

# 第1章 概述

## 1.1 引言

本章介绍伯克利(Berkeley)联网程序代码。开始我们先看一段源代码并介绍一些通篇要用的印刷约定。对各种不同代码版本的简单历史回顾让我们可以看到本书中的源代码处于什么位置。接下来介绍了两种主要的编程接口，它们在 Unix与非Unix系统中用于编写TCP/IP协议。

然后我们介绍一个简单的用户程序，它发送一个 UDP数据报给一个位于另一主机上的日期/时间服务器，服务器返回一个 UDP数据报，其中包含服务器上日期和时间的 ASCII码字符串。这个进程发送的数据报经过所有的协议栈到达设备驱动器，来自服务器的应答从下向上经过所有协议栈到达这个进程。通过这个例子的这些细节介绍了很多核心数据结构和概念，这些数据结构和概念在后面的章节中还要详细说明。

本章的最后介绍了在本书中各源代码的组织，并显示了联网代码在整个组织中的位置。

## 1.2 源代码表示

不考虑主题，列举 15 000行源代码本身就是一件难事。下面是所有源代码都使用的文本格式：

```
-----tcp_subr.c
381 void
382 tcp_quench(inp, errno)
383 struct inpcb *inp;
384 int     errno;
385 {
386     struct tcpcb *tp = intotcpcb(inp);

387     if (tp)
388         tp->snd_cwnd = tp->t_maxseg;
389 }
```

### 1.2.1 将拥塞窗口设置为 1

387-388 这是文件tcp\_subr.c中的函数tcp\_quench。这些源文件名引用4.4BSD-Lite发布的文件。4.4BSD在1.13节中讨论。每个非空白行都有编号。正文所描述的代码的起始和结束位置的行号记于行开始处，如本段所示。有时在段前有一个简短的描述性题头，对所描述的代码提供一个概述。

这些源代码同4.4BSD-Lite发行版一样，偶尔也包含一些错误，在遇到时我们会提出来并加以讨论，偶尔还包括一些原作者的编者评论。这些代码已通过了 GNU 缩进程序的运行，使它们从版面上看起来具有一致性。制表符的位置被设置成 4个栏的界线使得这些行在一个页面中显示得很合适。在定义常量时，有些 #ifdef语句和它们的对应语句 #endif被删去(如：GATEWAY和MROUTING，因为我们假设系统被作为一个路由器或多播路由器)。所有register说

明符被删去。有些地方加了一些注释，并且一些注释中的印刷错误被修改了，但代码的其他部分被保留下来。

这些函数大小不一，从几行(如前面的tcp\_quench)到最大1100行(tcp\_input)。超过大约40行的函数一般被分成段，一段一段地显示。虽然尽量使代码和相应的描述文字放在同一页或对开的两页上，但为了节约版面，不可能完全做到。

本书中有很多对其他函数的交叉引用。为了避免给每个引用都添加一个图号和页码，书封底内页中有一个本书中描述的所有函数和宏的字母交叉引用表和描述的起始页码。因为本书的源代码来自公开的4.4BSD\_Lite版，因此很容易获得它的一个拷贝：附录B详细说明了各种方法。当你阅读文章时，有时它会帮助你搜索一个在线拷贝[例如Unix程序grep(1)]。

描述一个源代码模块的各章通常以所讨论的源文件的列表开始，接着是全局变量、代码维护的相关统计以及一个实际系统的一些例子统计，最后是与所描述协议相关的SNMP变量。全局变量的定义通常跨越各种源文件和头文件，因此我们将它们集中到的一个表中以便于参考。这样显示所有的统计，简化了后面当统计更新时对代码的讨论。卷1的第25章提供了SNMP的所有细节。我们在本文中关心的是由内核中的TCP/IP例程维护的、支持在系统上运行的SNMP代理的信息。

### 1.2.2 印刷约定

通篇的图中，我们使用一个等宽字体表示变量名和结构成员名(m\_next)，用斜体等宽字体表示定义常量(NULL)或常量的值(512)的名称，用带花括号的粗体等宽字体表示结构名称(mbuf{ })。这里有一个例子：

mbuf{ }	
m_next	<i>NULL</i>
m_len	<i>512</i>

在表中，我们使用等宽字体表示变量名称和结构成员名称，用斜体等宽字体表示定义的常量。这里有一个例子：

m_flags	说 明
<i>M_BCAST</i>	以链路层广播发送/接收

通常用这种方式显示所有的#define符号。如果必要，我们显示符号的值(M\_BCAST的值无关紧要)并且所列符号按字母排序，除非对顺序有特殊要求。

通篇我们会使用像这样的缩进的附加说明来描述历史的观点或实现的细节。

我们用有一个数字在圆括号里的命令名称来表示Unix命令，如grep(1)。圆括号中的数字是4.4BSD手册“manual page”中此命令的节号，在那里可以找到其他的信息。

## 1.3 历史

本书讨论在伯克利的加利福尼亚大学计算机系统研究组的TCP/IP实现的常用引用。历史上，它曾以4.x BSD系统(伯克利软件发行)和“BSD联网版本”发行。这个源代码是很多其他实现的起点，不论是Unix或非Unix操作系统。

图1-1显示了各种BSD版本的年表，包括重要的TCP/IP特征。显示在左边的版本是公开可

源代码版，它包括所有联网代码：协议本身、联网接口的内核例程及很多应用和实用程序(如Telnet和FTP)。

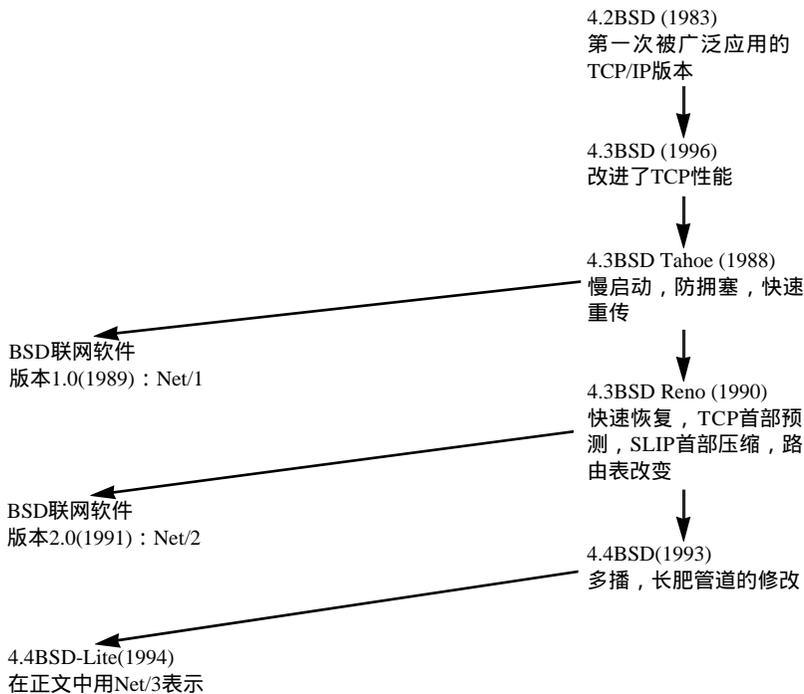


图1-1 带有重要TCP/IP特征的各种BSD版本

虽然本文描述的软件的官方名称为4.4BSD-Lite发行软件，但我们简单地称它为Net/3。

虽然源代码由U. C. Berkeley发行并被称为伯克利软件发行，但TCP/IP代码确实是各种研究者的工作的融合，包括伯克利和其他地区的研究人员。

通篇我们会使用术语源于伯克利的实现来谈及各厂商的实现，如SunOS 4.x、系统V版本4(SVR4)和AIX 3.2，它们的TCP/IP代码最初都是从伯克利源代码发展而来的。这些实现有很多共同之处，通常包括同样的错误！

在图1-1中没有显示的伯克利联网代码的第1版实际上是1982年的4.1cBSD，但是广泛发布的是1983年的版本4.2BSD。

在4.1cBSD之前的BSD版本使用的一个TCP/IP实现，是由Bolt Beranek and Newman(BBN)的Rob Gurwitz和Jack Haverty开发的。[Salus 1994]的第18章提供了另外一些合并到4.2BSD中的BBN代码细节。其他对伯克利TCP/IP代码有影响的实现是由Ballistics研究室的Mike Muuss为PDP-11开发的TCP/IP实现。

描述联网代码从一个版本到下一个版本的变化文档有限。 [Karels and McKusick 1986]描述了从4.2BSD到4.3BSD的变化，并且 [Jacobson 1990d]描述了从4.3BSD Tahoe到4.3BSD Reno的变化。

## 1.4 应用编程接口

在互联网协议中两种常用的应用编程接口 (API)是插口(socket)和TLI(运输层接口)。前者

有时称为伯克利插口(Berkeley socket),因为它被广泛地发布于4.2BSD系统中(见图1-1)。但它已被移植到很多非BSD Unix系统和很多非Unix系统中。后者最初是由AT&T开发的,由于被X/Open承认,有时叫作XTI(X/Open传输接口)。X/Open是一个计算机厂商的国际组织,它制定自己的标准。XTI是TLI的一个有效超集。

虽然本文不是一本程序设计书,但既然在Net/3(和所有BSD版本)中应用编程用插口来访问TCP/IP,我们还是说明一下插口。在各种非Unix系统中也实现了插口。插口和TLI的编程细节在[Stevens 1990]中可以找到。

