第1章 系统概貌

UNIX系统自从1969年问世以来已经变得相当流行，它运行在从微处理机到大型机的具有不同处理能力的机器上，并在这些机器上提供公共的执行环境。UNIX系统可分成两部分，第一部分由一些程序和服务组成，其中包括shell程序、邮件程序、正文处理程序包以及源代码控制系统等，正是这些程序和服务使得UNIX系统环境如此受欢迎，它们是用户立即可见的部分。第二部分由支持这些程序和服务的操作系统组成。本书给出了该操作系统的详细描述，它着重描述由美国电话电报公司（AT&T）生产的UNIX系统Ⅴ，但也考虑了其他版本所提供的颇有意义的特征。它考察了在该操作系统中使用的主要数据结构和算法，而这些数据结构和算法最终向用户提供了标准用户界面。

本章是UNIX系统的引言，它回顾了UNIX系统的历史并勾画出了整个系统结构的轮廓。下一章将对该操作系统做更详细的介绍。

1.1 历史

1965年，贝尔电话实验室和通用电气公司及麻省理工学院的MAC课题组一起联合开发一个被称为Multics[Organick 72]的新操作系统。Multics系统的目标是要向大的用户团体提供对计算机的同时访问，支持强大的计算能力与数据存储，以及允许用户在需要的时候容易地共享他们的数据。贝尔实验室中后来参加UNIX系统早期开发的许多人当时都参加了Multics工作。虽然Multics系统的原始版本于1969年在GE645计算机上运行了，但它既没能提供预定的综合计算服务，而且，连它自己也不清楚究竟什么时刻算达到开发目标了。结果，贝尔实验室退出了这一项目。

在他们结束了Multics工程上的工作的时候，贝尔实验室计算科学研究中心的成员们处于缺乏“方便的交互式计算服务”的境况之中[Ritchie 84a]。为了改善他们的程序设计环境，Ken Thompson、Dennis Ritchie及其他人勾画出了一个纸面上的文件系统设计——它后来就演化为UNIX文件系统的早期版本。Thompson编写了若干程序，模拟所建议的文件系统行为，以及模拟在请求调页环境下程序的行为。他甚至为GE645计算机的简单内核进行了编码。与此同时，他用Fortran语言为GECOS系统（Honeywell635）编写了名为“宇宙旅行”的游戏程序。但这个程序是不能令人满意的，因为它很难控制“宇宙飞船”，并且该程序运行开销太大。Thompson后来发现了一个几乎无人问津的PDP-7计算机能提供很好的图形显示和廉价的执行开销。为PDP-7开发“宇宙旅行”程序使Thompson学到了关于该机器的细节，但是它的程序开发环境要求先在GECOS机上进行程序的交叉汇编，而后把纸带带到PDP-7上输入。为了创建一个较好的开发环境，Thompson和Ritchie在PDP-7上实现了他们的系统设计，其中包括UNIX文件系统、进程子系统的早期版本及少量实用程序。终于，新系统再也不需要把GECOS系统作为开发环境，而能够自己支持自己了。这个新的系统被命名为UNIX。UNIX是针对Multics的双关语，它是计算科学研究中心的另一名成员Brian Kernighan想出来的。

虽然 UNIX 系统的早期版本是大有前途的，但直到它用于实际项目之前它并没能发挥出它的潜力。因此，为给贝尔实验室的专利部门提供一个正文处理系统，1971 年 UNIX 系统被移植到 PDP-11 上。该系统的特征是它的规模小：内存中 16K 字节用于系统，8K 字节用于用户程序；磁盘 512K 字节，每个文件限定长度为 64K 字节。在它初次成功之后，Thompson 开始动手为这个系统实现 Fortran 编译程序。但是在BCPL[Richards 69]的影响下开发出来的却是 B 语言。B 语言是解释性语言，在性能上有所退步——这是这类语言的共同特征。因此，Ritchie 把 B 发展成他称之为 C 的语言，C 语言允许产生机器代码、说明数据类型及定义数据结构。1973 年，用 C 语言重写了 UNIX 操作系统。这一步在当时并不太引人注目，但对其外部用户接受它却具有极大的影响。这之后，贝尔实验室的装机数目增加到 25 个，并且形成了一个 UNIX 系统小组，以提供内部支持。

由于美国电话电报公司 1965 年与联邦政府签署了反垄断法，所以这时它不能销售计算机产品。但是，美国电话电报公司把 UNIX 系统提供给了请求把 UNIX 用于教育目的的大学。该公司信守了反垄断法的条款，它既没有为 UNIX 系统做广告，也没有销售和支持 UNIX 系统。然而，UNIX 系统的声望却在稳步增长。1974 年，Thompson 与 Ritchie 在《ACM 通讯》上发表了一篇描述 UNIX 系统的文章[Thompson 74]，进一步促进了它的可接受性。到 1977 年，UNIX 系统的装机数目已经增长到大约 500 个，其中有 125 个在大学。UNIX 系统开始在业务电话公司流行起来，为程序开发、网络事务操作服务及实时服务（通过 MERT[Lycklama 78a]）提供了良好的环境。这时，UNIX 系统的许可证被提供给商业机构，同时也向大学提供。1977 年，交互系统公司（Interactive Systems Corporation）成了 UNIX系统的第一个增值转卖商（VAR）[⊖]，他们增强了它，使之在办公室自动化环境中使用。同样，1977 年也是标志 UNIX 系统首次被“移植”到非 PDP 机（即稍加改变或完全不变而在其他机种上运行）——Interdata 8/32 上的一年。

随着微处理机的日益普及，其他公司也把 UNIX 系统移植到新的机器上，但是它的简单清晰的特点吸引着很多开发者以他们自己的方式增强 UNIX 系统，结果在基本系统上产生若干变体。从 1977 年到 1982 年这一时期，贝尔实验室把若干 AT&T 变体综合成一个单个系统，这就是大家都知道的商用 UNIX 系统Ⅲ。随后，贝尔实验室又把若干特征加到系统Ⅲ上，称为新产品 UNIX 系统Ⅴ[⊖]，1983 年 1 月，美国电话电报公司发布它对系统Ⅴ的正式支持。然而，加利福尼亚大学伯克利分校的人们已经开发了一个 UNIX 系统的变体，它的最新版本称为 4.3BSD（Berkeley Software Distribution），是配在 VAX 机上的，它提供了一些新的有意义的特征。本书将着重描述 UNIX 系统Ⅴ，但也偶尔谈及 BSD 系统中所提供的那些特征。

到 1984 年初，世界上大约安装了 100 000 个 UNIX 系统，它们运行在从微处理机到大型机的具有宽广范围的计算能力的机器上，运行在出自不同的制造厂家的生产线的机器上。没有任何其他操作系统能与之匹敌。UNIX 系统的普及与成功可归结为如下一些原因：

- 该系统以高级语言书写，使之易读、易懂、易修改、易移植到其他机器上。据

⊖ 增值转卖商把具体应用加到计算机系统上以满足特定的市场需要。他们销售的是应用而不是销售这些应用赖以运行的操作系统。

⊖ 对系统Ⅳ发生了什么？事实上，系统Ⅳ是 UNIX 系统的一个内部版本，它被融化到系统Ⅴ中去了。

Ritchie估计，用C语言书写的第一个系统与用汇编语言书写的系统相比，要大而且慢20—40%。但是，采用高级语言的优点要远远超过它的缺点（见[Ritchie 786]第1965页）。

- 它有一个简单的用户界面但具有提供用户所希望的服务的能力。
- 它提供能够由较简单的程序构造出复杂程序的原语。
- 它使用了在维护上是容易的、在实现上是高效的层次式文件系统。
- 文件采用字节流这样的一致格式，使应用程序易于书写。
- 它为外围设备提供了简单一致的接口。
- 它是一个多用户、多进程系统，每个用户都能同时执行几个进程。
- 它向用户隐蔽了机器的体系结构，使用户易于书写在不同硬件实现上运行的程序。

简单性与一致性突出了UNIX系统的宗旨，上面列出的大部分原因都讲的是简单性与一致性。

虽然操作系统和很多命令程序是用C语言书写的，但是UNIX系统支持其他语言，包括Fortran、Basic、Pascal、Ada、Cobol、Lisp及Prolog等。UNIX系统能支持具有编译程序或解释程序的任何语言；UNIX系统还能支持一个系统接口，该接口把用户对操作系统服务的请求映射到UNIX系统使用的一组标准请求上。

1.2 系统结构

图1-1绘出了UNIX系统的高层次的体系结构。该图中心的硬件部分向操作系统提供将在1.5节中描述的基本服务。操作系统直接[⊖]与硬件交互，向程序提供公共服务，并使它们同硬件特性隔离。当我们把整个系统看成层的集合时，操作系统通常称为系统内核，或简称内核（kernel），此时强调的是它同用户程序的隔离。因为程序是不依赖于其下面的硬件的，所以，如果程序对硬件没有做什么假定的话，就容易把它们在不同硬件上运行的UNIX系统之间搬动。比如，那些假定了机器字长的程序比起没假定机器字长的程序来就较难于搬到其他机器上。

外层的程序，诸如shell及编辑程序（ed与vi），是通过引用一组明确定义的系统调用而与内核交互的。这些系统调用通知内核为调用程序做各种操作，并在内核与调用程序之间交换数据。图中出现的一些程序属于标准的系统配置，就是大家知道的命令，但是，由名为a.out的程序所指示的用户私用程序也可以存在于这一层。此处的a.out是被C编译程序产生的可执行文件的标准名字。其他应用程序能在较低层的程序之上构建，因此它们存在于本图的最外层。比如，标准的C编译程序cc就处在本图的最外层：它调用C预处理程序、两遍编译程序、汇编程序及装入程序（称为连接-编辑程序），这些都是彼此分开的低层程序。虽然该图对应用程序只描绘了两个级别的层次，但用户能够对层次进行扩充，直到级别的数目适合于自己的需要。确实，为UNIX系统所偏爱的程序设计风格鼓励把现存程序组合起来去完成一个任务。

⊖ 在UNIX系统的某些实现中，该操作系统同一个核（native operating system）接口，再由核与其下面的硬件接口，并向UNIX系统提供必要的服务。这样的配置可以使其他操作系统及其应用程序能够同UNIX系统平行地运行在装置上。这样的一种配置的经典例子是MERT系统[Lycklama 78a]。更近期的配置包括面向IBM系统/370计算机[Felton 84]及面向UNIVAC1100系列计算机[BodenStab 84]的实现。

一大批提供了对系统的高级看法的应用子系统及应用程序，诸如 shell、编辑程序、SCCS（source code control system）及文档准备程序包等，都逐渐变成了“UNIX 系统”这一名称的同义语。然而，它们最终都使用由内核提供的低层服务，并通过系统调用的集合来利用这些服务。系统Ⅴ中大约有 64 个系统调用，其中将近 32 个是常用的。它们有简单的可选项，这些可选项使系统调用容易使用，但是却向用户提供了很多能力。系统调用的集合及实现系统调用的内部算法形成了内核的主体，因而本书所要介绍的对 UNIX 操作系统的研究，就化为对系统调用及其相互作用的详细研究和分析。简言之，内核提供了 UNIX 系统全部应用程序所依赖的服务，它也定义了这些服务。本书将频繁使用“UNIX 系统”、“内核”或“系统”等术语，但其含义是指 UNIX 操作系统的内核，并且在上下文中是清楚的。

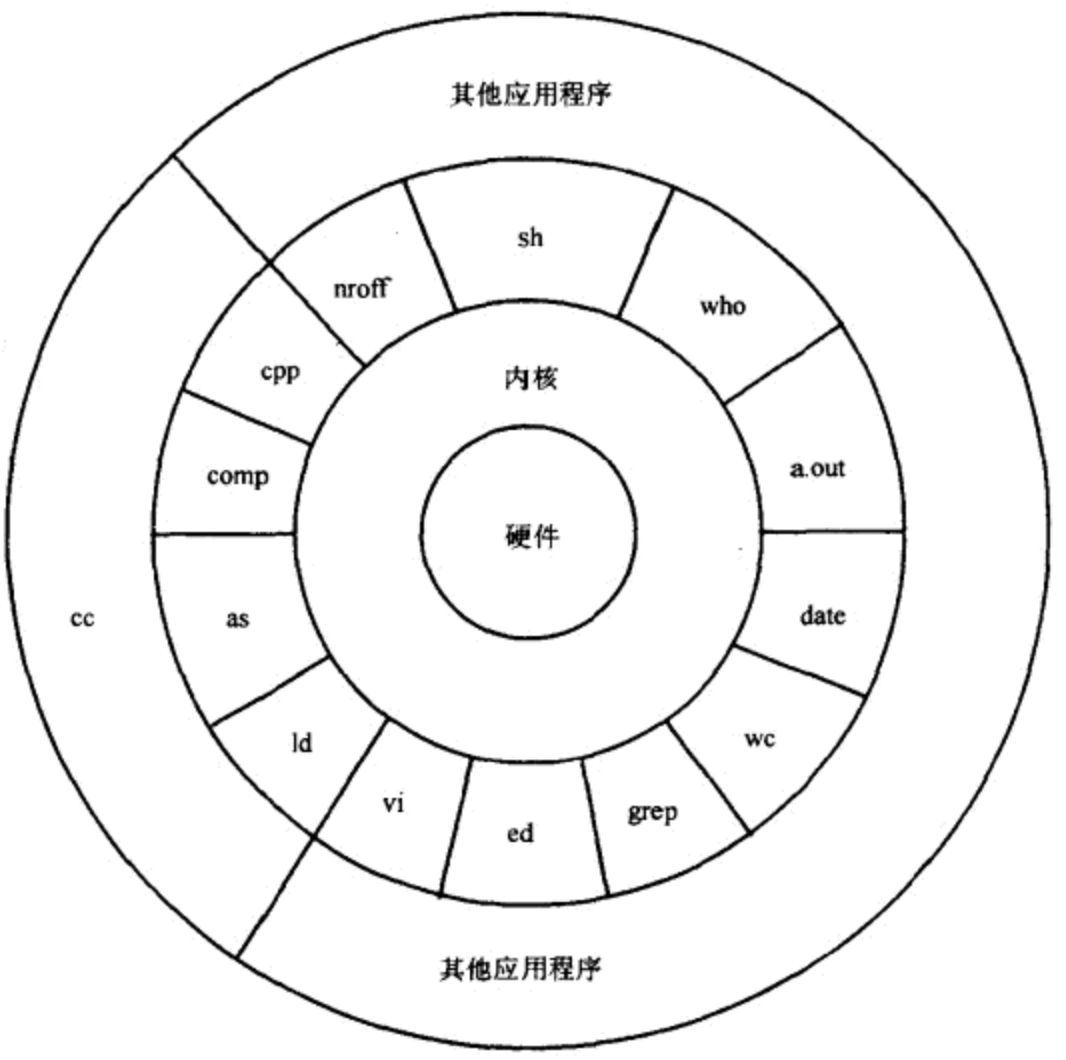


图 1-1　UNIX 系统的体系结构

1.3　用户看法

本节简要地考察 UNIX 系统的高层特征，诸如文件系统、处理环境及构件原语（比如，管道），后面几章将详细地探讨内核对这些特征的支持。

1.3.1　文件系统

UNIX 文件系统有如下特点：

- 层次结构。
- 对文件数据的一致的处理。

- 建立与删除文件的能力。
- 文件的动态增长。
- 文件数据的保护。
- 把外围设备（如终端及磁带装置）作为文件对待。

文件系统被组织成树状，树有一个称为根（root）的节点（记作“/”）。文件系统结构中的每个非树叶节点都是文件的一个目录（directory），树的叶节点上的文件既可以是目录，也可以是正规文件（regular files），还可以是特殊设备文件（special device files）。文件名由路径名（path name）给出，路径名描述了怎样在一个文件系统树中确定一个文件的位置。路径名是一个分量名序列，各分量名之间用斜杠符隔开。分量是一个字符序列，它指明一个被唯一地包含在前级（目录）分量中的文件名。一个完整的路径名由一个斜杠字符开始，并且指明一个文件，这个文件可以从文件系统的根开始，沿着该路径名的后继分量名所在的那个分支游历文件树而找到。因此，路径名“/etc/passwd”，“/bin/who”及“/usr/src/cmd/who.c”都是图 1-2 的树中的文件，但“/bin/passwd”及“/usr/src/date.c”则不是。一个路径名不一定非从根开始不可，可以省略掉路径名中的初始斜杠，由相对于正在执行的进程的当前目录来指明。因此，若从目录“/dev”开始，路径名“tty01”标明的是整个路径名为“/dev/tty01”的文件。

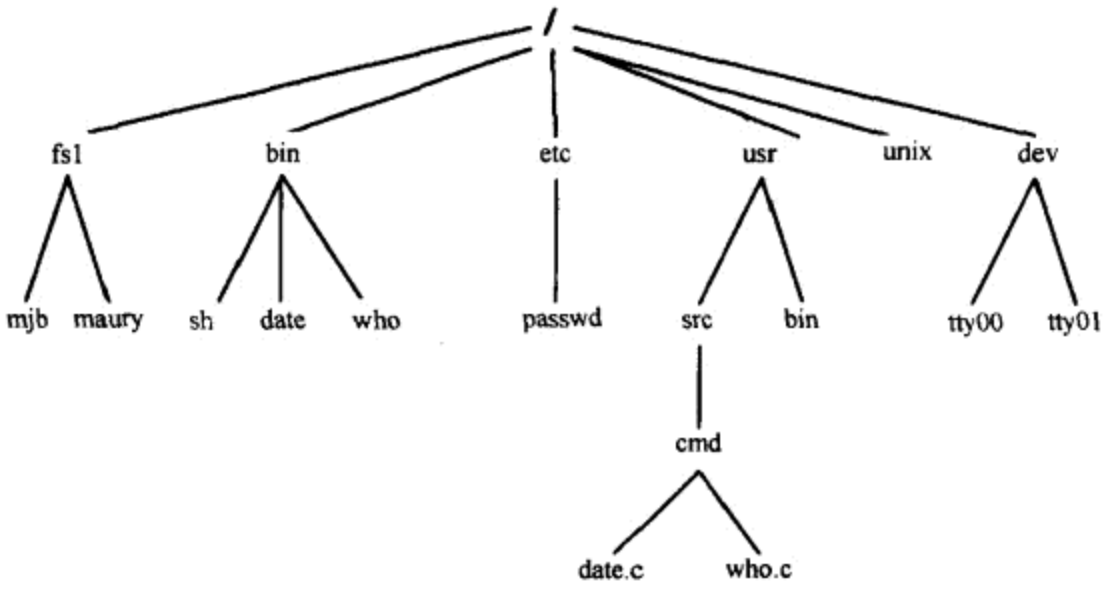


图 1-2 文件系统树示例

在 UNIX 系统中，程序不了解核心按怎样的内部格式存储文件，而把数据作为无格式的字节流看待。程序可以按它们自己的意愿去解释字节流，但这种解释与操作系统如何存储数据无关。因此，对文件中数据进行存取的语法是由系统定义的，并且对所有的程序都是同样的。但是，数据的语义是由程序加上去的。比如，文本格式化程序 troff 希望在文本的每一行的尾部找到“换行”符，而系统记帐程序 acctcom 则希望找到定长记录。两个程序都使用相同的系统服务，以存取文件中的作为字节流存在的数据，而在内部，它们通过分析把字节流解释成适当的格式。如果哪一个程序发现格式是错误的，则它负责采取适当的动作。

从这方面说，目录也像正规文件。系统把目录中的数据作为字节流看待。但是，由于该数据中包含着许多以预定格式记录的目录中的文件名，所以，操作系统以及诸如 ls（列出文件名和属性）这样的程序就能够在目录中发现文件。

对一个文件的存取许可权由与该文件相联系的 access permissions 所控制。存取许可权能够分别对文件所有者、同组用户及其他人这三类用户独立地建立存取许可权，以控制读、写及执行的许可权。如果目录存取许可权允许的话，则用户可以创建文件。新创建的文件是文件系统目录结构的树叶节点。

对于用户来说，UNIX 系统把设备看成文件。以特殊设备文件标明的设备，占据着文件系统目录结构中的节点位置。程序存取正规文件时使用什么语法，它们在存取设备时也使用什么语法。读、写设备的语义在很大程度上与读、写正规文件时相同。设备保护方式与正规文件的保护方式相同：都是通过适当建立它们的（文件）存取许可权实现的。由于设备名看起来像正规文件名，并且对于设备和正规文件能执行相同的操作，所以大多数程序在其内部不必知道它们所操纵的文件的类型。

例如，考察图 1-3 中为一个现存的文件创建新拷贝的程序。假设该程序的可执行版本的名字是 copy。终端上的用户通过敲入：

```
copy oldfile newfile
```

调用该程序。此处，oldfile 是一个现存文件名，而 newfile 是一个新文件名。系统引用 main 时需要提供 argc 作为表 argv 中的参数个数，并且对数组 argv 的每个成员赋初值，以指向由用户提供的参数。在上述例子中，argc 为 3，argv[0]指向字符串 copy（按惯例，第 0 个参数是程序名），argv[1]指向字符串 oldfile，argv[2]指向字符串 newfile。然后该程序检查它被调用时调用者提供的参数个数是否正确。如果正确，则调用系统调用 open，试图打开文件 oldfile，并对该文件做“只读”操作。如果该系统调用成功，则进一步调用系统调用 creat，以便创建 newfile。新创建的文件的存取权限将是 0666（八进制），即允许所有的用户读写该文件。所有的系统调用在失败时都返回 -1。如果 open 或 creat 调用失败，则该程序打印出一则消息，并以返回状态值“1”调用系统调用 exit，结束程序的执行并指出有些地方出错了。

系统调用 open 与 creat 返回一个称为文件描述符（file descriptor）的整数，程序随后就使用这一文件描述符访问该文件。接着，程序调用子程序 copy，copy 进入一个循环：调用系统调用 read，从现存文件中读满一缓冲区的字符，并调用系统调用 write，把这些数据写到新文件上。系统调用 read 返回所读的字节数，当它到达文件尾时返回 0。该程序在遇到文件尾时结束循环，或者在系统调用 read 出了什么错误时（它不检查写错误）结束循环。然后，它从 copy 返回并以返回状态值“0”调用系统调用 exit，以指示该程序成功地结束了。

该程序可以拷贝作为参数提供给它的任何文件——只要它对现存文件具有打开许可权以及对新文件具有建立许可权。此处所说的文件可以是一个可打印字符的文件，比如程序的源代码；或者，它包含不可打印的字符，甚至该程序本身。因此，两个调用

```
copy copy.c newcopy.c
copy copy newcopy
```

都工作。老文件也可以是一个目录，比如

```
copy.dircontents
```

```
#include <fcntl.h>
char buffer[2048];
int version = 1;        /* 第2章对此做了解释 */

main(argc,argv)
        int argc;
        char *argv[];
{
        int fdold, fdnew;

        if(argc ! = 3)
        {
                printf( "need 2 arguments for copy program \n ");
                exit(1);
        }
        fdold = open(argv[1],O_RDONLY);  /* 打开源文件只读 */
        if (fdold == -1)
        {
                printf( "cannot open file % s \n ",argv[1]);
                exit(1);
        }
        fdnew = creat(argv[2],0666);  /* 创建可为所有用户读写的目标文件 */
        if (fdnew = = -1)
        {
                printf( "cannot create file % s \n ",argv[2]);
                exit(1);
        }
        copy(fdold,fdnew);
        exit(0);
}

copy(old,new)
        int old,new;
{
        int count;

        while ((count = read(old,buffer,sizeof(buffer)))>0)
                write(new,buffer,count);
}
```

