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 }
```

tcp_subr.c

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。

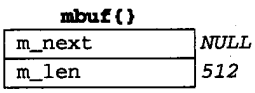
这些函数大小不一，从几行 (如前面的tcp_quench)到最大1100行(tcp_input)。超过大约40行的函数一般被分成段，一段一段地显示。虽然尽量使代码和相应的描述文字放在同一页或对开的两页上，但为了节约版面，不可能完全做到。

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：



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
M_BCAST	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

用源代码版，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)。

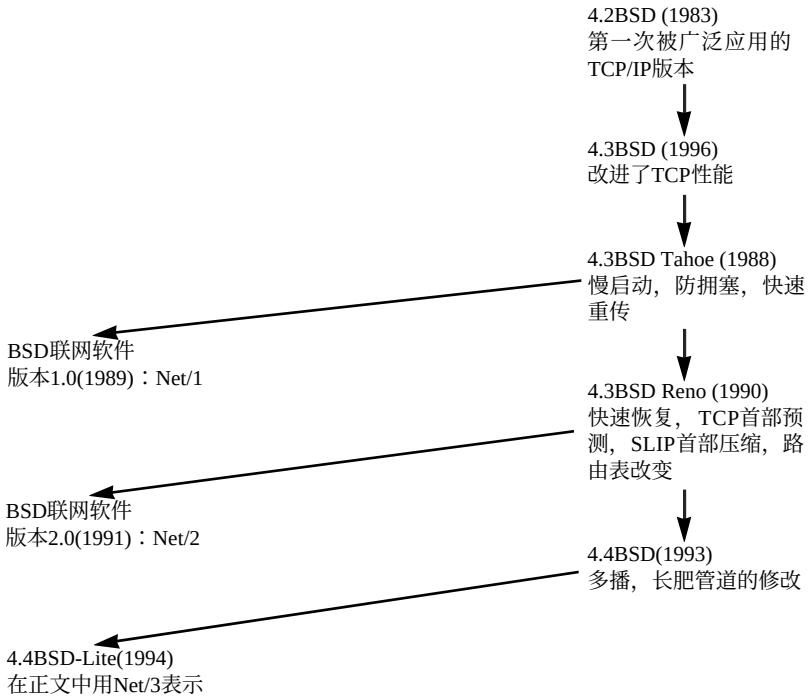


图1-1 带有重要TCP/IP特征的各种BSD版本

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的文档有限。[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V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一个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SVR 4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一个函数，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，它还不算是一套真正的协议，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。

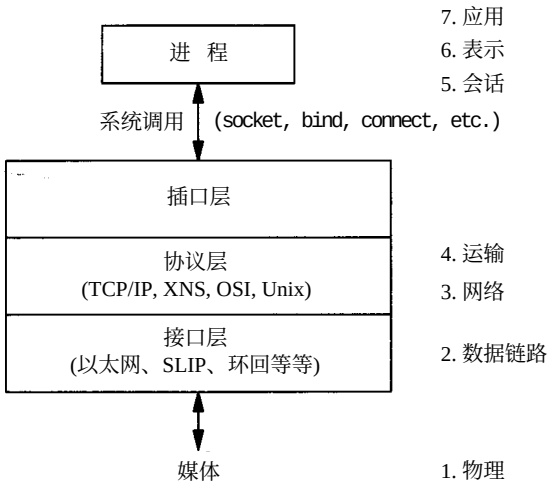


图1-3 Net/3联网代码的大概组织

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colse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。

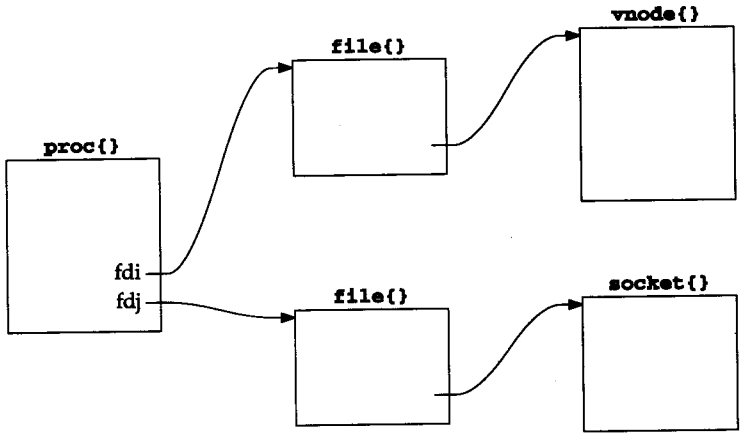


图1-4 从一个描述符开始的内核数据结构的基本关系

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。在这个结构中有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

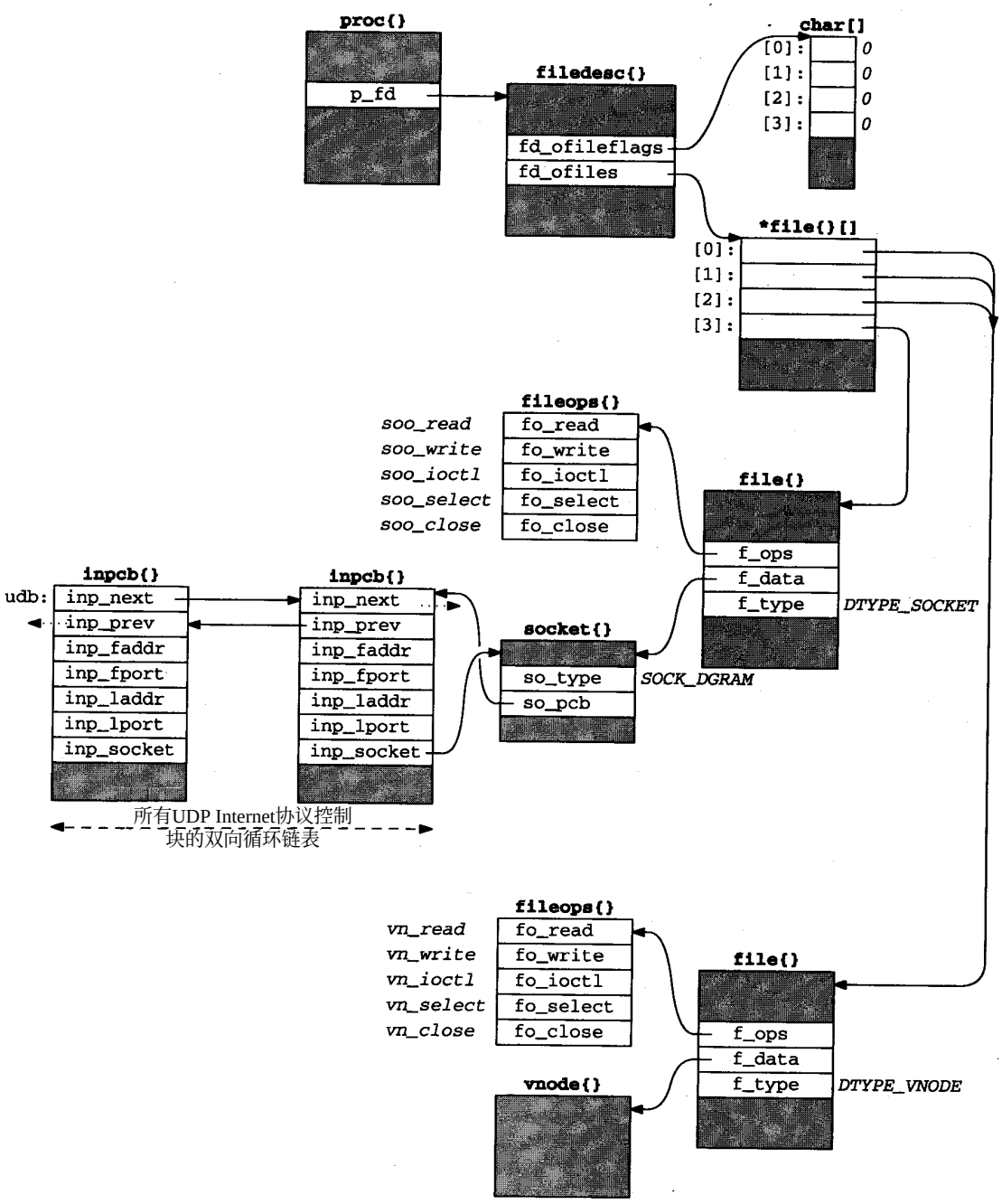


图1-5 在程序示例中调用socket后的内核数据结构

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。v-node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 v-node，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、writev、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)；默认的输入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。

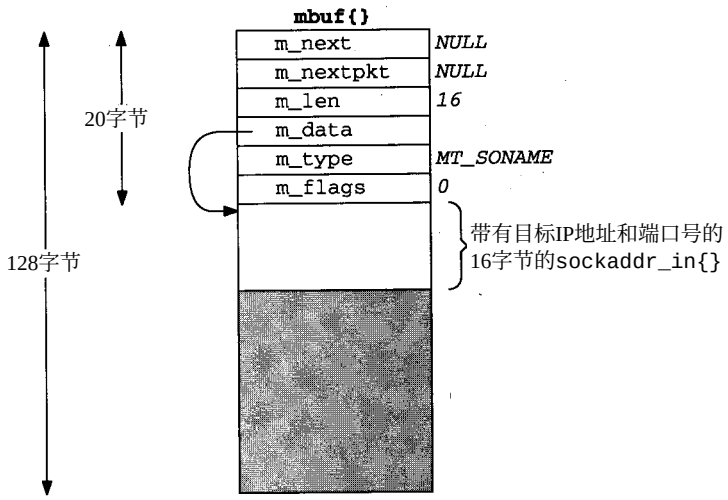


图1-6 mbuf中针对sendto的目的地址

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。

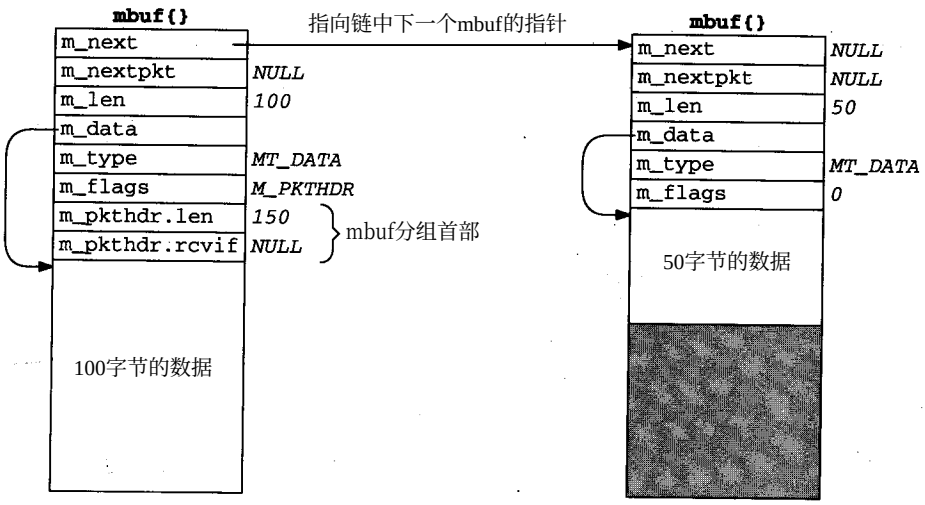


图1-7 用两个mbuf来存储150字节的数据

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。

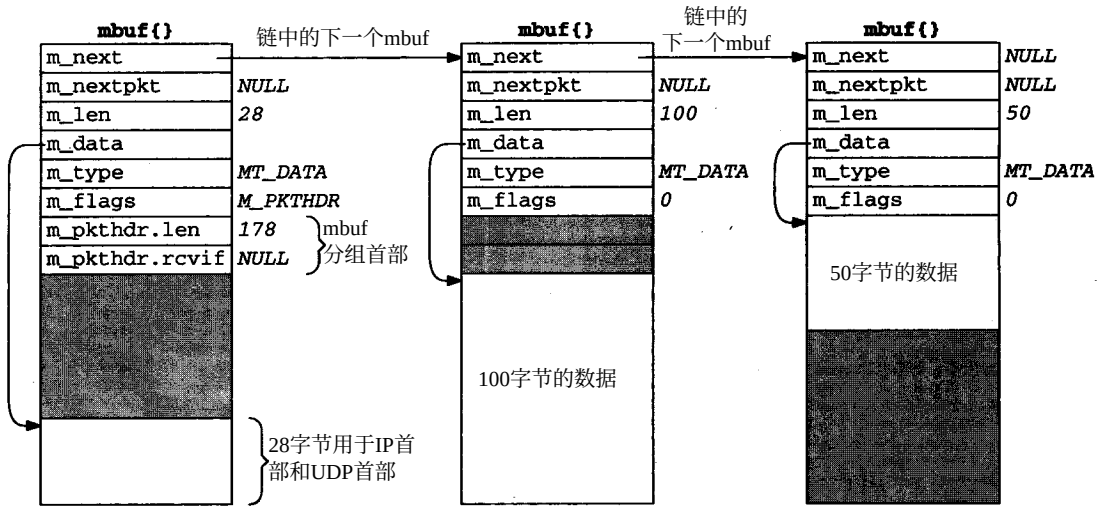


图1-8 在图1-7中的mbuf链表中添加另一个带有IP和UDP首部的mbuf

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。

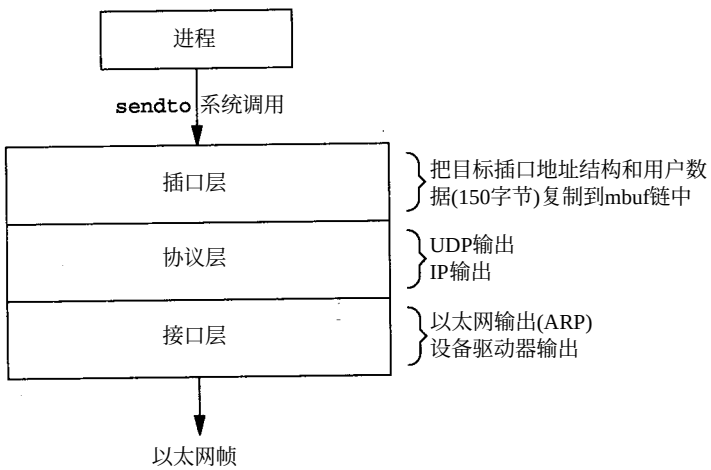


图1-9 三层处理一个简单UDP输出的执行过程

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。

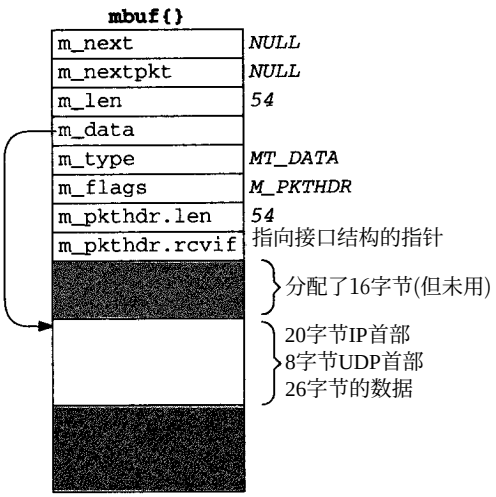


图1-10 用一个mbuf存储输入的以太网数据

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

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。

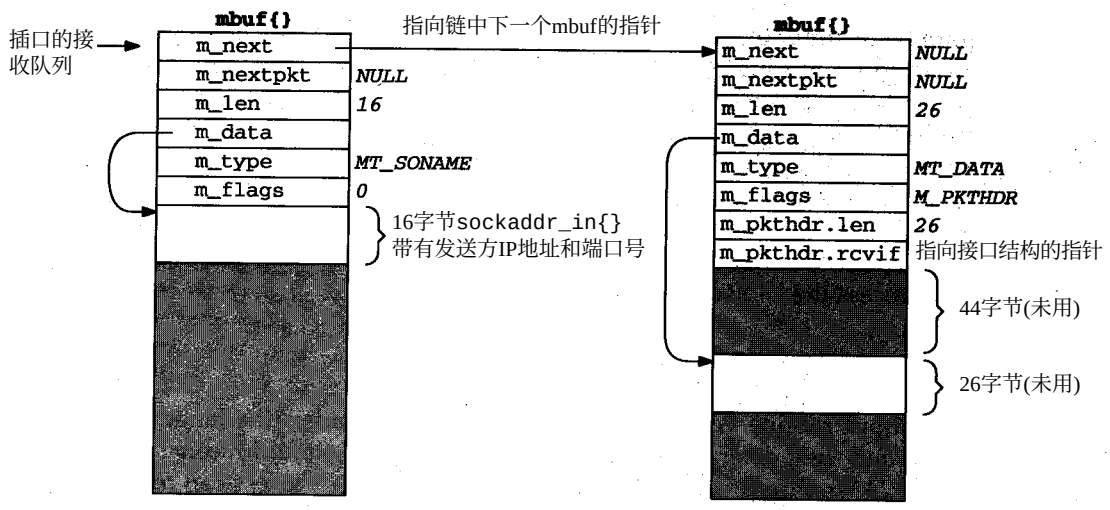


图1-11 发送方地址和数据

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recvform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recvform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

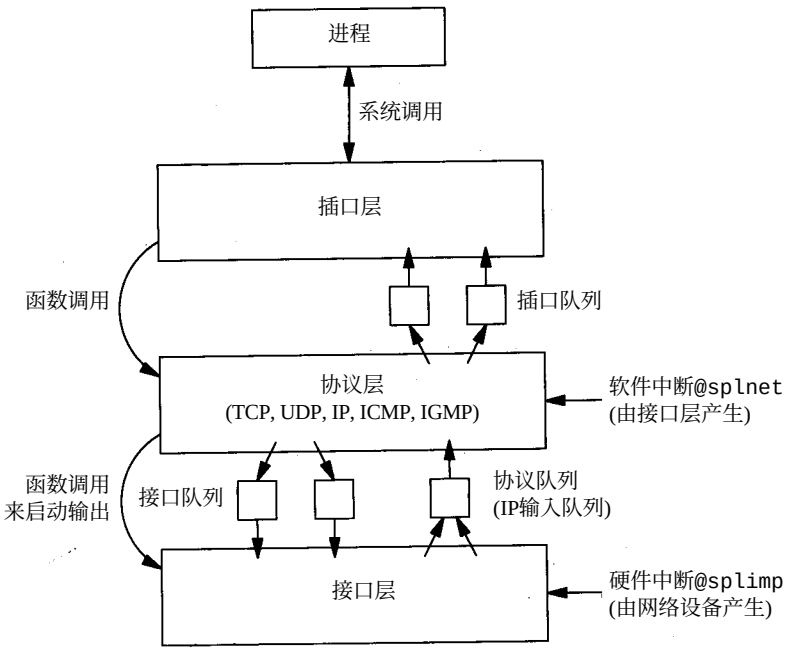


图1-12 网络输入输出的层间通信

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
spl0	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

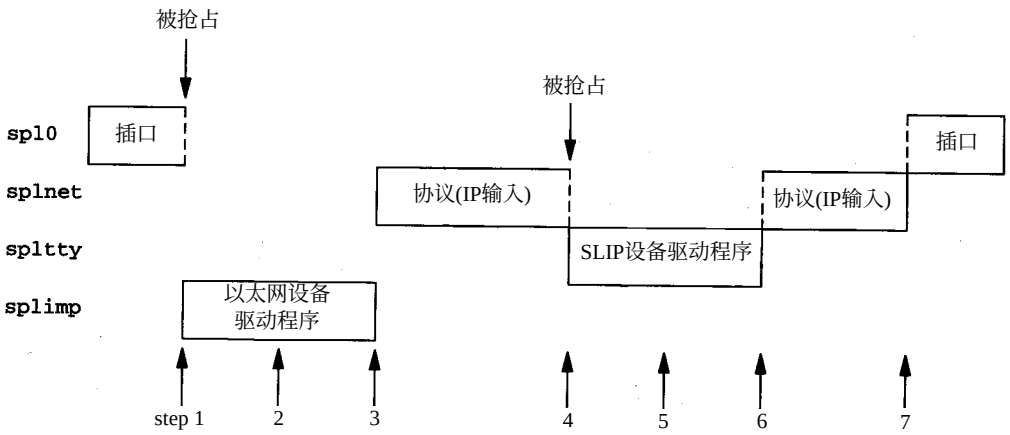


图1-14　优先级示例与内核处理

