadır. Çünkü herbir mbuf'ın farklı tip değerleri vardır (MT_SONAME, MT_DATA gibi).

Alıcı proses bundan sonra uyandırılır. Eğer proses verinin ulaşması için bekliyorsa, bu proses run-able olarak işaretlenir. Bir proses aynı zamanda select sistem çağrısı ya da SIGIO sinyali ile de sokete verinin ulaştığını anlayabilir.

Process Input

UDP katmanı tarafından soketin gelen kuyruğuna eklenen veri kernel tarafından mbuf'dan programın tampon belleğine kopyalanır.

recvfrom(fd, buffer, BUFFSIZE, 0, (struct sockaddr*)NULL, (int *) NULL)

Yukarıdaki recvfrom sistem çağrısında görüldüğü gibi beşinci ve altıncı parametresi null verilmiştir. Bu gönderici IP adresinin ve UDP port numarasının dikkate alınmayacağını belirtmektedir. Bu şekilde yapılan bir recvfrom sistem çağrısı ile zincirin ilk mbuf yapısı atlanır ve sadece diğer mbuf'lardaki veri döndürülür.Bu işlemden sonra kernel'ın recvfrom kodu mbuf zincirini yok eder.

1.5 Kesme Derecesi ve Eş Zamanlılık (Interrupt level and Concurrency)

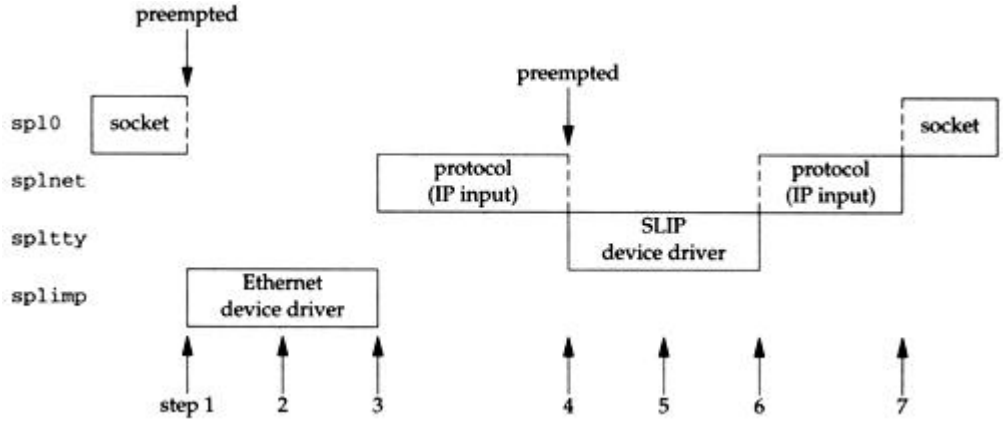
Input'taki paketlerin işlenmesi asenkron bir yapıdadır ve kesmelidir. İlk olarak, bir aygıt kesmesi (device interrupt) arayüz katamanı kodunun çalışmasını sağlar ve yazılımsal interrupt da protokol katmanı kodunun çalışmasını sağlar.Bu kesmeler bittiği zaman soket kodu çalışmaya başlar.

Yazılımsal ve donanımsal kesmelerin bir öncelik derecesi vardır. Şekil 1.9 düşük öncelikten yüksek önceliğe doğru sıralanmış sekiz öncelik derecesini gösterir.

Function	Description
spl0	normal operating mode, nothing blocked (lowest priority)
splsoftclock	low-priority clock processing
splnet	network protocol processing
spltty	terminal I/O
splbio	disk and tape I/O
splimp	network device I/O
splclock	high-priority clock processing
splhigh	all interrupts blocked (highest priority)
splx(s)	(see text)

Şekil 1.9 Kesmelerin kernel fonksiyonları

Kesmelerin öncelik derecelerinin olmasının anlamı, yüksek öncelikli bir kesme daha düşük öncelikli bir kesmeyi etkisiz hale getirip yüksek öncelikli kesmenin işlem görmesidir. Şekil 1.10 yüksek ve düşük öncelikli kesmelerin davranışını göstermektedir.