图 1-3 用于拷贝文件的程序

就把由名字“.”所表示的当前目录的内容拷贝到正规文件“dircontents”中；新文件中的数据与目录的内容是一个字节、一个字节地相同的，但该文件是一个正规文件（而系统调用mkond创建一个新目录）。最后我们要说的是，任一文件都可能是设备特殊文件。比如

```
copy /dev/tty terminalread
```

把在终端上键入的字符（特殊文件/dev/tty 是用户终端）读入，并把它们拷贝到文件 terminalread 上，仅当用户敲进字符 control-d 时才结束。类似地，

```
copy /dev/tty /dev/tty
```

则把终端上敲进的字符读入并把它们都拷贝回去。

1.3.2　处理环境

一个程序是一个可执行文件，而一个进程则是一个执行中的程序的实例。在 UNIX 系统上可以同时执行多个进程（这一特征有时称为多道程序设计或多道任务设计），对进程数目无逻辑上的限制，并且系统中可以同时存在一个程序（例如 copy）的多个实例。各种系统调用允许进程创建新进程、终止进程、对进程执行的阶段进行同步及控制对各种事件的反应。在进程使用系统调用的条件下，进程便互相独立地执行了。

比如，正在执行图 1-4 中的程序的进程执行系统调用 fork 以创建一个新进程。称为子（child）进程的新进程从 fork 那儿获得一个 0 返回值，并且调用 execl 以执行 copy 程序（图 1-3 中的程序）。系统调用 execl 用文件“copy”覆盖子进程的地址空间——当然，我们假设“copy”正处在当前目录中，并且，execl 根据用户提供的参数运行该程序。如果 execl 调用成功了，则它永不返回，这是因为该进程是在新地址空间中执行的，第 7 章我们将会看到这点。同时，调用 fork 的那个进程（父进程）从该调用收到一个非 0 返回值，然后调用 wait 将自己的执行挂起，直到 copy 结束时，打印出“拷贝完毕”的消息，而后退出（每个程序都是在主函数结束时退出，这是在编译处理期间与标准 C 程序库连接时由 C 程序库所安排的）。比如，若可执行程序的名字为 run，并且一个用户通过

```
run oldfile newfile
```

引用该程序的话，则进程把“oldfile”拷贝到“newfile”上并且打印出消息。虽然这个程序只往“copy”程序上添加了一点东西，但它呈现了用于进程控制的四个主要的系统调用：fork，exec，wait 及 exit。

```
main(argc,argv)
    int argc;
    char * argv[];
{
    /* 假设两个参数：源文件和目标文件 */
    if (fork() == 0)
        execl( "copy", "copy",argv[1],argv[2],0);
    wait((int *)0);
    printf( "copy done \n");
}
```

图 1-4　创建新进程以拷贝文件的程序

一般说来，系统调用允许用户书写做复杂操作的程序，其结果是：在其他系统中成为

“核心”部分的许多函数却不包含在UNIX系统的内核中。包括像编译程序和编辑程序这样的函数在UNIX系统中都是用户级程序。命令解释程序shell就是这样的程序的一个基本例子，在用户注册到系统中之后就执行该程序。shell把命令行的第一个字解释成命令名：对许多命令，shell都创建子进程，由子进程执行与该名字相联系的命令。把命令行中其余的字视为该命令的参数。

shell允许三种类型的命令。第一种，一个命令是一个可执行文件，它包含将源代码（比如说一个C程序）编译后产生的目标代码。第二种，一个命令是包含一系列shell命令行的可执行文件。最后，一个命令是一个内部shell命令（不是一个可执行文件）。内部命令使shell不仅是命令解释程序，而且是一种程序设计语言，它包括用于循环的命令（for-in-do-done与while-do-done），用于条件执行的命令（if-then-else-fi），一个“选择”（case）语句命令，一个改变进程的当前目录的命令（cd）及其他一些命令。shell语法允许模式匹配和参数处理。用户在执行命令时不一定非要知道它们的类型不可。

shell在一个给定的目录序列中搜索命令，而目录序列可以在每次调用shell时由用户请求来改变。通常shell是同步地执行一个命令的，要等候一个命令执行完毕再去读下一个命令行。然而，它也允许异步执行，这时，它不等待前一个命令执行完毕就去读下一个命令行。异步执行的命令被说成是在后台执行的。比如，键入命令

```
who
```

引起系统执行存储在文件/bin/who[⊖]中的程序，它打印当前在系统中注了册的用户名单。当who执行时，shell等候它结束，然后提示用户键入下一个命令。然而键入：

```
who &
```

系统则在后台执行程序who，并且shell做好了立即接收下一条命令的准备。

正在UNIX系统中执行着的每个进程都有一个包括当前目录在内的执行环境。一个进程的当前目录是不以斜杠字符开始的所有路径名的起始目录。用户可以执行shell命令cd，即改变目录，以便沿着文件系统树移动，并改变当前目录。命令行

```
cd  /usr/src/uts
```

把shell的当前目录变成目录“/usr/src/uts”。命令行

```
cd  ../..
```

把shell的当前目录移到向根结点“靠近”两层结点的目录：命令中的分量“..”指的是当前目录的父目录。

因为shell是用户程序而不是内核的一部分，所以易于修改它、剪裁它以适应特定环境的需要。举例来说，用户能使用C shell，而不是Bourne shell（以它的开发者Steve Bourne命名的），C shell提供一种历史（history）机制，以避免重新键入最近刚使用过的命令。这种

⊖ 目录“/bin”包含很多有用的命令，并且通常被包含在shell搜索的目录序列中。

机制是标准的系统 V 版本的一部分。或者，某些用户可以仅被允许使用一个有限的 shell——正规 shell 的简化版本。这就是说，系统能同时执行各种 shell。用户具有同时执行多个进程的能力，并且如果需要的话，进程能动态创建其他进程，并对它们的执行进行同步。这些特征向用户提供了一个强有力的执行环境。虽然 shell 的很多能力是从它作为一种程序设计语言的能力及从它对自变量的模式匹配能力引伸出来的，但本节着重叙述系统通过 shell 提供的进程环境。shell 的其他重要特征已超出了本书的范围（见[Bourne 78]中关于 shell 的详细描述）。

1.3.3　构件原语

正如前面所描述的那样，UNIX 系统的宗旨是提供操作系统原语，使用户能书写小的、模块化的程序，并把它们作为构件（building block）去构筑更复杂的程序。重定向 I/O（redirect I/O）的能力便是这样的为 shell 用户可见的一个原语。进程通常可以存取三个文件：它们从标准输入（standard input）文件上读，往标准输出（standard output）文件上写，以及把错误消息写到标准错误（standard error）文件上。在终端上执行的进程一般地是使用终端作为这三个文件，但每个文件都可以被独立地重定向。比如，命令行

```
ls
```

把当前目录中的所有文件都列到标准输出上。但是，命令行

```
ls>output
```

使用上面提到的系统调用 creat 把标准输出重定向到当前目录中称为“output”的文件上。类似地，命令行

```
mail mjb<letter
```

打开文件“letter”作为它的标准输入，并且把该文件的内容作为邮件寄给注册名为“mjb”的用户。进程能同时进行输入重定向与输出重定向，比如，在命令行

```
nroff - mm<doc1>doc1.out 2>errors
```

中，文本格式化程序 nroff 读输入文件 doc1，把它的标准输出重定向到文件 doc1.out 上，并且把错误消息重定向到文件 errors 上（记号“2>”意味着对文件描述符 2 进行输出重定向。按惯例，文件描述符 2 为标准错误文件）。程序 ls，mail 及 nroff 不知道它们的标准输入、标准输出或标准错误将是哪一个文件；shell 识别出“<”，“>”及“2>”等符号，并在执行这些进程之前适当地建立标准输入、标准输出及标准错误文件。

第二个构件原语是管道（pipe），这是允许在读者进程与写者进程之间传递数据流的机制。进程能够把它们的标准输出重新定向到一个管道上，另外一些进程把它们的标准输入重新定向为从这个管道来，即读这个管道。第一类进程写进管道的数据是第二类进程的输入。第二类进程也能对它们的输出重定向，等等，这就要看程序设计的需要了。还有，进程不需要知道它们的标准输出是什么类型的文件；不管它们的标准输出是正规文件，还是管道，还是设备，它们都能工作。当使用较小的程序作为构件组成较大的、较为复杂的程序时，程序

员使用管道原语及I/O重定向把几部分断片集成起来。不言而喻，系统确实鼓励这样的程序设计风格，以便使新的程序借助于现存的程序就能工作。

比如，程序grep搜索一组文件（作为grep参数）中的一个给定的模式：

```
grep main a.c b.c c.c
```

在a.c，b.c，c.c中搜索包含字符串“main”的行，把找到的那些行在标准输出上打印出来。下面就是一个输出实例：

```
a.c:main(argc,argv)
b.c:/* here is the main loop in the program */
c.c:main()
```

带有任选项“-1”的程序wc对标准输入文件中的行数进行计数。命令行

```
grep main a.c b.c c.c|wc-1
```

对上述文件中包含有字符串“main”的行进行计数；来自grep的输出被直接引送到wc命令中。对于前面的grep的输出实例而言，上面的管道化命令的输出是：

```
3
```

管道的应用常使得它不需要创建临时文件。

1.4 操作系统服务

图1-1描绘了紧挨在用户应用程序之下的内核层。内核代表用户进程完成各种原语操作，以便支持上面描述过的用户接口。由内核提供的服务有：

• 通过允许进程创建、终止、挂起及通信来控制进程的执行。

• 对进程在CPU上的执行进行公平调度。各个进程以分时方式共享CPU：CPU[⊖]执行一个进程，当它的时间片用完时，内核把它挂起并且调度另一个进程执行。以后，内核再次调度到被挂的进程。

• 对正在执行的进程分配主存。内核允许诸进程在一定条件下共享它们的部分地址空间，但要保护进程的私用地址空间免受外来的破坏。如果系统运行时空闲存储区不够了，则内核把一个进程临时写到二级存储器（称为对换设备（swap device））上来释放存储区。如果内核把整个进程都写到对换设备上，这种UNIX系统的实现称为对换系统（swapping system）；如果它把存储器的某些页写到对换设备上，则它被称为请求调页系统（paging system）。

• 为实现用户数据的有效存储和检索而分配二级存储。这一服务构成文件系统。内核为用户文件分配二级存储，回收不使用的存储区，以易于理解的方式构造文件系统，并且保护用户文件不被非法存取。

• 允许进程对诸如终端、磁带机、磁盘机以及网络设备等进行有控制的存取。

⊖ 第12章将考虑多处理机系统，而在那以前，我们都假定为单处理机方式。

内核提供的服务是透明的。比如，它能识别一个给定文件是正规文件还是一个设备，但它向用户进程隐蔽了它们的区别。类似地，它需要对文件中的数据格式化以便内部地存储，但它向用户进程隐蔽了内部格式，而返回一个非格式化的字节流。最后，它提供必要的服务，使用户级的进程能支持它们必须提供的服务，同时省略那些能够在用户级实现的服务。比如说，内核支持 shell 作为一个命令解释程序所需要的服务：它允许 shell 读终端输入，动态地产生进程，同步进程的执行，创建管道以及进行 I/O 重定向等。一个用户能在不影响其他用户的情况下构造 shell 的私用版本，裁剪他们的环境来适应他们的规范。像标准的 shell 一样，这些程序使用相同的内核服务。

1.5　关于硬件的假设

UNIX 系统上用户进程的执行分成两个级别：用户与内核。当一个进程执行一个系统调用时，进程的执行态从用户态变为核心态：由操作系统执行并试图为用户的请求服务。如果失败，则返回一个错误码。即使用户对操作系统有显式的没提出什么服务请求，操作系统仍然做与用户进程有关的内务操作、中断处理、进程调度、存储管理等等。很多机器体系结构（及其操作系统）都支持比这里所说的两个级别更多的级别，然而对于 UNIX 系统来说，用户态与核心态这两态已经足够用了。

两态之间的区别是：

• 用户态下的进程能存取它们自己的指令与数据，但不能存取内核指令和数据（或其他进程的指令和数据）。然而，核心态下的进程能够存取内核和用户地址。例如，一个进程的虚地址空间可划分成仅在核心态下可存取及在核心态、用户态下都可存取的两部分。

• 某些机器指令是特权指令，在用户态下执行特权指令会引起错误。比如，机器可以包含一条操纵处理机状态寄存器的指令。在用户态下执行的进程没有执行特权指令的能力。

简单地说，硬件是按核心态与用户态来观察世界的，而对这两种态下正在执行程序的多个用户是不做区别的。操作系统保存着内部记录以区分正在系统上执行着的多个进程。图 1-5 表明了这一区别：内核在横轴方向上把进程 A、B、C、D 区分开来，硬件在纵轴方向上把执行的态区分开来。

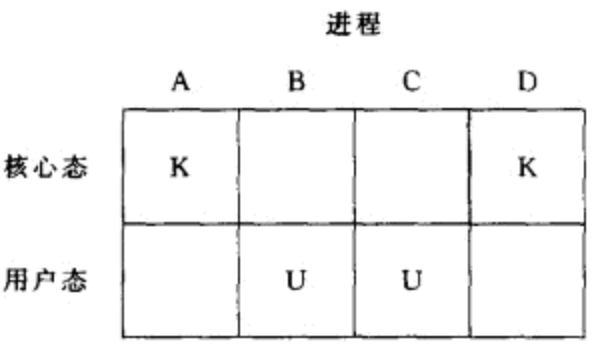


图 1-5　多个进程及执行态

虽然系统在执行时必处于两种态之一，但内核是为着用户进程运行的。内核不是与用户进程平行运行的孤立的进程集合，而是每个用户进程的一部分。本书以后将经常提到“内核”分配资源或“内核”进行各种操作，然而其含义是，一个在核心态下执行的进程分配资源或做各种操作。比如，shell 通过系统调用读用户终端：正在为该 shell 进程执行的内核对终端的操作进行控制，并把敲进的字符返回给 shell。然后 shell 在用户态下执行，对用户敲进来的字符流进行解释，执行特定的动作集合，在这些动作集合中又可以请求引用其他系统调用。

1.5.1　中断与例外

UNIX 系统允许 I/O 外围设备或系统时钟异步地中断 CPU。当接收到中断的时候，内核保存它的当前上下文（表示进程正在做什么的瞬时冻结映象），判定中断原因，为中断服

务。在内核为中断服务之后，内核恢复被中断了的上下文，并且继续进行，就好像什么事也没发生过一样。硬件通常按照中断被处理的次序给设备赋予优先权：当内核为某个中断服务时，它封锁较低优先级的中断，但为更高优先级的中断服务。

例外条件（exception condition）指的是由一个进程引起的非期望事件，例如非法存储寻址、执行特权指令、除数为零等等，它们与来自进程外部的事件所引起的中断是有区别的。例外事件发生在一条指令执行的过程中间，并且系统试图在处理完例外事件之后重新开始执行该指令；中断被认为是在两条指令的执行之间发生的，并且系统在对中断服务完毕之后，从下一条指令继续执行。UNIX 系统使用一种机制来处理中断及例外条件。

1.5.2 处理机执行级

在关键活动期间，有时内核必须阻止中断的发生，因为如果这时允许中断，可能会导致数据的误用。比如，正像在下一章将要看到的那样，当正在对链表操作时，内核可能不希望接收磁盘中断，因为这时处理中断会使指针混乱。计算机通常都有一组特权指令，它们在处理机状态字中建立处理机执行级（processor execation level），把处理机执行级置为一定的值，以便把同级或较低级的中断屏蔽掉，而仅允许较高级的中断。图 1-6 表明了一组执行级的示例。如果内核屏蔽掉磁盘中断，则除了时钟中断与机器错误中断之外，所有的中断都要被阻止。如果它屏蔽了软件中断，则所有其他中断都可能发生。

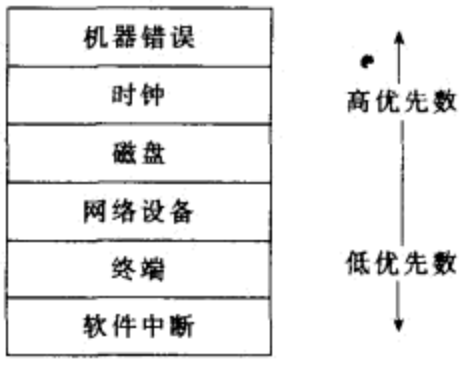


图 1-6 典型的中断级

1.5.3 存储管理

正如当前正在执行的进程（或者，至少是它的一部分）是驻留在主存中的那样，内核是永远驻留在主存中的。当对一个程序进行编译时，编译程序生成一个程序中的地址集合，它们表示变量和数据结构的地址，或诸如函数之类的指令的地址。编译程序产生的是虚机器地址，如同没有其他程序在物理机器上同时执行一样。

当程序在机器上运行时，内核为它分配主存空间，但是不要求由编译程序生成的虚地址与它们在机器中占据的物理地址完全相同。内核与机器硬件一起协作，建立虚地址到物理地址的转换（translation），把编译程序生成的地址映射成物理的机器地址。该映射依赖于机器硬件的能力，从而 UNIX 系统中负责映射的部分是依赖于机器的。例如，某些机器具有特殊的硬件以支持请求调页。第 6 章和第 9 章将更详细地讨论存储管理问题及它们与硬件的关系。

1.6 本章小结

本章描述了 UNIX 系统的整个结构、运行在用户态与核心态下的进程间的关系，及内核关于硬件所做的假设。进程在用户态或核心态下执行，这里，进程通过明确定义的系统调用集合来利用系统服务。系统设计鼓励程序员书写小程序，每个小程序都只做几个操作但做得很好，然后再使用管道及 I/O 重定向把这些小程序组合起来去做更复杂的处理。

系统调用使得进程能做要是没有系统调用就做不成的事。除了系统调用服务以外，内核还为用户团体做一般性内务操作、进程调度的控制、主存中的存储管理与进程保护、中断的

妥善处理、文件和设备管理以及系统错误条件的处理。UNIX 系统内核故意省略很多功能，而这些功能是其他操作系统的组成部分。它提供一小组系统调用，而这些系统调用允许进程在用户级上完成必要的功能。下一章将对内核做更详细介绍，描述它的体系结构及在它的实现中的某些基本概念。

第2章 内核导言

上一章给出了对 UNIX 系统环境的高层次的看法。本章重点放在内核上，对内核的体系结构提出一个总的看法，勾画出它的基本概念和结构，而这些对于了解本书的其余部分是必不可少的。

2.1 UNIX 操作系统的体系结构

Christian 曾提出，UNIX 系统支持文件系统有“空间”而进程有“生命”的假象（见[Christian 83]第 239 页）。文件和进程这两类实体是 UNIX 系统模型中的两个中心概念。图 2-1 给出了内核框图，示出了各种模块及它们之间的相互关系，特别地，它示出了内核的两个主要成分：左边的文件子系统和右边的进程控制子系统。虽然，在实际上，由于某些模块同其他模块的内部操作进行交互而使内核偏离该模型，但该图仍可作为观察内核的一个有用的逻辑观点。

图 2-1 让我们看到了三个层次：用户、内核及硬件。系统调用与库接口体现了图 1-1 中描绘的用户程序与内核间的边界。系统调用看起来象 C 程序中普通的函数调用，而库把这些函数调用映射成进入操作系统所需要的原语，这在第 6 章中有更详细的叙述。然而，汇编语言程序可以不经过系统调用库而直接引用系统调用。程序常常使用像标准 I/O 库这样一些其他的库程序以提供对系统调用的更高级的使用。在编译期间把这些库连接到程序上，因此，就这里所讨论的目的来说，这些库是用户程序的一部分。下面的一个例子将阐明这一点。

图 2-1 把系统调用的集合分成与文件子系统交互的部分及与进程控制子系统交互的部分。文件子系统管理文件，其中包括分配文件空间，管理空闲空间，控制对文件的存取，以及为用户检索数据。进程通过一个特定的系统调用集合，比如通过系统调用 open（为了读或写而打开一个文件），close，read，write，stat（查询一个文件属性），chown（改变文件所有者）及 chmod（改变文件存取许可权）等与文件子系统交互。这些及另外一些有关的系统调用将在第 5 章考察。

文件子系统使用一个缓冲机制存取文件数据，缓冲机制调节在内核与二级存储设备之间的数据流。缓冲机制同块 I/O 设备驱动程序交互，以便启动往内核去的数据传送及从内核来的数据传送。设备驱动程序是用来控制外围设备操作的内核模块。块 I/O 设备是随机存取存储设备，或者说，它们的设备驱动程序使得它们对于系统的其他部分来说好象是随机存取存储设备。例如，一个磁带驱动程序可以允许内核把一个磁带装置作为一个随机存取存储设备看待。文件子系统还可以在没有缓冲机制干预的情况下直接与“原始”I/O 设备驱动程序交互。原始设备，有时被称为字符设备，包括所有不是块设备的设备。

进程控制子系统负责进程同步、进程间通信、存储管理及进程调度。当要执行一个文件而把该文件装入存储器中时，文件子系统与进程控制子系统交互：进程子系统在执行可执行文件之前，把它们读到主存中。这些将在第 7 章看到。

用于控制进程的系统调用有 fork（创建一个新进程），exec（把一个程序的映象覆盖到

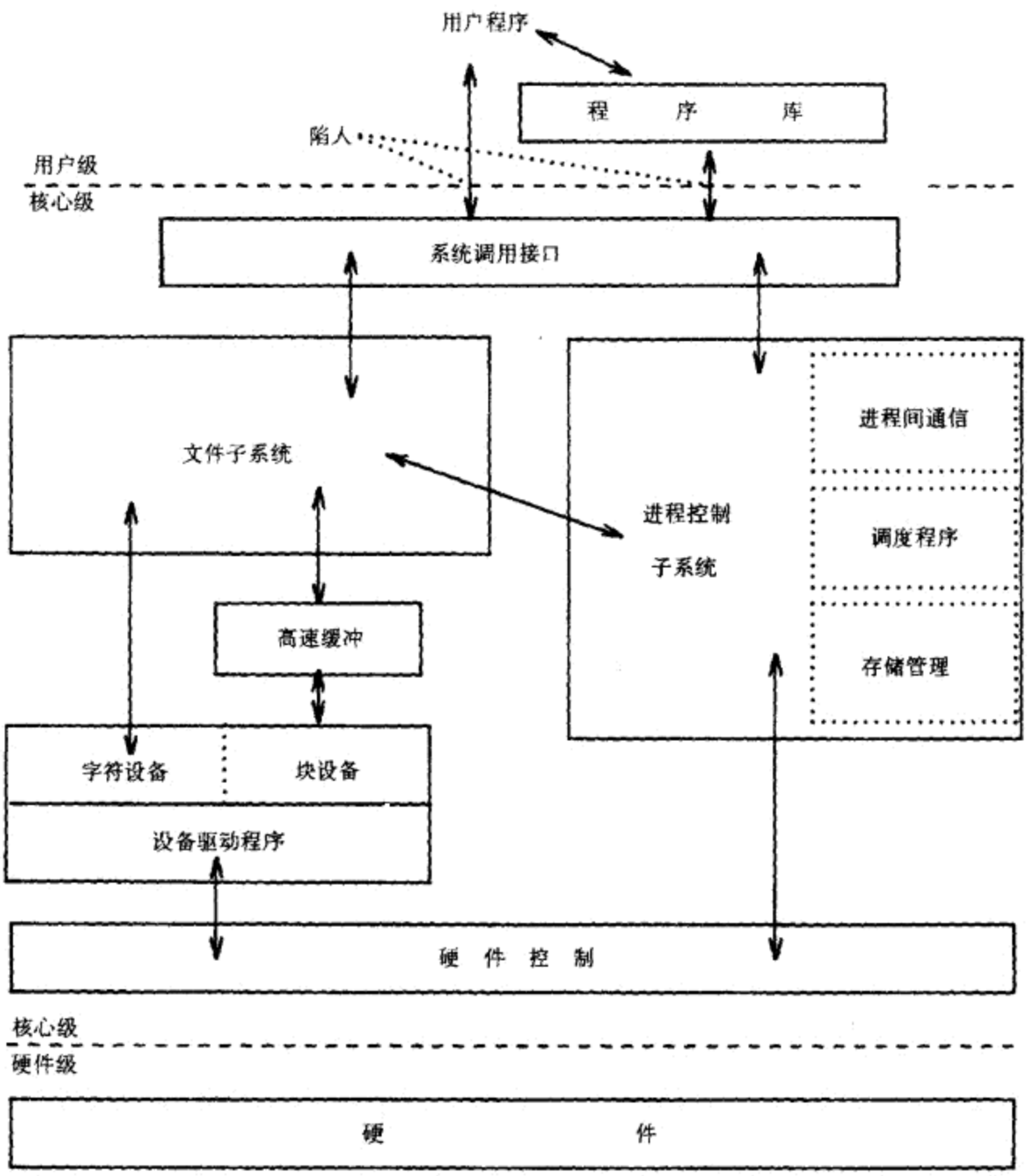


图 2-1 系统内核框图

正在运行的进程上），exit（结束一个进程的执行），wait（使进程的执行与先前创建的一个进程的 exit 相同步），brk（控制分配给一个进程的存储空间的大小）及 signal（控制进程对特别事件的响应）。第 7 章将考察这些及其他系统调用。