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。

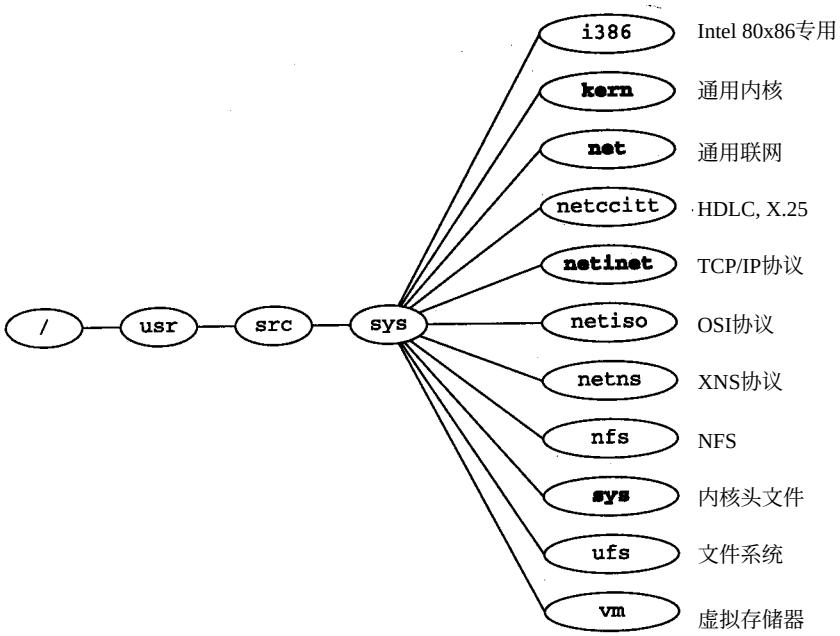


图1-15 Net/3源代码组织

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。

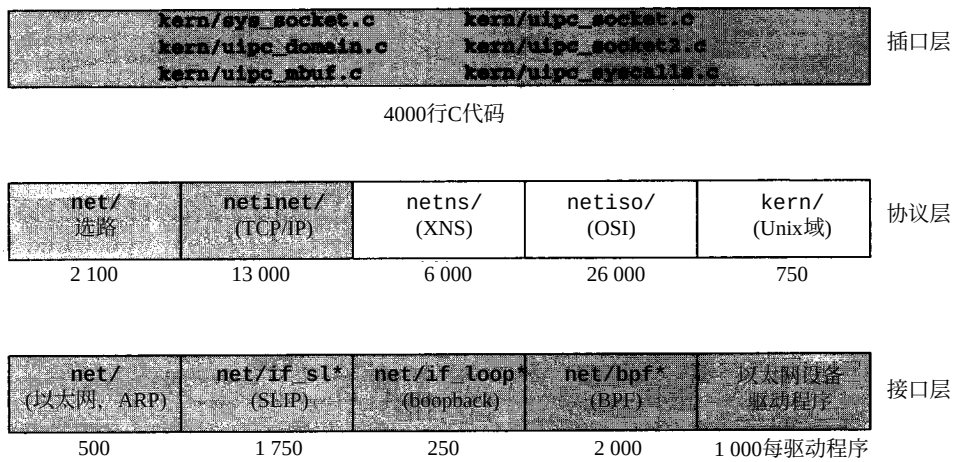


图1-16 映射到三个内核层的Net/3源代码组织

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。

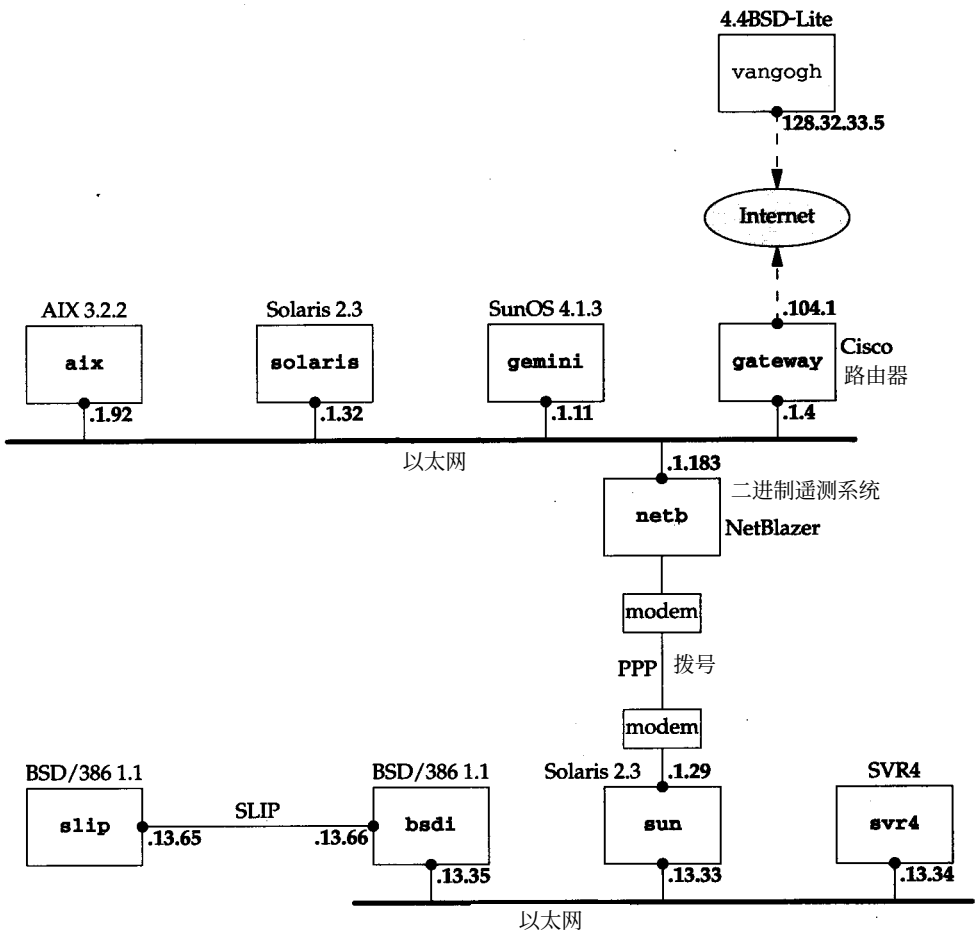


图1-17 用于本书中所有例子的测试网络

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？

第2章 mbuf：存储器缓存

2.1 引言

网络协议对内核的存储器管理能力提出了很多要求。这些要求包括能方便地操作可变长缓存，能在缓存头部和尾部添加数据 (如低层封装来自高层的数据)，能从缓存中移去数据 (如，当数据分组向上经过协议栈时要去掉首部)，并能尽量减少为这些操作所做的数据复制。内核中的存储器管理调度直接关系到联网协议的性能。

在第1章我们介绍了普遍应用于Net/3内核中的存储器缓存：mbuf，它是“ memory buffer ”的缩写。在本章，我们要查看 mbuf和内核中用于操作它们的函数的更多的细节，在本书中几乎每一页我们都会遇到mbuf。要理解本书的其他部分必须先要理解 mbuf。

mbuf的主要用途是保存在进程和网络接口间互相传递的用户数据。但 mbuf也用于保存其他各种数据：源与目标地址、插口选项等等。

图2-1显示了我们要遇到的四种不同类型的 mbuf，它们依据在成员 `m_flags`中填写的不同标志`M_PKTHDR`和`M_EXT`而不同。图2-1中四个mbuf的区别从左到右罗列如下：

1) 如果`m_flags`等于0，mbuf只包含数据。在mbuf中有108字节的数据空间(`m_dat`数组)。指针`m_data`指向这108字节缓存中的某个位置。我们所示的 `m_data`指向缓存的起始，但它能指向缓存中的任意位置。成员 `m_len`指示了从`m_data`开始的数据的字节数。图1-6是这类mbuf的一个例子。

   在图2-1中，结构`m_hdr`中有六个成员，它的总长是 20字节。当我们查看此结构的 C语言定义时，会看见前四个成员每个占用 4字节而后两个成员每个占用 2字节。在图2-1中我们没有区分4字节成员和2字节成员。

2) 第二类mbuf的`m_flags`值是`M_PKTHDR`，它指示这是一个分组首部，描述一个分组数据的第一个 mbuf。数据仍然保存在这个 mbuf中，但是由于分组首部占用了 8字节，只有100字节的数据可存储在这个 mbuf中(在`m_pktdat`数组中)。图1-10是这种mbuf的一个例子。

   成员`m_pkthdr.len`的值是这个分组的 mbuf链表中所有数据的总长度：即所有通过`m_next`指针链接的 mbuf的`m_len`值的和，如图 1-8所示。输出分组没有使用成员`m_pkthdr.rcvif`，但对于接收的分组，它包含一个指向接收接口 `ifnet`结构(图3-6)的指针。

3) 下一种mbuf不包含分组首部(没有设置`K_PKTHDR`)，但包含超过208字节的数据，这时用到一个叫“ 簇 ”的外部缓存 (设置`M_EXT`)。在此mbuf中仍然为分组首部结构分配了空间，但没有用 —— 在图2-1中，我们用阴影显示出来。 Net/3分配一个大小为 1024或2048字节的簇，而不是使用多个 mbuf来保存数据(第一个带有100字节数据，其余的每个带有108字节数据)。在这个mbuf中，指针`m_data`指向这个簇中的某个位置。

   Net/3版本支持七种不同的结构。定义了四种 1024字节的簇(惯例值)，三种2048字节的

簇。传统上用1024字节的原因是为了节约存储器：如果簇的大小是 2048，对于以太网分组(最大1500字节)，每个簇大约有四分之一没有用。在 27.5节中我们会看到 Net/3 TCP发送的每个 TCP报文段从来不超过一簇大小，因为当簇的大小为 1024时，每个1500字节的以太网帧几乎三分之一未用。但是 [Mogul 1993，图15-15]显示了当在以太网中发送最大帧而不是1024字节的帧时能明显提高以太网的性能。这就是一种性能 /存储器互换。老的系统使用 1024字节簇来节约存储器，而拥有廉价存储器的新系统用2048字节的簇来提高性能。在本书中我们假定一簇的大小是 2048字节。

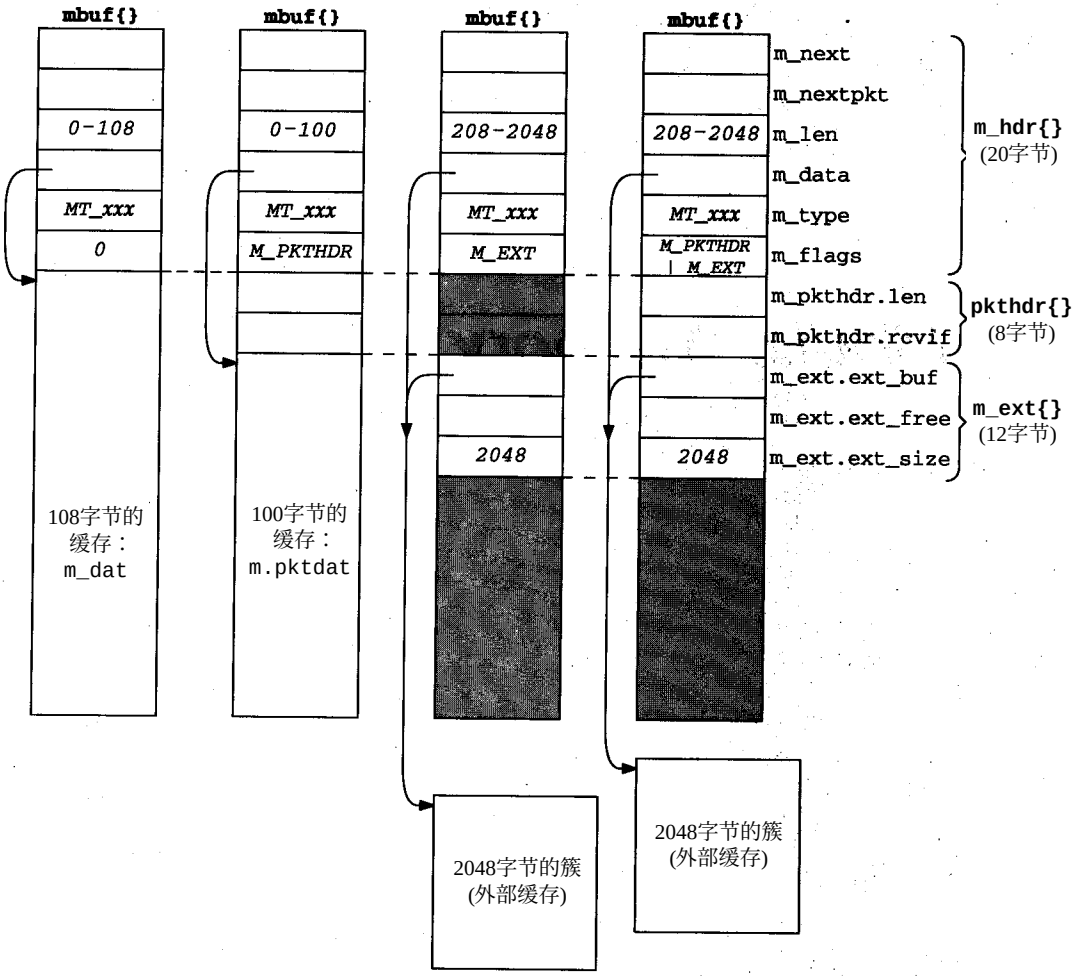


图2-1 根据不同m_flags 值的四种不同类型的mbuf

不幸的是，我们所说的“簇(cluster)”用过不同的名字。常量MCLBYTES是这些缓存(1024或2048)的大小，操作这些缓存的宏的名字是 MCLGET、MCLALLOC和MCLFREE。这就是为什么称它们为“簇”的原因。但我们还看到 mbuf的标志是M_EXT，它代表“外部的”缓存。最后，[Leffler et al. 1989]称它们为映射页(mapped page)。这后一种称法来源于它们的实现，在 2.9节我们会看到当要求一个副本时，这些簇是可以共享的。

我们可能会希望这种类型的 mbuf的m_len的最小值是209而不是我们在图中所示

的208。这是指，208字节数据的记录是可以存放在两个 mbuf中的，第一个mbuf存放100字节，第二个mbuf存放108字节。但在源代码中有一个差错：若超过或等于 208就分配一个簇。

4) 最后一类 mbuf包含一个分组首部，并包含超过 208字节的数据。同时设置了标志M_PKTHDR和M_EXT。

对于图2-1，我们还有另外几点需要说明：

- mbuf结构的大小总是 128字节。这意味着图 2-1右边两个mbuf在结构m_ext后面的未用空间为88字节(128 - 20 - 8 - 12)。
- 既然有些协议(例如UDP)允许零长记录，当然就可以有m_len为0的数据缓存。
- 在每个mbuf中的成员m_data指向相应缓存的开始(mbuf缓存本身或一个簇)。这个指针能指向相应缓存的任意位置，不一定是起始。

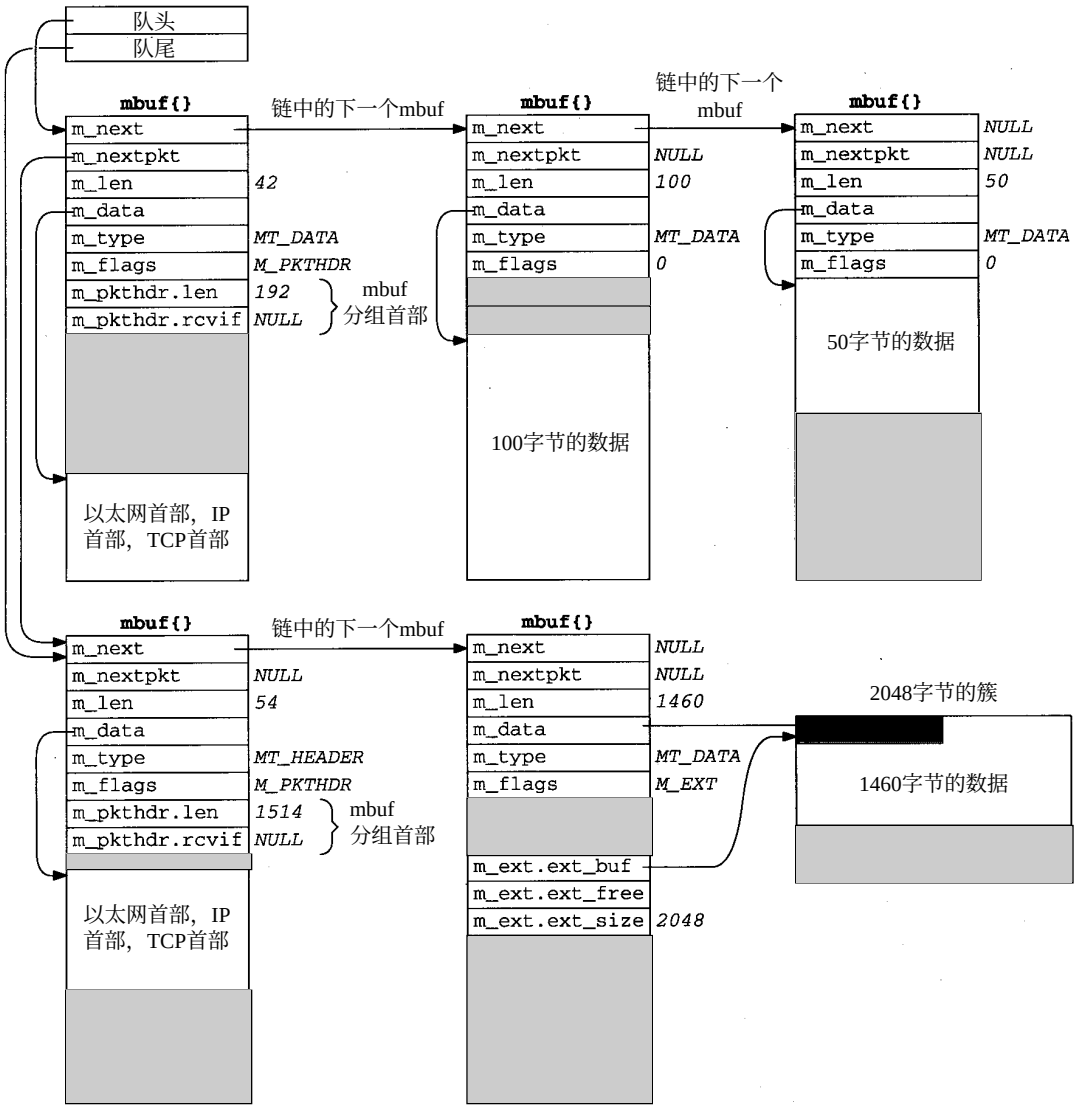


图2-2 在一个队列中的两个分组：第一个带有 192字节数据，第二个带有1514字节数据