系统版本4(SVR4)也为应用编程提供了一组插口API,在实现上与本文中列举的有所不同。在SVR4中的插口基于“流”子系统,在[Rago 1993]中有所说明。

## 1.5 程序示例

在本章我们用一个简单的C程序(图1-2)来介绍一些BSD网络实现的很多特点。

```
1 /*
2  * Send a UDP datagram to the daytime server on some other host,
3  * read the reply, and print the time and date on the server.
4  */
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <arpa/inet.h>
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #define BUFFSIZE 150 /* arbitrary size */
13 int
14 main()
15 {
16     struct sockaddr_in serv;
17     char buff[BUFFSIZE];
18     int sockfd, n;
19     if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
20         err_sys("socket error");
21     bzero((char *) &serv, sizeof(serv));
22     serv.sin_family = AF_INET;
23     serv.sin_addr.s_addr = inet_addr("140.252.1.32");
24     serv.sin_port = htons(13);
25     if (sendto(sockfd, buff, BUFFSIZE, 0,
26               (struct sockaddr *) &serv, sizeof(serv)) != BUFFSIZE)
27         err_sys("sendto error");
28     if ((n = recvfrom(sockfd, buff, BUFFSIZE, 0,
29                      (struct sockaddr *) NULL, (int *) NULL)) < 2)
30         err_sys("recvfrom error");
31     buff[n - 2] = 0; /* null terminate */
32     printf("%s\n", buff);
33     exit(0);
34 }
```

图1-2 程序示例：发送一个数据报给UDP日期/时间服务器并读取一个应答

### 1. 创建一个数据报插口

19-20 `socket`函数创建了一个UDP 插口，并且给进程返回一个保存在变量 `sockfd`中的描述符。差错处理函数 `err_sys`在[Stevens 1992]的附录B.2中给出。它接收任意数量的参数，并用 `vsprintf`对它们格式化，将系统调用产生的 `errno`值对应的Unix错误信息打印出来，并中断进程。

我们在不同的地方使用术语插口：(1)为4.2BSD开发的程序用来访问网络协议的API通常叫插口API或者就叫插口接口；(2) `socket`是插口API中的一个函数的名字；(3)我们把调用 `socket`创建的端点叫做一个插口，如评注“创建一个数据报插口”。

但是这里还有一些地方也使用术语插口：(4) `socket`函数的返回值叫一个插口描述符或者就叫一个插口；(5)在内核中的伯克利联网协议实现叫插口实现，相比较其他系统如：系统V的流实现。(6)一个IP地址和一个端口号的组合叫一个插口，IP地址和端口号对叫一个插口对。所幸的是引用哪一种术语是很明显的。

### 2. 将服务器地址放到结构 `sockaddr_in`中

21-24 在一个互联网插口地址结构中存放日期/时间服务器的IP地址(140.252.1.32)和端口号(13)。大多数TCP/IP实现都提供标准的日期/时间服务器，它的端口号为13 [Stevens 1994，图1-9]。我们对服务器主机的选择是随意的——直接选择了提供此服务的本地主机(图1-17)。

函数 `inet_addr`将一个点分十进制表示的IP地址的ASCII字符串转换成网络字节序的32 bit二进制整数。(Internet协议族的网络字节序是高字节在后)。函数 `htons`把一个主机字节序的短整数(可能是低字节在后)转换成网络字节序(高字节在后)。在Sparc这种系统中，整数是高字节在后的格式，`htons`典型地是一个什么也不做的宏。但是在低字节在后的80386上的BSD/386系统中，`htons`可能是一个宏或者是一个函数，来完成一个16 bit整数中的两个字节的交换。

### 3. 发送数据报给服务器

25-27 程序调用 `sendto`发送一个150字节的数据报给服务器。因为是运行时栈中分配的未初始化数组，150字节的缓存内容是不确定的。但没有关系，因为服务器根本就不看它收到的报文的内容。当服务器收到一个报文时，就发送一个应答给客户端。应答中包含服务器以可读格式表示的当前时间和日期。

我们选择的150字节的客户数据报是随意的。我们有意选择一个报文长度在100~208之间的值，来说明在本章的后面要提到的 `mbuf`链表的使用。为了避免拥塞，在以太网中，我们希望长度要小于1472。

### 4. 读取从服务器返回的数据报

28-32 程序通过调用 `recvfrom`来读取从服务器发回的数据报。Unix服务器典型地发回一个如下格式的26字节字符串

```
Sat Dec 11 11:28:05 1993\r\n
```

`\r`是一个ASCII回车符，`\n`是ASCII换行符。我们的程序将回车符替换成一个空字节，然后调用 `printf`输出结果。

在本章和下一章我们在分析函数 `socket`、`sendto`和 `recvfrom`的实现时，要进入这个例子的一些细节部分。

## 1.6 系统调用和库函数

所有的操作系统都提供服务访问点，程序可以通过它们请求内核中的服务。各种 Unix 都提供精心定义的有限个内核入口点，即系统调用。我们不能改变系统调用，除非我们有内核的源代码。Unix 第7版提供大约 50 个系统调用，4.4BSD 提供大约 135 个，而 SVR4 大约有 120 个。

在《Unix 程序员手册》第2节中有系统调用接口的文档。它是以 C 语言定义的，在任何给定的系统中无需考虑系统调用是如何被调用的。

在各种 Unix 系统中，每个系统调用在标准 C 函数库中都有一个相同名字的函数。一个应用程序用标准 C 的调用序列来调用此函数。这个函数再调用相应的内核服务，所使用的技术依赖于所在系统。例如，函数可能把一个或多个 C 参数放到通用寄存器中，并执行几条机器指令产生一个软件中断进入内核。对于我们来说，可以把系统调用看成 C 函数。

在《Unix 程序员手册》的第3节中为程序员定义了一般用途的函数。虽然这些函数可能调用一个或多个内核系统调用但没有进入内核的入口点。如函数 `printf` 可能调用了系统调用 `write` 去执行输出，而函数 `strcpy` (复制一个串) 和 `atoi` (将 ASCII 码转换成整数) 完全不涉及操作系统。

从实现者的角度来看，一个系统调用和库函数有着根本的区别。但在用户看来区别并不严重。例如，在 4.4BSD 中我们运行图 1-2 中的程序。程序调用了三个函数：`socket`、`sendto` 和 `recvfrom`，每个函数最终调用了内核中同样名称的函数。在本书的后面我们可以看到这三个系统调用的 BSD 内核实现。

如果我们在 SVR4 中运行这个程序，在那里，用户库中的插口函数调用“流”子系统，那么三个函数同内核的相互作用是完全不同的。在 SVR4 中对 `socket` 的调用最终调用内核 `open` 系统调用，操作文件 `/dev/udp` 并将流模块 `sockmod` 放置到结果流。调用 `sendto` 导致一个 `putmsg` 系统调用，而调用 `recvfrom` 导致一个 `getmsg` 系统调用。这些 SVR4 的细节在本书中并不重要，我们仅仅想指出的是：实现可能不同但都提供相同的 API 给应用程序。

最后，从一个版本到下一个版本的实现技术可能会改变。例如，在 Net/1 中，`send` 和 `sendto` 是分别用内核系统调用实现的。但在 Net/3 中，`send` 是一个调用系统调用 `sendto` 的库函数：

```
send(int s, char *msg, int len, int flags)
{
    return(sendto(s, msg, len, flags, (struct sockaddr *) NULL, 0));
}
```

用库函数实现 `send` 的好处是仅调用 `sendto`，减少了系统调用的个数和内核代码的长度。缺点是由于多调用了函数，增加了进程调用 `send` 的开销。

因为本书是说明 TCP/IP 的伯克利实现的，大多数进程调用的函数 (`socket`、`bind`、`connect` 等) 是直接由内核系统调用来实现。

## 1.7 网络实现概述

Net/3 通过同时对多种通信协议的支持来提供通用的底层基础服务。的确，4.4BSD 支持四种不同的通信协议族：

- 1) TCP/IP (互联网协议族)，本书的主题。
- 2) XNS (Xerox 网络系统)，一个与 TCP/IP 相似的协议族；在 80 年代中期它被广泛应用于连

接Xerox设备(如打印机和文件服务器),通常使用的是以太网。虽然Net/3仍然发布它的代码,但今天已很少使用这个协议了,并且很多使用伯克利TCP/IP代码的厂商把XNS代码删去了(这样他们就不需要支持它了)。

3) OSI协议[Rose 1990; Piscitello and Chapin 1993]。这些协议是在80年代作为开放系统技术的最终目标而设计的,来代替所有其他通信协议。在90年代初它没有什么吸引力,以致于在真正的网络中很少被使用。它的历史地位有待进一步确定。

4) Unix域协议。从通信协议是用来在不同的系统之间交换信息的意义上来说,它还不是一套真正的协议,但它提供了一种进程间通信(IPC)的形式。

相对于其他IPC,例如系统V消息队列,在同一主机上两个进程间的IPC使用Unix域协议的好处是Unix域协议用与其他三种协议同样的API(插口)访问。另一方面,消息队列和大多数其他形式IPC的API与插口和TLI完全不同。在同一主机上的两进程间的IPC使用网络API,更容易将一个客户/服务器应用程序从一台主机移植到多台主机上。在Unix域中提供两个不同的协议——一个是可靠的,面向连接的,与TCP相似的字节流协议;一个是不可靠的,无连接的,与UDP相似的数据报协议。