存储管理模块控制存储分配。在任何时刻，只要系统没有足够的物理存储供所有进程使用，内核就在主存与二级存储之间对进程进行迁移，以便所有的进程都得到公平的执行机会。第 9 章将描述存储管理的两个策略：对换与请求调页。对换进程有时被称为调度程序，因为它为进程进行存储分配的调度，并且影响到 CPU 调度程序的操作。然而，本书仍将称它为对换程序，以避免与 CPU 调度程序混淆。

调度程序（scheduler）模块把 CPU 分配给进程。该模块调度各进程依次运行，直到它们因等待资源自愿放弃 CPU，或它们最近一次的运行时间超过一个时间量，从而内核抢占它们。于是调度程序选择最高优先权的合格进程投入运行；当原来的进程成为最高优先权的

合格进程时，还会再次投入运行。进程间通信有几种形式，从事件的异步软中断信号到进程间消息的同步传输，等等。

最后，硬件控制负责处理中断及与机器通信。像磁盘或终端这样的设备可以在一个进程正在执行时中断 CPU。如果出现这种情况，在对中断服务完毕之后内核可以恢复被中断了的进程的执行：中断不是由特殊的进程服务的，而是由内核中的特殊函数服务的，这些特殊函数是在当前运行的进程的上下文中被调用的。

2.2 系统概念介绍

本节将概述一些主要的内核数据结构，并且更详细地描述图 2-1 中给出的各模块的功能。

2.2.1 文件子系统概貌

一个文件的内部表示由一个索引节点（inode）给出，索引节点描述了文件数据在磁盘上的布局，并且包含诸如文件所有者、存取许可权及存取时间等其他信息。索引节点这一术语是 index node 的缩写，并且普遍地用于 UNIX 系统的文献中。每个文件都有一个索引节点，但是它可以有几个名字，且这几个名字都映射到该索引节点上。每个名字都被称为一个联结（link）。当进程使用名字访问一个文件时，内核每次分析文件名中的一个分量，检查该进程是否有权搜索路径中的目录，并且最终检索到该文件所对应的索引节点。例如，如果一个进程调用

```
open ( "/fs2/mjb/rje/sourcefile", 1);
```

则内核检查“/fs2/mjb/rje/sourcefile”所对应的索引节点。当一个进程建立一个新文件时，内核分配给它一个尚未使用的索引节点。正如我们很快就会看到的那样，索引节点被储存在文件系统中，但是当操纵文件时，内核把它们读到内存（in-core）[⊖]索引节点表中。

内核还包含另外两个数据结构，文件表（file table）和用户文件描述符表（user file descriptor table）。文件表是一个全局核心结构，但用户文件描述符表对每个进程分配一个。当一个进程打开或建立一个文件时，内核在每个表中为相应于该文件的索引节点分配一个表项。这样一共有三种结构表——用户文件描述符表、文件表和索引结点表（inode table），用这三种结构表中的表项来维护文件的状态及用户对它的存取。文件表保存着文件中的字节偏移量——下一次读或写将从那里开始，并保存着对打开的进程所允许的存取权限。用户文件描述符表标识着一个进程的所有打开文件。图 2-2 表明了这三张表及它们之间的相互关系。对于系统调用 open 和系统调用 creat，内核返回一个文件描述符（file descriptor），它是在用户文件描述符表中的索引值。当执行系统调用 read 和 write 时，内核使用文件描述符以存取用户文件描述符表，循着指向文件表及索引节点表表项的指针，从索引节点中找到文件中的数据。第 4、5 章将详细地描述这些结构，此刻，我们只要说使用这三张表可以实现对一个文件的不同程度的存取共享就够了。

UNIX 系统把正规文件（regular file）及目录保存在诸如磁带或磁盘这样的块设备上。由于磁带和磁盘在存取时间上的差别，所以没有什么 UNIX 系统装置使用磁带实现它们的文

⊖ “core”这一术语指的是机器的原始存储，不是指硬件技术。

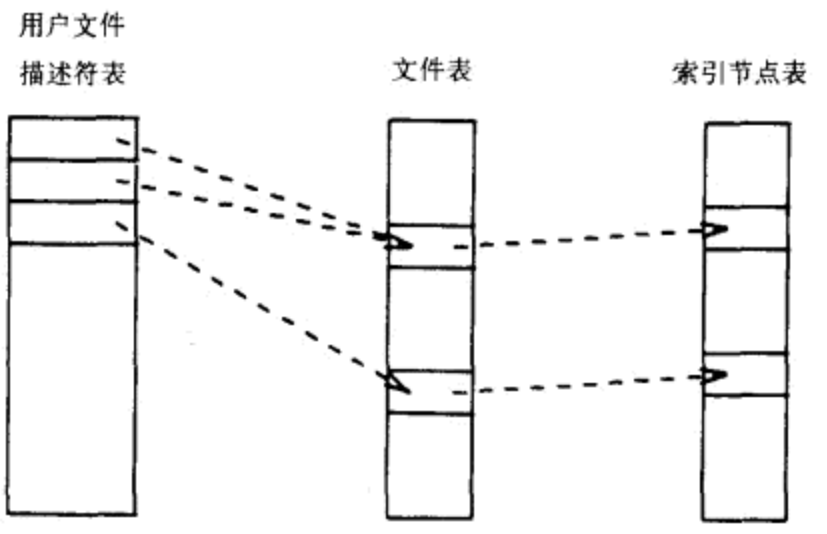


图 2-2　文件描述符、文件表和索引节点表

件系统。今后，无盘工作站将用得很普遍。在无盘工作站中，文件被存放在一个远程系统上，并通过网络进行存取（见第 13 章）。然而，为简单起见，下面假设讨论的是有磁盘的系统。一套系统装置可以有若干物理磁盘设备，每个物理磁盘设备包含一个或多个文件系统。把一个磁盘分成几个文件系统可以使管理人员易于管理存储在那儿的数据。内核在逻辑级上只涉及文件系统，而不涉及磁盘，把每个文件系统都当作由一个逻辑设备号标识的逻辑设备(logical device)。由磁盘驱动程序实现逻辑设备（文件系统）地址与物理设备（磁盘）地址之间的转换。除非另有明确的说明，否则，本书在使用“设备”这一术语时总是意味着一个逻辑设备。

一个文件系统由一个逻辑块（logical block）序列组成，每个块都包含 512、1024、2048 个字节或 512 个字节的任意倍数，这要依赖于系统实现。在一个文件系统中逻辑块大小是完全相同的，但是在一个系统配置中的不同文件系统间逻辑块大小可以是不同的。使用大的逻辑块增加了在磁盘与主存之间的有效数据传送率，因为内核在每次磁盘操作中能传送较多的数据，所以只执行很少几次费时的操作。比如，一次从磁盘读 1K 字节的读操作，会比读两次每次读 512 字节的操作要快。然而，正如将在第 5 章中看到的那样，如果一个逻辑块太大，将失掉有效的存储能力。为简单起见，本书将使用“块”这一术语表示一个逻辑块，并且它将假设一个逻辑块包含 1K 字节数据，除非另有明确说明。

一个文件系统具有如下结构（图 2-3）：

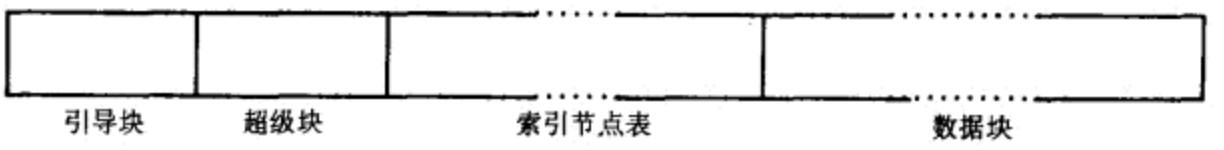


图 2-3　文件系统布局

• 引导块（boot block）占据文件系统的开头，典型地，是一个扇区。它可以含有被读入机器中起引导或初启操作系统作用的引导代码。虽然为了引导系统只需一个引导块，但每个文件系统都有一个（可能是空的）引导块。

• 超级块（super block）描述了文件系统的状态——如它有多大，它能储存多少文件，在文件系统的何处可找到空闲空间，以及其他信息。

• 索引节点表（inode list）是一张装有索引节点的表，它在文件系统中跟在超级块后面。当配置一个文件系统时，管理人员应指明索引节点表的大小。内核通过索引来访问索引节点表中的索引节点。有一个索引节点是文件系统的根索引节点（root inode）：在执行了系统调用 mount（见 5.14 节）之后，该文件系统的目录结构就可以从这个根索引节点开始进行存取了。

• 数据块（data blocks）在索引节点表结束后开始，并且包含文件数据与管理数据。一个已被分配的数据块，能且仅能属于文件系统中的一个文件。

2.2.2 进程

本节将更进一步考察进程子系统。先描述一个进程的结构及用于存储管理的若干进程数据结构。然后给出进程状态图的初步看法，并考虑状态转换中的各种问题。

一个进程是一个程序的执行，它是由一系列有格式字节组成的，这些有格式字节被解释成机器指令（以下被称为“正文（text）”）、数据和栈区（stack）。当内核调度各个进程使之执行时，看起来这些进程像是同时执行的。而且，可以有几个进程是一个程序的实例。一个进程循着一个严格的指令序列执行，这个指令序列是自包含的，并且不会跳转到别的进程的指令序列上。它读、写它自己的数据和栈区，但它不能读或写其他进程的数据和栈区。进程通过系统调用与其他进程及外界进行通信。

用实际的术语来说，UNIX 系统上的进程是被系统调用 fork 所创建的实体。除了 0 进程以外，每个进程都是被另一个进程执行系统调用 fork 时创建的。调用系统调用 fork 的进程是父进程（parent process），而新创建的进程是子进程（child process）。每个进程都有一个父进程，但一个进程可以有多个子进程。内核用各进程的进程标识号（process ID）来标识每个进程，进程标识号简称为进程 ID（或 PID，见第 6 章）。0 进程是一个特殊进程，它是在系统引导时被“手工”创建的；当它创建了一个子进程（1 进程）之后，0 进程就变成对换进程。正如在第 7 章所解释的那样，1 进程被称为 init 进程，是系统中其他每个进程的祖先，并且享有同它们之间的特殊关系。

用户对一个程序的源代码进行编译以建立一个可执行文件，可执行文件由以下几部分组成：

• 描述文件属性的一组“头标（header）”；

• 程序正文；

• 数据的机器语言表示，它给出程序开始执行时的初值；一个空间指示，它指出内核应该为被称为 bss[⊖] 的未初始化数据分配多大的空间（在运行时内核把 bss 的初值置为 0）；

• 其他段，诸如符号表信息。

对于图 1-3 中的程序，可执行文件的正文是函数 main 与 copy 所生成的代码，其中，变量 version 是初始化数据（放在本程序中仅仅是为让它有初始化数据），数组 buffer 是未初始化的数据。系统 V 的 C 编译程序版本在缺省时创建一个分离的正文段，但它支持一种选择，该选择允许数据段中包含程序指令，这是在系统的较老的版本中使用的。

在系统调用 exec 期间，内核把一个可执行文件装入主存中，被装入的进程至少由被称为正文区、数据区及栈区的三部分组成。正文区和数据区相应于可执行文件中的正文段和数据 bss 段。但是栈区是自动创建的，而且它的大小在运行时是被内核动态调节的。栈区由逻

⊖ bss 这一名字来自 IBM 7090 机的汇编伪运算符，它代表“block started by symbol”。

辑栈帧（stack frame）组成，当调用一个函数时栈帧被压入，当返回时栈帧被弹出。一个称为栈指针（stack pointer）的特殊寄存器指示出当前栈深度。一个栈帧包含着用于函数调用的参数、它的逻辑变量及为恢复先前的栈帧所需要的数据——其中包括函数调用时程序计数器的值及栈指针的值。程序代码包含管理栈增长的指令序列，并且当需要时内核为栈区分配空间。在图 1-3 的程序中，当 main 被调用时（按惯例，在每个程序中被调用一次），函数 main 中的参数 argc 和 argv、变量 fdold 和 fdnew 就会在栈区上出现。并且，无论何时函数 copy 被调用，copy 中的参数 old 与 new 及变量 count 都在栈区上出现。

因为 UNIX 系统中的一个进程能在两种态——核心态（kernel mode）或用户态（user mode）下执行，所以 UNIX 系统中核心栈（kernel stack）与用户栈（user stack）是分开的。用户栈含有在用户态下执行时函数调用的参数、局部变量及其他数据。图 2-4 中的左半部表明一个进程在 copy 程序中做系统调用 write 时进程的用户栈。进程启动过程（此过程是包含在库中的）用两个参数调用函数 main，并将第 1 帧压入用户栈；第 1 帧含有 main 的两个

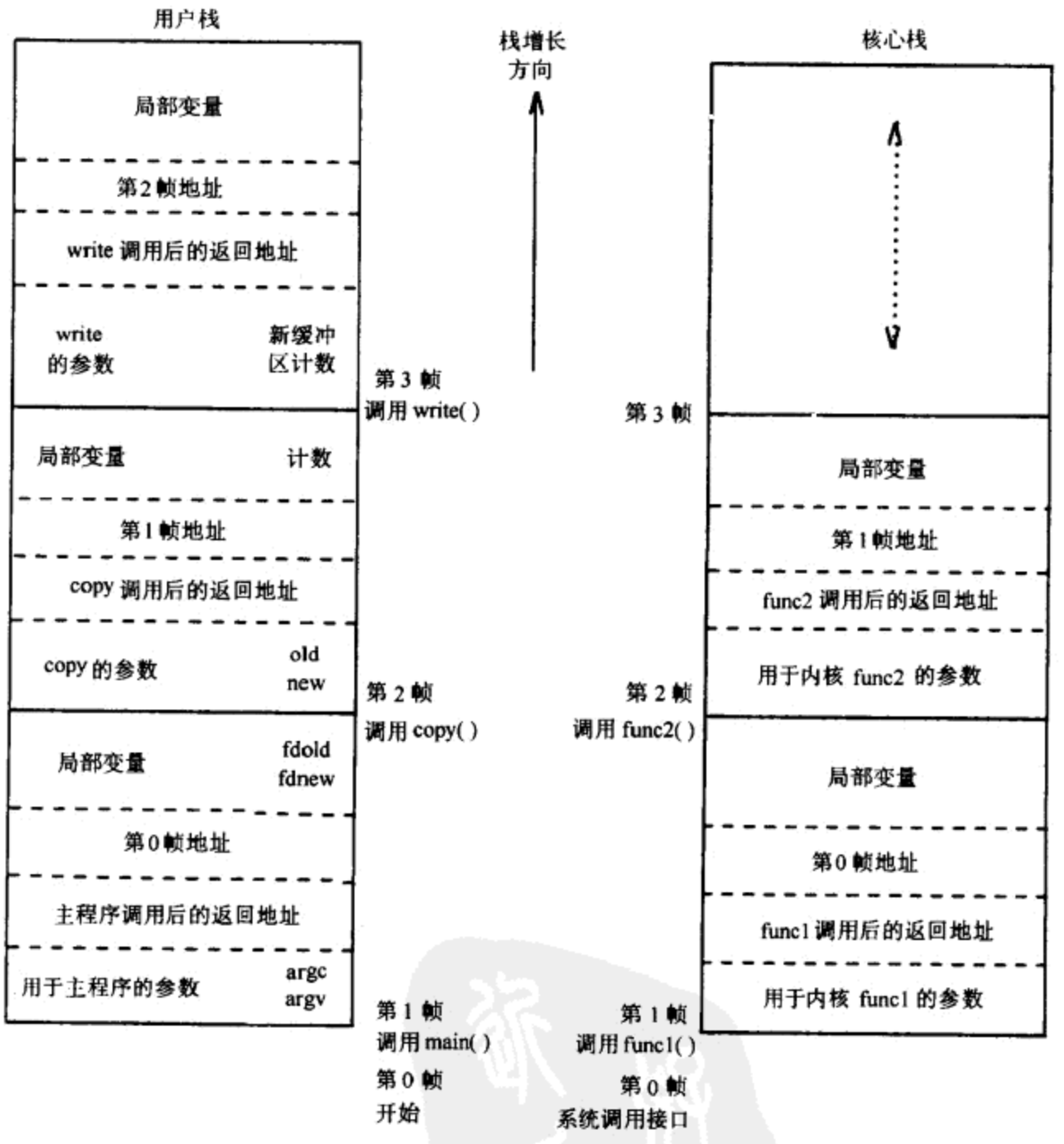


图 2-4　程序 copy 的用户栈及核心栈

局部变量的空间。然后 main 用两个参数 old 与 new 调用 copy，并将第 2 帧压入到用户栈中，第 2 帧包含局部变量 count 的空间。最后，进程通过调用库函数 write 来引用系统调用 write，每个系统调用都在系统调用库中有一个入口点；系统调用库按汇编语言编码并包含一个特殊的 trap 指令，当该指令被执行时，它引起一个“中断”，从而导致硬件转换到核心态。一个进程调用一个特定的系统调用库的入口点，正像它调用任何函数一样。对于库函数也要创建一个栈帧。当进程执行特定的指令时，它将处理机执行态转换到核心态，执行内核代码，并使用核心栈。

核心栈中含有在核心态下执行的函数的栈帧。核心栈上的函数及数据项涉及的是内核中的而不是用户程序中的函数和数据，但它的构成与用户栈的构成相同。当一个进程在用户态下执行时，它的核心栈为空的。图 2-4 的右半部给出了一个在 copy 程序中执行系统调用 write 的进程的核心栈的表示。在以后的章节中对系统调用 write 进行详细讨论时，再叙述各算法名称。

每个进程在内核进程表（process table）中都有一个表项，并且每个进程都被分配一个 u 区[⊖]，u 区包含仅被内核操纵的私用数据。进程表包含（或指向）一个本进程区表（per process region table），本进程区表的表项指向区表（region table）的表项。一个区是进程地址空间中连续的区域，如正文区、数据区及栈区等。区表登记项描述区的属性，诸如它是否包含正文或数据，它是共享的还是私用的，以及区的“数据”位于主存的何处，等等。从本进程区表到区表的额外间接级允许彼此独立的进程对区的共享。当一个进程调用系统调用 exec 时，在释放了进程一直在使用着的老区之后，内核为它的正文、数据和栈分配新区。当一个进程调用系统调用 fork 时，内核拷贝老进程的地址空间，在可能时允许进程对区共享，否则再建立一个物理拷贝。当一个进程调用系统调用 exit 时，内核释放进程使用过的区。图 2-5 示出了一个运行中的进程的有关数据结构：进程表指向本进程区表，本进程区表有指向该进程的正文区、数据区或栈区的区表表项的指针。

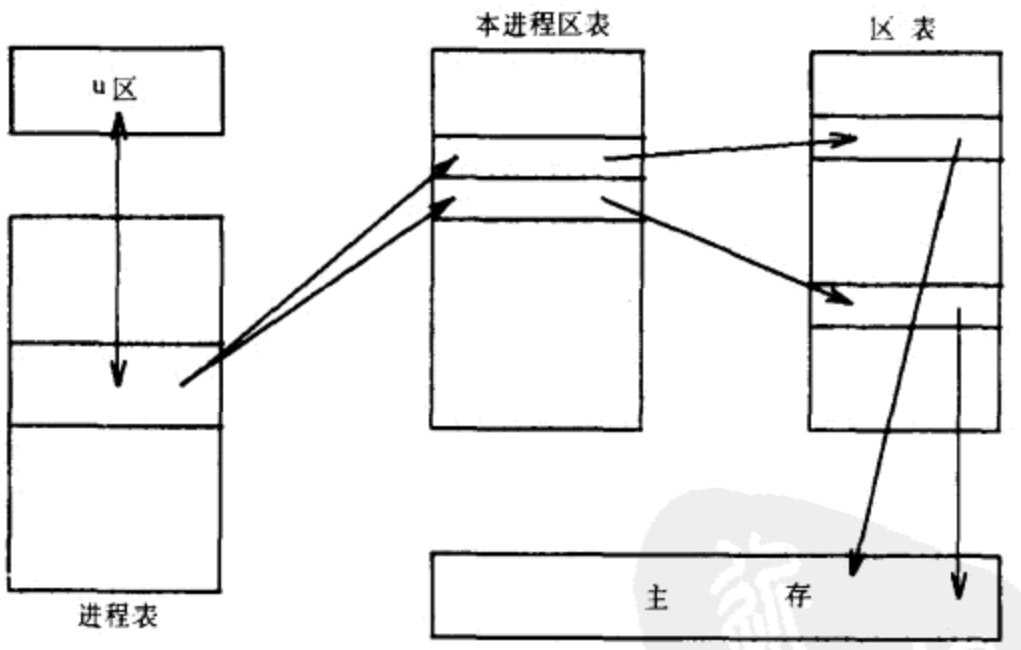


图 2-5 进程的数据结构

⊖ u 区中的 u 代表用户。u 区的另一个名称是 ublock；本书则总是称它为 u 区。

进程表表项及 u 区包含进程的控制信息和状态信息。u 区是进程表表项的扩展，第 6 章将考察这两个表的区别。在后几章中讨论的进程表中的字段是：

- 状态字段。
- 标识符——指示拥有该进程的用户（用户 ID 或 UID，见第 6 章）。
- 当一个进程被挂起（在 sleep 状态）时的事件描述符集合。

u 区包含的是用来描述进程的信息，这些信息仅当进程正在执行时才是可存取的。重要的字段有：

- 指向当前正在执行的进程的进程表项的指针。
- 当前系统调用的参数，返回值及错误码。
- 所有的打开文件的文件描述符。
- 内部 I/O 参数。
- 当前目录（current directory）和当前根（current root）（见第 5 章）。
- 进程及文件大小的限制。