- 带有簇的mbuf总是包含缓存的起始地址(m_ext.ext_buf)和它的大小(m_ext.ext_size)。我们在本书采用的大小为2048。成员m_data和m_ext.ext_buf值是不同的(如我们所示)，除非m_data也指向缓存的第一个字节。结构 m_ext的第三个成员 ext_free，Net/3当前未用。
- 指针m_next把mbuf链接在一起，把一个分组(记录)形成一条mbuf链表，如图1-8所示。
- 指针m_nextpkt把多个分组(记录)链接成一个mbuf链表队列。在队列中的每个分组可以是一个单独的mbuf，也可以是一个mbuf链表。每个分组的第一个mbuf包含一个分组首部。如果多个mbuf定义一个分组，只有第一个mbuf的成员m_nextpkt被使用——链表中其他mbuf的成员m_nextpkt全是空指针。

图2-2所示的是在一个队列中的两个分组的例子。它是图 1-8的一个修改版。我们已经把UDP数据报放到接口输出队列中(显示出14字节的以太网首部已经添加到链表中第一个 mbuf的IP首部前面)，并且第二个分组已经被添加到队列中： TCP段包含1460字节的用户数据。 TCP数据包含在一个簇中，并且有一个 mbuf包含了它的以太网、 IP与TCP首部。通过这个簇，我们可以看到指向簇的数据指针 (m_data)不需要指向簇的起始位置。我们所示的队列有一个头指针和一个尾指针。这就是Net/3处理接口输出队列的方法。我们给有 M_EXT标志的mbuf还添加了一个m_ext结构，并且用阴影表示这个mbuf中未用的pkthdr结构。

带有UDP数据报分组首部的第一个 mbuf的类型是MT_DATA，但带有TCP报文段分组首部的第一个mbuf的类型是MT_HEADER。这是由于UDP和TCP采用了不同的方式往数据中添加首部造成的，但没有什么不同。这两种类型的 mbuf本质上一样。链表中第一个mbuf的m_flags的值M_PKTHDR指示了它是一个分组首部。

仔细的读者可能会注意到我们显示一个 mbuf的图(Net/3 mbuf，图2-1)与显示一个Net/1 mbuf的图[Leffler et al. 1989，p.290]的区别。这个变化是在Net/2中造成的：添加了成员 m_flags，把指针 m_act改名为 m_nextpkt，并把这个指针移到这个mbuf的前面。

在第一个mbuf中，UDP与TCP协议首部位置的不同是由于 UDP调用M_PREPEND(图23-15和习题23.1)而TCP调用MGETHDR(图26-25)造成的。

2.2 代码介绍

mbuf函数在一个单独的C文件中，并且mbuf宏与各种mbuf定义都在一个单独的头文件中，如图2-3所示。

文 件	说 明
sys/mbuf.h	mbuf结构、mbuf宏与定义
kern/uipc_mbuf.c	mbuf函数

图2-3 本章讨论的文件

2.2.1 全局变量

在本章中有一个全局变量要介绍，如图 2-4所示。

变 量	数 据 类 型	说 明
mbstat	struct mbstat	mbuf的统计信息(图2-5)

图2-4 本章介绍的全局变量

2.2.2 统计

在全局结构mbstat中维护的各种统计，如图2-5所示。

mbstat成员	说 明
m_clfree	自由簇
m_clusters	从页池中获得的簇
m_drain	调用协议的drain函数来回收空间的次数
m_drops	寻找空间(未用)失败的次数
m_mbufs	从页池(未用)中获得的mbuf数
m_mtypes[256]	当前mbuf的分配数：MT_*xxx*索引
m_spare	剩余空间(未用)
m_wait	等待空间(未用)的次数

图2-5 在结构mbstat 中维护的mbuf统计

这个结构能被命令netstat -m检测；图2-6所示的是一些输出示例。关于所用映射页的数量的两个值是：m_clusters(34)减m_clfree (32)——当前使用的簇数(2)和m_clusters(34)。

分配给网络的存储器的千字节数是 mbuf存储器(99×128字节)加上簇存储器(34×2048字节)再除以1024。使用百分比是mbuf存储器(99×128字节)加上所用簇的存储器(2×2048字节)除以网络存储器总数(80千字节)，再乘100。

netstat -m output	mbstat member
99 mbufs in use:	
1 mbufs allocated to data	m_mtypes[MT_DATA]
43 mbufs allocated to packet headers	m_mtypes[MT_HEADER]
17 mbufs allocated to protocol control blocks	m_mtypes[MT_PCB]
20 mbufs allocated to socket names and addresses	m_mtypes[MT_SONAME]
18 mbufs allocated to socket options	m_mtypes[MT_SOOPTS]
2/34 mapped pages in use	(see text)
80 Kbytes allocated to network (20% in use)	(see text)
0 requests for memory denied	m_drops
0 requests for memory delayed	m_wait
0 calls to protocol drain routines	m_drain

图2-6 mbuf统计例子

2.2.3 内核统计

mbuf统计显示了在Net/3源代码中的一种通用技术。内核在一个全局变量 (在本例中是结构mbstat)中保持对某些统计信息的跟踪。当内核在运行时，一个进程 (在本例中是netstat程序)可以检查这些统计。

不是提供系统调用来获取由内核维护的统计，而是进程通过读取链接编辑器在内核建立时保存的信息来获得所关心的数据结构在内核中的地址。然后进程调用函数 kvm(3)，通过使用特殊文件/dev/mem读取在内核存储器中的相应位置。如果内核数据结构从一个版本改变

为下一版本，任何读取这个结构的程序也必须改变。

2.3 mbuf的定义

处理mbuf时，我们会反复遇到几个常量。它们的值显示在图 2-7中。除了MCLBYTES定义在文件/usr/include/machine/param.h中外，其他所有常量都定义在文件mbuf.h中。

常　量	值(字节数)	说　明
MCLBYTES	2048	一个mbuf簇(外部缓存)的大小
MHLEN	100	带分组首部的mbuf的最大数据量
MINCLSIZE	208	存储到簇中的最小数据量
MLEN	108	在正常mbuf中的最大数据量
MSIZE	128	每个mbuf的大小

图2-7 mbuf.h 中的mbuf常量

2.4 mbuf结构

图2-8所示的是mbuf结构的定义。

mbuf.h