Şekil 1.10 Öncelik dereceleri ve kernel'ın kesmeleri işlemesi

1. Soket katmanı spl0'da çalışıyorken, ethernet aygıt sürücüsü kesmesinin oluşması arayüz katmanının splimp'de çalışmasına neden olur.Bu kesme soket katmanı kodunun çalışmasını etkisizleştirir.Bu, Arayüz input rutininin çalışması asenkrondur.
2. Ethernet aygıt sürücüsü çalışıyorken, IP input kuyruğuna bir paket yerleşir ve splnet seviyesinde çalışması için yazılımsal bir kesme oluşturulur. Fakat kernel daha yüksek

bir seviyede (splimp) çalıştığından bu yazılımsal kesmenin birden bir etkisi olmayacaktır.

3. Ethernet aygıt sürücüsü işlemini bitirdiği zaman, protokol katmanı splnet'de çalışır.Bu, IP input rutinlerinin çalışması asenkrondur.
4. Bir teminal aygıtının kesmesi oluşur ve terminal I/O (spltty) protoko katmanından (splnet) daha yüksek önceliği olduğundan bu kesme dikkate alınır ve protokol katmanı etkisiz hale getirilir.Bu, arayüz input rutininin çalışması asenkrondur.
5. SLIP sürücüsü alınan paketleri IP input kuyruğuna yerleştirir ve protokol katmanı için başka bir yazılımsal kesme oluşturur.
6. SLIP sürücüsü tamamlandığı zaman, etkisiz hale getirilen protokol katmanı splnet'de çalışmaya devam eder. Ethernet aygıt sürücüsünden alınan paketlerin işlenmesini bitirir ve sonra SLIP sürücüsünden alınan paketleri işleme alır.Input kuyruğunda işleme alınacak daha fazla paket kalmadığı zaman, daha önceden etkisiz hale getirilen her ne ise kontrol ona geçenecektir (bu örnekte soket katmanıdır).
7. Soket katmanı etkisiz hale getirildiği yerden devam eder.

Farklı seviyeler arasındaki paylaşılmış veri yapıları farklı öncelik seviyelerinin nasıl işleme aldığı dikkat edilmesi gereken diğer bir konudur.Paylaşılmış veri yapılarına örnek olarak farklı seviyelerdeki katmanlar verilebilir, soket, arayüz ve protokol kuyrukları gibi. Örneğin, IP input rutini kendi input kuyruğundan bir paket alıyorken, bir aygıt kesmesi oluşsun, protokol katmanını etkisiz hale getirsin ve aygıt sürücüsü IP input kuyruğuna başka bir paket eklesin.Bu durumda paylaşılmış veri yapıları(bu örnekte IP input kuyruğu, protokol katmanı ve arayüz katmanı arasında paylaşılır) bozulabilir eğer hiçbirşey yapılmazsa.

Net/3 kodu splimp ve splnet fonksiyon çağrıları arasına yayılmıştır.İşlemcinin bir önceki seviyeye döndürmek için bu iki fonksiyon çağrısı her zaman splx çağrısı ile birlikte kullanılır.Örnek olarak, Aşağıdaki kod input kuyruğunda işleme alınacak başka bir paket olup olmadığının kontrolü için protokol katmanında IP input fonksiyonları tarafından kullanılır.

```
        struct mbuf *m;
   int  s;

   s = splimp();
   IF_DEQUEUE(&ipintrq, m);
   splx(s);

   if (m == 0)
      return;
```

Herhangi bir network aygıt sürücüsü kesmesinin oluşmasını engellemek için CPU önceliğini network aygıt sürücülerinin kullandığı seviyeye splimp çağrısı ile yükseltilir. Daha önce kullanılan seviye fonksiyondan döndürülür ve s değişkenine katdedilir.Daha sonra IF_DEQUEUE macrosu çalıştırılır ve IP input kuyruğunun başındaki paket çıkarılır ve m değişkenine mbuf zincirini gösteren bir pointer yerleştirilir.Son olarak splimp çağrılmadan önceki seviye ne ise splx çağrısının s parametresi ile o seviyeye getirilir.

splimp ve splx arasındaki bütün network aygıt sürücüsü kesmeleri iptal edildiğinden, bu çağrılar arasındaki kod miktarı en az seviyededir. Eğer daha uzun bir süre kesmeler iptal edilirse, ilave aygıt kesmeleri dikkate alınmayabilir ve veri kaybı oluşabilir.Bu nedenle splx çağrısından sonra –önce değil- m değişkeninin testi gerçekleştirilir (işlenecek başka bir paket olup olmadığını görmek için).