虽然Unix域协议可以作为一种同一主机上两进程间的IPC,但也可以用TCP/IP来完成它们之间的通信。进程间通信并不要求使用在不同的主机上的互联网协议。

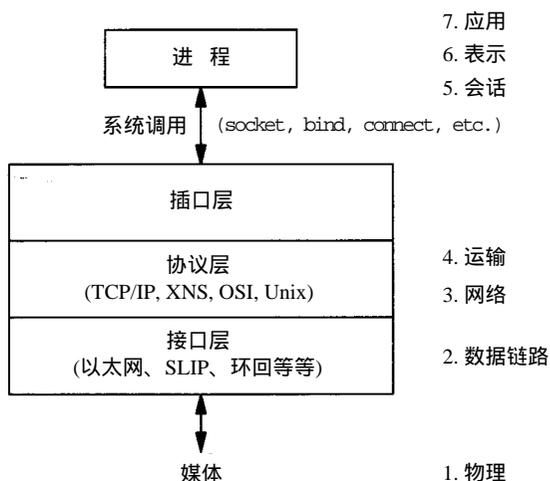
内核中的联网代码组织成三层,如图1-3所示。在图的右侧我们注明了OSI参考模型[Piscitello和Chapin 1994]的七层分别对应到BSD组织的哪里。

1) 插口层是一个到下面协议相关层的协议无关接口。所有系统调用从协议无关的插口层开始。例如:在插口层中的bind系统调用的协议无关代码包含几十行代码,它们验证的第一个参数是一个有效的插口描述符,并且第二个参数是一个进程中的有效指针。然后调用下层的协议相关代码,协议相关代码可能包含几百行代码。

2) 协议层包括我们前面提到的四种协议族(TCP/IP, XNS, OSI和Unix域)的实现。

每个协议族可能包含自己的内部结构,在图1-3中我们没有显示出来。例如,在Internet协议族中,IP(网络层)是最低层,TCP和UDP两运输层在IP的上面。

3) 接口层包括同网络设备通信的设备驱动程序。



## 1.8 描述符

图1-2中,一开始调用socket,这要求定义插口类型。Internet协议族(PF\_INET)和数据报插口(SOCK\_DGRAM)组合成一个UDP协议插口。

socket的返回值是一个描述符,它具有其他Unix描述符的所有特性:可以用这个描述符调用read和write;可以用dup复制它,在调用了fork后,父进程和子进程可以共享

它；可以调用 `fcntl` 来改变它的属性，可以调用 `close` 来关闭它，等等。在我们的例子中可以看到插口描述符是函数 `sendto` 和 `recvfrom` 的第一个参数。当程序终止时（通过调用 `exit`），所有打开的描述符，包括插口描述符都会被内核关闭。

我们现在介绍在进程调用 `socket` 时被内核创建的数据结构。在后面的几章中会更详细地描述这些数据结构。

首先从进程的进程表表项开始。在每个进程的生存期内都会有一个对应的进程表表项存在。

一个描述符是进程的进程表表项中的一个数组的下标。这个数组项指向一个打开文件表的结构，这个结构又指向一个描述此文件的 `i-node` 或 `v-node` 结构。图 1-4 说明了这种关系。

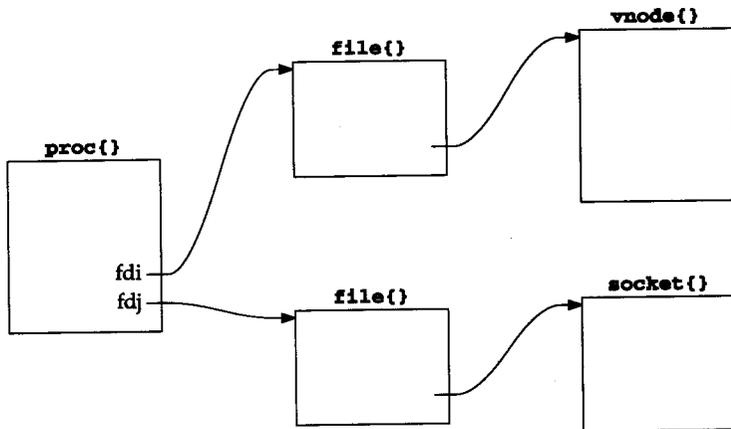


图 1-4 从一个描述符开始的内核数据结构的基本关系

在这个图中，我们还显示了一个涉及插口的描述符，它是本书的焦点。由于进程表表项是由以下 C 语言定义的，我们把记号 `proc{}` 放在进程表项的上面。并且在本书所有的图中都用它来标注这个结构。

```

struct proc {
    ...
}
  
```

[Stevens 1992, 3.10 节] 显示了当进程调用 `dup` 和 `fork` 时，描述符、文件表结构和 `i-node` 或 `v-node` 之间的关系是如何改变的。这三种数据结构的关系存在于所有版本的 Unix 中，但不同的实现细节有所变化。在本书中我们感兴趣的是 `socket` 结构和它所指向的 Internet 专用数据结构。但是既然插口系统调用以一个描述符开始，我们就需要理解如何从一个描述符导出一个 `socket` 结构。

如果程序如此执行

```
a.out
```

不重定向标准输入（描述符 0）、标准输出（描述符 1）和标准错误处理（描述符 2），图 1-5 显示了程序示例中的 Net/3 数据结构的更多细节。在这个例子中，描述符 0、1 和 2 连接到我们的终端，并且当 `socket` 被调用时未用描述符的最小编号是 3。

当进程执行了一个系统调用，如 `socket`，内核就访问进程表结构。在这个结构中的项 `p_fd` 指向进程的 `filedesc` 结构。在这个结构中两个我们现在关心的成员：一个是

fd\_ofileflags，它是一个字符数组指针（每个描述符有一个描述符标志）；一个是fd\_ofiles，它是一个指向文件表结构的指针数组的指针。描述符标志有8 bit，只有两位可为任何描述符设置：close-on-exec标志和mapped-from-device标志。在这里我们显示的所有标志都是0。

由于Unix描述符与很多东西有关，除了文件外，还有：插口、管道、目录、设

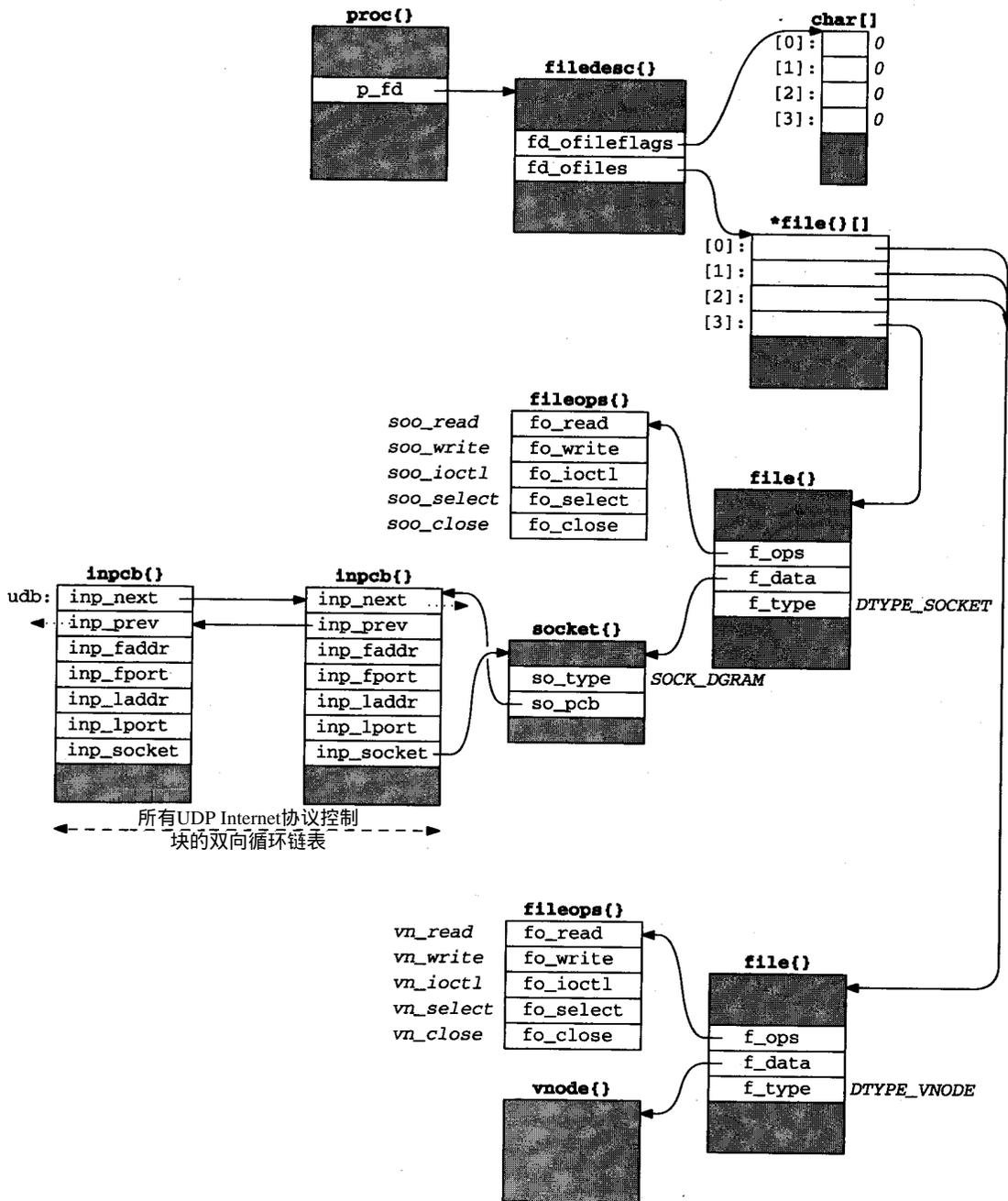


图1-5 在程序示例中调用socket后的内核数据结构

备等等，因此，我们有意把本节叫做“描述符”而不是“文件描述符”。但是很多 Unix 文献在谈到描述符时总是加上“文件”这个修饰词，其实没有必要。虽然我们要说明的是插口描述符，但这个内核数据结构叫 `filedesc{}`。我们尽可能地使用描述符这个未加修饰的术语。