```
60 /* header at beginning of each mbuf: */
61 struct m_hdr {
62     struct mbuf *mh_next;        /* next buffer in chain */
63     struct mbuf *mh_nextpkt;     /* next chain in queue/record */
64     int      mh_len;             /* amount of data in this mbuf */
65     caddr_t mh_data;             /* pointer to data */
66     short    mh_type;            /* type of data (Figure 2.10) */
67     short    mh_flags;           /* flags (Figure 2.9) */
68 };

69 /* record/packet header in first mbuf of chain; valid if M_PKTHDR set */
70 struct pkthdr {
71     int      len;                /* total packet length */
72     struct ifnet *rcvif;         /* receive interface */
73 };

74 /* description of external storage mapped into mbuf, valid if M_EXT set */
75 struct m_ext {
76     caddr_t ext_buf;             /* start of buffer */
77     void    (*ext_free) ();      /* free routine if not the usual */
78     u_int   ext_size;            /* size of buffer, for ext_free */
79 };

80 struct mbuf {
81     struct m_hdr m_hdr;
82     union {
83         struct {
84             struct pkthdr MH_pkthdr;     /* M_PKTHDR set */
85             union {
86                 struct m_ext MH_ext;     /* M_EXT set */
87                 char     MH_databuf[MHLEN];
88             } MH_dat;
89         } MH;
90         char     M_databuf[MLEN];     /* !M_PKTHDR, !M_EXT */
91     } M_dat;
92 };
```

图2-8 mbuf 结构

```
 93 #define m_next       m_hdr.mh_next
 94 #define m_len        m_hdr.mh_len
 95 #define m_data       m_hdr.mh_data
 96 #define m_type       m_hdr.mh_type
 97 #define m_flags      m_hdr.mh_flags
 98 #define m_nextpkt    m_hdr.mh_nextpkt
 99 #define m_act        m_nextpkt
100 #define m_pkthdr     M_dat.MH.MH_pkthdr
101 #define m_ext        M_dat.MH.MH_dat.MH_ext
102 #define m_pktdat     M_dat.MH.MH_dat.MH_databuf
103 #define m_dat        M_dat.M_databuf
```

mbuf.h

图2-8 (续)

结构mbuf是用一个m_hdr结构跟着一个联合来定义的。如注释所示，联合的内容依赖于标志M_PKTHDR和M_EXT。

93-103 这11个#define语句简化了对mbuf结构中的结构与联合的成员的访问。我们会看到这种技术普遍应用于Net/3源代码中，只要是一个结构包含其他结构或联合这种情况。

我们在前面说明了在结构 mbuf中前两个成员的目的：指针 m_next把mbuf链接成一个mbuf链表，而指针m_nextpkt把mbuf链表链接成一个mbuf队列。

图1-8显示了每个mbuf的成员m_len与分组首部中的成员 m_pkthdr.len的区别。后者是链表中所有mbuf的成员m_len的和。

图2-9所示的是成员m_flags的五个独立的值。

m_flags	说 明
M_BCAST	作为链路层广播发送/接收
M_EOR	记录结束
M_EXT	此mubf带有簇(外部缓存)
M_MCAST	作为链路层多播发送/接收
M_PKTHDR	形成一个分组(记录)的第一个mbuf
M_COPYFLAGS	*M_PKTHDR/M_EOR/M_BCAST/M_MCAST*

图2-9 m_flags 值

我们已经说明了标志 M_EXT和M_PKTHDR。M_EOR在一个包含记录尾的 mbuf中设置。Internet协议(例如TCP)从来不设置这个标志，因为TCP提供一个无记录边界的字节流服务。但是OSI与XNS运输层要用这个标志。在插口层我们会遇到这个标志，因为这一层是协议无关的，并且它要处理来自或发往所有运输层的数据。

当要往一个链路层广播地址或多播地址发送分组，或者要从一个链路层广播地址或多播地址接收一个分组时，在这个 mubf中要设置接下来的两个标志 M_BCAST和M_MCAST。这两个常量是协议层与接口层之间的标志(图1-3)。

对于最后一个标志值 M_COPYFLAGS，当一个mbuf包含一个分组首部的副本时，这个标志表明这些标志是复制的。

图2-10所示的常量MT_*xxx*用于成员m_type，指示存储在mbuf中的数据的类型。虽然我们总认为一个mbuf是用来存放要发送或接收的用户数据，但 mbuf可以存储各种不同的数据结构。回忆图1-6中的一个mbuf被用来存放一个插口地址结构，其中的目标地址用于系统调用sendto。它的m_type成员被设置为MT_SONAME。

不是图2-10中所有的mbuf类型值都用于Net/3。有些已不再使用(MT_HTABLE)，还有一些不用于TCP/IP代码中，但用于内核的其他地方。例如，MT_OOBDATA用于OSI和XNS协议，但是TCP用不同方法来处理带外(out-of-band)数据(我们在29.7节说明)。当我们在本书的后面遇到其他mbuf类型时会说明它们的用法。

Mbuf m_type	用于Net/3 TCP/IP代码	说　明	存储类型
MT_CONTROL	•	外部数据协议报文	M_MBUF
MT_DATA	•	动态数据分配	M_MBUF
MT_FREE		应在自由列表中	M_FREE
MT_FTABLE	•	分片重组首部	M_FTABLE
MT_HEADER	•	分组首部	M_MBUF
MT_HTABLE		IMP主机表	M_HTABLE
MT_IFADDR		接口地址	M_IFADDR
MT_OOBDATA		加速(带外)数据	M_MBUF
MT_PCB		协议控制块	M_PCB
MT_RIGHTS		访问权限	M_MBUF
MT_RTABLE		路由表	M_RTABLE
MT_SONAME	•	插口名称	M_MBUF
MT_SOOPTS	•	插口选项	M_SOOPTS
MT_SOCKET		插口结构	M_SOCKET

图2-10 成员m_type 的值

本图的最后一列所示的M_*xxx*值与内核为不同类型mbuf分配的存储器片有关。这里有大约60个可能的M_*xxx*值指派给由内核函数malloc和宏MALLOC分配的不同类型的存储器空间。图2-6所示的是来源于命令netstat -m的mbuf分配统计信息，它包括每种MT_*xxx*类型的统计。命令vmstat -m显示了内核的存储分配统计，包括每个M_*xxx*类型的统计。

由于mbuf有一个固定长度(128字节)，因此对于mbuf的使用有一个限制——包含的数据不能超过108字节。Net/3用一个mbuf来存储一个TCP协议控制块(在第24章我们会涉及到)，这个mbuf的类型为MT_PCB。但是4.4BSD把这个结构的大小从108字节增加到140字节，并为这个结构使用一种不同的内核存储器分配类型。

仔细的读者会注意到图2-10中我们表明未使用MT_PCB类型的mbuf，而图2-6显示这个类型的计数不为零。Unix域协议使用这种类型的mbuf，并且mbuf的统计功能用于所有协议，而不只是Internet协议，记住这一点很重要。

2.5 简单的mbuf宏和函数

有超过两打的宏和函数来处理mbuf(分配一个mbuf，释放一个mbuf，等等)。让我们来查看几个宏与函数的源代码，看看它们是如何实现的。

有些操作既提供了宏也提供了函数。宏版本的名称是以M开头的大写字母名称，而函数是以m_开始的小写字母名称。两者的区别是一种典型的时间-空间互换。宏版本在每个被用到的地方都被C预处理器展开(要求更多的代码空间)，但是它在执行时更快，因为它不需要执行函数调用(对于有些体系结构，这是费时的)。而对于函数版本，它在每个被调用的地方变成了一些指令(参数压栈，调用函数等)，要求较少的代码空间，但会花费更多的执行时间。

2.5.1 m_get函数

让我们先看一下图2-11中分配mbuf的函数：m_get。这个函数仅仅就是宏MGET的展开。

uipc_mbuf.c

```
134 struct mbuf *
135 m_get(nowait, type)
136 int     nowait, type;
137 {
138     struct mbuf *m;
139     MGET(m, nowait, type);
140     return (m);
141 }
```

uipc_mbuf.c

图2-11 m_get 函数：分配一个mbuf

注意，Net/3代码不使用ANSI C参数声明。但是，如果使用一个ANSI C编译器，所有Net/3系统头文件为所有的内核函数都提供了ANSI C函数原型。例如，<sys/mbuf.h>头文件中包含这样的行：

```
struct mbuf *m_get(int, int);
```

这些函数原型为所有内核函数的调用提供编译期间的参数与返回值的检查。

这个调用表明参数nowait的值为M_WAIT或M_DONTWAIT，它取决于在存储器不可用时是否要求等待。例如，当插口层请求分配一个 mbuf来存储sendto系统调用(图1-6)的目标地址时，它指定M_WAIT，因为在此阻塞是没有问题的。但是当以太网设备驱动程序请求分配一个mbuf来存储一个接收的帧时 (图1-10)，它指定M_DONTWAIT，因为它是作为一个设备中断处理来执行的，不能进入睡眠状态来等待一个 mbuf。在这种情况下，若存储器不可用，设备驱动程序丢弃这个帧比较好。

2.5.2 MGET宏

图2-12所示的是MGET宏。调用MGET来分配存储sendto系统调用(图1-6)的目标地址的mbuf如下所示：