内核能直接存取正在执行的进程的 u 区的字段，但不能存取其他进程的 u 区的字段。在其内部，内核引用结构变量 u 以存取当前正在运行的进程的 u 区，并且当另一进程执行时，内核重新安排它的虚地址空间，以使结构变量 u 引用的是新进程的 u 区。由于这一实现方式给出了从 u 区到它的进程表表项的指针，所以内核很容易识别出当前进程。

1. 进程上下文

一个进程的上下文（context）包括被进程正文所定义的进程状态、进程的全局用户变量和数据结构的值、它使用的机器寄存器的值、存储在它的进程表项与 u 区中的值以及它的用户栈和核心栈的内容。操作系统的正文和它的全局数据结构被所有的进程所共享，因而不是进程上下文的一部分。

当执行一个进程时，系统被说成在该进程的上下文中执行。当内核决定它应该执行另一个进程时，它做一次上下文切换（context switch），以使系统在另一个进程的上下文中执行。正如将要看到的，内核仅允许在指定条件下进行上下文切换。当进行上下文切换时，内核保留足够信息，为的是以后它能切换回第一个进程，并恢复它的执行。类似地，当从用户态移到核心态时，内核保留足够信息以便它后来能返回到用户态，并从它的断点继续执行。在用户态与核心态之间的移动是态的改变，而不是上下文切换。再看一下图 1-5，当它把上下文从进程 A 变成进程 B 时，内核做的是上下文切换；当发生从用户到核心态或从内核到用户态的改变时，所改变的是执行态，但仍在同一个进程（例如进程 A）的上下文中执行。

内核在被中断了的进程的上下文中对中断服务，即使该中断可能不是由它引起的。被中断的进程可以是正在用户态下执行的，也可以是正在核心态下执行的。内核保留足够的信息以便它在后来能恢复被中断了的进程的执行，并在核心态下对中断进行服务。内核并不产生或调度一个特殊进程来处理中断。

2. 进程状态

一个进程的生存周期能被划分为一组状态，每个状态都具有一定的用来描述该进程的特点。第 6 章将描述所有的进程状态，但现在了解如下状态是重要的：

（1）进程正在用户态下执行。

（2）进程正在核心态下执行。

(3) 进程未正在执行，但是它已准备好运行——一旦调度程序选中了它，它就可以投入运行。很多进程可以处于这一状态，而调度算法决定哪个进程将成为下一个执行的进程。

(4) 进程正在睡眠。当进程再也不能继续执行下去的时候，如正在等候 I/O 完成时，进程使自己进入睡眠状态。

因为任何时刻一个处理机仅能执行一个进程，所以至多有一个进程可以处在第 1 种状态和第 2 种状态。这两个状态相应于两种执行态：用户态与核心态。

3. 状态转换

以上描述的进程状态给出了进程的一种静态观点，但是，实际上，各个进程是按照明确定义的规则连续地在各种状态间移动的。状态转换图是一个有向图，它的节点表示一个进程能进入的状态，而它的边表示引起一个进程从一种状态到另一种状态的事件。如果从第一种状态到第二种状态存在着一条边，则这两种状态之间的状态转换是合法的。可以从一种状态发出多个转换，但是，就处于某种状态的一个进程来说，依赖于所发生的系统事件，完成一个且只完成一个转换。图 2-6 给出了上述定义的进程状态的状态转换图。

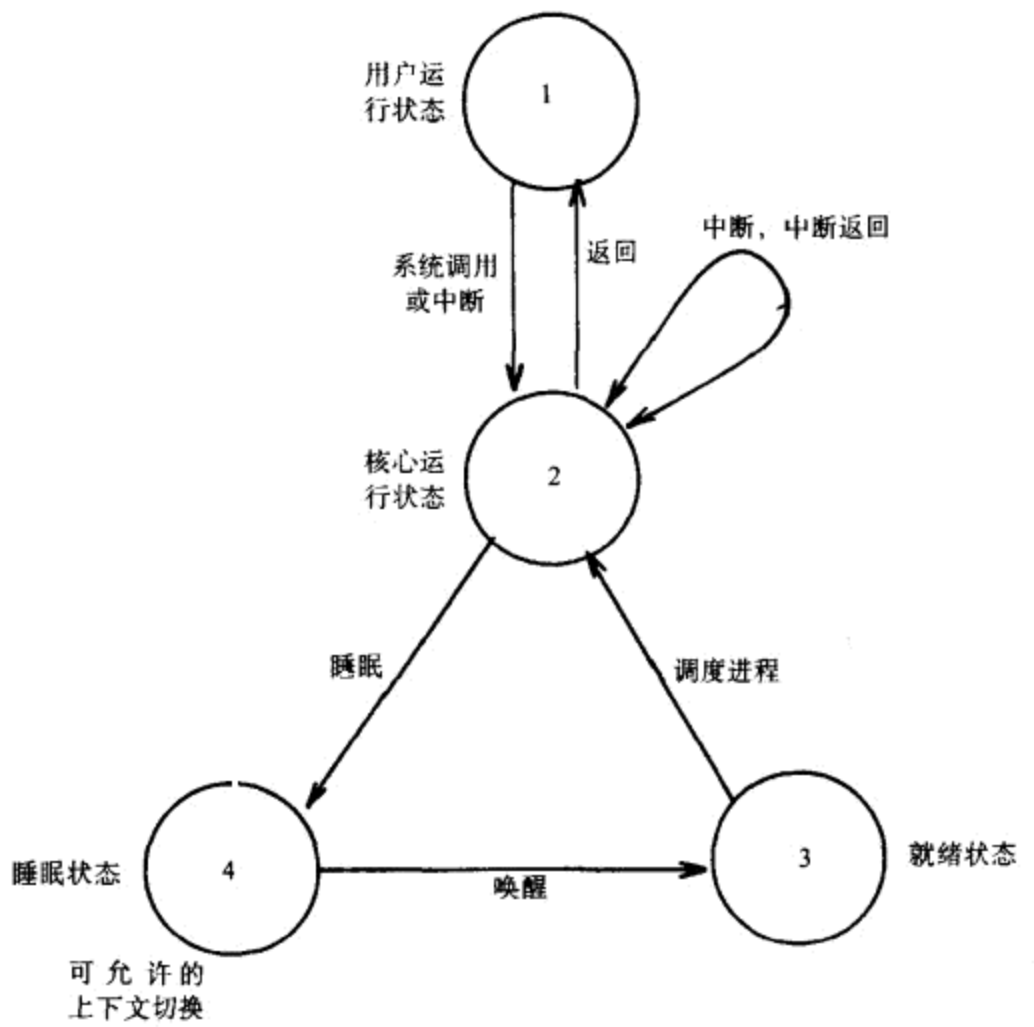


图 2-6 进程状态及转换

如前所述，在一个分时方式中，几个进程能同时执行，并且它们可能都要在核心态下运行。如果对它们在核心态下的运行不加以限制，则它们会破坏全局核心数据结构中的信息。通过禁止任意的上下文切换和控制中断的发生，内核可保护它的一致性。

仅当进程从“核心态运行”状态转移到“在内存中睡眠”状态时，内核才允许上下文切换。在核心态下运行的进程不能被其他进程所抢占，因此内核有时被称为不可抢先（non-

preemptive）的，虽然系统并不抢占处于用户态下的进程。因为内核处于不可抢先状态，所以内核可保持它的数据结构的一致性，从而解决了互斥（mutual exclusion）问题——保证在任何时刻至多一个进程执行临界区代码。

比如，让我们考虑图 2-7 中的示例代码。该代码段要把其地址在指针变量 bp1 中的数据结构，插入到双向链表中地址在指针变量 bp 中的数据结构之后。如果当内核执行这一代码段时系统允许上下文切换，则会发生如下情形。假设直到注释出现之前内核执行该代码，然后做一个上下文切换，这时双向链表处于非一致性状态：结构 bp1 一半被插在该链表上，另一半在该链表外。如果进程沿着向前的指针，则它能在该链表上找到 bp1；但如果沿着向后的指针，则它不能找到 bp1（图 2-8）。如果其他进程在原来的进程再次运行之前操纵链表上的这些指针，则双向链表结构能被永久性地毁坏。UNIX 系统通过一个进程在核心态下执行时不允许上下文切换来防止这种情况发生。如果一个进程进入睡眠从而允许上下文切换，则必须使内核算法的编码实现能确保系统数据结构处于安全、一致的状态。

```
struct queue {

    } * bp, * bp1;
bp1->forp = bp->forp;
bp1->backp = bp;
bp->forp = bp1;
/* 考虑可能的上下文切换 */
bp1->forp->backp = bp1;
```

图 2-7　创建双链表的示例代码

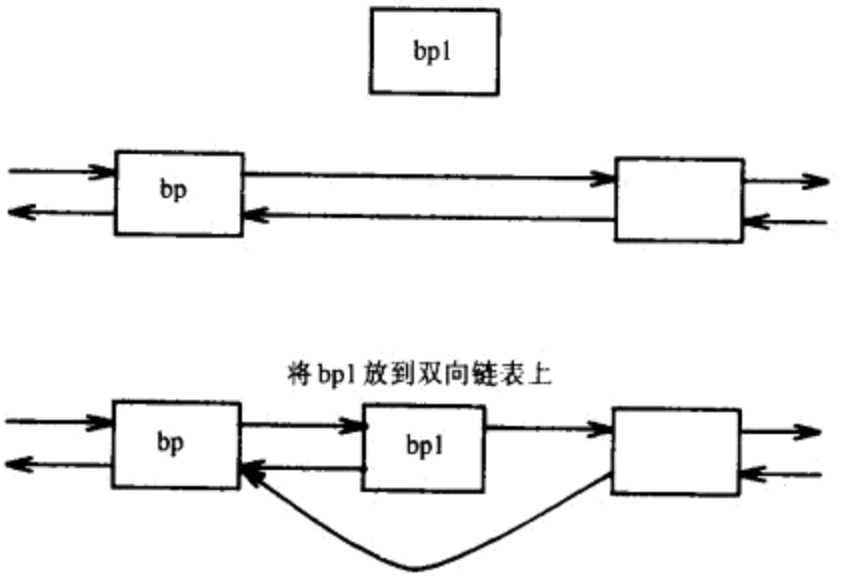


图 2-8　由于上下文切换而造成的不正确链表

与能引起内核数据的非一致性的有关问题是中断的处理。中断处理能改变内核状态信息。举例来说，如果内核正在执行图 2-7 中的代码，当执行到注释行时接收了一个中断，并且中断处理程序是如前所述的那样操纵指针，则中断处理程序就会破坏该链表中的信息。若规定在核心态下执行时系统禁止所有的中断，就可以解决这一问题。但是这可能会使中断的服务推迟，或者可能会损害系统吞吐量，为此，改为当进入代码临界区（critical region）时内核把处理机执行级提高，以禁止中断。一个代码段是临界的，如果任意的中断处理程序的执行会导致一致性问题的话。比如，如果一个磁盘中断处理程序操纵图中的缓冲区队列，则内核操纵缓冲区队列的那个代码段是关于磁盘中断处理程序的代码临界区。临界区应小且不经常出现，以便系统吞吐量不大会被它们的存在所影响。其他操作系统解决这一问题的方法是：规定在系统状态下执行时封锁所有的中断，或者采用完善的加锁方案以保证一致性。第 12 章将面对多处理机系统再回过头来讨论这一问题。这里所给出的解答在那时就不够了。

现在让我们回顾一下本节内容：内核通过仅当一个进程使自己进入睡眠时才允许上下文切换，以及通过禁止一个进程改变另一个进程的状态来保护它的一致性。它还在代码临界区周围提高处理机执行级，以封锁其他能引起非一致性的中断。进程调度程序定期地抢占用户

态下的进程执行，以使进程不能独占式地使用CPU。

4. 睡眠与唤醒

一个在核心态下执行的进程在决定它对系统事件的反应上它打算做什么方面有很大的自主权。进程能互相通信并且“建议”各种可供选择的方法，但由它们自己做出最后的决定。正如我们将要看到的，存在着一组进程在面临各种情况时所应服从的规则，但是每个进程最终都是自主地遵循这些规则的。例如，当一个进程必须暂停它的执行（“进入睡眠”）时，它能自由地按自己的意图去做。然而，一个中断处理程序不能睡眠，因为如果中断处理程序能睡眠，就意味着被中断的进程会被投入睡眠。

进程会因为它们正在等待某些事件的发生而进入睡眠，例如：等待来自外围设备的I/O完成；等待一个进程退出；等待获得系统资源，等等。当我们说进程在一个事件上睡眠时，这意味着，直到该事件发生时，它们一直处于睡眠状态；当事件发生时它们被唤醒，并且进入“就绪”状态。很多进程能同时睡眠在一个事件上；当一个事件发生时，由于这个事件的条件再也不为真了，所以所有睡眠在这个事件上的进程都被唤醒。当一个进程被唤醒时，它完成一个从“睡眠”状态到“就绪”状态的状态转换，对于随后的调度来说，该进程就是个合格者了，但它并不立即执行。睡眠进程不耗费CPU资源；内核并不是经常地去查看一个进程是否仍处于睡眠状态，而是等待事件的发生，那时把进程唤醒。

举例来说，一个在核心态执行的进程有时必须锁住一个数据结构，如果发生后来它进入睡眠的情况，其他企图操纵该上了锁的数据结构的进程必须检查上锁情况，并且因为别的进程已经占有该锁，则它们去睡眠。内核按如下方式实现这样的锁：

```
while（条件为真）
    sleep（事件：条件变为假）；
置条件为真；
```

它按如下方式解锁并唤醒睡眠在该锁上的所有进程：

```
置条件为假；
wakeup（事件：条件变为假）。
```

图2-9描绘了三个进程A、B、C为一个上了锁的缓冲区进行竞争的情况。睡眠的条件是缓冲区处于上锁状态。在任一时刻只能有一个进程在执行，它发现缓冲区是上了锁的，就在缓冲区变为开锁状态的事件上等待。终于，缓冲区的锁解开了，所有的进程被唤醒并且进入“就绪”状态。内核最终选择一个进程（比如说B）执行。进程B执行“while”循环，发现缓冲区处于开锁状态，于是为缓冲区上锁，并且继续执行。如果后来进程B在为缓冲区解锁之前再次去睡眠（例如等候I/O操作的完成），则内核能调度其他进程去运行。如果它选择了进程A，进程A执行“while”循环，发现缓冲区处于上锁状态，那么它就再次去睡眠。进程C可以做同样的事情。最后，进程B醒来并为缓冲区解锁，允许进程A也允许进程C存取缓冲区。因此，“while-sleep”循环保证至多一个进程能获得对资源的存取。

第6章将极其详细地介绍睡眠与唤醒的算法。在此期间它们应被考虑成是“原子的”：一个进程瞬时地进入睡眠状态，并停留在那儿直至它被唤醒。在它睡眠之后，内核调度另一个进程去运行，并切换成后者的上下文。

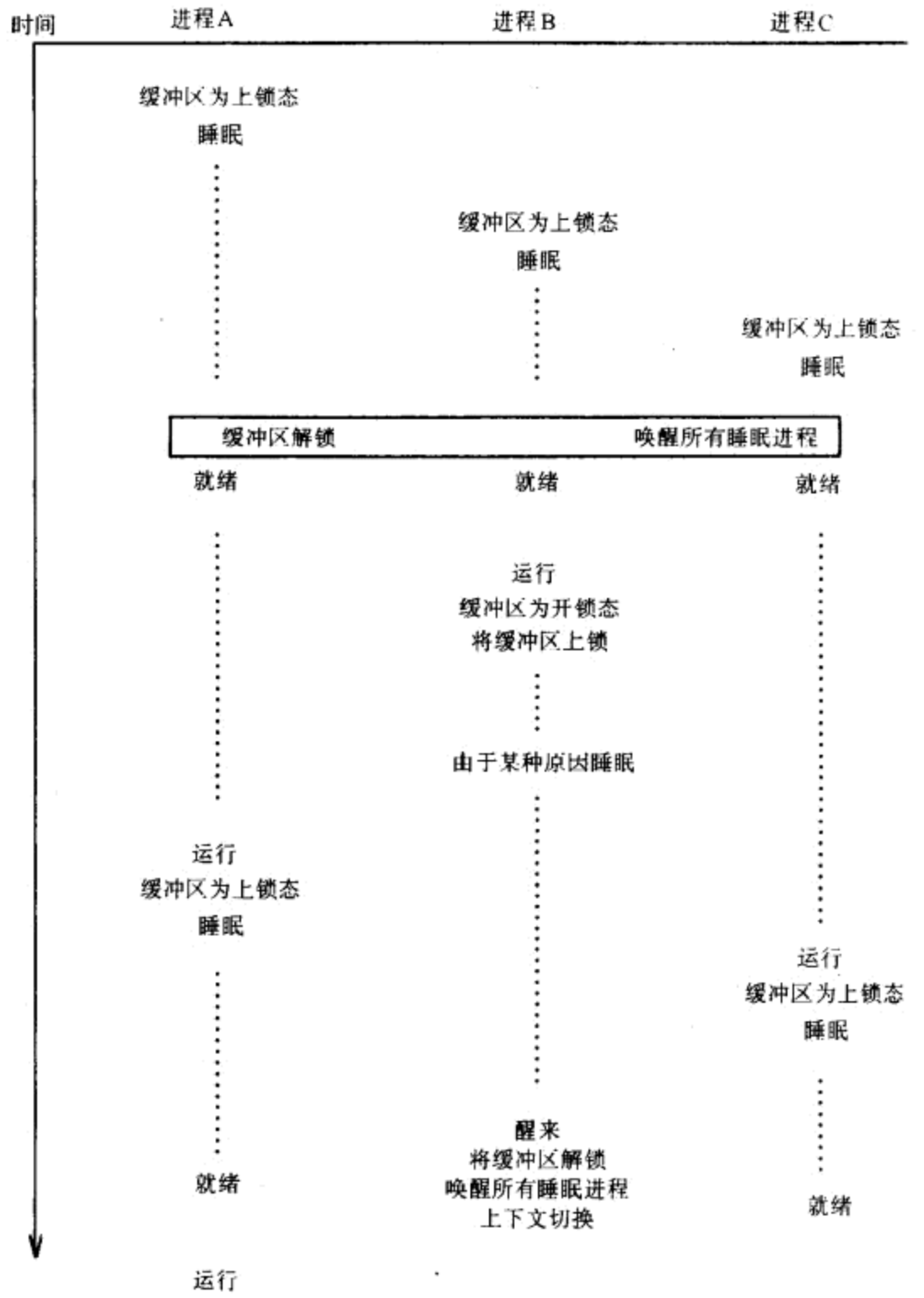


图 2-9 在一个锁上睡眠的多个进程

2.3 内核数据结构

大多数内核数据结构都占据固定长度的表而不是动态地分配空间，这一方法的优点是内核代码简单。但是它限制了一种数据结构的表项的数目，为系统生成时原始配置的数目。如果在系统操作期间，内核用完了一种数据结构的表项，则它不能动态地为新的表项分配空间，而是必须向发出请求的用户报告一个错误。另一方面，如果内核被配置得具有不可能用完的表空间，则由于不能用于其他目的而使多余的表空间浪费了。然而，一般都认为内核算法的简单性比挤出主存中每一个仅有的字节的必要性更重要一些。算法通常使用简单的循环，来寻找表中的空闲表项，这是一个较易于理解的方法，而且有时比复杂的分配方案更为有效。

2.4 系统管理

管理进程可以非严格地归入为用户团体的公共福利提供各种功能的那类进程。这些功能包括磁盘格式化、新文件系统的创建、修复被破坏的文件系统、内核调试及其他。从概念上说，管理进程与用户进程没有区别：它们都使用为一般用户团体可用的相同的一组系统调用。它们仅在被允许的权限与特权上区别于一般用户进程。例如，文件存取权限允许管理进程操纵对一般用户来说禁止进入的文件。在内部，内核把一个称为超级用户（superuser）的特殊用户区别出来，赋予它特权，这一点我们即将看到。通过履行一次注册-口令序列或通过执行特殊程序可使一个用户成为超级用户。超级用户特权的其他用途将在随后的章节中研究。简而言之，内核不识别一个分离的管理进程类。

2.5 本章小结

本章描述了内核的体系结构；它的两个主要成分是文件子系统与进程子系统。文件子系统控制用户文件中数据的存储与检索。文件被组织到文件系统中，而文件系统被看作一个逻辑设备。像磁盘这样的一个物理设备能包含几个逻辑设备（文件系统）。每个文件系统都有一个用来描述文件系统的结构和内容的超级块，并且文件系统中的每个文件都由索引节点描述，索引结点给出了文件的属性。操纵文件的系统调用通过索引节点来实现其功能。

进程有各种状态，并且按照明确定义的转换规则在这些状态之间转移。特别地，在核心态下执行的进程能暂停它们的执行而进入睡眠状态，但是没有哪一个进程能把另一进程投入睡眠状态。内核是不可被抢占的，这意味着，一个在核心态下执行的进程将连续执行，直至它进入睡眠状态或直至它返回到用户态下执行时为止。内核通过实施不可抢占策略以及通过在执行代码临界区时封锁中断来维护它的数据结构的一致性。

本章的其余部分详细描述了图 2-1 所示的子系统及它们的交互作用。从文件子系统开始，继之以进程子系统。下一章将涉及高速缓冲问题，并描述在第 4、5、7 章要介绍的算法中所使用的缓冲区分配算法。第 4 章考察文件系统的内部算法，包括索引节点的操纵、文件的结构及路径名到索引节点的转换。第 5 章解释若干系统调用，例如系统调用 open，close，read 及 write，这些系统调用使用了第 4 章中的算法来访问文件系统。第 6 章论述进程上下文的基本思想及其地址空间；第 7 章涉及有关进程管理及使用第 6 章的算法的系统调用；第 8 章考察进程调度；第 9 章讨论存储管理算法；第 10 章讲的是设备驱动程序，直到这时终端驱动程序与进程管理之间的相互关系才能被解释；第 11 章介绍了进程间通信的某些形式；最后两章涉及若干高级专题，包括多处理机系统与分布式系统。

2.6 习题

1. 考虑如下命令序列：

```
grep main a.c b.c c.c > grepout &
wc -l < grepout &
rm grepout &
```

每一命令行尾部的“&”都通知 shell 在后台运行这些命令，并且它能并行地执行每个命令。

为什么这不等价于如下的命令行？

```
grep main a.c b.c c.c|wc-l
```

2. 考虑图 2-7 中的内核代码示例。假设当代码到达注释处时发生上下文切换，并且假设另一进程通过执行如下代码而从链表中摘掉一个缓冲区：

```
remove(qp)
    struct queue *qp;
{
    qp->forp->backp=qp->backp;
    qp->backp->forp=qp->forp;
    qp->forp=qp->backp=NULL;
}
```