项 `fd_ofiles` 指向的数据结构用 `*file{ }[]` 来表示。它是一个指向 `file` 结构的指针数组。这个数组及描述符标志数组的下标就是描述符本身：0、1、2 等等，是非负整数。在图 1-5 中我们可以看到描述符 0、1、2 对应的项指向图底部的同一个 `file` 结构（由于这三个描述符都对应终端设备）。描述符 3 对应的项指向另外一个 `file` 结构。

结构 `file` 的成员 `f_type` 指示描述符的类型是 `DTYPE_SOCKET` 和 `DTYPE_VNODE`。`vnode` 是一个通用机制，允许内核支持不同类型的文件系统——磁盘文件系统、网络文件系统（如 NFS）、CD-ROM 文件系统、基于存储器的文件系统等等。在本书中关心的不是 `vnode`，因为 TCP/IP 插口的类型总是 `DTYPE_SOCKET`。

结构 `file` 的成员 `f_data` 指向一个 `socket` 结构或者一个 `vnode` 结构，根据描述符类型而定。成员 `f_ops` 指向一个有 5 个函数指针的向量。这些函数指针用在 `read`、`readv`、`write`、`writew`、`ioctl`、`select` 和 `close` 系统调用中，这些系统调用需要一个插口描述符或非插口描述符。这些系统调用每次被调用时都要查看 `f_type` 的值，然后做出相应的跳转，实现者选择了直接通过 `fileops` 结构的相应项来跳转的方式。

我们用一个等宽字体 (`fo_read`) 来醒目地表示一个结构成员的名称，用斜体等宽字体 (`soo_read`) 来表示一个结构成员的内容。注意，有时我们用一个箭头指向一个结构的左上角（如结构 `filedesc`），有时用一个箭头指向右上角（如结构 `file` 和 `fileops`）。我们用这些方法来简化图例。

下面我们来查看结构 `socket`，当描述符的类型是 `DTYPE_SOCKET` 时，结构 `file` 指向结构 `socket`。在我们的例子中，`socket` 的类型（数据报插口的类型是 `SOCK_DGRAM`）保存在成员 `so_type` 中。还分配了一个 Internet 协议控制块 (PCB)：一个 `inpcb` 结构。结构 `socket` 的成员 `so_pcb` 指向 `inpcb`，并且结构 `inpcb` 的成员 `inp_socket` 指向结构 `socket`。对于一个给定插口的操作可能来自两个方向：“上”或“下”，因此需要有指针来互相指向。

1) 当进程执行一个系统调用时，如 `sendto`，内核从描述符值开始，使用 `fd_ofiles` 索引到 `file` 结构指针向量，直到描述符所对应的 `file` 结构。结构 `file` 指向 `socket` 结构，结构 `socket` 带有指向结构 `inpcb` 的指针。

2) 当一个 UDP 数据报到达一个网络接口时，内核搜索所有 UDP 协议控制块，寻找一个合适的，至少要根据目标 UDP 端口号，可能还要根据目标 IP 地址、源 IP 地址和源端口号。一旦定位所找的 `inpcb`，内核就能通过 `inp_socket` 指针来找到相应的 `socket` 结构。

成员 `inp_faddr` 和 `inp_laddr` 包含远地和本地 IP 地址，而成员 `inp_fport` 和 `inp_lport` 包含远地和本地端口号。IP 地址和端口号的组合经常叫做一个插口。

在图 1-5 的左边，我们用名称 `udb` 来标注另一个 `inpcb` 结构。这是一个全局结构，它是所有 UDP PCB 组成的链表表头。我们可以看到两个成员 `inp_next` 和 `inp_prev` 把所有的 UDP PCB 组成了一个双向环型链表。为了简化此图，我们用两条平行的水平箭头来表示两条链，而不是用箭头指向 PCB 的顶角。右边的 `inpcb` 结构的成员 `inp_prev` 指向结构 `udb`，而不是它的成员 `inp_prev`。来自 `udb.inp_prev` 和另一个 PCB 成员 `inp_next` 的虚线箭头表示这里

还有其他PCB在这个双链表上，但我们没有画出。

在本章，我们已看了不少内核数据结构，大多数还要在后续章节中说明。现在要理解的关键是：

1) 我们的进程调用socket，最后分配了最小未用的描述符（在我们的例子中是3）。在后面，所有针对此socket的系统调用都要用这个描述符。

2) 以下内核数据结构是一起被分配和链接起来的：一个DTYPE\_SOCKET类型file结构、一个socket结构和一个inpcb结构。这些结构的很多初始化过程我们并没有说明：file结构的读写标志（因为调用socket总是返回一个可读或可写的描述符）；默认的输出和输出缓存大小被设置在socket结构中，等等。

3) 我们显示了标准输入、输出和标准错误处理的非socket描述符的目的是为了说明所有描述符最后都对应一个file结构，虽然socket描述符和其他描述符之间有所不同。

## 1.9 mbuf与输出处理

在伯克利联网代码设计中的一个基本概念就是存储器缓存，称作一个mbuf，在整个联网代码中用于存储各种信息。通过我们的简单例子（图1-2）分析一些mbuf的典型用法。在第2章中我们会更详细地说明mbuf。

### 1.9.1 包含插口地址结构的mbuf

在sendto调用中，第5个参数指向一个Internet插口地址结构（叫serv），第6个参数指示它的长度（后面我们将要看到是16个字节）。插口层为这个系统调用做的第一件事就是验证这些参数是有效的（即这个指针指向进程地址空间的一段存储器），并且将插口地址结构复制到一个mbuf中。图1-6所示的是这个所得到的mbuf。

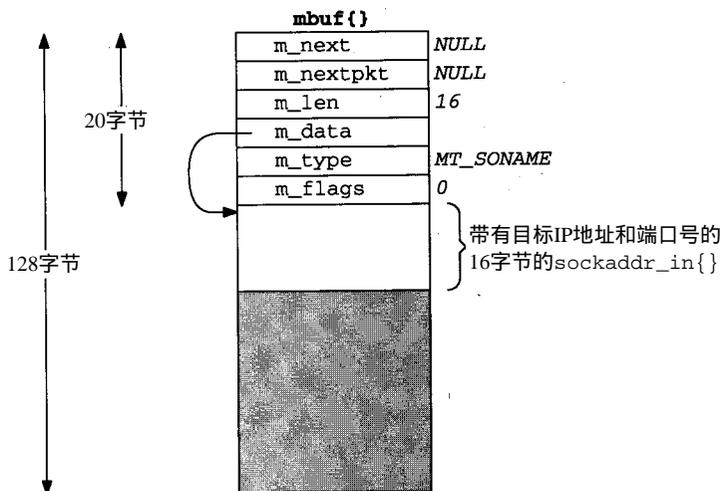


图1-6 mbuf中针对sendto的目的地址

mbuf的前20个字节是首部，它包含关于这个mbuf的一些信息。这20个字节的首部包括四个4字节字段和两个2字节字段。mbuf的总长为128个字节。

稍后我们会看到，mbuf可以用成员m\_next和m\_nextpkt链接起来。在这个例子中都是

空指针，它是一个独立的 mbuf。

成员 `m_data` 指向 mbuf 中的数据，成员 `m_len` 指示它的长度。对于这个例子，`m_data` 指向 mbuf 中数据的第一个字节（紧接着 mbuf 首部）。mbuf 后面的 92 个字节（108-16）没有用（图 1-6 的阴影部分）。

成员 `m_type` 指示包含在 mbuf 中数据的类型，在本例中是 `MT_SONAME`（插口名称）。首部的最后一个成员 `m_flags`，在本例中是零。

### 1.9.2 包含数据的 mbuf

下面继续讨论我们的例子，插口层将 `sendto` 调用中指定的数据缓存中的数据复制到一个或多个 mbuf 中。`sendto` 的第二个参数指示了数据缓存 (buff) 的开始位置，第三个参数是它的大小 (150 字节)。图 1-7 显示了 150 字节的数据是如何存储在两个 mbuf 中的。