```
MGET(m, M_WAIT, MT_SONAME);
if (m == NULL)
   return(ENOBUFS);
```

mbuf.h

```
154 #define MGET(m, how, type) { \
155     MALLOC((m), struct mbuf *, MSIZE, mbtypes[type], (how)); \
156     if (m) { \
157         (m)->m_type = (type); \
158         MBUFLOCK(mbstat.m_mtypes[type]++;) \
159         (m)->m_next = (struct mbuf *)NULL; \
160         (m)->m_nextpkt = (struct mbuf *)NULL; \
161         (m)->m_data = (m)->m_dat; \
162         (m)->m_flags = 0; \
163     } else \
164         (m) = m_retry((how), (type)); \
165 }
```

mbuf.h

图2-12 MGET 宏

虽然调用指定了M_WAIT，但返回值仍然要检查，因为，如图2-13所示，等待一个mbuf并不保证它是可用的。

154-157 MGET一开始调用内核宏MALLOC，它是通用内核存储器分配器进行的。数组mbtypes把mbuf的MT_*xxx*值转换成相应的M_*xxx*值(图2-10)。若存储器被分配，成员m_type被设置为参数中的值。

158 用于跟踪统计每种mbuf类型的内核结构加1(mbstat)。当执行这句时，宏MBUFLOCK把它作为参数来改变处理器优先级(图1-13)，然后把优先级恢复为原值。这防止在执行语句mbstat.m_mtypes[type]++；时被网络设备中断，因为mbuf可能在内核中的各层中被分配。考虑这样一个系统，它用三步来实现一个C中的++运算：(1)把当前值装入一个寄存器；(2)寄存器加1；(3)把寄存器值存入存储器。假设计数器值为77并且MGET在插口层执行。假设执行了步骤1和2(寄存器值为78)，并且一个设备中断发生。若设备驱动也执行MGET来获得同种类型的mbuf，在存储器中取值(77)，加1(78)，并存回在存储器。当被中断执行的MGET的步骤3继续执行时，它将寄存器的值(78)存入存储器。但是计数器应为79，而不是78，这样计数器就被破坏了。

159-160 两个mbuf指针，m_next和m_nextpkt，被设置为空指针。若必要，由调用者把这个mbuf加入到一个链或队列。

161-162 最后，数据指针被设置为指向108字节的mbuf缓存的起始，而标志被设置为0。

163-164 若内核的存储器分配调用失败，调用m_retry(图2-13)。第一个参数是M_WAIT或M_DONTWAIT。

2.5.3 m_retry函数

图2-13所示的是m_retry函数。

uipc_mbuf.c

```
 92 struct mbuf *
 93 m_retry(i, t)
 94 int     i, t;
 95 {
 96     struct mbuf *m;

 97     m_reclaim();
 98 #define m_retry(i, t)    (struct mbuf *)0
 99     MGET(m, i, t);
100 #undef m_retry
101     return (m);
102 }
```

uipc_mbuf.c

图2-13 m_retry 函数

92-97 被m_retry调用的第一个函数是m_reclaim。在7.4节我们会看到每个协议都能定义一个“drain”函数，在系统缺乏可用存储器时能被m_reclaim调用。在图10-32中我们还会发现当IP的drain函数被调用时，所有等待重新组成IP数据报的IP分片被丢弃。TCP的drain函数什么都不做，而UDP甚至就没有定义一个drain函数。

98-102 因为在调用了m_reclaim后有可能有机会得到更多的存储器，因此再次调用宏MGET，试图获得mbuf。在展开宏MGET(图2-12)之前，m_retry被定义为一个空指针。这可以防止当存储器仍然不可用时的无休止的循环：这个MGET展开会把m设置为空指针而不是调用m_retry函数。在MGET展开以后，这个m_retry的临时定义就被取消了，以防在此之后

有对MGET的其他引用。

2.5.4 mbuf锁

在本节中我们所讨论的函数和宏并不调用 spl函数，而是调用图2-12中的MBUFLOCK来保护这些函数和宏不被中断。但在宏 MALLOC的开始包含一个splimp，在结束时有一个splx。宏MFREE中包含同样的保护机制。由于 mbuf在内核的所有层中被分配和释放，因此内核必须保护那些用于存储器分配的数据结构。

另外，用于分配和释放mbuf簇的宏MCLALLOC与MCLFREE要用一个splimp和一个splx包括起来，因为它们修改的是一个可用簇链。

因为存储器分配与释放及簇分配与释放的宏被保护起来防止被中断，我们通常在 MGET和m_get这样的函数和宏的前后不再调用 spl函数。

2.6 m_devget和m_pullup函数

我们在讨论IP、ICMP、IGMP、UDP和TCP的代码时会遇到函数m_pullup。它用来保证指定数目的字节 (相应协议首部的大小)在链表的第一个 mbuf中紧挨着存放；即这些指定数目的字节被复制到一个新的 mbuf并紧接着存放。为了理解 m_pullup的用法，必须查看它的实现及相关的函数 m_devget和宏mtod与dtom。在分析这些问题的同时我们还可以再次领会Net/3中mbuf的用法。

2.6.1 m_devget函数

当接收到一个以太网帧时，设备驱动程序调用函数 m_devget来创建一个mbuf链表，并把设备中的帧复制到这个链表中。根据所接收的帧的长度 (不包括以太网首部)，可能导致4种不同的mbuf链表。图2-14所示的是前两种。

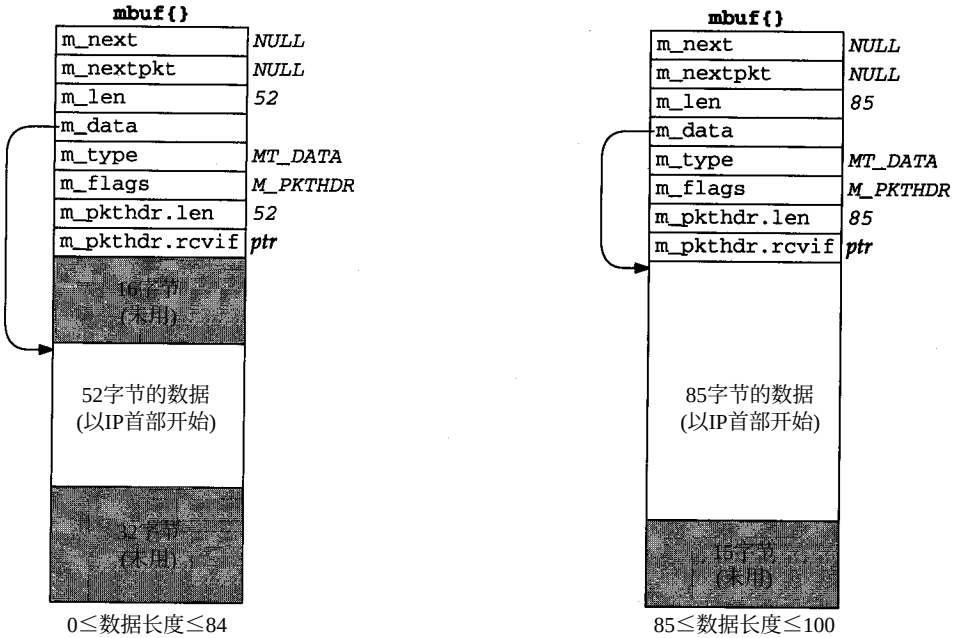


图2-14 m_devget 创建的前两种类型的mbuf

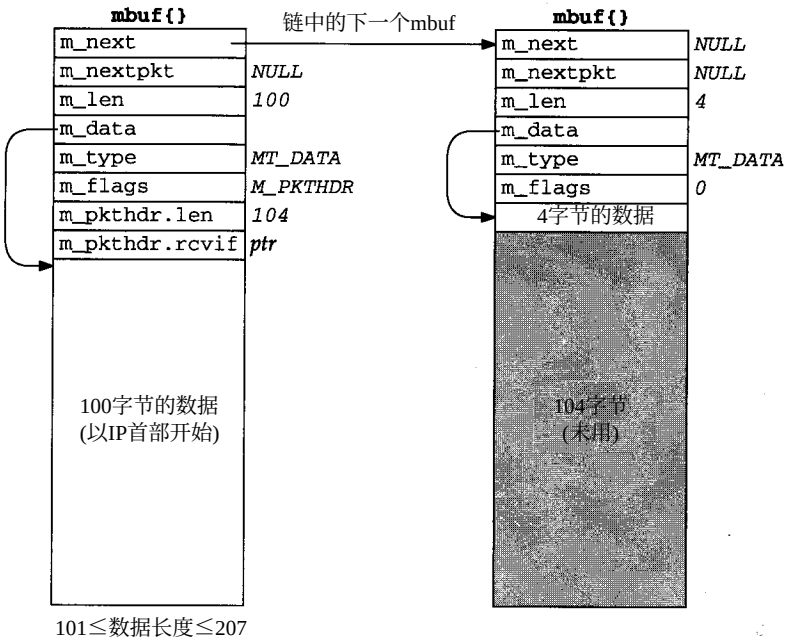


图2-15 m_devget 创建的第3种mbuf

1) 图2-14左边的mbuf用于数据的长度在0~84字节之间的情况。在这个图中，我们假定有52字节的数据：一个20字节的IP首部和一个32字节的TCP首部(标准的20字节的TCP首部加上12字节的TCP选项)，但不包括TCP数据。既然m_devget返回的mbuf数据从IP首部开始，m_len的实际最小值是28：20字节的IP首部，8字节的UDP首部和一个0长度的UDP数据报。m_devget在这个mbuf的开始保留了16字节未用。虽然14字节的以太网首部不存放在这里，还是分配了一个14字节的用于输出的以太网首部，这是同一个mbuf，用于输出。我们会遇到两个函数：icmp_reflect和tcp_respond，它们通过把接收到的mbuf作为输出mbuf来产生一个应答。在这两种情况中，接收的数据报应该少于84字节，因此很容易在前面保留16字节的空间，这样在建立输出数据报时可以节省时间。分配16字节而不是14字节的原因是为了在mbuf中用长字对准方式存储IP首部。

2) 如果数据在85~100字节之间，就仍然存放在一个分组首部mbuf中，但在开始没有16字节

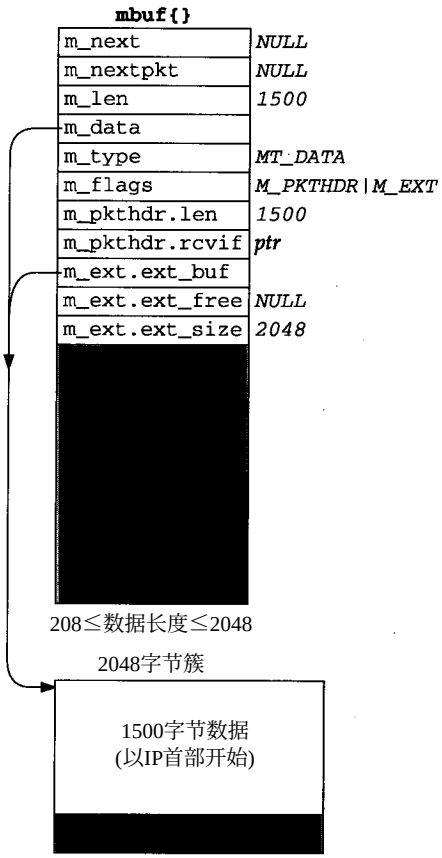


图2-16 m_devget 创建的第4种mbuf

的空间。数据存储在数组m_pktdat的开始，并且任何未用的空间放在这个数组的后面。例如在图2-14的右边的mbuf显示的就是这个例子，假设有85字节数据。

3) 图2-15所示的是m_devget创建的第3种mbuf。当数据在101~207字节之间时，要求有两个mbuf。前100字节存放在第一个mbuf中(有分组首部的mbuf)，而剩下的存放在第二个mbuf中。在此例中，我们显示的是一个104字节的数据报。在第一个mbuf的开始没有保留16字节的空间。

4) 图2-16所示的是m_devget创建的第四种mbuf。如果数据超过或等于208字节(MINCLBYTES)，要用一个或多个簇。图中的例子假设了一个1500字节的以太网帧。如果使用1024字节的簇，本例子需要两个mbuf，每个mbuf都有标志M_EXT，和指向一个簇的指针。

2.6.2 mtod和dtom宏

宏mtod和dtom也定义在文件mbuf.h中。它们简化了复杂的mbuf结构表达式。

```
#define  mtod(m,t)   ((t)((m)->m_data))
#define  dtom(x)     ((struct mbuf *)((int)(x) & ~(MSIZE-1)))
```

mtod(“mbuf到数据”)返回一个指向mbuf数据的指针，并把指针声名为指定类型。例如代码

```
struct mbuf *m;
struct ip *ip;

ip = mtod(m, struct ip *);
ip->ip_v = IPVERSION;
```

存储在mbuf的数据(m_data)指针ip中。C编译器要求进行类型转换，然后代码用指针ip引用IP首部。我们可以看到当一个C结构(通常是一个协议首部)存储在一个mbuf中时会用到这个宏。当数据存放在mbuf本身(图2-14和图2-15)或存放在一个簇中(图2-16)时，可以用这个宏。

宏dtom (“数据到mbuf”)取得一个存放在一个mbuf中任意位置的数据的指针，并返回这个mbuf结构本身的一个指针。例如，若我们知道ip指向一个mbuf的数据区，下面的语句序列

```
struct mbuf *m;
struct ip *ip;

m = dtom(ip);
```

把指向这个mbuf开始的指针存放到m中。我们知道MSIZE(128)是2的幂，并且内核存储器分配器总是为mbuf分配连续的MSIZE字节的存储块，dtom仅仅是清除参数中指针的低位来发现这个mbuf的起始位置。

宏dtom有一个问题：当它的参数指向一个簇，或在一个簇内，如图2-16时，它不能正确执行。因为那里没有指针从簇内指回mbuf结构，dtom不能被使用。这导致了下一个函数：m_pullup。

2.6.3 m_pullup函数和连续的协议首部

函数m_pullup有两个目的。第一个是当一个协议(IP、ICMP、IGMP、UDP或TCP)发现在第一个mbuf的数据量(m_len)小于协议首部的最小长度(例如：IP是20，UDP是8，TCP是20)时。调用m_pullup是基于假定协议首部的剩余部分存放在链表中的下一个mbuf。

m_pullup重新安排mbuf链表，使得前*N*字节的数据被连续地存放在链表的第一个 mbuf中。*N*是这个函数的一个参数，它必须小于或等于 100(MHLEN)。如果前*N*字节连续存放在第一个mbuf中，则可以使用宏mtod和dtom。

例如，我们在IP输入例程中会遇到下面这样的代码：