考虑三种情况：

- 进程从链表中摘掉 bp 结构。
- 进程从链表中摘掉 bp1 结构。
- 进程摘掉链表上当前跟在 bp1 之后的结构。

这三种情况中哪种是原来的进程执行完注释以后的代码时链表的状态？

3. 如果内核试图唤醒睡眠在一个事件上的所有进程，但是在唤醒时没有进程睡眠在那个事件上，那么会发生什么情况？

第3章　数据缓冲区高速缓冲

正如在上一章所提到的那样，内核维护诸如磁盘这样的大容量存储器上的文件，并且内核允许进程存储新的信息，或再访问先前存储的信息。当进程想要从一个文件上存取数据时，内核把文件中的数据移入主存，在那里该进程能观察它，修改它，然后请求将数据再次保留到文件系统中。比如，让我们回忆一下图 1-3 中的 copy 程序：内核把数据从第一个文件读到主存中，并且随后把数据写到第二个文件上。正如内核必须把文件数据移到主存中一样，内核也必须把辅助性数据移到主存，以便操纵这些数据。比如，文件系统的超级块描述了在文件系统上可获得的空闲空间（当然还描述了其他内容）。内核把超级块读入主存以存取它的数据，并且当内核希望保留它的数据时，又把它写回文件系统。类似地，索引节点描述了一个文件的布局。当内核想要存取一个文件中的数据时，内核把该文件对应的索引节点读入主存；而当它想要修改文件的布局时，它把索引节点写回文件系统。它用不着对正在运行的进程有显式的知识或进行请求，就可以操纵这些辅助数据。

对文件系统的一切存取操作，内核都能通过每次直接从磁盘上读或往磁盘上写来实现。但是，慢的磁盘传输速率会使系统响应时间加长、吞吐率降低。因此，内核通过保持一个称为数据缓冲区高速缓冲[⊖]（buffer cache）（简称高速缓冲）的内部数据缓冲区池来试图减小对磁盘的存取频率。高速缓冲含有最近被使用过的磁盘块的数据。

图 2-1 表明，内核体系结构中的高速缓冲模块的位置是在文件子系统与（块）设备驱动程序之间。当从磁盘中读数据的时候，内核试图先从高速缓冲中读。如果数据已经在该高速缓冲中，则内核可以不必从磁盘上读。如果数据不在该高速缓冲中，则内核从磁盘上读数据，并将其缓冲起来。所使用的算法试图把尽可能多的好数据保存在高速缓冲中。类似地，要往磁盘上写的数据也被暂存于高速缓冲中，以便如果内核随后又试图读它时，它能在高速缓冲中。内核也通过判定是否数据必须真的需要存储到磁盘上，或数据是否是将要很快被重写的暂时性数据，来减少磁盘写操作的频率。高层内核算法让高速缓冲模块把数据预先缓存起来，或延迟写数据以扩大高速缓冲的效果。本章就来描述内核用于操纵高速缓冲中的缓冲区的算法。

3.1　缓冲头部

在系统初启期间，内核按照存储器的大小及系统性能的约束条件来为若干个缓冲区分配空间。一个缓冲区由两部分组成：一个含有磁盘上的数据的存储器数组及一个用来标识该缓冲区的缓冲头部（buffer header）。由于缓冲头部到数据数组之间有一到一的映射关系，所以下面的讨论常把两部分统称作“缓冲区”，而所讨论的究竟是指哪部分，从上下文中可以清楚地判断出来。

一个缓冲区的数据与文件系统上一个逻辑磁盘块中的数据相对应，并且内核通过考察缓冲头部中的标识符字段来识别缓冲区内容。缓冲区是磁盘块（disk block）在主存中的拷贝，

⊖ 数据缓冲区高速缓冲是一个软件结构，请不要与加速主存访问的硬件高速缓冲混淆。

磁盘块的内容映射到缓冲区中。但该映射是临时的，过一段时间后，内核会决定将另一个磁盘块映射到该缓冲区中。在同一时刻，绝不能将一个磁盘块映射到多个缓冲区中。如果有两个缓冲区包含的是同一个磁盘块的数据，则内核无法知道哪一个缓冲区包含着当前数据，并且会把不正确的数据写回磁盘。比如，一个磁盘块映射到两个缓冲区 A 和 B 上。如果内核先把数据写到缓冲区 A，而后又写入缓冲区 B，而且所有的写操作把缓冲区 B 完全填满时，则磁盘块应包含缓冲区 B 的内容。然而，如果内核把缓冲区往磁盘上拷贝时颠倒了次序，则磁盘块将包含不正确的数据。

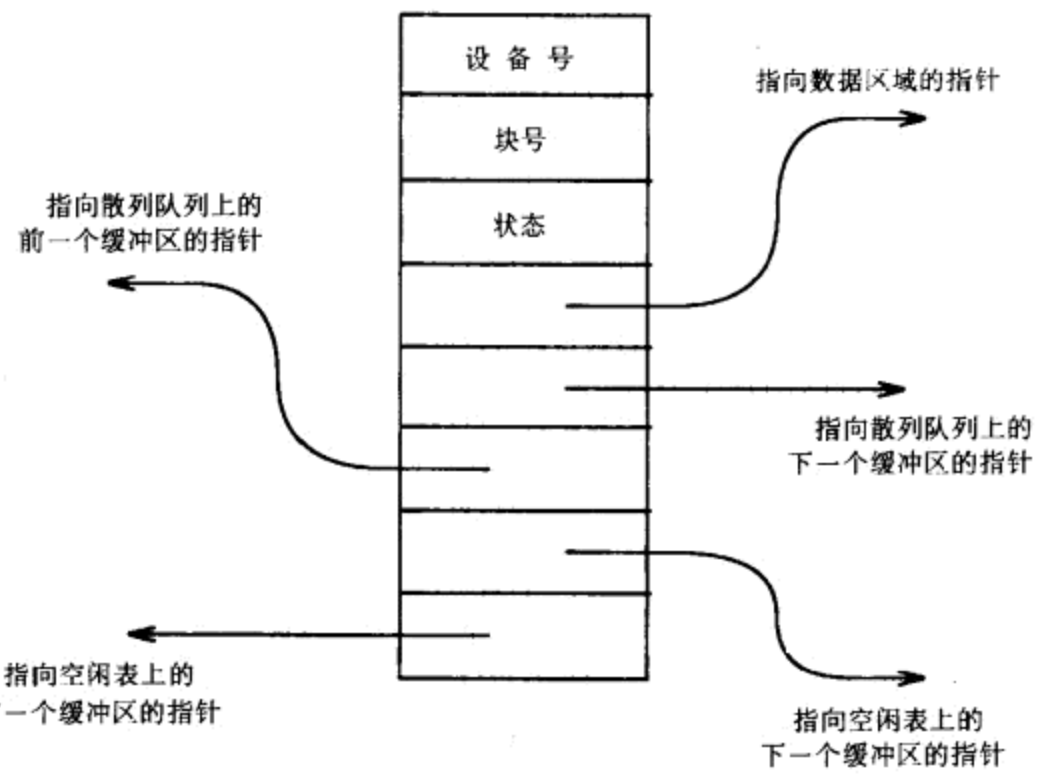


图 3-1　缓冲头部

缓冲头部（图 3-1）包含一个设备号字段与一个块号字段，这两个字段指明了文件系统与磁盘上的数据的块号，并且唯一地标识了该缓冲区。设备号是逻辑文件系统号（见 2.2.1 节），而不是物理设备（磁盘）号。缓冲头部也包含一个指向该缓冲区的数据数组的指针，该数组大小至少必须有磁盘块那么大。缓冲头部还包含状态字节，它概括了缓冲区的当前状态。一个缓冲区状态是如下条件的组合：

- 缓冲区当前为“上锁”（术语“上锁”与“忙”将互换使用，正如“开锁”与“闲”将互换使用一样）；
- 缓冲区包含有效数据；
- 内核在把某缓冲区重新分配出去之前必须把该缓冲区内容写到磁盘上，这一条件叫做“延迟写”（delayed write）；
- 内核当前正在从磁盘往缓冲区读信息或把缓冲区的内容写到磁盘上；
- 一个进程当前正在等候缓冲区变为闲。

缓冲头部还包含两组指针，缓冲区分配算法使用这两组指针来维护缓冲池的整体结构，这将在下一节解释。

3.2 缓冲池的结构

内核按最近最少使用算法把数据缓存于缓冲池中：在它把一个缓冲区分配给磁盘块之后，只要不是所有其他缓冲区都在更近的时间内被使用过了，它就不能让另一磁盘块使用该缓冲区。内核维护一个缓冲区的空闲表，它保存被最近使用的次序。空闲表是缓冲区的双向链接循环表，具有一个哑缓冲区头标，以标志缓冲区空闲表的开始和结束（图 3-2）。当系统初启时，每个缓冲区都放到空闲表中。当内核想要一个任意的空闲缓冲区时，它从空闲表的头部取出一个缓冲区。但是，如果它标识出缓冲池中的一个特定块的话，它会从空闲表的中间取出一个缓冲区。在这两种情况下，它都从空闲表中摘下缓冲区。当内核把一个缓冲区还给缓冲区池时，它通常把该缓冲区附到空闲表的尾部，偶尔也将其附到空闲表的头部（在出错的情况下），但是从来都不插到中间。当内核从空闲表上不断地摘下缓冲区时，装有有效数据的缓冲区会越来越近地移动到空闲表的头部（图 3-2）。因此，离空闲表的头部近的缓冲区比离空闲表的头部远的缓冲区是最近最少使用的。

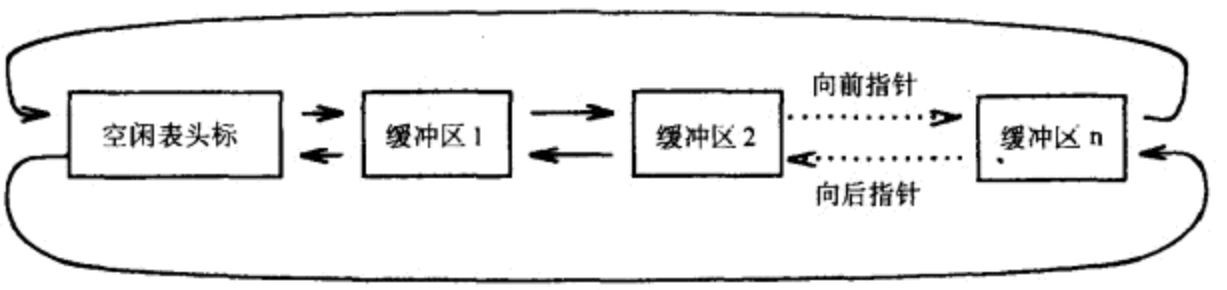


图 3-2 缓冲区的空闲表

当内核存取一个磁盘块时，它使用适当的设备号和块号的组合去找相应的缓冲区。它并不是去搜索整个缓冲区池。它把缓冲区组织成一个个队列，这些队列是以设备号和块号来散列的。内核把一个散列队列上的缓冲区链接成一个个类似于空闲表结构的双向链接循环表。正如将要看到的那样，一个散列队列上的缓冲区的数目在系统生存期间是变化的。内核必须使用一个散列函数，该散列函数把诸缓冲区均匀地分布在一组散列队列中。散列函数也必须简单，以便使性能不受损失。当生成操作系统时由系统的管理人员配置散列队列的数目。

图 3-3 示出了在若干散列队列上的缓冲区：散列队列的头标在图中的左部，每一行上的方块是散列队列上的缓冲区。因此，标记着 28、4、64 的方块代表“blkno 0 mod 4”这一散列队列上的那些缓冲区（以 4 取模后块号为 0）。缓冲区之间的虚线代表散列队列中向前与向后的指针，为简单起见，本章中后面的图将不再画出这些指针，但它们的存在是不言而喻的。类似地，该图仅用它们的块号来标识，并且仅使用依赖于块号的散列函数，然而实现中

也使用设备号。

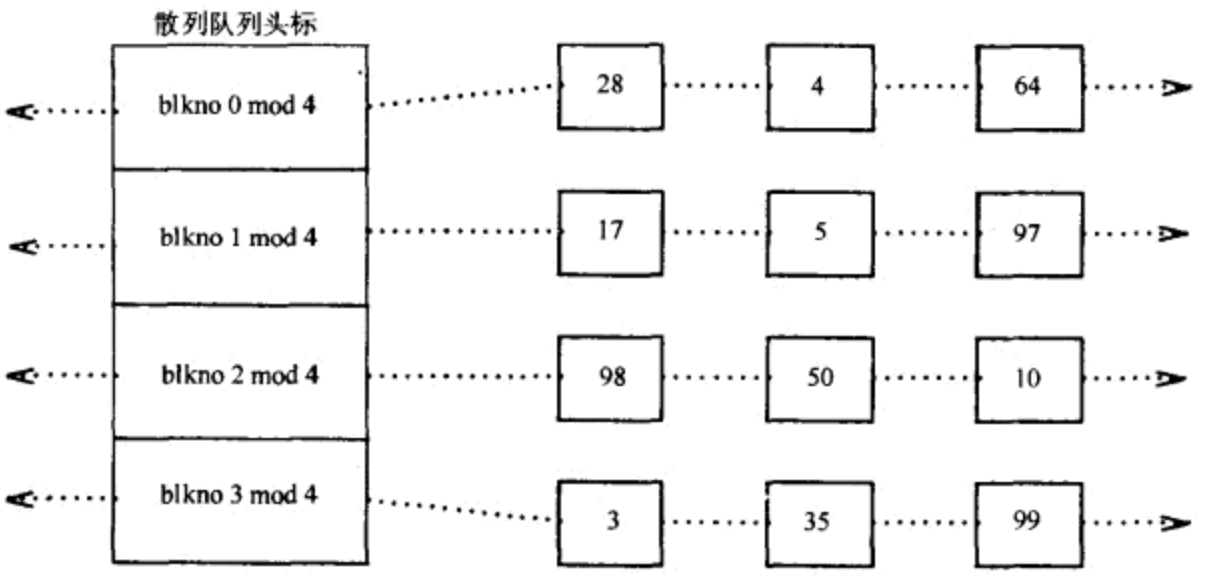


图 3-3　散列队列上的缓冲区

每个缓冲区总是存在于一个散列队列中，然而它位于队列上的什么位置是不重要的。正如前面陈述的那样，没有两个缓冲区可以同时包含同一磁盘块上的内容，因此，缓冲区池中的每个磁盘块，存在于且仅存在于一个散列队列中，并且仅在那个散列队列中呆一阵。然而，如果一个缓冲区为空闲状态，则它也可以在空闲表中。因为一个缓冲区可以同时既存在于一个散列队列中又存在于一个空闲队列中，所以内核有两个方法找到它：如果它要寻找一个特定的缓冲，则它搜索散列队列；如果它要寻找任何一个空闲缓冲区，则它从空闲表中摘下一个缓冲区。下一节将让大家看到内核如何在高速缓冲中找到特定的磁盘块，以及它如何操纵散列队列及空闲表中的缓冲区。概括地说，一个缓冲区总是在某个散列队列上，但是它可以在或不在空闲表中。

3.3　缓冲区的检索

正如在图 2-1 中所看到的那样，文件子系统中的高层内核算法引用管理高速缓冲的算法。当它们试图检索一个块时，由高层算法决定它们想要存取的逻辑设备号和块号。举例来说，正如在第 4 章将要看到的那样，如果一个进程想要从一个文件中读数据，则内核需判定哪一个文件系统包含该文件，以及该文件系统中的哪一块包含该数据。当要从一个特定的磁盘块上读数据时，内核检查是否该块在缓冲区池中。如果不在，则分配给它一个空闲缓冲区。当要把数据写到一个特定磁盘块上时，内核检查是否该块在缓冲区池中。如果不在，则为那个块分配一个空闲缓冲区。读、写磁盘块的算法使用算法 getblk（图 3-4）来对池中的缓冲区进行分配。

本节描述在算法 getblk 中内核把一个缓冲区分配给磁盘块时可能出现的五种典型情况：

（1）内核发现该块在它的散列队列中，并且它的缓冲区是空闲的。

（2）内核在散列队列中找不到该块，因此，它从空闲表中分配一个缓冲区。

（3）内核在散列队列中找不到该块，并且，在试图从空闲表中分配一个缓冲区（像第二种情况那样）的时候，在空闲表中找到一个已标上了“延迟写”标记的缓冲区。内核必须把

```
算法 getblk
输入:文件系统号
     块号
输出:现在能被磁盘块使用的上了锁的缓冲区
{
    while (没找到缓冲区)
    {
        if (块在散列队列中)
        {
            if (块忙)    /* 第五种情况 */
            {
                sleep(等候"缓冲区变为空闲"事件);
                continue;    /* 回到 while 循环 */
            }
            为缓冲区标记上"忙";    /* 第一种情况 */
            从空闲表上摘下缓冲区;
            return(缓冲区);
        }
        else    /* 块不在散列队列中 */
        {
            if(空闲表上无缓冲区)    /* 第四种情况 */
            {
                sleep(等候"任何缓冲区变为空闲"事件);
                continue;    /* 回到 while 循环 */
            }
            从空闲表上摘下缓冲区;
            if (缓冲区标记着延迟写)    /* 第三种情况 */
            {
                把缓冲区异步写到磁盘上;
                continue;    /* 回到 while 循环 */
            }
            /* 第二种情况——找到一个空闲缓冲区 */
            从旧散列队列中摘下缓冲区;
            把缓冲区投入新散列队列;
            return(缓冲区);
        }
    }
}
```

图 3-4 缓冲区分配算法

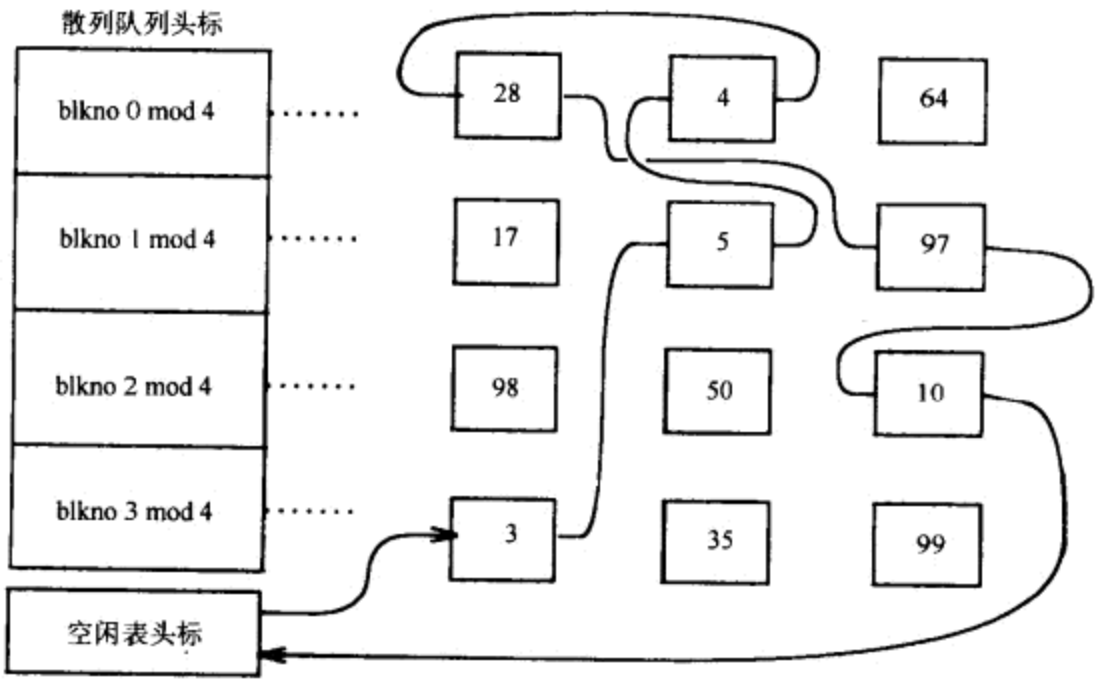
"延迟写"缓冲区的内容写到磁盘上，并分配另一个缓冲区。

(4) 内核在散列队列中找不到该块，并且空闲缓冲区表已空。

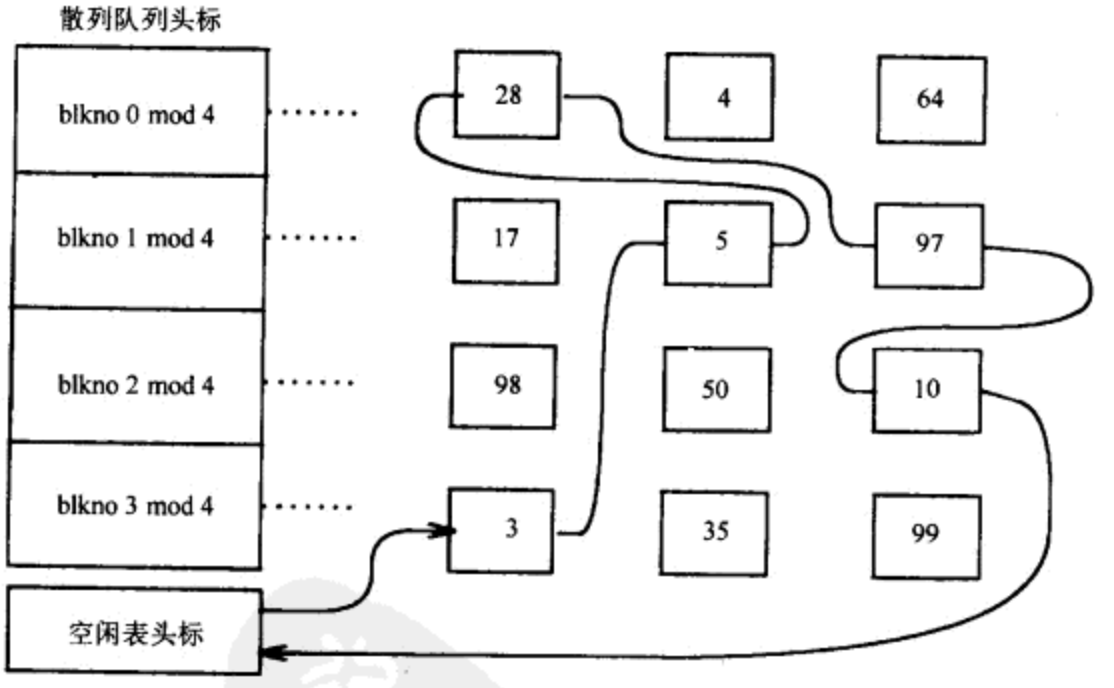
(5) 内核在散列队列中找到该块，但它的缓冲区当前为忙。现在，让我们更详细地讨论每种情况。

当根据设备号和块号的组合在缓冲池中搜索一个块时，内核找到含有该块的散列队列。

它沿着缓冲区链表搜索散列队列，直到（在第一种情况中）它找到了其设备号和块号与所要搜索的设备号和块号相匹配的缓冲区。内核检查该缓冲区是否空闲。如果是，则将该缓冲区标记上“忙”以使其他进程[⊖]不能存取它。然后内核从空闲表上摘下该缓冲区，因为一个缓冲区不能既处于忙状态又位于空闲表中。正如所见到的那样，如果当缓冲区忙时其他进程企图存取该块，则它们去睡眠，直至该缓冲区被释放时为止。图 3-5 描绘了第一种情况。在那



a)



b)

图 3-5　缓冲区分配的第一种情况

a）在第一个散列队列上搜索第 4 块　b）从空闲表上摘下第 4 块

⊖ 回忆上一章中正在核心态下执行的一个进程的上下文中所做的所有内核操作。因此，术语“其他进程”意味着它们也正在核心态下执行。当描述正在核心态下执行的几个进程间的交互时将使用这一术语。如果没有进程间的交互，则将使用“内核”这一术语。