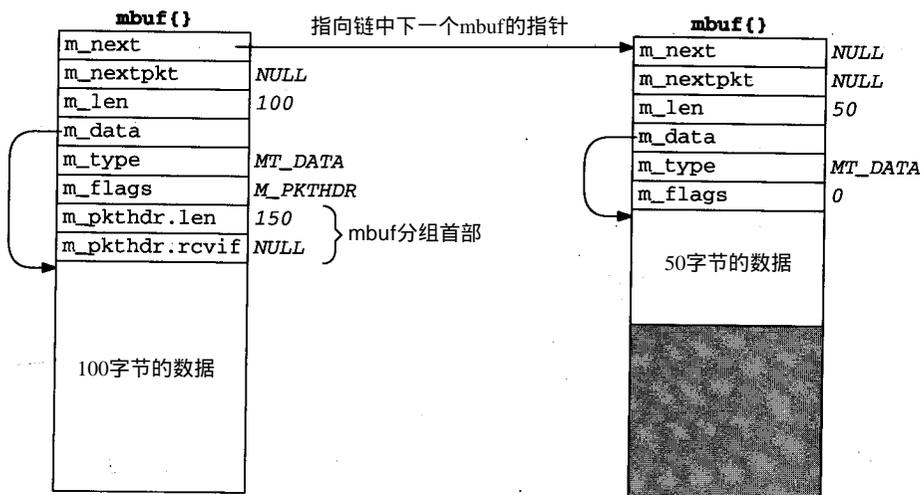


图 1-7 用两个 mbuf 来存储 150 字节的数据

这种安排叫做 mbuf 链表。在每个 mbuf 中的成员 `m_next` 把链表中所有的 mbuf 都链接在一起。

我们看到的另一个变化是链表中第一个 mbuf 的 mbuf 首部的另外两个成员：`m_pkthdr.len` 和 `m_pkthdr.rcvif`。这两个成员组成了分组首部并且只用在链表的第一个 mbuf 中。成员 `m_flags` 的值是 `M_PKTHDR`，指示这个 mbuf 包含一个分组首部。分组首部结构的成员 `len` 包含了整个 mbuf 链表的总长度（在本例中是 150），下一个成员 `rcvif` 在后面我们会看到，它包含了一个指向接收分组的接收接口结构的指针。

因为 mbuf 总是 128 个字节，在链表的第一个 mbuf 中提供了 100 字节的数据存储能力，而后面所有的 mbuf 有 108 字节的存储空间。在本例中的两个 mbuf 需要存储 150 字节的数据。我们稍后会看到当数据超过 208 字节时，就需要 3 个或更多的 mbuf。有一种不同的技术叫“簇”，一种大缓存，典型的有 1024 或 2048 字节。

在链表的第一个 mbuf 中维护一个带有总长度的分组首部的原因是，当需要总长度时可以避免查看所有 mbuf 中的 `m_len` 来求和。

## 1.9.3 添加IP和UDP首部

在插口层将目标插口地址结构复制到一个 mbuf 中，并把数据复制到 mbuf 链中后，与此插口描述符（一个 UDP 描述符）对应的协议层被调用。明确地说，UDP 输出例程被调用，指向 mbuf 的指针被作为一个参数传递。这个例程要在这 150 字节数据的前面添加一个 IP 首部和一个 UDP 首部，然后将这些 mbuf 传递给 IP 输出例程。

在图 1-7 中的 mbuf 链表中添加这些数据的方法是分配另外一个 mbuf，把它放在链首，并将分组首部从带有 100 字节数据的 mbuf 复制到这个 mbuf。在图 1-8 中显示了这三个 mbuf。

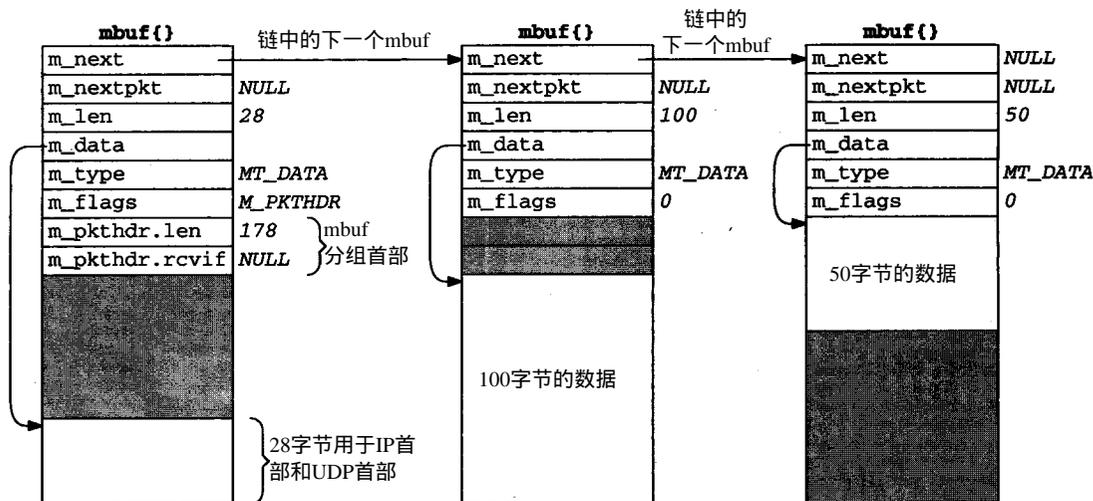


图 1-8 在图 1-7 中的 mbuf 链表中添加另一个带有 IP 和 UDP 首部的 mbuf

IP 首部和 UDP 首部被放置在新 mbuf 的最后，这个新 mbuf 就成了整个链表的首部。如果需要，它允许任何其他低层协议（例如接口层）在 IP 首部前添加自己的首部，而不需要再复制 IP 和 UDP 首部。在第一个 mbuf 中的 m\_data 指针指向这两个首部的起始位置，m\_len 的值是 28。在分组首部和 IP 首部之间有 72 字节的未用空间留给以后的首部，通过适当地修改 m\_data 指针和 m\_len 添加在 IP 首部的前面。稍后我们会看见以太网首部就是用这种方法建立的。

注意，分组首部已从带有 100 字节数据的 mbuf 中移到新 mbuf 中去了。分组首部必须放在 mbuf 链表的第一个 mbuf 中。在移动分组首部的同时，在第一个 mbuf 设置 M\_PKTHDR 标志并且在第二个 mbuf 中清除此标志。在第二个 mbuf 中分组首部占用的空间现在未用。最后，在此分组首部中的长度成员由于增加了 28 字节而变成了 178。

然后 UDP 输出例程填写 UDP 首部和 IP 首部中它们所能填写的部分。例如，IP 首部中的目标地址可以被设置，但 IP 检验和要留给 IP 输出例程来计算和存放。

UDP 检验和计算后存储在 UDP 首部中。注意，这要求遍历存储在 mbuf 链表中的所有 150 字节的数据。这样，内核要对这 150 字节的用户数据做两次遍历：一次是把用户缓存中的数据复制到内核中的 mbuf 中，而现在是在计算 UDP 检验和。对整个数据的额外遍历会降低协议的性能，在后续章节中我们会介绍另一种可选的实现技术，它可以避免不必要的遍历。

接着，UDP 输出例程调用 IP 输出例程，并把此 mbuf 链表的指针传递给 IP 输出例程。

#### 1.9.4 IP输出

IP输出例程要填写IP首部中剩余的字段，包括IP检验和；确定数据报应发到哪个输出接口（这是IP路由功能）；必要时，对IP报文分片；以及调用接口输出函数。

假设输出接口是一个以太网接口，再次把此 mbuf链表的指针作为一个参数，调用一个通用的以太网输出函数。

#### 1.9.5 以太网输出

以太网输出函数的第一个功能就是把 32位IP地址转换成相应的 48位以太网地址。在使用 ARP(地址解析协议)时会使用这个功能，并且会在以太网上发送一个 ARP请求并等待一个ARP应答。此时，要输出的mbuf链表已得到，并等待应答。

然后以太网输出例程把一个 14字节的以太网首部添加到链表的第一个 mbuf中，紧接在IP首部的前面(图1-8)。以太网首部包括6字节以太网目标地址、6字节以太网源地址和2字节以太网帧类型。

之后此mbuf链表被加到此接口的输出队列队尾。如果接口不忙，接口的“开始输出”例程立即被调用。若接口忙，在它处理完输出队列中的其他缓存后，它的输出例程会处理队列中的这个新mbuf。

当接口处理它输出队列中的一个 mbuf时，它把数据复制到它的传输缓存中，并且开始输出。在我们的例子中，192字节被复制到传输缓存中：14字节以太网首部、20字节IP首部、8字节UDP首部及150字节用户数据。这是内核第三次遍历这些数据。一旦数据从 mbuf链表被复制到设备传输缓存，mbuf链表就被以太网设备驱动程序释放。这三个 mbuf被放回到内核的自由缓存池中。

#### 1.9.6 UDP输出小结

我们在图1-9中给出了一个进程调用 sendto传输一个UDP数据报时的大致处理过程。在图中我们说明的处理过程与三层内核代码(图1-3)的关系也显示出来了。

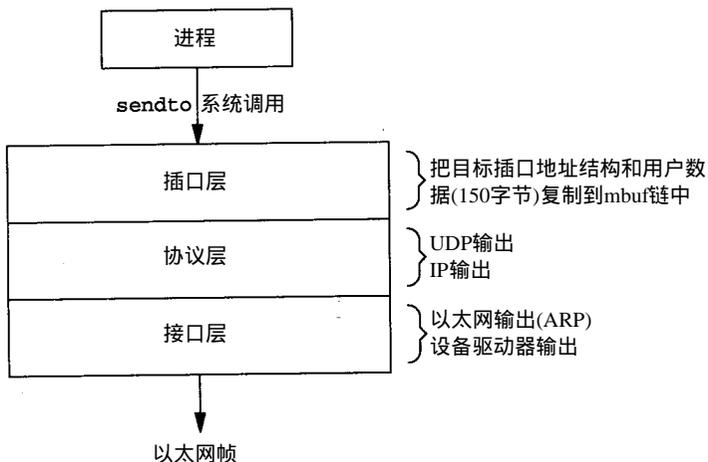


图1-9 三层处理一个简单UDP输出的执行过程

函数调用控制从插口层到 UDP输出例程，到 IP输出例程，然后到以太网输出例程。每个函数调用传递一个指向要输出的 mbuf的指针。在最低层，设备驱动程序层，mbuf链表被放置到设备输出队列并启动设备。函数调用按调用的相反顺序返回，最后系统调用返回给进程。注意，直到 UDP数据报到达设备驱动程序前，UDP数据没有排队。高层仅仅添加它们的协议首部并把mbuf传递给下一层。