```
if (m->m_len < sizeof(struct ip) &&
    (m = m_pullup(m, sizeof(struct ip))) == 0) {
        ipstat.ips_toosmall++;
        goto next;
}
ip = mtod(m, struct ip *);
```

如果第一个mbuf中的数据少于20(标准IP首部的大小)，m_pullup被调用。函数m_pullup有两个原因会失败：(1)如果它需要其他mbuf并且调用MGET失败；或者(2)如果整个mbuf链表中的数据总数少于要求的连续字节数 (即我们所说的*N*，在本例中是20)。通常，失败是因为第二个原因。在此例中，如果m_pullup失败，一个IP计数器加1，并且此IP数据报被丢弃。注意，这段代码假设失败的原因是mbuf 链表中数据少于20字节。

实际上，在这种情况下，m_pullup很少能被调用(注意，C语言的&&操作符仅当mbuf长度小于期待值时才调用它)，并且当它被调用时，它通常会失败。通过查看图 2-14~图2-16，我们可以找到它的原因：在第一个mbuf中，或在簇中，从IP首部开始有至少100字节的连续字节。这允许60字节的最大IP首部，并且后面跟着40字节的TCP首部(其他协议——ICMP，IGMP和UDP——它们的协议首部不到40字节)。如果mbuf链表中的数据可用(分组不小于协议要求的最小值)，则所要求的字节数总能连续地存放在第一个 mbuf中。但是，如果接收的分组太小(m_len小于期待的最小值)，则m_pullup被调用，并且它返回一个差错，因为在 mbuf链表中没有所要求数目的可用数据。

> 源于伯克利的内核维护一个叫MPFail的变量，每次m_pullup失败时，它都加1。在一个Net/3系统中曾经接收了超过2700万的IP数据报，而MPFail只有9。计数器ipstat.ips_toosmall也是9，并且所有其他协议计数器(ICMP、IGMP、UDP和TCP等)所计的m_pullup失败次数为0。这证实了我们的断言：大多数 m_pullup的失败是因为接收的IP数据报太小。

2.6.4 m_pullup和IP的分片与重组

使用m_pullup的第二个用途涉及到IP和TCP的重组。假定IP接收到一个长度为296的分组，这个分组是一个大的IP数据报的一个分片。这个从设备驱动程序传到 IP输入的mbuf看起来像我们在图2-16中所示的一个mbuf：296字节的数据存放在一个簇中。我们将这显示在图 2-17中。

问题在于，IP的分片算法将各分片都存放在一个双向链表中，使用 IP首部中的源与目标IP地址来存放向前与向后链表指针 (当然，这两个IP地址要保存在这个链表的表头中，因为它们还要放回到重组的数据报中。我们在第 10章讨论这个问题)。但是如果这个IP首部在一个簇中，如图2-17所示，这些链表指针会存放在这个簇中，并且当以后遍历链表时，指向 IP首部的指针(即指向这个簇的起始的指针)不能被转换成指向mbuf的指针。这是我们在本节前面提到的问题：如果m_data指向一个簇时不能使用宏dtom，因为没有从簇指回mbuf的指针。IP分片

不能如图2-17所示的把链指针存储在簇中。

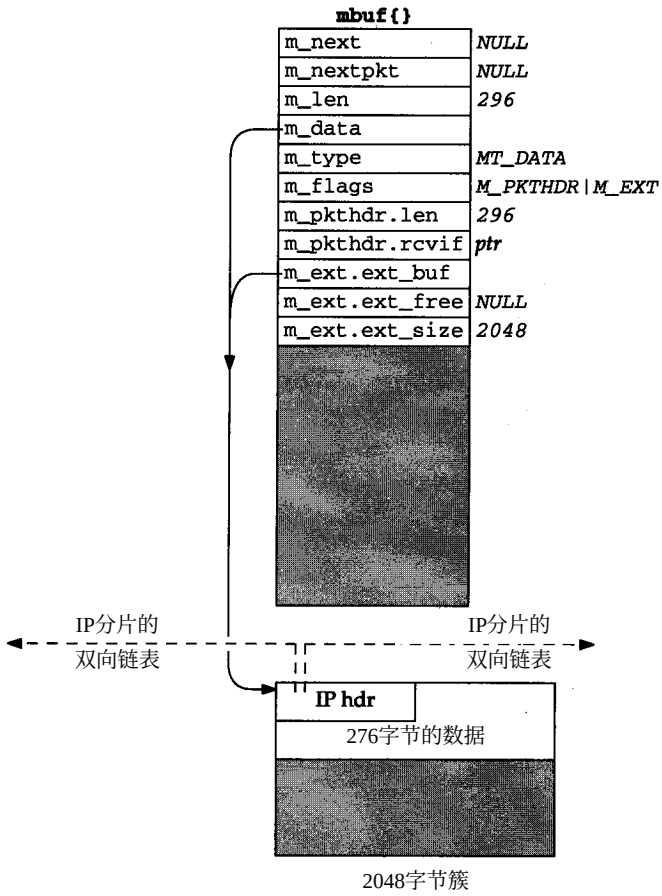


图2-17 一个长度为296的IP分片

为解决这个问题，当接收到一个分片时，若分片存放在一个簇中， IP分片例程总是调用m_pullup。它强行将20字节的IP首部放到它自己的mbuf中。代码如下：