儿，内核在标记有“blkno 0 mod 4”的队列上搜索第4块。找到该缓冲区后，内核把它从空闲表中摘下，而将第5块和第28块相邻接并留在空闲表上。

在继续谈另外几种情况之前，让我们考虑在把一个缓冲区分配了之后对它会发生什么。内核可以把数据从磁盘读至缓冲区，并进而操纵它，或把数据写到缓冲区，并且可能进而写到磁盘上。内核把标有“忙”的缓冲区留在那里，当它忙时，没有别的进程能存取它以及改变它的内容，因此可保持该缓冲区中的数据的完整性。当内核结束使用该缓冲区时，按照算法 brelse（图3-6）释放该缓冲区。它唤醒那些因该缓冲区忙而睡眠的进程，也唤醒由于空闲表上没有缓冲区而睡眠的那些进程。这两类唤醒意味着被投入睡眠的那些进程现在可以使用释放了的缓冲区了——虽然得到该缓冲区的第一个进程锁上了它并且阻止其他进程得到它（回忆2.2，2.4节）。内核把该缓冲区放到空闲表尾部；但是，如果发生了一个I/O错误或者内核明确地在该缓冲区上标记上“旧”，则内核把该缓冲区放到空闲表头部——本章的后面部分将看到这一点。现在，该缓冲区对于要索取它的所有进程都是空闲的。

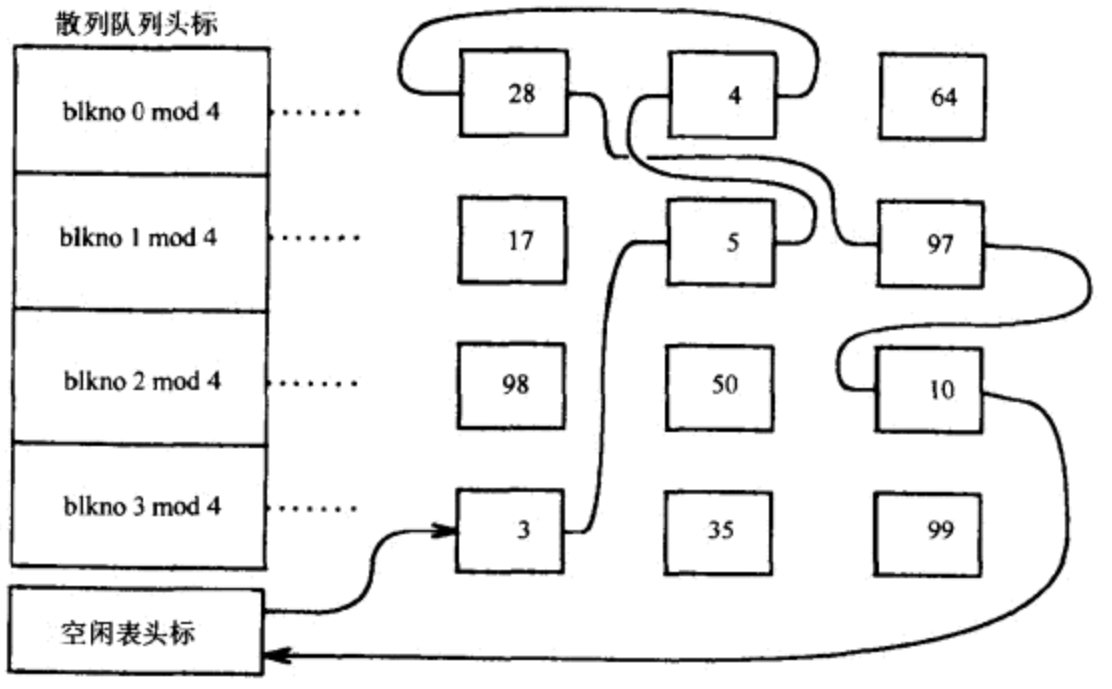
```
算法 brelse
输入:上锁态的缓冲区
输出:无
{
    唤醒正在等待“无论哪个缓冲区变为空闲”这一事件发生的所有进程;
    唤醒正在等待“这个缓冲区变为空闲”这一事件发生的所有进程;
    提高处理机执行级以封锁中断;
    if (缓冲区内容有效且缓冲区非“旧”)
        将缓冲区送入空闲表尾部
    else
        将缓冲区送入空闲表头部
    降低处理机执行级以允许中断;
    给缓冲区解锁;
}
```

图3-6 释放缓冲区算法

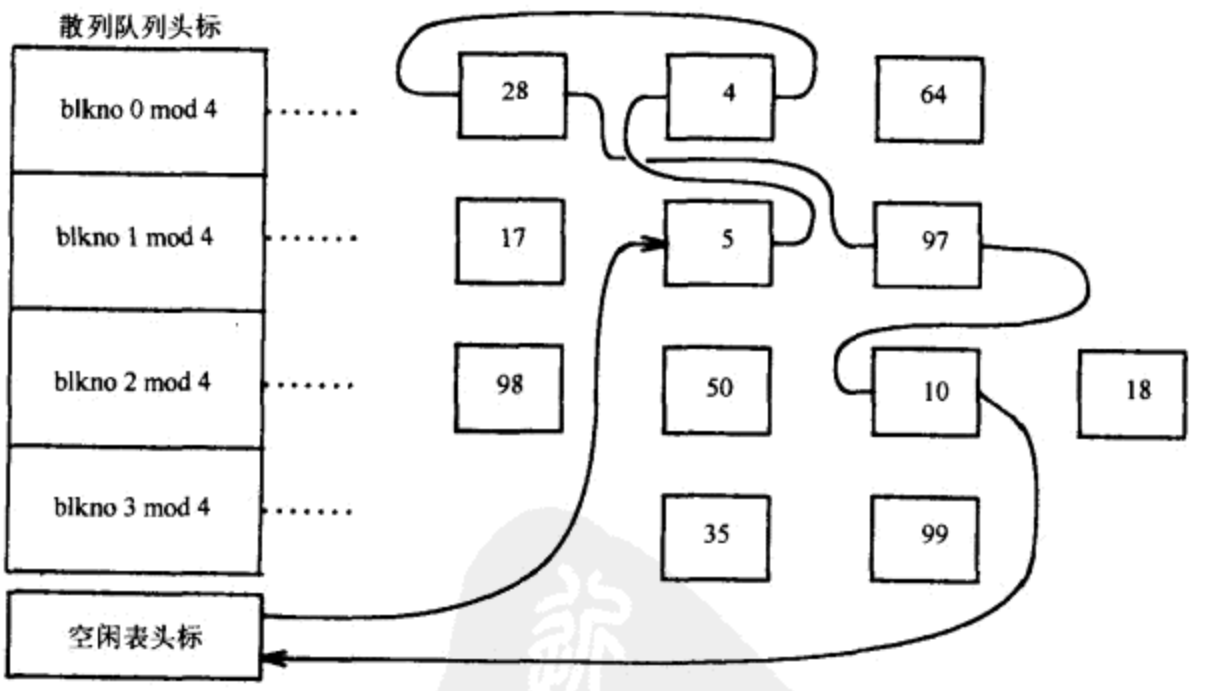
正如当内核不再需要一个缓冲区时它引用算法 brelse 一样，在处理一个磁盘中断过程中，释放用于往磁盘写和从磁盘读的异步I/O的缓冲区时，也要引用该算法。这将在3.4节看到。当操纵空闲表时内核把处理机执行级提高以禁止磁盘中断，从而防止了由于嵌套调用 brelse 而引起的缓冲区指针的错误。如果当一个进程正在执行 getblk 时，一个中断处理程序引用 brelse，则也会发生类似的坏结果。因此，在 getblk 中对全局有重要意义的地方也要提高处理机执行级。本章的练习详细探讨了这些情况。

在算法 getblk 中的第二种情况下，内核搜索应该包含该块的那个散列队列，但在那个散列队列上找不到该块。因为它不能“散列”在别处，所以该块不可能在另一个散列队列上，于是可以判定它不在高速缓冲中。因此，内核从空闲表中摘下第一个缓冲区。该缓冲区曾分配给另一个磁盘块，并且也正在某个散列队列上。如果该缓冲区没有被标记上延迟写（象随后将要描述的那样），则内核标记该缓冲区为忙，将它从当前所在的散列队列中摘出，把该缓冲头部的设备号和块号重新指定为当前进程正在搜索的磁盘块的设备号和块号，并且把该

缓冲区放到正确的散列队列中。内核使用该缓冲区，但是并未记录该缓冲区从前包含有另一个磁盘块的数据。如果这时有一个进程搜索旧磁盘块，则它在池中将找不到这个旧磁盘块，它必须严格地按照前面所描述的那样，从空闲表中为这个旧磁盘块分配一个新缓冲区。当内核结束对该缓冲区的使用时，它按如上所描述的那样把该缓冲区释放。例如，在图 3-7 中，内核搜索第 18 块但是在标有“blkno 2 mod 4”的散列队列上没有找到它，因此，内核从空闲表上摘下第一个缓冲区(第 3 块)，将它分配给第 18 块，并把它放到适当的散列队列中。



a)



b)

图 3-7　缓冲区分配的第二种情况

a) 搜索第 18 块——不在高速缓冲中　b) 从空闲表中摘下第一个缓冲区，分配给第 18 块

在算法 getblk 的第三种情况下，内核也必须从空闲表中分配一个缓冲区。然而，它发现，从空闲表中摘下的缓冲区已被标记上“延迟写”，因此，它必须在使用该缓冲区之前，

将该缓冲区的内容写到磁盘上。内核开始了一个往磁盘的异步写，并且试图从空闲表上分配另一个缓冲区。当异步写完成时，内核把该缓冲区释放，并把它放到空闲表头部。在异步写之前，该缓冲区已经从空闲表尾部出发迁移到了空闲表头部。假使在异步写之后，内核把该缓冲区放到空闲表尾部，则该缓冲区会做一次多余的贯穿空闲表的移动，这就与最近最少使用算法相违背了。举例来说，在图 3-8 中，内核没有找到第 18 块，但是，当它试图分配空闲表上的头两个缓冲区（每次一个）时，发现它们标记着延迟写。内核把它们从空闲表中摘下，为这两块启动往磁盘上写的操作。内核分配空闲表上块号为 4 的第三个缓冲区。它适当地重新对该缓冲区的设备号和块号字段赋值，并且把该缓冲区——现在就是标记着第 18 块的缓冲区放到它的新的散列队列中。

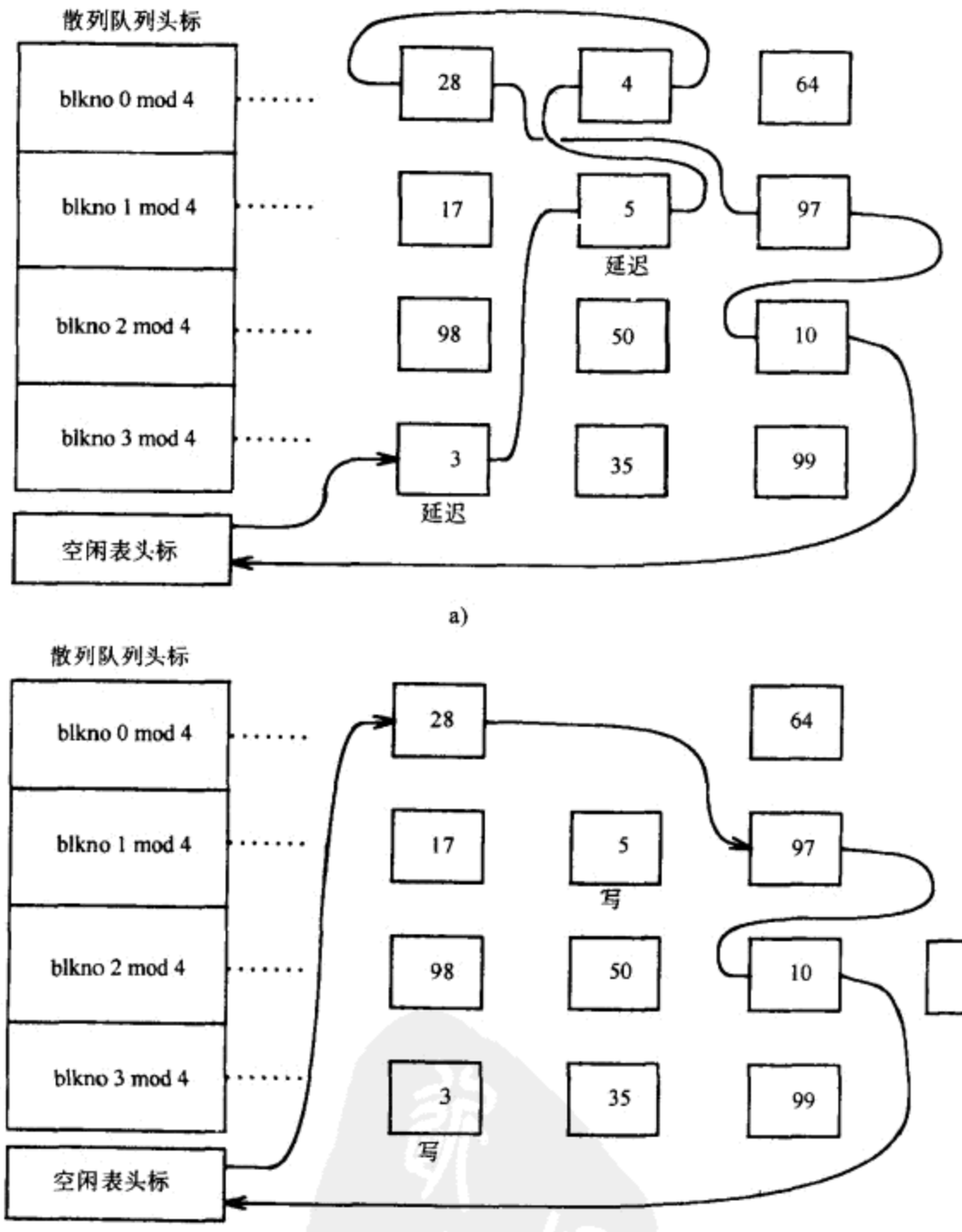


图 3-8　缓冲区分配的第三种情况

a）搜索第 18 块，空闲表上头两个缓冲区标记着延迟写

b）写第 3 块、第 5 块，把原分配给第 4 块的缓冲区改为分配给第 18 块

第四种情况（图 3-9）下，为进程 A 而动作着的内核没有在它的散列队列上找到该磁盘块，因此，像第二种情况那样，它试图从空闲表上分配一个新的缓冲区。然而，在空闲表上已没有缓冲区可供分配，因此，进程 A 去睡眠直至另一进程执行算法 brelse，释放一个缓冲区。当内核调度到进程 A 时，它必须为该块重新搜索散列队列。它不能立即从空闲表中分配缓冲区，因为可能有多个进程正在等候空闲缓冲区，并且可能其中有一个进程已经把一个新的空闲缓冲区分配给进程 A 所寻找的目标磁盘块了。因此，需要再次搜索该块，保证仅仅一个缓冲区包含有该块。图 3-10 描绘了两个进程间为一个空闲缓冲区而进行的竞争。

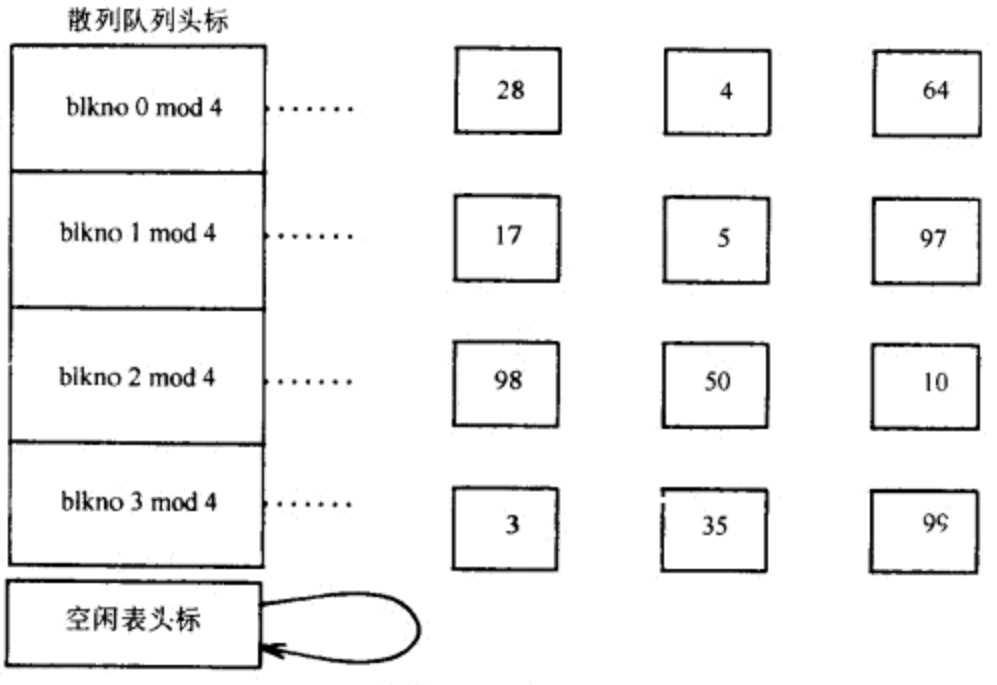


图 3-9　缓冲区分配的第四种情况

最后一种情况（图 3-11）是复杂的，因为它牵涉到几个进程间的复杂关系。假设内核正在为进程 A 动作，搜索一个磁盘块，并分配一个缓冲区，但是在释放该缓冲区之前去睡眠了。举例来说，如果进程 A 试图读一磁盘块并且象第二种情况那样分配了一个缓冲区，则当它等候从磁盘上的 I/O 传输完成时，它将去睡眠。当进程 A 睡眠时，假设内核调度到第二个进程 B，B 试图存取那个缓冲区刚刚被进程 A 锁上了的磁盘块。进程 B（进入到第五种情况）将在散列队列上找到那个上了锁的块。因为使用上着锁的缓冲区是非法的，并且为一个磁盘块分配第二个缓冲区也是非法的，所以进程 B 在缓冲区上打上“有需求”的标记，然后去睡眠，等候进程 A 释放该缓冲区。

进程 A 将最终释放该缓冲区并注意到该缓冲区有需求者。它唤醒在事件“缓冲区变为空闲”上睡眠的所有进程，其中包括进程 B。当内核再次调度到进程 B 时，进程 B 必须证实该缓冲区是空闲的。另一进程 C，可能一直等待同一个缓冲区；并且内核可能先于进程 B 而调度到 C 去运行；进程 C 可能已经去睡眠，而该缓冲区仍是上了锁的。因此，进程 B 必须确认该块真的是空闲的。

进程 B 也必须验证该缓冲区是否包含着它原来请求的磁盘块，因为可能会像第二种情况那样，进程 C 已经把该缓冲区分配给另一块了。当进程 B 执行时，它可能发现它那时正

等候着错误的缓冲区，因此它必须再次搜索该磁盘块；假若它自动地从空闲表中分配一个缓冲区，那么它将察觉不出另一个进程刚把一个缓冲区分配给该块的可能性。

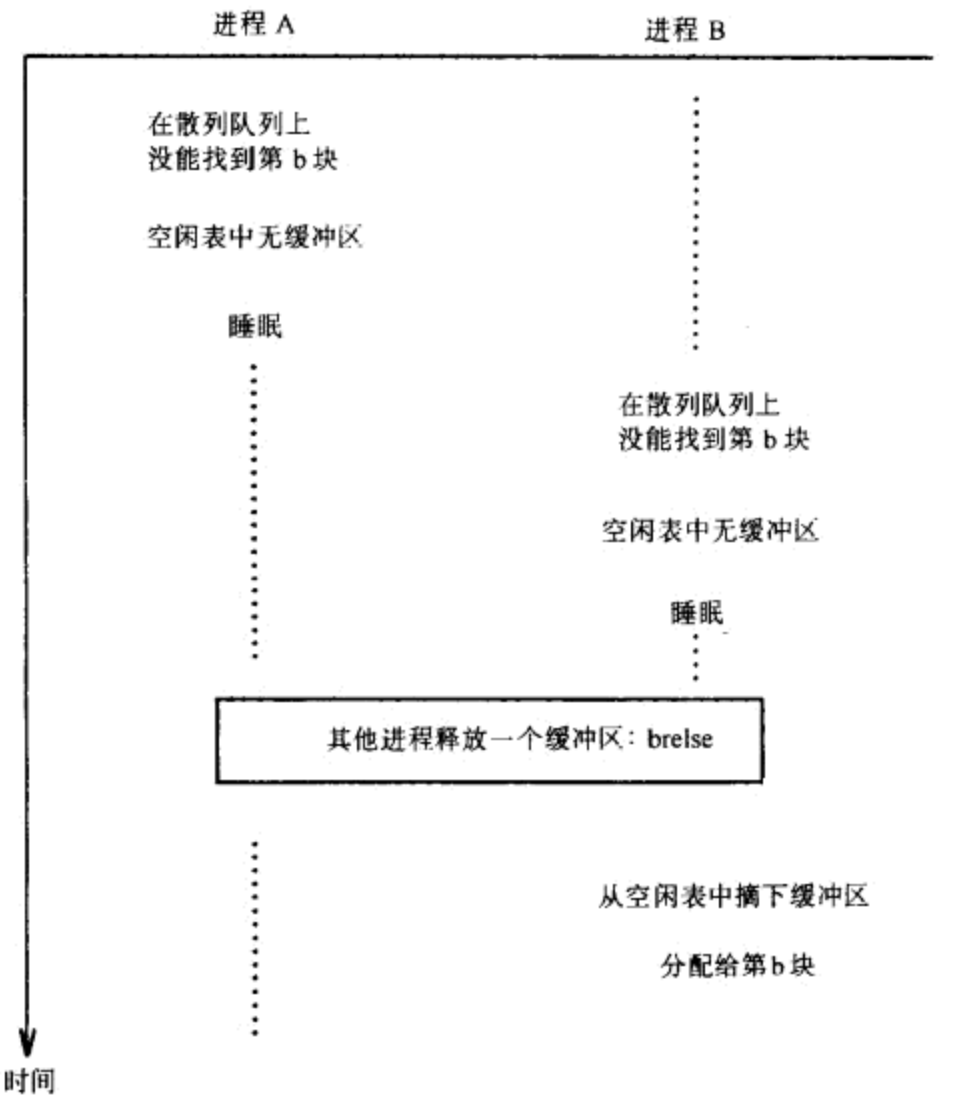


图 3-10　竞争缓冲区

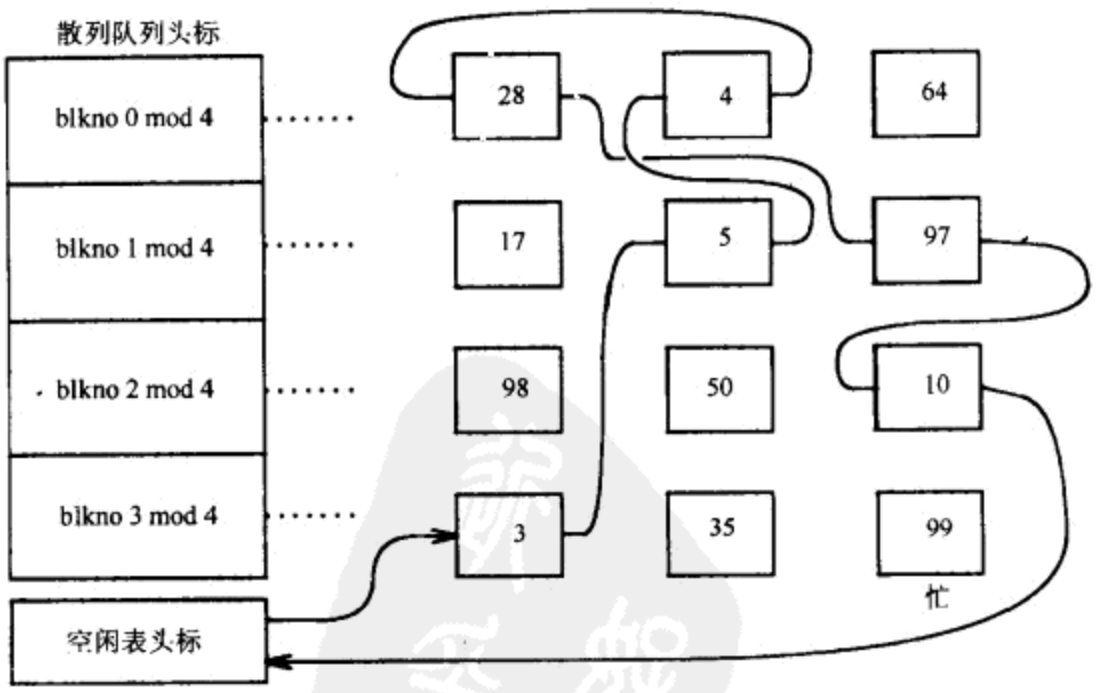


图 3-11　缓冲区分配的第五种情况

最后，进程 B 将找到它的块，可能像第二种情况那样从空闲表中分配一个新缓冲区。例如，在图 3-11 中，一个正在搜索第 99 块的进程在它的散列队列上找到了它，但是该块被标记为忙。该进程睡眠直至该块变为空闲，并且随后重新开始该算法。图 3-12 描绘了对一个上了锁的缓冲区进行的竞争。