这时，在我们的程序示例中调用 `recvfrom`去读取服务器的应答。因为该插口的输入队列是空的(假设应答还没有到达)，进程就进入睡眠状态。

## 1.10 输入处理

输入处理与刚讲过的输出处理不同，因为输入是异步的。就是说，它是通过一个接收完成中断驱动以太网设备驱动程序来接收一个输入分组，而不是通过进程的系统调用。内核处理这个设备中断，并调度设备驱动程序进入运行状态。

### 1.10.1 以太网输入

以太网设备驱动程序处理这个中断，假定它表示一个正常的接收已完成，数据从设备读到一个mbuf链表中。在我们的例子中，接收了 54 字节的数据并复制到一个 mbuf中：20字节IP首部、8字节UDP首部及26字节数据(服务器的时间与日期)。图1-10所示的是这个mbuf的格式。

这个mbuf是一个分组首部(`m_flags`被设置成`M_PKTHDR`)，它是一个数据记录的第一个mbuf。分组首部的成员 `len`包含数据的总长度，成员 `rcvif`包含一个指针，它指向接收数据的接口的接口结构(第3章)。我们可以看到成员 `rcvif`用于接收分组而不是输出分组(图1-7和图1-8)。

mbuf的前16字节数据空间被分配给一个接口层首部，但没有使用。数据就存储在这个mbuf中，54字节的数据存储在剩余的84字节的空间中。

设备驱动程序把 mbuf传给一个通用以太网输入例程，它通过以太网帧中的类型字段来确定哪个协议层来接收此分组。在这个例子中，类型字段标识一个 IP数据报，从而mbuf被加入到IP输入队列中。另外，会产生一个软中断来执行 IP输入例程。这样，这个设备中断处理就完成了。

### 1.10.2 IP输入

IP输入是异步的，并且通过一个软中断来执行。当接口层在系统的一个接口上收到一个IP数据报时，它就设置这个软中断。当 IP输入例程执行它时，循环处理在它的输入队列中的每一个IP数据报，并在整个队列被处理完后返回。

IP输入例程处理每个接收到的IP数据报。它验证IP首部检验和，处理IP选项，验证数据报

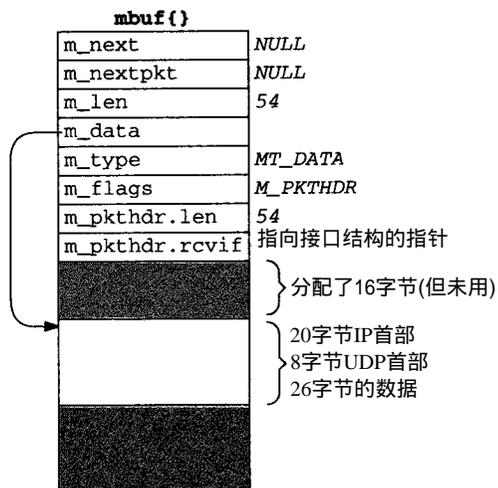


图1-10 用一个mbuf存储输入的以太网数据

被传递到正确的主机(通过比较数据报的目标IP地址与主机IP地址),并当系统被配置为一个路由器,且数据报被表注为其他的IP地址时,转发此数据报。如果IP数据报到达它的最终目标,调用IP首部中标识的协议的输入例程:ICMP,IGMP,TCP或UDP。在我们的例子中,调用UDP输入例程去处理UDP数据报。

### 1.10.3 UDP输入

UDP输入例程验证UDP首部中的各字段(长度与可选的检验和),然后确定是否一个进程应该接收此数据报。在第23章我们要详细讨论这个检查是如何进行的。一个进程可以接收到一指定UDP端口的所有数据报,或让内核根据源与目标IP地址及源与目标端口号来限制数据报的接收。

在我们的例子中,UDP输入例程从一个全局变量udb(图1-5)开始,查看所有UDP协议控制块链表,寻找一个本地端口号(inp\_lport)与接收的UDP数据报的目标端口号匹配的协议控制块。这个PCB是由我们调用socket创建的,它的成员inp\_socket指向相应插口结构,并允许接收的数据在此插口排队。

在程序示例中,我们从未为应用程序指定本地端口号。在习题23.3中,我们会看到在写第一个UDP程序时创建一个插口而不绑定一个本地端口号会导致内核自动地给此插口分配一个本地端口号(称为短期端口)。这就是为什么插口的PCB成员inp\_lport不是一个空值的原因。

因为这个UDP数据报要传递给我们的进程,发送方的IP地址和UDP端口号被放置到一个mbuf中,这个mbuf和数据(在我们的例子中是26字节)被追加到此插口的接收队列中。图1-11所示的是被追加到这个插口的接收队列中的这两个mbuf。

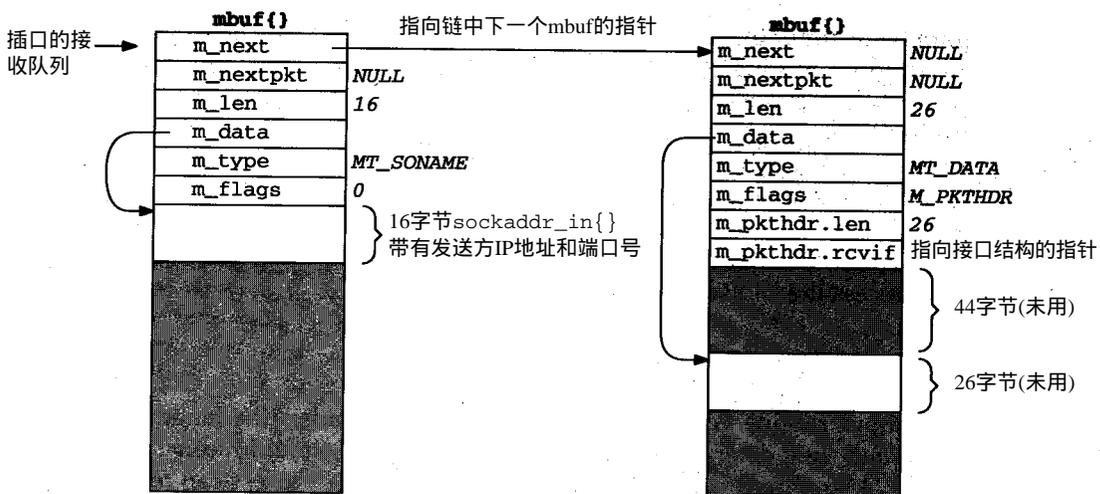


图1-11 发送方地址和数据

比较这个链表中的第二个mbuf(MT\_DATA类型)与图1-10中的mbuf,成员m\_len和m\_pkthdr.len都减小了28字节(20字节的IP首部和8字节UDP首部),并且指针m\_data也减小了28字节。这有效地将IP和UDP首部删去,只保留了26字节数据追加到插口接收队列。

在链表的第一个 mbuf 中包括一个 16 字节 Internet 插口地址结构，它带有发送方 IP 地址和 UDP 端口号。它的类型是 MT\_SONAME，与图 1-6 中的 mbuf 类似。这个 mbuf 是插口层创建的，将这些信息返回给通过调用系统调用 `recvfrom` 或 `recvmsg` 的调用进程。即使在这个链表的第二个 mbuf 中有空间 (16 字节) 存储这个插口地址结构，它也必须存放到它自己的 mbuf 中，因为它们的类型不同 (一个是 MT\_SONAME，一个是 MT\_DATA)。

然后接收进程被唤醒。如果进程处于睡眠状态等待数据的到达 (我们例子中的情况)，进程被标志为可运行状态等待内核的调度。也可以通过 `select` 系统调用或 SIGIO 信号来通知进程数据的到达。

#### 1.10.4 进程输入

我们的进程调用 `recvfrom` 时被阻塞，在内核中处于睡眠状态，现在进程被唤醒。UDP 层追加到插口接收队列中的 26 字节的数据 (接收的数据报) 被内核从 mbuf 复制到我们程序的缓存中。

注意，我们的程序把 `recvfrom` 的第 5，第 6 个参数设置为空指针，告诉系统在接收过程中不关心发送方的 IP 地址和 UDP 端口号。这使得系统调用 `recvfrom` 时，略过链表中的第一个 mbuf (图 1-11)，仅返回第二个 mbuf 中的 26 字节的数据。然后内核的 `recvfrom` 代码释放图 1-11 中的两个 mbuf，并把它们放回到自由 mbuf 池中。

### 1.11 网络实现概述 (续)

图 1-12 总结了在各层间为网络输入输出而进行的通信。图 1-12 是对图 1-3 进行了重画，它只考虑 Internet 协议，并且强调层间的通信。符号 `splnet` 与 `splimp` 在下一节讨论。