```
if (m->m_flags & M_EXT) {
    if ((m = m_pullup(m, sizeof(struct ip))) == 0) {
        ipstat.ips_toosmall++;
        goto next;
    }
    ip = mtod(m, struct ip *);
}
```

图2-18所示的是在调用了m_pullup后得到的mbuf链表。m_pullup分配了一个新的mubf，挂在链表的前面，并从簇中取走40字节放入到这个新的mbuf中。之所以取40字节而不是仅要求的20字节，是为了保证以后在IP把数据报传给一个高层协议(例如：ICMP，IGMP，UDP或TCP)时，高层协议能正确处理。采用不可思议的40(图7-17中的max_protohdr)是因为最大协议首部通常是一个20字节的IP首部和20字节的TCP首部的组合(这假设其他协议族，例如OSI协议，并不编译到内核中)。

在图2-18中，IP分片算法在左边的mbuf中保存了一个指向IP首部的指针，并且可以用dtom将这个指针转换成一个指向mbuf本身的指针。

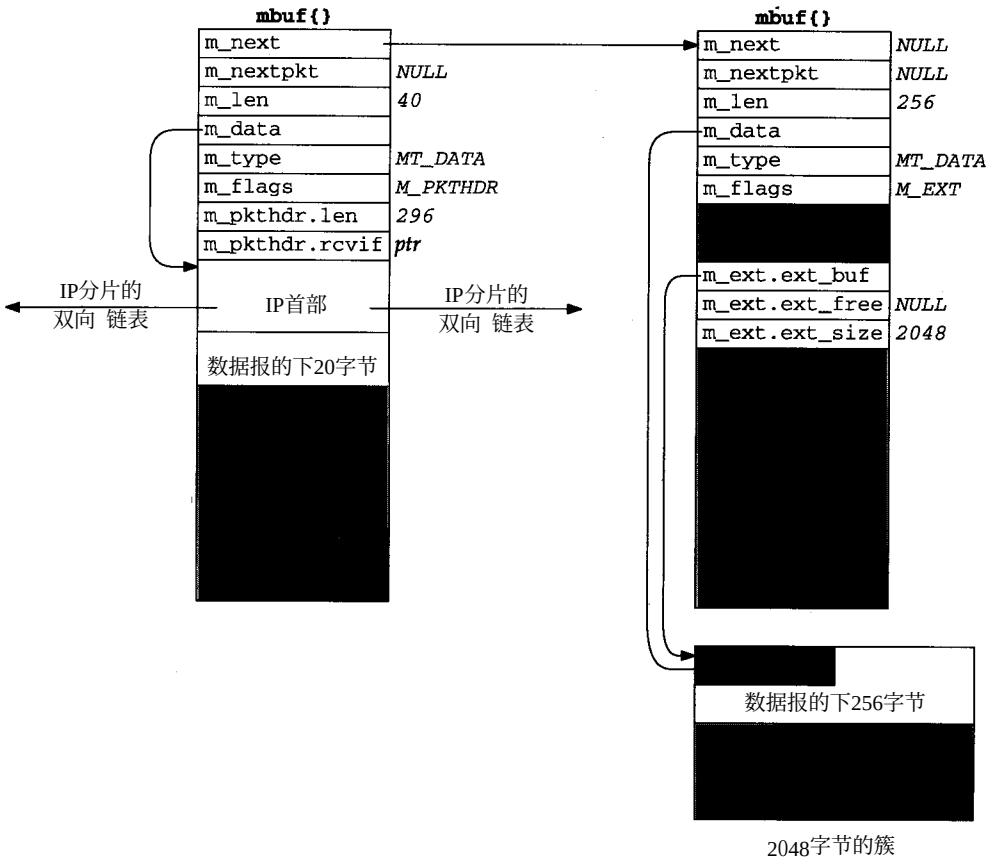


图2-18 调用m_pullup 后的长度为296的IP分片

2.6.5 TCP重组避免调用m_pullup

重组TCP报文段使用一个不同的技术而不是调用 m_pullup。这是因为m_pullup开销较大：分配存储器并且数据从一个簇复制到一个 mbuf中。TCP试图尽可能地避免数据的复制。

卷1的第19章提到大约有一半的 TCP数据是批量数据 (通常每个报文段有 512或更多字节的数据)，并且另一半是交互式数据 (这里面有大约90%的报文段包含不到 10字节的数据)。因此，当TCP从IP接收报文段时，它们通常是如图 2-14左边所示的格式 (一个小量的交互数据，存储在mbuf本身)或图2-16所示的格式 (批量数据，存储在一个簇中)。当TCP报文段失序到达时，它们被TCP存储到一个双向链表中。如IP分片一样，在IP首部的字段用于存放链表的指针，既然这些字段在TCP接收了IP数据报后不再需要，这完全可行。但当 IP首部存放在一个簇中，要将一个链表指针转换成一个相应的 mbuf指针时，会引起同样的问题 (图2-17)。

为解决这个问题，在27.9节中我们会看到TCP把mbuf指针存放在TCP首部中的一些未用的字段中，提供一个从簇指回 mbuf的指针，来避免对每个失序的报文段调用 m_pullup。如果IP首部包含在mbuf的数据区(图2-18)，则这个回指指针是无用的，因为宏 dtom对这个链表指针会正常工作。但如果IP首部包含在一个簇中，这个回指指针将被使用。当我们在讨论 27.9节的tcp_reass时，会研究实现这种技术的源代码。

2.6.6 m_pullup使用总结

我们已经讨论了关于使用m_pullup的三种情况：

- 大多数设备驱动程序不把一个IP数据报的第一部分分割到几个mbuf中。假设协议首部都紧挨着存放，则在每个协议(IP、ICMP、IGMP、UDP和TCP)中调用m_pullup的可能性很小。如果调用m_pullup，通常是因为IP数据报太小，并且m_pullup返回一个差错，这时数据报被丢弃，并且差错计数器加1。
- 对于每个接收到的IP分片，当IP数据报被存放在一个簇中时，m_pullup被调用。这意味着，几乎对于每个接收的分片都要调用m_pullup，因为大多数分片的长度大于208字节。
- 只要TCP报文段不被IP分片，接收一个TCP报文段，不论是否失序，都不需调用m_pullup。这是避免IP对TCP分片的一个原因。

2.7 mbuf宏和函数的小结

在操作mbuf的代码中，我们会遇到图2-19中所列的宏和图2-20中所列的函数。图2-19中的宏以函数原型的形式显示，而不是以#define形式来显示参数的类型。由于这些宏和函数主要用于处理mbuf数据结构并且不涉及联网问题，因此我们不查看实现它们的源代码。还有另外一些mbuf宏和函数用于Net/3源代码的其他地方，但由于我们在本书中不会遇到它们，因此没有把它们列于图中。

宏	描　述
MCLGET	获得一个簇(一个外部缓存)并将*m*指向的mbuf中的数据指针(m_data)设置为指向这个簇。如果存储器不可用，返回时不设置mbuf中的M_EXT标志 `void MCLGET(struct mbuf *m, int nowait);`
MFREE	释放一个*m*指向的mbuf。若*m*指向一个簇(设置了M_EXT)，这个簇的引用计数器减1，但这个簇并不被释放，直到它的引用计数器降为0(如2.9节所述)。返回*m*的后继(由m->m_next指向，可以为空)存放在*n*中 `void MFREE(struct mbuf *m, struct mbuf *n);`
MGETHDR	分配一个mbuf，并把它初始化为一个分组首部。这个宏与MGET(图2-12)相似，但设置了标志M_PKTHDR，并且数据指针(m_data)指向紧接分组首部后的100字节的缓存 `void MGETHDR(struct mbuf *m, int nowait, int type);`
MH_ALIGN	设置包含一个分组首部的mbuf的m_data，在这个mbuf数据区的尾部为一个长度为len字节的对象提供空间。这个数据指针也是长字对准方式的 `void MH_ALIGN(struct mbuf *m, int len);`
M_PREPEND	在*m*指向的mbuf中的数据的前面添加*len*字节的数据。如果mbuf有空间，则仅把指针(m_data)减len字节，并将长度(m_len)增加*len*字节。如果没有足够的空间，就分配一个新的mbuf，它的m_next指针被设置为*m*。一个新mbuf的指针存放在*m*中。并且新mbuf的数据指针被设置，这样*len*字节的数据放置到这个mbuf的尾部(例如，调用MH_ALIGN)。如果一个新mbuf被分配，并且原来的mbuf的分组首部标志被设置，则分组首部从老mbuf中移到新mbuf中 `void M_PREPEND(struct mbuf *m, int len, int nowait);`
dtom	将指向一个mbuf数据区中某个位置的指针*x*转换成一个指向这个mbuf的起始的指针。 `struct mbuf *dtom(void *x);`
mtod	将*m*指向的mbuf的数据区指针的类型转换成*type*类型 `type mtod(struct mbuf *m, type);`

图2-19　我们在本书中会遇到的mbuf宏

函 数	说 明
m_adj	从*m*指向的mbuf中移走*len*字节的数据。如果*len*是正数，则所操作的是紧排在这个mbuf的开始的*len*字节数据；否则是紧排在这个mbuf的尾部的*len*绝对值字节数据 `void m_adj(struct mbuf *m, int len);`
m_cat	把由*n*指向的mbuf链表链接到由*m*指向的mbuf链表的尾部。当我们讨论IP重组时(第10章)会遇到这个函数 `void m_cat(struct mbuf *m, struct mbuf *n);`
m_copy	这是m_copym的三参数版本，它隐含的第4个参数的值为M_DONTWAIT `struct mbuf * m_copy(struct mbuf *m, int offset, int len);`
m_copydata	从*m*指向的mbuf链表中复制*len*字节数据到由*cp*指向的缓存。从mbuf链表数据区起始的*offset*字节开始复制 `void m_copydata(struct mbuf *m, int offset, int len, caddr_t cp);`
m_copyback	从*cp*指向的缓存复制*len*字节的数据到由*m*指向的mbuf，数据存储在mbuf链表起始*offset*字节后。必要时，mbuf链表可以用其他mbuf来扩充 `void m_copyback(struct mbuf *m, int offset, int len, caddr_t cp);`
m_copym	创建一个新的mbuf链表，并从*m*指向的mbuf链表的开始*offset*处复制*len*字节的数据。一个新mbuf链表的指针作为此函数的返回值。如果*len*等于常量M_COPYALL，则从这个mbuf链表的*offset*开始的所有数据都将被复制。在2.9节中，我们会更详细地介绍这个函数 `struct mbuf *m_copym(struct mbuf *m, int offset, int len, int nowait);`
m_devget	创建一个带分组首部的mbuf链表，并返回指向这个链表的指针。这个分组首部的len和rcvif字段被设置为*len*和*ifp*。调用函数*copy*从设备接口(由*buf*指向)将数据复制到mbuf中。如果*copy*是一个空指针，调用函数bcopy。由于尾部协议不再被支持，*off*为0。我们在2.6节讨论了这个函数 `struct mbuf *m_devget(char *buf, int len, int off, struct ifnet *ifp, void (*copy)(const void *, void *, u_int));`
m_free	宏MFREE的函数版本 `struct mbuf *m_free(struct mbuf *m);`
m_freem	释放*m*指向的链表中的所有mbuf `void m_freem(struct mbuf *m);`
m_get	宏MGET的函数版本。我们在图2-12中显示过此函数 `struct mbuf *m_get(int nowait, int type);`
m_getclr	此函数调用宏MGET来得到一个mbuf，并把108字节的缓存清零 `struct mbuf *m_getclr(int nowait, int type);`
m_gethdr	宏MGETHDR的函数版本 `struct mbuf *m_gethdr(int nowait, int type);`
m_pullup	重新排列由*m*指向的mbuf中的数据，使得前*len*字节的数据连续地存储在链表中的第一个mbuf中。如果这个函数成功，则宏mtod能返回一个正好指向这个大小为*len*的结构。我们在2.6节讨论了这个函数 `struct mbuf *m_pullup(struct mbuf *m, int len);`

图2-20 在本书中我们要遇到的mbuf函数

所有原型的参数*nowait*是M_WAIT或M_DONTWAIT，参数*type*是图2-10中所示的MT_*xxx*中的一个。

M_PREPEND的一个例子是，从图1-7转换到图1-8的过程中，当IP和UDP首部被添加到数据的前面时要调用这个宏，因为另一个 mbuf要被分配。但当这个宏再次被调用 (从图1-8转换成图2-2)来添加以太网首部时，在那个mbuf中已有存放这个首部的空间。