Ethernet giden bir paketi arayüzün kuyruğuna yerleştirdiği zaman , ethernet output rutini spl çağrılarına ihtiyaç duyar ve arayüzün meşgul olup olmadığını test eder, meşgul değilse arayüzü başlatır.

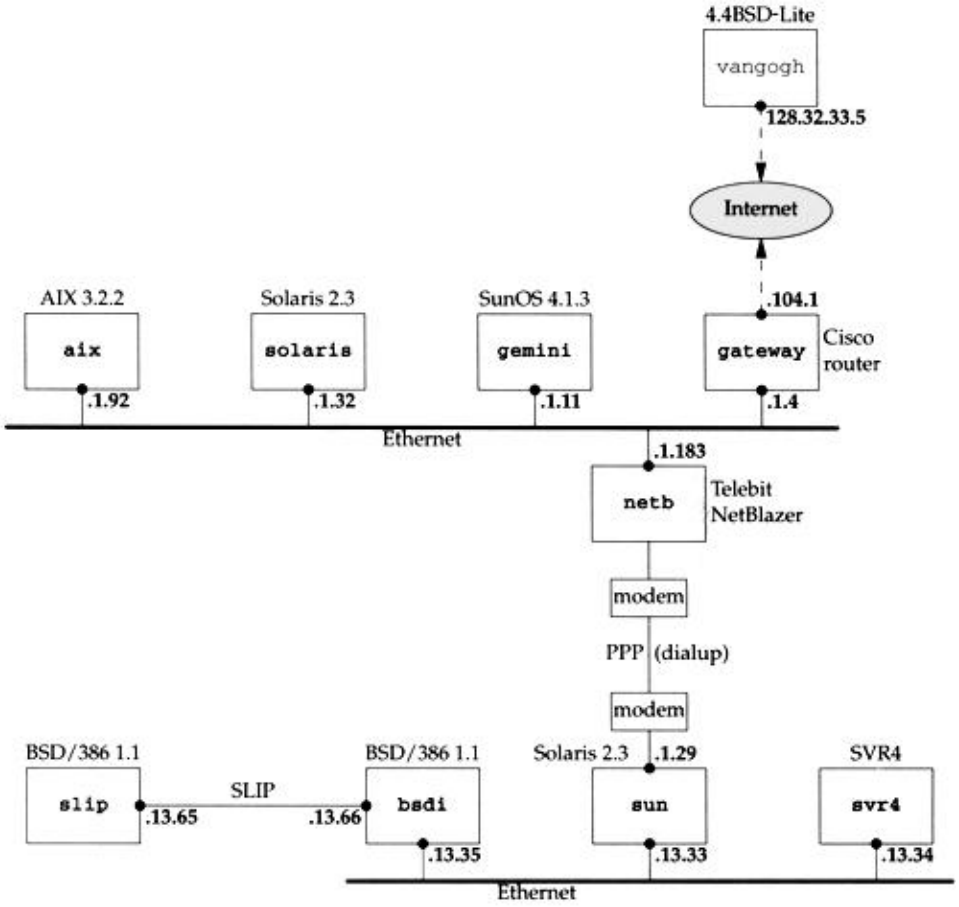
```
struct mbuf  *m;
   int  s;

   s = splimp();
   /*
```

```
     * Queue message on interface, and start output if interface not active.
 */
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);    /* queue is full, drop packet */
    splx(s);
    error = ENOBUFS;
    goto bad;
}

IF_ENQUEUE(&ifp->if_snd, m);  /* add the packet to interface queue */
if ((ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp);    /* start interface */

 splx(s);
```

Bu örnekte aygıt kesmelerinin iptal edilmesinin nedeni protokol katmanı giden kuyruğuna (send queue) paket ekliyorken, aygıt sürücüsünün giden kuyruğundan paket almasını engellemektir. Sürücünün giden kuyruğu protokol katmanı ve arayüz katmanı arasında bir paylaşılmış veri yapısıdır.

1.6 Test Network

Bundan sonra kullanılacak örneklerde Şekil 1.11'deki örnek network kullanılacaktır.Vangogh hostu haric, ip adreslerin tamamı B sınıfı networke ait olup network ID'si 140.252'dir. Host isimleri .tuc.noao.edu domain'ine aittir. Örnek olarak, sağ alttaki sistemin tam host ismi svr4.tuc.noao.edu olup IP adresi 140.252.13.34'dür.



Şekil 1.11 Diğer örneklerde kullanılacak test networkü