我们使用复数术语插口队列 (socket queues) 和接口队列 (interface queues)，因为每个插口

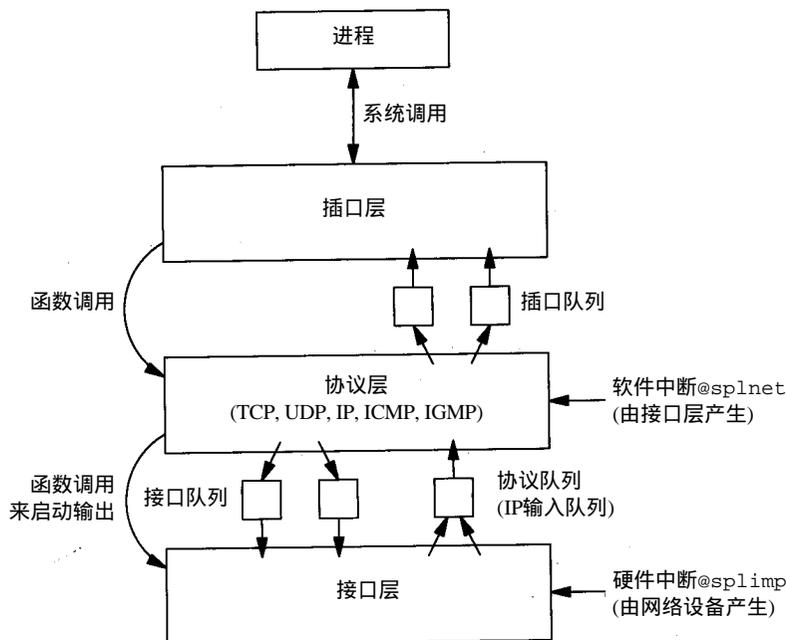


图 1-12 网络输入输出的层间通信

和每个接口(以太网、环回、SLIP、PPP等)都有一个队列,但我们使用单数术语协议队列(protocol queue),因为只有一个IP输入队列。如果考虑其他协议层,我们就会有一个队列用于XNS协议,一个队列用于OSI协议。

## 1.12 中断级别与并发

我们在1.10节看到网络代码处理输入分组用的是异步和中断驱动的方式。首先,一个设备中断引发接口层代码执行,然后它产生一个软中断引发协议层代码执行。当内核完成这些级别的中断后,执行插口代码。

在这里给每个硬件和软件中断分配一个优先级。图 1-13所示的是8个优先级别的顺序,从最低级别(不阻塞中断)到最高级别(阻塞所有中断)。

函 数	说 明
sp10	正常操作方式,不阻塞中断 (最低优先级)
Splsoftclock	低优先级时钟处理
splnet	网络协议处理
spltty	终端输入输出
splbio	磁盘与磁带输入输出
splimp	网络设备输入输出
splclock	高优先级时钟处理
splhigh	阻塞所有中断 (最高优先级)
splx(s)	(见正文)

图1-13 阻塞所选中断的内核函数

[Leffler et al. 1989]的表4-5显示了用于VAX实现的优先级别。386的Net/3的实现使用图1-13所示的8个函数,但splsoftclock与splnet在同一级别,splclock与splhigh也在同一级别。

用于网络接口级的名称 *imp* 来自于缩写IMP(接口报文处理器),它是在ARPANET中使用的路由器的最初类型。

不同优先级的顺序意味着高优先级中断可以抢占一个低优先级中断。看图 1-14所示的事

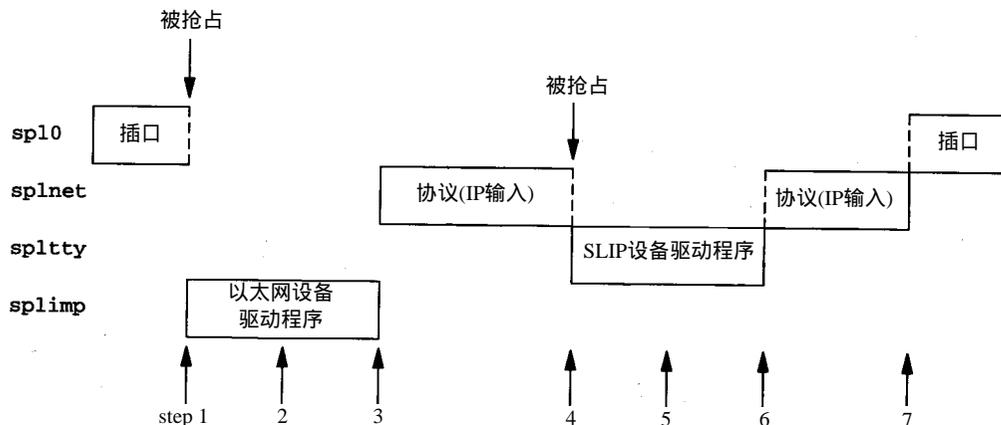


图1-14 优先级示例与内核处理

件顺序。

- 1) 当插口层以级别 `spl0` 执行时，一个以太网设备驱动程序中断发生，使接口层以级别 `splimp` 执行。这个中断抢占了插口层代码的执行。这就是异步执行接口输入例程。
- 2) 当以太网设备驱动程序在运行时，它把一个接收的分组放置到 IP 输出队列中并调度一个 `splnet` 级别的软中断。软中断不会立即有效，因为内核正在一个更高的优先级 (`splimp`) 上运行。
- 3) 当以太网设备驱动程序完成后，协议层以级别 `splnet` 执行。这就是异步执行 IP 输入例程。
- 4) 一个终端设备中断发生 (完成一个 SLIP 分组)，它立即被处理，抢占协议层，因为终端输入/输出 (`spltty`) 优先级比图 1-13 中的协议层 (`splnet`) 更高。
- 5) SLIP 驱动程序把接收的分组放到 IP 输入队列中并为协议层调度另一个软中断。
- 6) 当 SLIP 驱动程序结束时，被抢占的协议层继续以级别 `splnet` 执行，处理完从以太网设备驱动程序收到的分组后，处理从 SLIP 驱动程序接收的分组。仅当没有其他输入分组要处理时，它会把控制权交还给被它抢占的进程 (在本例中是插口层)。
- 7) 插口层从它被中断的地方继续执行。

对于这些不同优先级，一个要关心的问题就是如何处理那些在不同级别的进程间共享的数据结构。在图 1-2 中显示了三种在不同优先级进程间共享的数据结构——插口队列、接口队列和协议队列。例如，当 IP 输入例程正在从它的输入队列中取出一个接收的分组时，一个设备中断发生，抢占了协议层，并且那个设备驱动程序可能添加另一个分组到 IP 输入队列。这些共享的数据结构 (本例中的 IP 输入队列，它共享于协议层和接口层)，如果不协调对它们的访问，可能会破坏数据的完整性。

Net/3 的代码经常调用函数 `splimp` 和 `splnet`。这两个调用总是与 `splx` 成对出现，`splx` 使处理器返回到原来的优先级。例如下面这段代码，被协议层 IP 输入函数执行，去检查是否有其他分组在它的输入队列中等待处理：

```
struct mbuf *m;
int s;

s = splimp ();
IF_DEQUEUE (&ipintrq, m);
splx(s);
if (m == 0)
    return;
```

调用 `splimp` 把 CPU 的优先级升高到网络设备驱动程序级，防止任何网络设备驱动程序中断发生。原来的优先级作为函数的返回值存储到变量 `s` 中。然后执行宏 `IF_DEQUEUE` 把 IP 输入队列 (`ipintrq`) 头部的第二个分组删去，并把指向此 `mbuf` 链表的指针放到变量 `m` 中。最后，通过调用带有参数 `s` (其保存着前面调用 `splimp` 的返回值) 的 `splx`，CPU 的优先级恢复到调用 `splimp` 前的级别。

由于在调用 `splimp` 和 `splx` 之间所有的网络设备驱动程序的中断被禁止，在这两个调用间的代码应尽可能的少。如果中断被禁止过长的时间，其他设备会被忽略，数据会被丢失。因此，对变量 `m` 的测试 (看是否有其他分组要处理) 被放在调用 `splx` 之后而不是之前。

当以太网输出例程把一个要输出的分组放到一个接口队列，并测试接口当前是否忙时，若接口不忙则启动接口，这时例程需要调用这些 `spl` 调用。

```

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);

```

在这个例子中，设备中断被禁止的原因是防止在协议层正在往队列添加分组时，设备驱动程序从它的发送队列中取走下一个分组。设备发送队列是一个在协议层和接口层共享的数据结构。