M_copydata的最后一个参数的类型是caddr_t，它代表“内核地址”。这个数据类型通常定义在<sys/types.h>中，为char *。它最初在内核中使用，但被某些系统调用使用时被外露出来。例如， mmap系统调用，不论是4.4BSD或SVR4都把caddr_t作为第一个参数的类型并作为返回值类型。

2.8 Net/3联网数据结构小结

本节总结我们在Net/3联网代码中要遇到的数据结构类型。在 Net/3内核中用到其他数据结构(感兴趣的读者可以查看头文件<sys/queue.h>)，但下面这些是我们在本书中要遇到的。

1) 一个mbuf链：一个通过m_next指针链接的mbuf链表。我们已经看过几个这样的例子。

2) 只有一个头指针的 mbuf链的链表。 mbuf链通过每个链的第一个 mbuf中的m_nextpkt指针链接起来。

图2-21所示的就是这种链表。这种数据结构的例子是一个插口发送缓存和接收缓存。

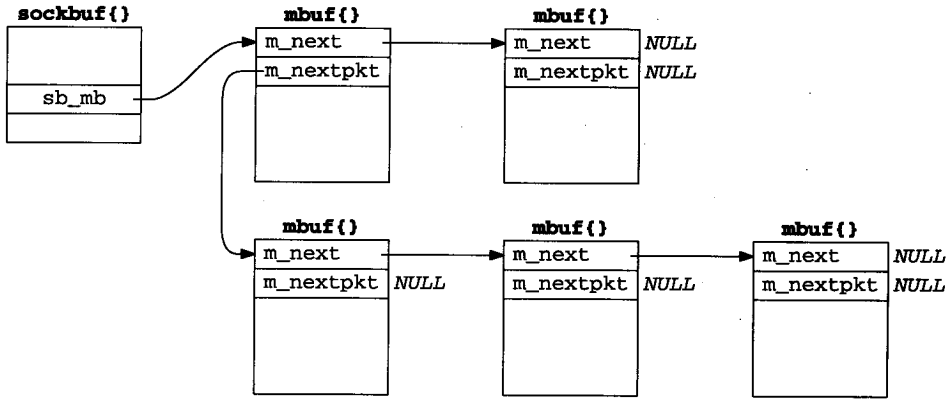


图2-21 只有头指针的mbuf链的链表

顶部的两个mbuf形成这个队列中的第一个记录，底下三个 mbuf形成这个队列的第二个记录。对于一个基于记录的协议，如 UDP，我们在每个队列中能遇到多个记录。但对于像TCP这样的协议，它没有记录的边界，每个队列我们只能发现一个记录 (一个mbuf链可能包含多个mbuf)。

把一个mbuf追加到队列的第一个记录中要遍历所有第一个记录的 mbuf，直到遇到m_next为空的mbuf。而追加一个包含新记录的 mbuf链到这个队列中，要查找所有记录直到遇到m_nextpkt为空的记录。

3) 一个有头指针和尾指针的mbuf链的链表。

图2-22显示的是这种类型的链表。我们在接口队列中会遇到它 (图3-13)，并且在图2-2中已显示过它的一个例子。

在图2-21中仅有一点改变：增加了一个尾指针，来简化增加一个新记录的操作。

4) 双向循环链表。

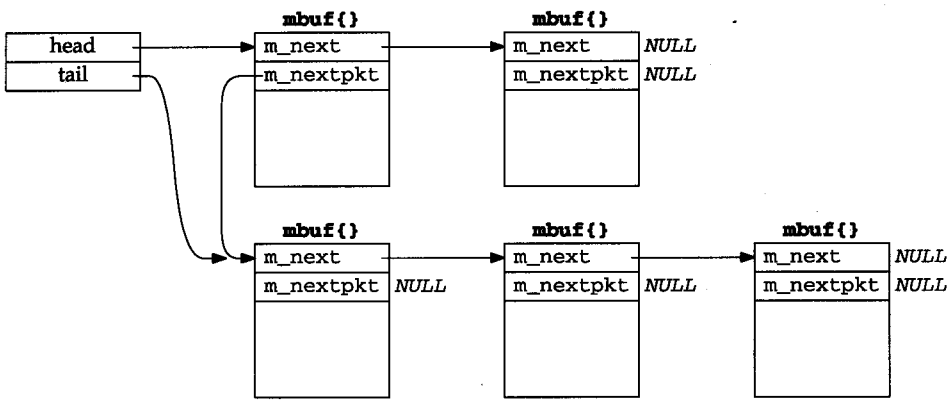


图2-22 有头指针和尾指针的链表

图2-23所示的是这种类型的链表，我们在IP分片与重装(第10章)、协议控制块(第22章)及TCP失序报文段队列(第27.9节)中会遇到这种数据结构。

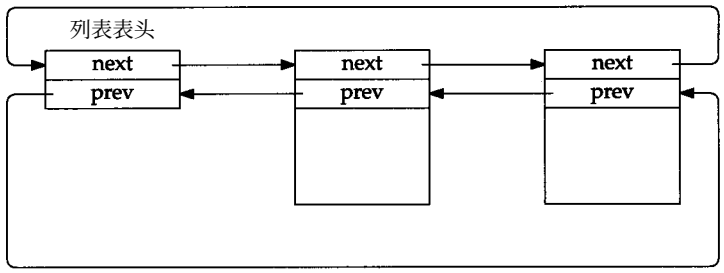


图2-23 双向循环链表

在这个链表中的元素不是mbuf——它们是一些定义了两个相邻的指针的结构：一个next指针跟着一个previous指针。两个指针必须在结构的起始处。如果链表为空，表头的next和previous指针都指向这个表头本身。

在图中我们简单地把向后指针指向另一个向后指针。显然所有的指针应包含它所指向的结构的地址，即向前指针的地址(因为向前和向后指针总是放在结构的起始处)。

这种类型的数据结构能方便地向前向后遍历，并允许方便地在链表中任何位置进行插入与删除。

函数insque和remque(图10-20)被调用来对这个链表进行插入和删除。

2.9 m_copy和簇引用计数

使用簇的一个明显的好处就是在要求包含大量数据时能减少mbuf的数目。例如，如果不使用簇，要有10个mbuf才能包含1024字节的数据：第一个mbuf带有100字节的数据，后面8个每个存放108字节数据，最后一个存放60字节数据。分配并链接10个mbuf比分配一个包含1024字节簇的mbuf开销要大。簇的一个潜在缺点是浪费空间。在我们的例子中使用一个簇(2048 + 128)要2176字节，而1280字节不到1簇(10×128)。

簇的另外一个好处是在多个mbuf间可以共享一个簇。在TCP输出和m_copy函数中我们遇到过这种情况，但现在我们要更详细地说明这个问题。

例如，假设应用程序执行一个write，把4096字节写到TCP插口中。假设插口发送缓存原来是空的，接收窗口至少有4096，则会发生以下操作。插口层把前2048字节的数据放在一个簇中，并且调用协议的发送例程。TCP发送例程把这个mbuf追加到它的发送缓存后，如图2-24所示，并调用tcp_output。结构socket中包含sockbuf结构，这个结构中存储着发送缓存mbuf链的链表的表头：so_snd.sb_mb。

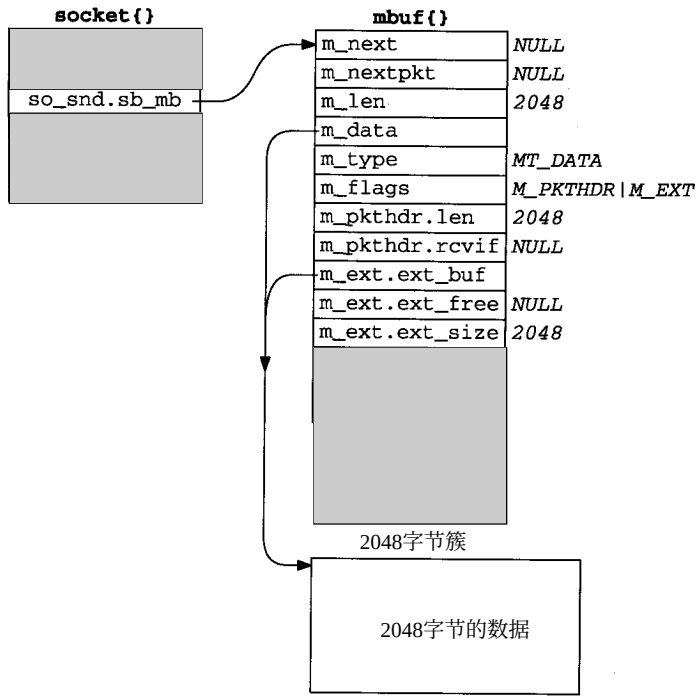


图2-24 包含2048字节数据的TCP插口发送缓存

假设这个连接(典型的是以太网)的一个TCP最大报文段大小(MSS)为1460，tcp_output建立一个报文段来发送包含前1460字节的数据。它还建立一个包含IP和TCP首部的mbuf，为链路层首部(16字节)预留了空间，并将这个mbuf链传给IP输出。在接口输出队列尾部的mbuf链显示在图2-25中。

在1.9节的UDP例子中，UDP用mbuf链来存放数据报，在前面添加一个mbuf来存放协议首部，并把此链传给IP输出。UDP并不把这个mbuf保存在它的发送缓存中。而TCP不能这样做，因为TCP是一个可靠协议，并且它必须维护一个发送数据的副本，直到数据被对方确认。

在这个例子中，tcp_output调用函数m_copy，请求复制1460字节的数据，从发送缓存起始位置开始。但由于数据被存放在一个簇中，m_copy创建一个mbuf(图2-25的右下侧)并且对它初始化，将它指向那个已存在的簇的正确位置(此例中是簇的起始处)。这个mbuf的长度是1460，虽然有另外588字节的数据在簇中。我们所示的这个mbuf链的长度是1514，包括以太网首部、IP首部和TCP首部。

在图2-25的右下侧我们还显示了这个mbuf包含一个分组首部，但它不是链中的第一个mbuf。当m_copy复制一个包含一个分组首部的mbuf并且从原来mbuf的起始地址开始复制时，分组首部也被复制下来。因为这个mbuf不是链中的第一个mbuf，

这个额外的分组首部被忽略。而在这个额外的分组首部中的 m_pkthdr.len的值2048也被忽略。

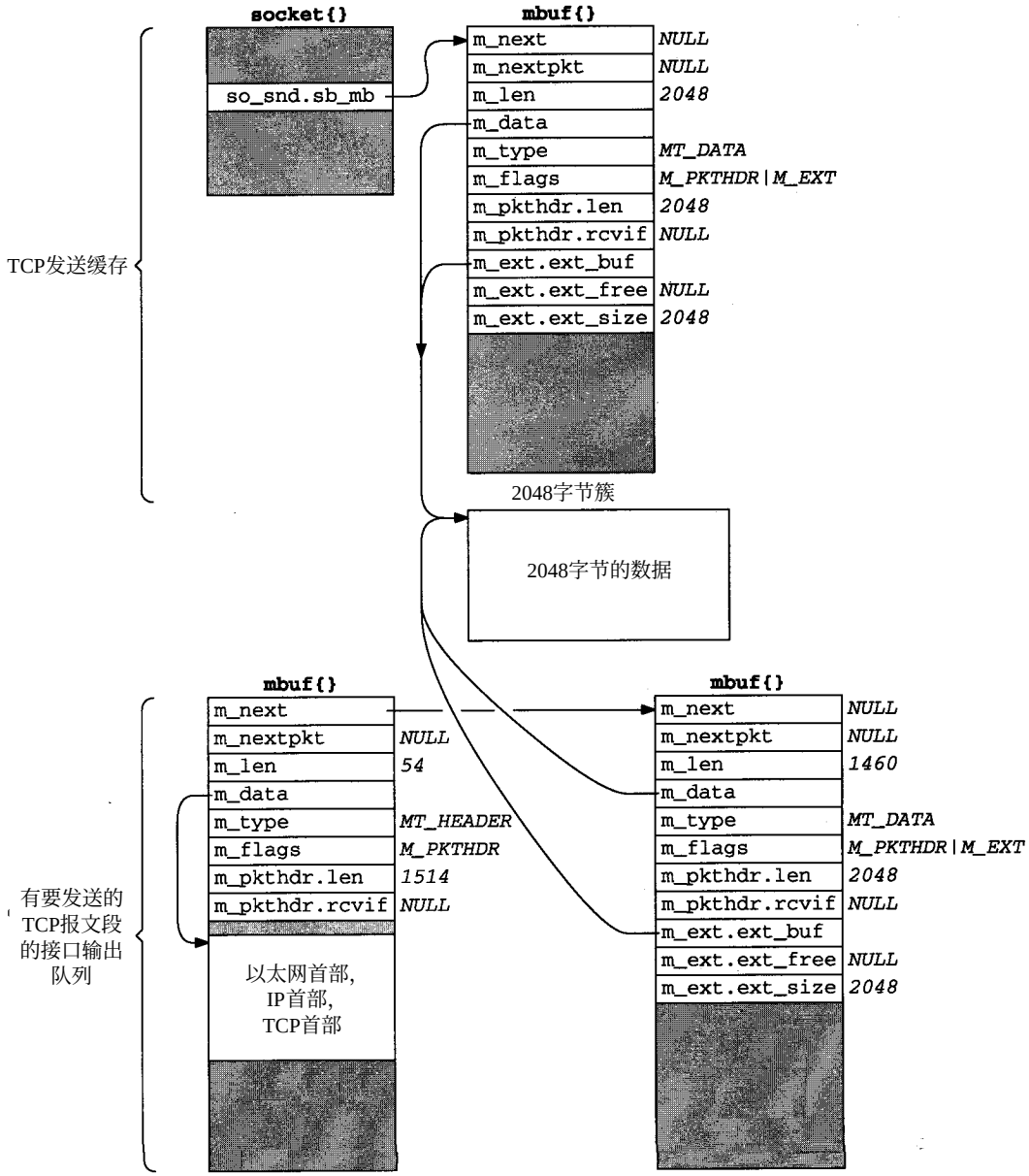


图2-25 TCP插口发送缓存和接口输出队列中的报文段

这个共享的簇避免了内核将数据从一个 mbuf复制到另一个 mbuf中 —— 这节约了很多开销。它是通过为每个簇提供一个引用计数来实现的，每次另一个 mbuf指向这个簇时计数加 1，当一个簇释放时计数减 1。仅当引用计数到达 0时，被这个簇占用的存储器才能被其他程序使用(见习题2.4)。

例如，当图2-25底部的mbuf链到达以太网设备驱动程序并且它的内容已被复制给这个设备时，驱动程序调用 m_freem。这个函数释放带有协议首部的第一个 mbuf，并注意到链中第

二个mbuf指向一个簇。簇引用计数减 1，但由于它的值变成了 1，它仍然保存在存储器中。它不能被释放，因为它仍在TCP发送缓存中。

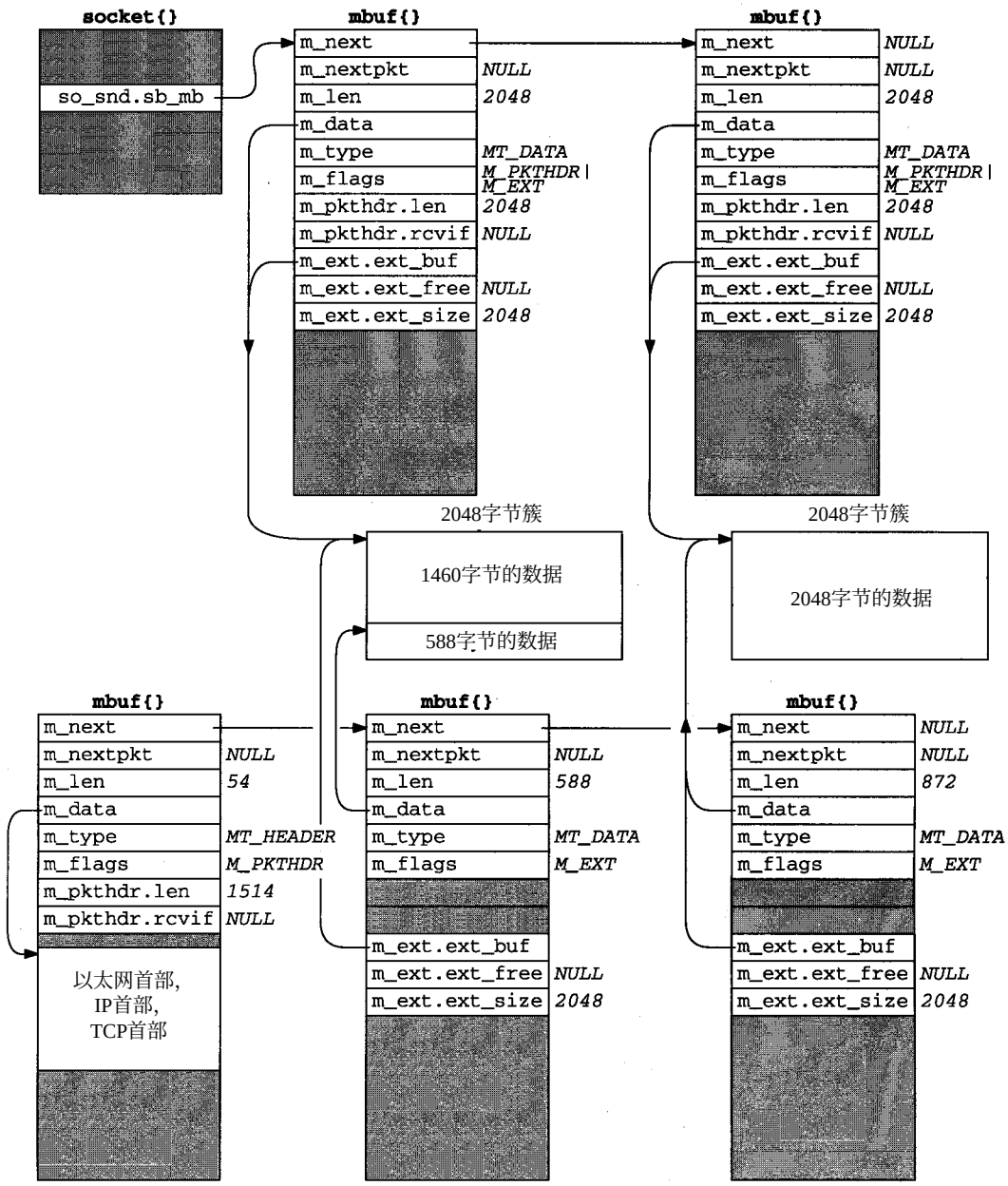


图2-26 用于发送1460字节TCP报文段的mbuf链

继续我们的例子，由于在发送缓存中剩余的 588字节不能组成一个报文段，tcp_output在把1460字节的报文段传给IP后返回(在第26章我们要详细说明在这种条件下 tcp_output发送数据的细节)。插口层继续处理来自应用程序的数据：剩下的 2048字节被存放到一个带有一个簇的mbuf中，TCP发送例程再次被调用，并且新的 mbuf被追加到插口发送缓存中。因为能发送一个完整的报文段，tcp_output建立另一个带有协议首部和1460字节数据的mbuf链表。

m_copy的参数指定了1460字节的数据在发送缓存中的起始位移和长度 (1460字节)。这显示在图2-26中，并假设这个mbuf链在接口输出队列中(这个链中的第一个mbuf的长度反映了以太网首部、IP首部及TCP首部)。

这次1460字节的数据来自两个簇：前 588字节来自发送缓存的第一个簇而后面的 872字节来自发送缓存的第二个簇。它用两个 mbuf来存放1460字节，但m_copy还是不复制这1460字节的数据 —— 它引用已存在的簇。

这次我们没有在图 2-26右下侧的任何 mbuf中显示一个分组首部。原因是调用m_copy的起始位移为零。但在插口发送缓存中的第二个 mbuf包含一个分组首部，而不是链中的第一个mbuf。这是函数sosend的特点，这个额外的分组首部被简单地忽略了。

我们在通篇中会多次遇到函数m_copy。虽然这个名字隐含着对数据进行物理复制，但如果数据被包含在一个簇中，却是仅引用这个簇而不是复制。

2.10 其他选择

mbuf远非完美，并且时常遭到批评。但不管怎样，它们形成了所有今天正使用着的伯克利联网代码的基础。

一种由Van Jacobson [Partridge 1993]完成的Internet协议的研究实现，它废除了支持大量连续缓存的复杂的mbuf数据结构。[Jacobson 1993]提出了一种速度能提高一到两个数量级的改进方案，还包括其他改进，及废除mbuf。

这个mbuf的复杂性是一种权衡，以避免分配大的固定长度的缓存，这样的大缓存很少能被装满。而在这种情况下， mbuf要进行设计，一个VAX-11/780有4兆存储器，是一个大系统，并且存储器是昂贵的资源，需要仔细分配。今天存储器已不昂贵了，而焦点已经转向更高的性能和代码的简单性。

mbuf的性能基于存放在mbuf中数据量。[Hutchinson and Peterson 1991]显示了处理mbuf的时间与数据量不是线性关系。

2.11 小结

在本书几乎所有的函数中我们都会遇到 mbuf。它们的主要用途是在进程和网络接口之间传递用户数据时用来存放用户数据，但 mbuf还用于保存其他各种数据：源地址和目标地址、插口选项等等。

根据M_PKTHDR和M_EXT标志是否被设置，这里有4种类型的mbuf：

- 无分组首部， mbuf本身带有0~108字节数据；
- 有分组首部， mbuf本身带有0~100字节数据；
- 无分组首部，数据在簇(外部缓存)中；
- 有分组首部，数据在簇(外部缓存)中。

我们查看了几个mbuf宏和函数的源代码，但不是所有的mbuf例程源代码。图2-19和图2-20提供了所有我们在本书中遇到的mbuf例程的函数原型和说明。

查看了我们要遇到的两个函数的操作： m_devget，很多网络设备驱动程序调用它来存

储一个收到的帧；m_pullup，所有输入例程调用它把协议首部连续放置在一个mbuf中。

由一个mbuf指向的簇(外部缓存)能通过m_copy被共享。例如，用于TCP输出，因为一个被传输的数据的副本要被发送端保存，直到数据被对方确认。比起进行物理复制来说，通过引用计数，共享簇提高了性能。

习题

2.1 在图2-9中定义了M_COPYFLAGS。为什么不复制标志M_EXT？

2.2 在2.6节中，我们列出了两个m_pullup失败的原因。实际上有三个原因。查看这个函数的源代码(附录B)，并发现另外一个原因。

2.3 为避免宏dtom遇到在2.6节中我们所讨论的问题——当数据在簇中时，为什么不仅仅给每个簇加一个指向mbuf的回指指针？

2.4 既然一个mbuf簇的大小是2的幂(典型的是1024或2048)，簇内的空间不能用于引用计数。查看Net/3的源代码(附录B)，并确定这些引用计数存储在什么地方。

2.5 在图2-5中，我们注意到两个计数器m_drops和m_wait现在没有实现。修改mbuf例程增加这些计数器。