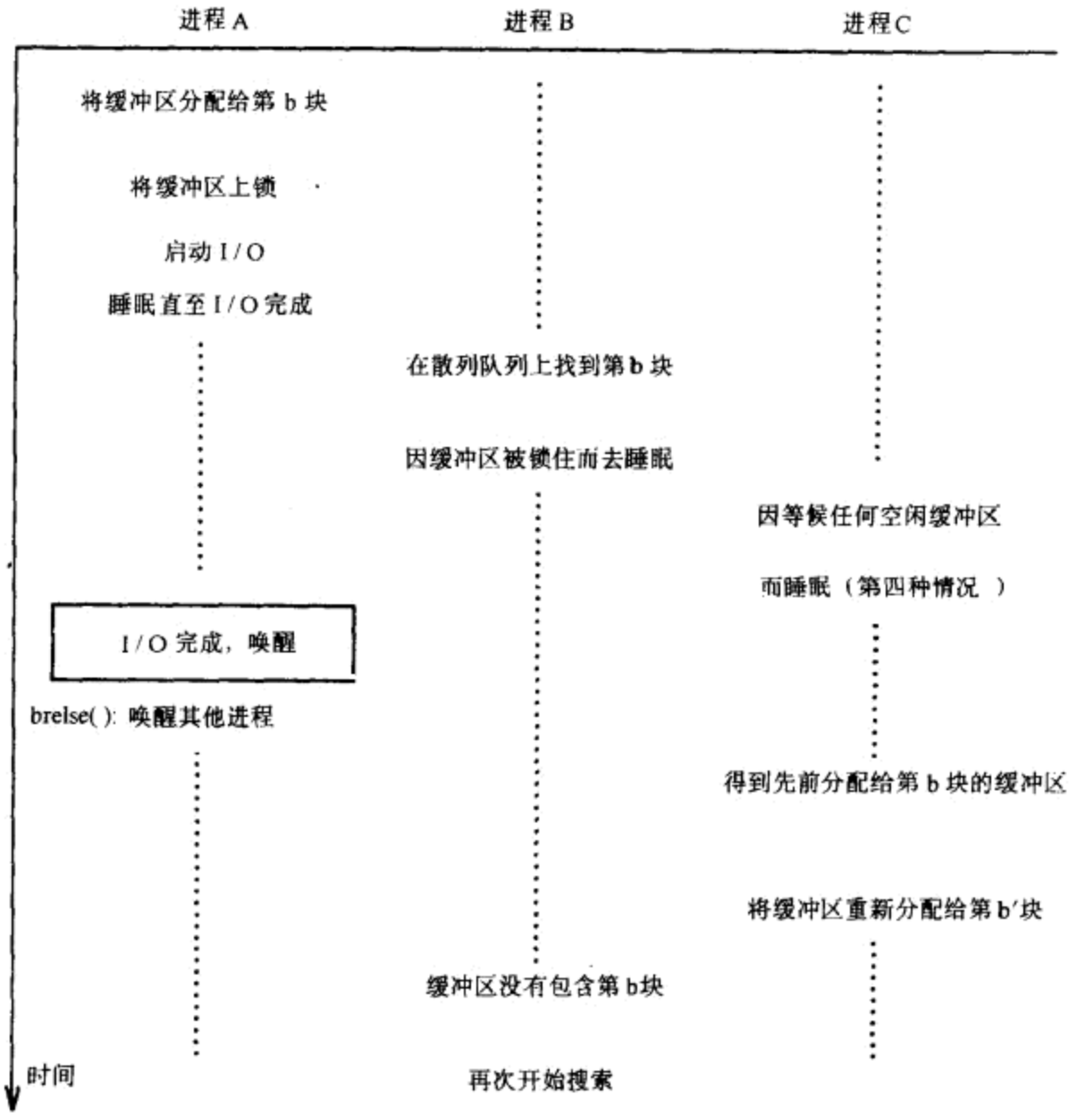


图 3-12　竞争一个上了锁的缓冲区

缓冲区分配的算法必须是安全的：必须使进程不永远睡眠，必须使它们最终能得到一个缓冲区。因为内核在系统调用执行期间分配缓冲区，并在返回之前释放它们[⊖]，所以它保证等候缓冲区的所有进程都能醒来。在用户态下的进程不直接控制内核缓冲区的分配，因此，它们不能故意地“霸占”缓冲区。仅当内核等待着一个缓冲区与磁盘之间的 I/O 完成时，内核才失去对这个缓冲区的控制。可以想象得到，若一个磁盘驱动器是讹误的，造成它不能中断 CPU，这就使内核总是不能释放该缓冲区。磁盘驱动程序必须对硬件发生这样的情况进行监视，并返回一个错误给内核，说明磁盘工作不正常了。简言之，内核能保证正在因一

⊖ 系统调用 mount 是个例外，因为它分配一个缓冲区直到以后的 umount 调用为止。这一例外不是关键的，因为缓冲区总数远远超过了活动的被安装的文件系统的数目。

个缓冲区而睡眠的进程最终能被唤醒。

也可想象到这样的情况是可能的：一个进程总也不能存取一个缓冲区。比如，在第四种情况下，如果几个进程因等待一个缓冲区变为空闲而睡眠，内核不保证它们能按它们请求的次序获得缓冲区。当一个缓冲区变为空闲时，一个进程可能继续睡眠，也可能被唤醒，因为当其他进程首先获得了对该缓冲区的控制权时，它只能再次去睡眠。从理论上说，这种情形可能会永远继续下去，但在实际上，由于系统中一般都配置了很多缓冲区，所以这是不成问题的。

3.4　读磁盘块与写磁盘块

既然已叙述了缓冲区分配算法，那么，读磁盘块与写磁盘块的过程就应该很容易理解了。为了读一个磁盘块（图3-13），进程使用算法 getblk 在高速缓冲中搜索这个磁盘块。如果它就在高速缓冲中，则内核可不必物理地从磁盘上读该块，就能立即将它返回。如果它不在高速缓冲中，则内核调用磁盘驱动程序，以“安排”一个读请求，而后去睡眠，等候 I/O 完成事件的发生。磁盘驱动程序通知磁盘控制硬件，说它想要读取数据。磁盘控制器过一会儿将数据传送到该缓冲区。最后，当 I/O 完成时，磁盘控制器中断处理机，由磁盘中断处理程序唤醒正在睡眠的进程。现在，磁盘块的内容已在缓冲区中了。请求这一特定磁盘块的那些模块现在拥有该数据；当它们不再需要该缓冲区时，它们释放该缓冲区，以便其他进程能存取它。

```
算法　bread　　/*　读块　*/
输入：文件系统块号
输出：含有数据的缓冲区
{
    得到该块的缓冲区（算法 getblk）；
    if （缓冲区数据有效）
        return（缓冲区）；
    启动磁盘读；
    sleep（等待“读盘完成”事件）；
    return（缓冲区）；
}
```

图 3-13　读磁盘块算法

第5章中我们将看到，当一个进程顺序地读一个文件时，较高层次的内核模块（比如文件子系统）可能会预期到对另一个磁盘块的需要，因而该模块异步地请求第二个 I/O，希望一旦需要这部分数据时，这部分数据将在主存中，这样可以改善性能。为了做到这点，内核执行提前读磁盘块算法 breada（图 3-14）；内核检查第一块是否在高速缓冲中，并且，如果它不在，则请求磁盘驱动程序读该块。如果第二块不在高速缓冲中，则内核指示磁盘驱动程序异步地读它。然后该进程去睡眠以等候“第一个块的 I/O 完成”事件的发生。当它醒来时，它返回第一块的缓冲区，并不关心第二块的 I/O 何时完成。当第二个块的 I/O 完成时，磁盘控制器向系统发中断，中断处理程序识别出该 I/O 是异步的，并释放该缓冲区（算法 brelse）。如果它不释放该缓冲区，则该缓冲区保持上锁状态，从而对于所有的进程它都是不

可存取的。事先为缓冲区解锁是不可能的，因为对该缓冲区的 I/O 是活动的，从而该缓冲区的内容是无效的。随后，如果该进程想要读第二块，那么若 I/O 在此期间已经完成，则应在高速缓冲中找得到第二块。如果在 breada 开始时第一块已在高速缓冲中，则内核立即检查第二块是否也在高速缓冲中，并且按以上描述的那样进行。

```
算法 breada    /* 读块与提前读 */
输入:(1)立即读的文件系统块号
     (2)异步读的文件系统块号
输出:含有可立即读的数据的缓冲区
{
    if (第一块不在高速缓冲中)
    {
        为第一块获得缓冲区(算法 getblk);
        if (缓冲区数据无效)
            启动磁盘读;
    }
    if (第二块不在高速缓冲中)
    {
        为第二块获得缓冲区(算法 getblk);
        if (缓冲区数据有效)
            释放缓冲区(算法 brelse);
        else
            启动磁盘读;
    }
    if (第一块本来就在高速缓冲中)
    {
        读第一块(算法 bread);
        return(缓冲区);
    }
    sleep(第一个缓冲区包含有效数据的事件);
    return(缓冲区);
}
```

图 3-14　提前读块算法

把一个缓冲区的内容写到一个磁盘块上去的算法是类似的(图 3-15),内核通知磁盘驱动程序说它有一个缓冲区的内容应该被输出。磁盘驱动程序调度到该块以便进行 I/O。如果写是同步的,则调用者进程进入睡眠等候 I/O 完成,并且当它醒来时释放该缓冲区。如果写是异步的,则内核开始磁盘写,但是不等待写完成。当 I/O 完成时,内核将释放该缓冲区。

在某些场合下,内核并不立即地把数据写到磁盘上,这将在下面两章中描述。如果它执行这样的一个“延迟写”,则它相应地为该缓冲区做标记,使用 brelse 算法释放该缓冲区,并且不调度 I/O 就继续向下执行。在别的进程可能把该缓冲区重新分配给另一个块之前,内核要把该块写到磁盘上,如同我们在 getblk 的第三种情况中所描述的那样。在此期间内,内核希望有进程在该缓冲区必须被写到磁盘上之前存取该块。如果那个进程后来又改变了该缓冲区的内

容,则内核节省了一次额外的磁盘操作。

```
算法 bwrite    /* 写块 */
输入:缓冲区
输出:无
{
    启动磁盘写;
    if (I/O同步)
    {
        sleep(等待'I/O完成'事件);
        释放缓冲区(算法 brelse);
    }
    else if (缓冲区标记着延迟写)
            为缓冲区做标记以放到空闲表头部;
}
```

图 3-15 写磁盘块算法

延迟写不同于异步写。当做一个异步写时，内核立即开始这一写磁盘操作，但是不等候它的完成。对于“延迟写”，内核则尽可能长时间地推迟物理地往磁盘上写。那么，再回忆一下算法 getblk 中的第三种情况，它把缓冲区标记为“旧”，并把该块异步写到磁盘上。随后，磁盘控制器中断系统，并使用算法 brelse 释放该缓冲区。由于该缓冲区是“旧”的，所以该缓冲区到达空闲表的头部时结束。因为有提前读与延迟写磁盘块这两个异步 I/O 操作，所以才发生内核从一个中断处理程序中调用 brelse 的情形。由于 brelse 要把缓冲区放到空闲表上，所以它必须在对缓冲区空闲表进行操作的任何过程中都禁止中断。

3.5 高速缓冲的优点与缺点

高速缓冲的使用有某些优点，也有某些缺点。

• 缓冲区的使用提供了统一的磁盘存取方法，因为内核不需要知道 I/O 的原因。不论数据是文件的一部分，还是一个索引节点，还是超级块的一部分，内核都是往缓冲区或从缓冲区拷贝数据。磁盘 I/O 的缓冲技术使代码更加模块化，因为内核中进行磁盘 I/O 的那些部分，对于所有的目的都使用同一个接口。简而言之，系统设计较为简单。

• 系统对进行 I/O 的用户进程没有做数据对齐限制，因为内核在内部实现了数据对齐。硬件实现常常需要对磁盘 I/O 的特别的数据对齐，例如，使主存中的数据按两字节边界对齐，或按四字节边界对齐等等。若不用缓冲区机制，则程序员必须保证他们的数据缓冲区是正确地对齐了的。这将会产生很多程序员的错误，并且程序不能移植到运行在具有严格的地址对齐性质的机器的 UNIX 系统中。通过把数据从用户缓冲区拷贝到系统缓冲区（并且反之亦然），内核消除了对用户缓冲区的特殊对齐的需要，使得用户程序较为简单，且易于移植。

• 高速缓冲的使用可减少访盘次数，从而提高整个系统的吞吐量，减少响应时间。欲从文件系统中读数据的进程可以在高速缓冲中找到数据块，从而避免了对磁盘 I/O 的需要。内核经常使用延迟写以避免不必要的磁盘写，把该块留在高速缓冲中，并希望高速缓冲命中该块。显然，对于具有很多缓冲区的系统来说高速缓冲命中的机会是较大的。然而，一个系

统能够有益地配置的缓冲区的数目受到主存总量的限制——它必须保证正在执行的诸进程有够用的主存。如果缓冲区占用了过多的主存，则系统会由于过量的进程对换或调页而慢下来。

• 缓冲区算法有助于确保文件系统的完整性，因为它们维护一个公共的、包含在高速缓冲中的磁盘块的单一映象。如果两个进程同时试图操纵一个磁盘块，则缓冲区算法（比如 getblk）把它们的存取按顺序排列，防止数据信息被破坏。

• 访盘次数的减少对于良好的吞吐量与响应时间来说是重要的，但是高速缓冲策略也引进了一些缺点。由于延迟写使得内核没有立即地把数据写到磁盘上，所以，当系统发生瘫痪使磁盘数据处于错误状态时，系统显得无能为力。虽然最近的系统实现已经减少了由于灾难性事件引起的破坏，但仍然留下了一个基本问题：发出一个写系统调用的用户从来不能确定这些数据到底什么时候真正地写到磁盘上了[⊖]。

• 高速缓冲的使用，使得当往用户进程中读或从用户进程写时需要一次额外的数据拷贝过程。写数据的进程把数据拷贝到内核，内核把数据拷贝到磁盘上；读数据的进程则把数据从磁盘读进内核，再从内核读到用户进程。当传输的数据量很大时，这种过量的拷贝使性能减慢，但是当传输的数据量小时，它改进了性能——因为内核（使用算法 getblk 及延迟写）把数据缓冲起来，直至它认为往磁盘写或从磁盘读是合算的时候。

3.6　本章小结

本章介绍了高速缓冲的结构，及内核寻找高速缓冲中的磁盘块的各种方法。缓冲区算法把几个简单思想结合起来以提供一个高级的高速缓冲机制。内核采用最近最少使用替换算法，把块保持在高速缓冲中，这是基于最近被存取过的块可能很快就会被再次存取的假设。缓冲区在空闲表中出现的次序指明了它们最近一次被使用的次序。其他缓冲区替换算法，比如先进先出算法或最不经常使用算法等，要么在实现上更复杂，要么导致高速缓冲命中率降低。散列函数和散列队列能使内核很快找到特定的块，双向链表的使用使它易于从表中摘下缓冲区。

内核通过提供一个逻辑设备号与块号来标识它们需要的块。算法 getblk 搜索高速缓冲中的一个块，并且，如果该缓冲区存在且空闲，则将该缓冲区上锁并返回它。如果该缓冲区已处于上锁状态，则请求者进程去睡眠直至该块变为空闲。锁机制保证在任一时刻仅有一个进程操纵一个缓冲区。如果该块不在高速缓冲中，则内核重新分配一个缓冲区给该块，将它上锁，并且返回它。算法 bread 为一个块分配一个缓冲区，并且把数据读入该缓冲区——如果需要的话。算法 bwrite 把数据拷贝到一个预先分配好了的缓冲区中。在某些高层次算法的执行中，如果内核决定不需要把数据立即拷贝到磁盘上，则它为该缓冲区打上“延迟写”的标记，以避免不必要的 I/O。不幸的是，“延迟写”方案意味着一个进程从来也不能断定数据究竟何时物理地记录到磁盘上。如果内核同步地往磁盘上写数据，则它调用磁盘驱动程序把该块写到该文件系统中，并等候 I/O 完成中断。

内核以多种方式使用高速缓冲。内核通过高速缓冲在应用程序与文件系统之间传输数据，

⊖ C 语言程序可使用的标准 I/O 程序包中含有一个函数 fflush。该函数调用使数据从用户地址空间的缓冲区流到内核。然而，用户仍然不知道什么时候内核把数据写到磁盘上。

并且它在高层内核算法与文件系统之间传输像索引节点这样的辅助系统数据。当把程序读入主存以备执行时，它也使用高速缓冲。下章将要描述的很多算法都使用本章中描述的过程。把索引节点和主存页予以高速缓冲的其他算法也使用了与本章所描述的高速缓冲算法类似的技术。

3.7 习题

1. 考虑图 3-3 中的散列函数。最好的散列函数是使块均匀分布在散列队列集合上的散列函数。什么是最理想的散列函数？散列函数在它的计算中应该使用逻辑设备号吗？

2. 在算法 getblk 中，如果内核要从空闲表中摘下一个缓冲区，则它必须提高处理机优先级，以便在检查空闲表之前封锁中断，为什么？

*3. 在算法 getblk 中，在检查一个块是否正处于忙状态之前，内核必须提高处理机优先级以封锁中断（在正文中这一点没有表示出来）。为什么？

4. 在算法 brelse 中，如果缓冲区内容无效，则内核把该缓冲区加入到空闲表队列头部。如果缓冲区内容无效，该缓冲区应该出现在散列队列中吗？

5. 假设内核进行一个块的延迟写，当另一个进程从它的散列队列中取那个块时会发生什么？从空闲表中呢？

*6. 如果几个进程竞争一个缓冲区，内核担保没有一个进程永远睡眠，但是它不保证进程不会总也得不到使用缓冲区的机会。重新设计 getblk 以保证一个进程最终能用上一个缓冲区。

7. 重新设计算法 getblk 与 brelse，使内核不遵循“最近最少使用”方案，而遵循“先进先出”方案。使用“最不经常使用”方案重复做这一题。

8. 描述一种在算法 bread 中的缓冲区数据已经有效的情况。

*9. 描述算法 breada 中可能发生的各种情况。当前的提前读块将被读时，下一个 bread 或 breada 的引用将发生什么？在算法 breada 中，如果第一个或第二个块不在高速缓冲中，则随后测试该缓冲区数据是否有效时，就意味着该块能在该缓冲区池中。这是怎么成为可能的？

10. 描述一个从缓冲区池请求及接收任何一个空闲缓冲区的算法。将这一算法与 getblk 比较。

11. 像 umount 与 sync（第 5 章）这样几个不同的系统调用，请求内核把对一个特定的文件系统“延迟写”的所有缓冲区都清理到磁盘上。描述一个实现缓冲区清理的算法。作为清理操作的结果，对于空闲表上的缓冲区次序发生了什么？内核怎样才能保证当清理进程因等候一次 I/O 完成而睡眠时，没有别的进程偷偷溜进来用延迟写的方法把一个缓冲区写到这个文件系统中？

12. 把系统响应时间定义为它完成一个系统调用所占用的平均时间。把系统吞吐量定义为在一个给定时间段内系统所能执行的进程数目。描述高速缓冲能够怎样有助于缩短响应时间。它必定有助于提高系统吞吐量吗？

第4章　文件的内部表示

正如在第2章中所看到的那样，UNIX系统中的每个文件都有一个唯一的索引节点。索引节点包含着为进程存取文件所必需的信息，如文件所有者、存取权限、文件长度、文件数据在文件系统中的位置。进程通过一组明确定义的系统调用来存取文件，并且通过一个字符串即路径名来指明一个文件。每个路径名都唯一地指明一个文件，内核把路径名转换成文件的索引节点。

本章描述UNIX系统中文件的内部结构，下一章描述对文件的系统调用接口。4.1节考察索引节点以及内核如何操纵索引节点；4.2节考察正规文件的内部结构及内核怎样读、写它们的数据；4.3节研究目录结构，这是使内核能把文件系统组织成文件树的那些文件；4.4节介绍把用户文件名转换成索引节点的算法；4.5节给出了超级块结构；4.6节和4.7节介绍了向文件分配磁盘索引节点和磁盘块的算法；最后，4.8节谈论了本系统中的其他文件类型——管道与设备文件。

本章中描述的算法所在的层次居于上一章所解释的高速缓冲算法之上（图4-1）。算法iget返回一个先前标识了的索引节点，它可能是经由高速缓冲而从磁盘上读出的。算法iput释放索引节点。算法bmap为存取一个文件设置内核参数。算法namei使用算法iget、iput及bmap把一个用户级路径名转换成一个索引节点。算法alloc与free为文件分配及释放磁盘块，而算法ialloc与ifree为文件分配及释放索引节点。

较低层次的文件系统算法

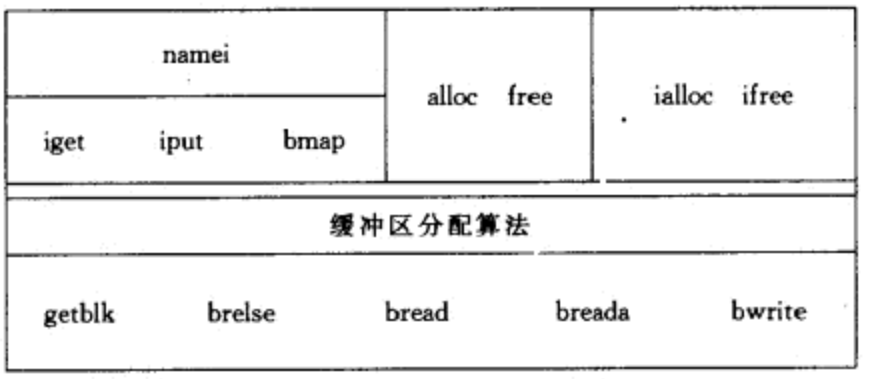


图4-1　文件系统算法

4.1　索引节点

4.1.1　定义

索引节点以静态形式存在于磁盘上，内核把它们读进内存索引节点表中以便操纵它们。磁盘索引节点由如下字段组成：

• 文件所有者标识号。所有权定义了对一个文件具有存取权的用户集合，分为个体所有与“用户组”所有。超级用户具有对系统中所有文件的存取权。

• 文件类型。文件可以是正规类型、目录类型、字符类型或块特殊类型，或 FIFO（管道）类型的文件。

• 文件存取许可权。系统按如下三个类别对文件施行保护：文件所有者、文件的用户组所有者及其他用户。每类都具有读、写、执行该文件的存取权，并且能分别地设置。由于目录不能执行，所以对一个目录的执行许可权实际上给出了为一个文件名搜索该目录的权力。