在整个源代码中到处都会看到 spl 函数。

### 1.13 源代码组织

图1-15所示的是 Net/3 网络源代码的组织，假设它位于目录 `/usr/src/sys`。

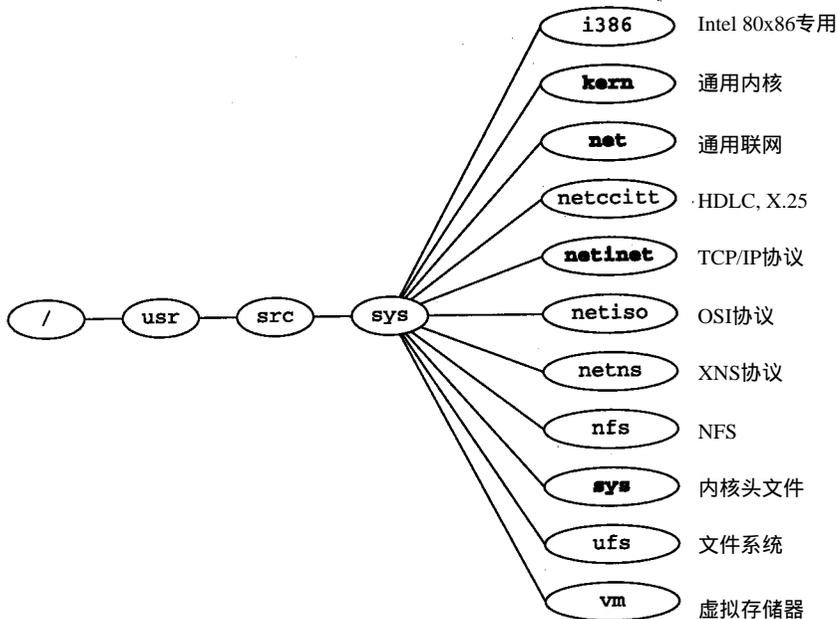


图1-15 Net/3源代码组织

本书的重点在目录 `netinet`，它包含所有 TCP/IP 源代码。在目录 `kern` 和 `net` 中我们也可找到一些文件。前者是协议无关的插口代码，而后者是一些通用联网函数，用于 TCP/IP 例程，如路由代码。

包含在每个目录中的文件简要地列于下面：

- `i386`：因特80x86专用目录。例如，目录 `i386/isa` 包含专用于ISA总线的设备驱动程序。目录 `i386/stand` 包含单机引导程序代码。
- `kern`：通用的内核文件，不属于其他目录。例如，处理系统调用 `fork` 和 `exec` 的内核文件在这个目录。在这个目录中，我们只考察少数几个文件——用于插口系统调用的文件(插口层在图1-3)。
- `net`：通用联网文件，例如，通用联网接口函数，BPF(BSD分组过滤器)代码、SLIP驱动程序和路由代码。在这个目录中我们考察一些文件。
- `netccitt`：OSI协议接口代码，包括HDLC(高级数据链路控制)和X.25驱动程序。
- `netinet`：Internet协议代码：IP，ICMP，IGMP，TCP和UDP。本书的重点集中在这个目录中的文件。
- `netiso`：OSI协议。
- `netns`：施乐(Xerox)XNS协议。
- `nfs`：SUN公司的网络文件系统代码。
- `sys`：系统头文件。在这个目录中我们考察几个头文件。这个目录中的文件还出现在目录 `/usr/include/sys` 中。
- `ufs`：Unix文件系统的代码，有时叫伯克利快速文件系统。它是标准磁盘文件系统。
- `vm`：虚拟存储器系统代码。

图1-16所示的是源代码组织的另一种表现形式，它映射到我们的三个内核层。忽略 `netimp` 和 `nfs` 这样的目录，在本书中我们不关心它们。

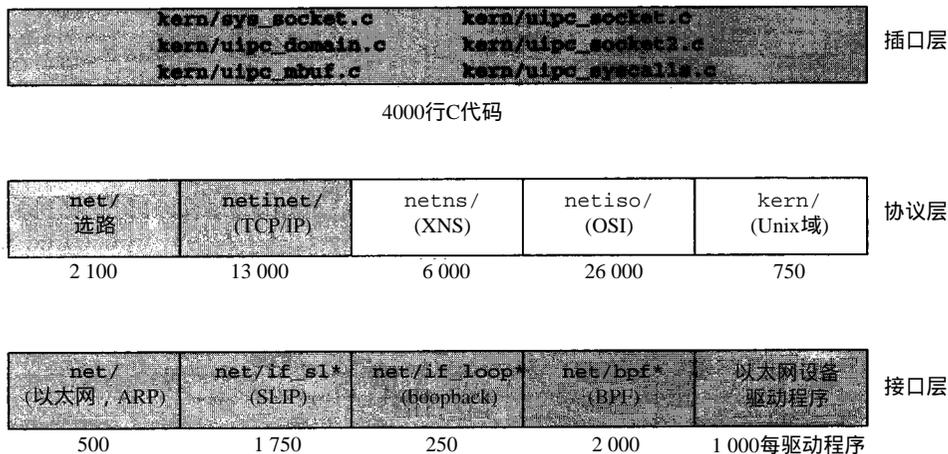


图1-16 映射到三个内核层的Net/3源代码组织

在每个表格框底下的数字是对应功能的C代码的近似行数，包括源文件中的所有注释。

我们不考察图中所有的源代码。显示目录 `netns` 与 `netiso` 是为了与Internet协议比较。我们仅考虑有阴影的表格框。

## 1.14 测试网络

图1-17所示的测试网络用于本书中所有的例子。除了在图顶部的主机 `vangogh`，所有的

IP地址属于B类网络地址140.252，并且所有主机名属于域.tuc.noao.edu (noao代表“国家光学天文台”，tuc代表Tucson)。例如，在右下角的系统的主机全名是svr4.tuc.noao.edu，IP地址是140.252.13.34。在每个框图顶上的记号是运行在此系统上操作系统的名称。

在图顶的主机的全名是vangogh.cs.berkeley.edu，其他主机通过Internet可以连接到它。

这个图与卷1中的测试网络几乎一样，有一些操作系统升级了，在sun与netb之间的拨号链路现在用PPP取代了SLIP。另外，我们用Net/3网络代码代替了BSD/386 V1.1提供的Net/2网络代码。

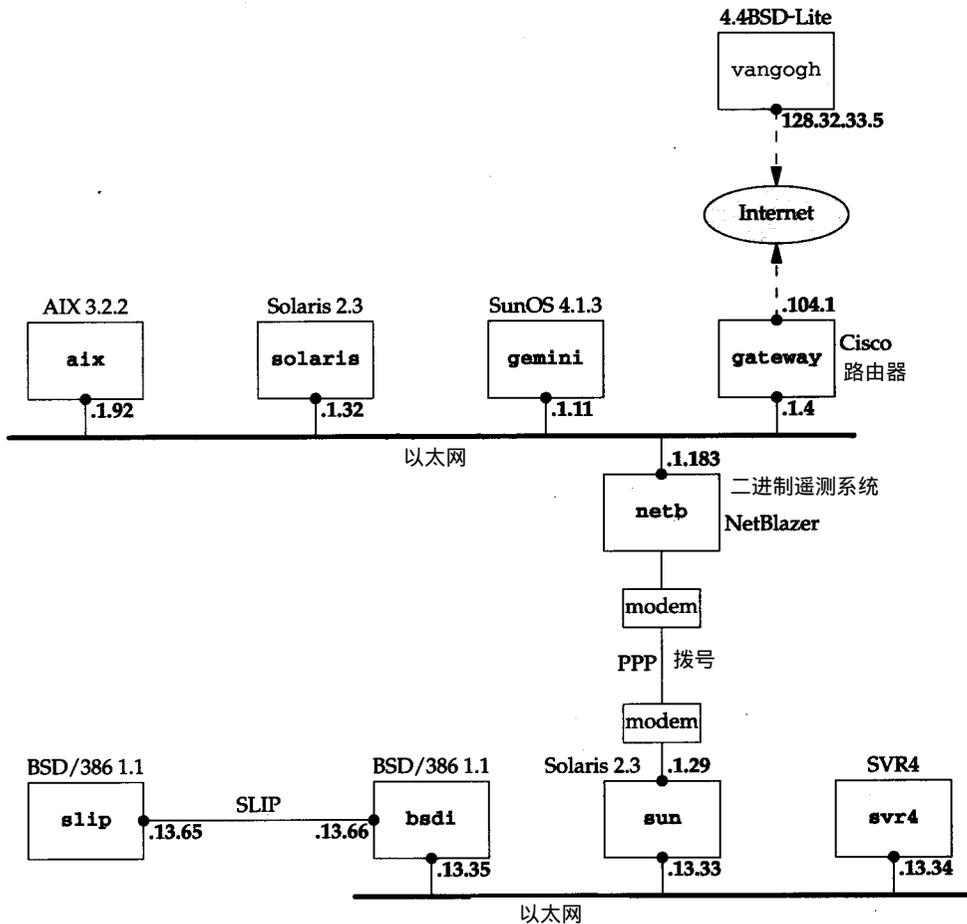


图1-17 用于本书中所有例子的测试网络

## 1.15 小结

本章是对Net/3网络代码的概述。通过一个简单的程序示例(图1-2)——发送一个UDP数据报给一个日期时间服务器并接收应答，我们分析了通过内核进行输入输出的过程。mbuf中保存要输出的信息和接收的IP数据报。下一章我们要查看mbuf的更多细节。

当进程执行sendto系统调用时，产生UDP输出，而IP输入是异步的。当一个设备驱动程序

序接收了一个IP数据报，数据报被放到IP输入队列中并且产生一个软中断使IP输入函数执行。我们考察了在内核中用于联网代码的不同中断级别。由于很多联网数据结构被不同的层所共享，而这些层在不同的中断级别上执行，因此当访问或修改这些共享结构时要特别小心。几乎所有我们要查看的函数中都会遇到spl函数。

本章结束时我们查看了Net/3源代码的整个组织结构，及本书关注的代码。

## 习题

- 1.1 输入程序示例(图1-2)并在你的系统上运行。如果你的系统有系统调用跟踪能力，如trace (SunOS 4.x)、truss (SVR4)或ktrace (4.4BSD)，用它检测本例中调用的系统调用。
- 1.2 在1.12节调用IF\_DEQUEUE的例子中，我们注意到调用splimp来防止网络设备驱动程序的中断。当以太网驱动程序以这个级别执行时，SLIP驱动程序会发生什么？