• 文件存取时间。给出了文件最后一次被修改的时间，最后一次被存取的时间，最后一次被修改索引节点的时间。

• 文件联结数目。表示在本目录树中有多少个文件名指向该文件。第 5 章将详细解释文件的联结。

• 文件数据的磁盘地址明细表。虽然用户们把文件中的数据在逻辑上作为字节流看待，但是内核把这些数据保存在不连续的磁盘块上。索引节点标识出了含有文件数据的磁盘块。

• 文件大小。一个文件中的数据可以用它偏离文件起始点的字节数来编址——设起始点的字节偏移量为 0。整个文件大小比该文件中数据的最大字节偏移量大 1。比如，如果一个用户创建了一个文件并且仅在文件中字节偏移量为 1000 的位置上写了 1 个字节的数据，则该文件的大小为 1001。

索引节点并不标明存取该文件的路径名。

图 4-2 给出了一个示例文件的磁盘索引节点。该索引节点是一个正规文件的索引节点，其所有者是“mjb”，含有 6030 个字节。系统允许“mjb”读、写或执行该文件；“os”用户组的成员以及其他用户仅能读或执行该文件，但不能往该文件上写。某人最近一次读该文件的时间是 1984 年 10 月 23 日下午 1 点 45 分。某人最近一次写该文件的时间是 1984 年 10 月 22 日上午 10 点 30 分。最近一次对该索引节点进行修改的时间是 1984 年 10 月 23 日下午 1 点 30 分，虽然那时并没有往文件上写数据。内核对索引节点中的上述信息进行编码。请注意把索引节点的内容往磁盘上写与把文件的内容往磁盘上写的区别；仅当写文件时才改变文件内容，而当改变文件内容、改变文件所有者、改变存取许可权或改变联结状况时，都要改变索引节点的内容。改变一个文件的内容自动地暗示着其索引节点的改变，但改变索引节点并不意味着文件内容的改变。

图 4-2　磁盘索引节点示例

内存中的索引节点拷贝除了磁盘索引节点所包含的那些字段之外，还包含如下字段：

- 内存索引节点的状态，它指示
  ——索引节点是否被上锁了。
  ——是否有进程正在等待索引节点变为开锁状态。
  ——作为对索引节点中的数据进行更改的结果，索引节点的内存表示是否与它的磁盘拷贝不同。
  ——作为对文件数据更改的结果，文件的内存表示是否与它的磁盘拷贝不同。
  ——是否该文件为安装点（见 5.15 节）。
- 含有该文件的文件系统的逻辑设备号。
- 索引节点号。因为索引节点存储在磁盘上的线性数组中（回顾 2.2.1 节），所以内核用索引节点在该数组中的位置来标识磁盘索引节点号。磁盘索引节点不需要这个字段。
- 指向其他内存索引节点的指针。就象把缓冲区链接到缓冲区散列队列上和缓冲区空闲表上一样，内核按照与此相同的方式把索引节点链接到散列队列上和空闲表上。内核是用索引节点的逻辑设备号和索引节点号来标识一个散列队列的。对于一个磁盘索引节点来说，内核至多只能包含它的一个内存拷贝，但索引节点能够同时既位于散列队列上，又位于空闲表上。
- 引用数。指示出该文件的活跃实例数目（例如以系统调用 open 打开了的）。

内存索引节点中的很多字段都与缓冲首部的那些字段类似。当索引节点处于上锁状态时，阻止其他进程存取该索引节点；当其他进程试图存取该索引节点时，它们在索引节点中设置一个标志，以指示出当锁被释放时应该唤醒它们。内核设置另外的标志以指示磁盘索引节点与它的内存拷贝之间的差异。当内核需要把对文件或索引节点的变更记录下来时，在考察了这些标志之后，它把索引节点的内存拷贝写到磁盘上。

内存索引节点与缓冲首部的显著区别是内存引用计数，它对该文件的活跃的实例数进行计数。当一个进程分配了一个索引节点时，比如当打开一个文件时，该索引节点就是活跃的。仅当索引节点的引用计数为 0 时它才位于空闲表上，以表示内核能把这个内存索引节点重新分配给另一个磁盘索引节点。因此，可把空闲索引节点表当作不活跃的索引节点的高速缓冲。如果一个进程试图存取一个文件，而该文件的索引节点当前不在内存索引节点池中，则内核从空闲表中重新分配一个内存索引节点供它使用。另一方面，缓冲区没有引用计数；当且仅当缓冲区为开锁状态时，它位于空闲表中。

4.1.2 对索引节点的存取

内核用文件系统和索引节点号来标识特定的索引节点，并在高层算法请求时分配内存索引节点。算法 iget 分配一个索引节点的内存拷贝（图 4-3），它与为在高速缓冲中找到一个磁盘块的 getblk 算法几乎是完全相同的。内核把设备号和索引节点号映射到一个散列队列上，并且搜索该队列以便找到该索引节点。如果它不能找到这个索引节点，它就从空闲表中分配一个索引节点，并且对它上锁。随后，内核准备把待存取的索引节点的磁盘拷贝读进内存拷贝中。它已经知道索引节点号和逻辑设备，再根据一个磁盘块中能装多少个磁盘索引节点，来计算出包含该索引节点的逻辑磁盘块。计算按以下公式：

```
算法　iget
输入:文件系统索引节点号
输出:上锁状态的索引节点
{
    while （未完成）
    {
        if （是索引节点高速缓冲中的索引节点）
        {
            if(索引节点为上锁状态)
            {
                sleep(索引节点变成开锁状态事件);
                continue;     /*  循环回到 while  */
            }
            /*  对安装点进行特殊处理(第 5 章)  */
            if （是空闲索引节点表上的索引节点）
                从空闲表上移去该节点;
            索引节点引用计数增值 1;
            return(索引节点);
        }
        /*  不是索引节点高速缓冲中的索引节点  */
        if （空闲表上没有索引节点）
            return(错误);
        从空闲表上移出一个新索引节点;
        重置索引节点号及文件系统;
        从老的散列队列中撤掉索引节点,把索引节点放到新的散列队列中;
        从磁盘上读索引节点(算法 bread);
        索引节点初始化(例如访问计数置为 1);
        return(索引节点);
    }
}
```

图 4-3　分配内存索引节点算法

$$块号=((索引节点号-1)/每块的索引节点数目)+索引节点表的起始块号$$

此处的除操作取商的整数部分。例如，假设索引节点表从第 2 块开始并且每块有 8 个索引节点，则第 8 号索引节点在第 2 磁盘块中，第 9 号索引节点在第 3 磁盘块中。如果每个磁盘块上有 16 个索引节点，则第 8 号与第 9 号索引节点都在第 2 磁盘块中，第 17 号索引节点则是第 3 磁盘块中的第一个索引节点。

当内核知道了设备和磁盘块号时，它使用算法 bread（第 2 章）读出该块，然后使用下述公式计算本索引节点在该块中的字节偏移量：

$$((索引节点号-1)\ \bmod\ (每块的索引节点数目))\ *\ 磁盘索引节点大小$$

例如，如果每个磁盘索引节点占据 64 个字节，并且每磁盘块有 8 个索引节点，则第 8 号索引节点从磁盘块中字节偏移量为 448 的位置上开始。内核从空闲表中移去内存索引节点，把

它放到正确的散列队列中，并且将它的内存引用计数置为1。它把文件类型、所有者、存取权限、联结计数、文件大小及内容表等从磁盘索引节点拷贝到内存索引节点上，并且返回一个上锁状态的索引节点。

内核独立地操纵索引节点锁和引用计数。在系统调用执行期间把锁关闭，防止其他进程存取正在使用中的索引节点（可能会引起不一致性）。在系统调用结束时内核把锁释放：两次系统调用间隔期间，索引节点是从不上锁的。对文件的每次活跃的引用，内核都把引用计数加1。比如，5.1节将表明，当进程打开一个文件时，它就把索引节点引用计数增1；当引用变为不活跃的时候，比如，当进程关闭一个文件时，它就把索引节点引用计数减1。在系统调用与系统调用之间锁是空闲的，以允许多个进程共享对一个文件的同时存取；在系统调用与系统调用之间引用计数保持不变，以使内核不把活跃的内存节点重新分配出去。因此，内核可以不依赖于引用计数的值而为一个已分配了的索引节点上锁和解锁。第5章中我们将看到，open以外的其他系统调用都分配和释放索引节点。

现在再回到算法iget。如果内核试图从空闲表中取出一个索引节点，但发现空闲表是空的，则它报告一个错误。这与内核处理磁盘缓冲区所遵从的准则是不同的，在那里，进程进入睡眠直至缓冲区变为空闲；而在此处，诸进程通过执行系统调用open和close而在用户级上控制索引节点的分配，结果使内核不能保证什么时候一个索引节点将变成可用的。因此，一个因等候得到空闲索引节点而去睡眠的进程可能总是不能醒来。内核与其把这样的进程“挂起”，不如让这样的系统调用失败。然而，进程不能这样来控制缓冲区：因为进程不能使缓冲区在系统调用间隔期间保持上锁状态，内核能保证缓冲区将很快地变为空闲，所以让进程睡眠，直到有一个可用的缓冲区。

前面几段讲了内核分配不在索引节点高速缓冲中的一个索引节点的情况。如果索引节点在高速缓冲中，那么进程（A）能在它的散列队列上找到它，并且检查该索引节点当前是否被另一个进程（B）锁住了。如果该索引节点为上锁状态，则进程A设置内存索引节点标志，以指示它正在等待该索引节点变为空闲，与此同时它去睡眠。而后，当进程B为这个索引节点解锁时，它唤醒等待着该索引节点变为空闲的所有进程（包括进程A）。当终于轮到进程A使用该索引节点时，它将该索引节点上锁，以使其他进程不能分配它。如果引用计数原来为0，则该索引节点也出现在空闲表中，因此内核把它从空闲表中移去：该索引节点不再是空闲的了。内核使该索引节点的引用计数加1，并且返回一个上了锁的索引节点。

综上所述，算法iget用于当进程初次存取一个文件时所引用的系统调用的开始阶段。该算法返回一个上了锁的索引节点数据结构，其引用计数比原来大1。内存索引节点包含着关于该文件状态的最新的信息。在从系统调用返回之前内核给索引节点解锁，使想要存取该索引节点的其他进程能够如愿以偿。第5章会较为详细地讨论这一情况。

4.1.3 释放索引节点

当内核释放一个索引节点时（算法iput，图4-4），将它的索引节点引用计数值减1。如果该计数值降为0，且它的内存拷贝与磁盘拷贝不同，则内核还要往磁盘上写该索引节点。文件数据的改变、文件存取时间的改变、文件所有者或存取许可权的改变，都会引起它的内存拷贝与磁盘拷贝的不同。内核还把该索引节点放到空闲索引节点表中，如果发生再次需要该索引节点的情况，那么把该索引节点高速缓冲起来是非常高效的。如果该文件的联结数为

0，则内核也可以释放与该文件有关的所有数据块，并且把该索引节点变为空闲。

```
算法  iput    /*  释放对内存索引节点的存取  */
输入：指向内存索引节点的指针
输出：无
{
      如果索引节点未上锁，则将其上锁；
      将索引节点引用计数值减 1；
      if  (访问计数为 0)
       {
          if  (索引节点联结计数值为 0)
           {
              释放文件所占的磁盘块（算法 free，4.7 节）；
              将文件类型置为 0；
              释放索引节点（算法 ifree，4.6 节）；
           }
          if  (文件被存取或索引节点被改变或文件被改变)
                修改磁盘索引节点；
          将索引节点放到空闲表中；
       }
      为索引节点解锁；
}
```

图 4-4　释放索引节点算法

4.2　正规文件的结构

如上所述，索引节点包含着文件数据在磁盘上的位置的明细表。因为磁盘上的每一块都是编了号的，所以明细表是由磁盘块号的集合组成的。如果文件中的数据被储存到一个连续的磁盘段上（即该文件占据了磁盘块的线性序列），那么，只要把起始块地址与文件大小存储在索引节点中便足以存取文件中的所有数据了。然而，如果不想冒把磁盘上的空闲存储区分割成碎片的风险的话，那么，像这样一种分配策略是不允许文件系统中文件的简单扩展与收缩的。而且，在允许增加文件大小的操作进行之前，内核必须在文件系统中分配并储备好连续的空间。

举例来说，假设某用户创建了三个文件 A，B，C，每个文件都由 10 个磁盘块组成，并且假设系统为三个文件分配了相邻的存储区。如果用户想要往中间的那个文件即 B 上增加 5 个数据块的话，那么内核必须把文件 B 拷贝到文件系统中有连续的 15 个存储块的位置上去。除了这样的操作的损失之外，还有一个问题是：原来被文件 B 所占据的磁盘块没有用了，除非有小于 10 块的文件（图 4-5）。内核可以通过周期地运行废区收集过程，把可用的存储区连到一起，以使存储空间碎片降到最低限，但这将增加处理能力的消耗。

为能有较大的灵活性，内核一次分配一块文件空间，并且允许文件中的数据遍布整个文件系统。但是这种分配方案使数据定位的任务复杂化了。地址明细表实际上是一张包含着属于该文件的那些数据的磁盘块块号的清单。但简单的计算表明，如果一个逻辑块包含 1K 字节，那么，由 10K 个字节组成的文件就需要 10 个块号的索引，由 100K 个字节组成的文件

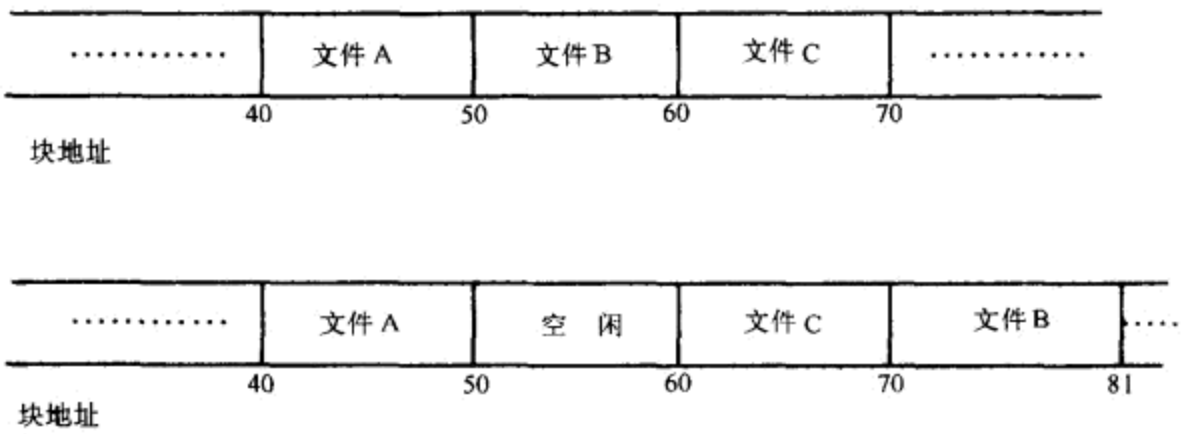


图 4-5 相邻文件分配与空闲空间碎片

就需要 100 个块号的索引。要么，索引节点的大小随着文件大小变化；要么，对文件大小施加一个相对小的限制。

为使索引节点保持较小的结构又能允许组织大文件，磁盘块地址表取图 4-6 所示的形式。

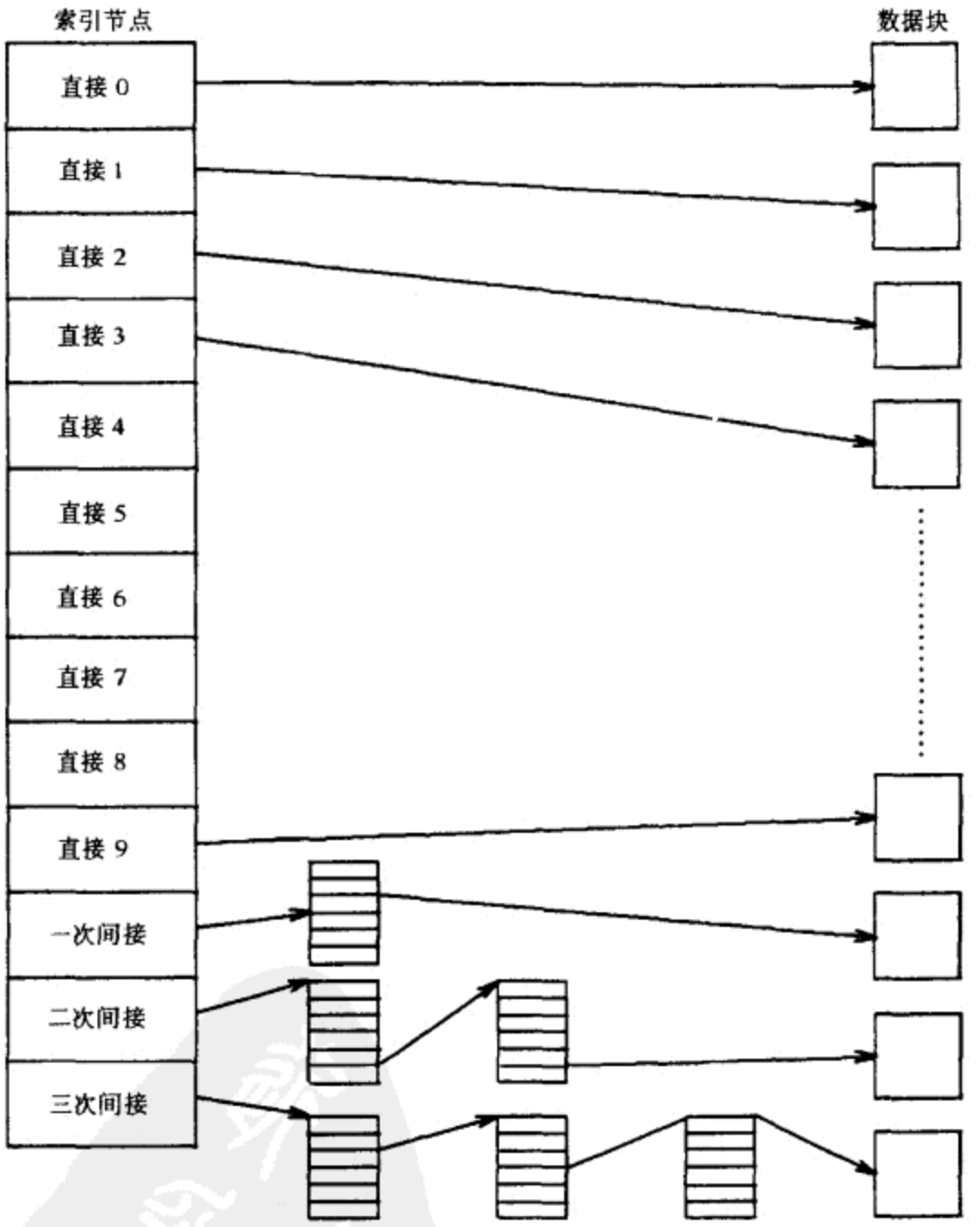


图 4-6 索引节点中的直接块和间接块

UNIX 系统 V 的索引节点中的地址表有 13 个表项，但此处所述的原理是不依赖于表项的个数的。图中，标有“直接”的那些块指示含有实际数据的磁盘块的块号，标有“一次间接”的那些块则指向一个含有直接块号表的磁盘块。若要通过间接块存取数据，内核必须先把间接块读出，找到相应的直接块块号，然后读出直接块以找到数据。标有“二次间接”的块包含一个间接块号表，而标有“三次间接”的块则包含一个二次间接块号表。

从原则上说，这一方法可以扩展到支持“四次间接块”、“五次间接块”，等等。但在实际上，当前的这一结构就足够了。假设文件系统中的一个逻辑块容纳 1K 字节，并且假设一个块号用 32 位（4 字节）的整数编址，于是每块可容纳 256 个块号。一个文件可容纳的字节数的最大值可以计算出来（图 4-7）——若在索引节点中使用 10 个直接块、1 个间接块、1 个二次间接块和 1 个三次间接块，则可达 16 千兆。设索引节点中的“文件大小”字段为 32 位，则一个文件的大小不能超过 4 千兆（2^{32}）。

10 个直接块，每个按 1K 字节计 =	10K 字节
1 个具有 256 个直接块的一次间接块 =	256K 字节
1 个具有 256 个一次间接块的二次间接块 =	64M 字节
1 个具有 256 个二次间接块的三次间接块 =	16G 字节

图 4-7　一个文件的字节容量——按每块 1K 字节计

进程存取文件中的数据时使用了字节偏移量。它们是采用字节计数的方法工作的，并且把一个文件看作从字节地址 0 开始直至整个文件大小的字节流。内核把用户的字节流看法转换成块的看法：文件从第 0 个逻辑块开始，继续到相应于该文件大小的那个逻辑块号为止。内核存取索引节点并且把逻辑文件块转换成适当的磁盘块。图 4-8 给出了把一个文件字节偏

```
算法　bmap　　/*　从逻辑文件字节偏移量到文件系统块的块映射　*/
输入:(1) 索引节点
     (2) 字节偏移量
输出:(1) 文件系统中的块号
     (2) 块中的字节偏移量
     (3) 块中 I/O 字节数
     (4) 提前读块号
{
    由字节偏移量计算出在文件中的逻辑块号;
    为 I/O 计算出块中的起始字节;　　/*　输出 2　*/
    计算出拷贝给用户的字节数;　　/*　输出 3　*/
    检查是否可用提前读并标记索引节点;　　/*　输出 4　*/
    决定间接级;
    while　(没在所必须的间接级上)
    {
        从文件中的逻辑块号计算索引节点中或间接块中的下标;
        从索引节点或间接块上得到磁盘块号;
        如果需要,应从先前的磁盘读释放缓冲区(算法 brelse);
        if　(再也没有间接级了)
            return(块号);
        读间接磁盘块(算法 bread);
        按照间接级调节文件中的逻辑块号;
    }
}
```

图 4-8　文件系统中字节偏移量到块号的转换算法

移量转换成一个物理磁盘块的算法 bmap。

考虑图 4-9 所示的文件的块布局，并假设每个磁盘块包含 1024 字节。如果某进程想要存取偏移量为 9000 的字节，则内核计算出该字节在文件中的第 8 个直接块中（从 0 开始计数），于是它就存取第 367 号磁盘块，在这块中的第 808 字节（从 0 开始）即为文件中第 9000 字节。如果某进程想要存取文件中偏移量为 350000 的字节，则它必须存取一个二次间接块——图中序号为 9156 的那块。由于一个间接块可容纳 256 个块号，所以经由二次间接块所存取的第一个字节是第 272384（256K + 10K）个字节；从而文件中的第 350000 字节是二次间接块中的第 77616 字节。因为每个一次间接块存取 256K 字节，所以第 350000 字节必定在二次间接块的第 0 个一次间接块中——其块号为 331。因为在一次间接块中每个间接块都包含 1K 字节，所以一次间接块中的第 77616 字节位于一次间接块的第 75 个直接块中——其块号为 3333。最终得到：文件中的第 350000 字节是第 3333 块中的第 816 字节。

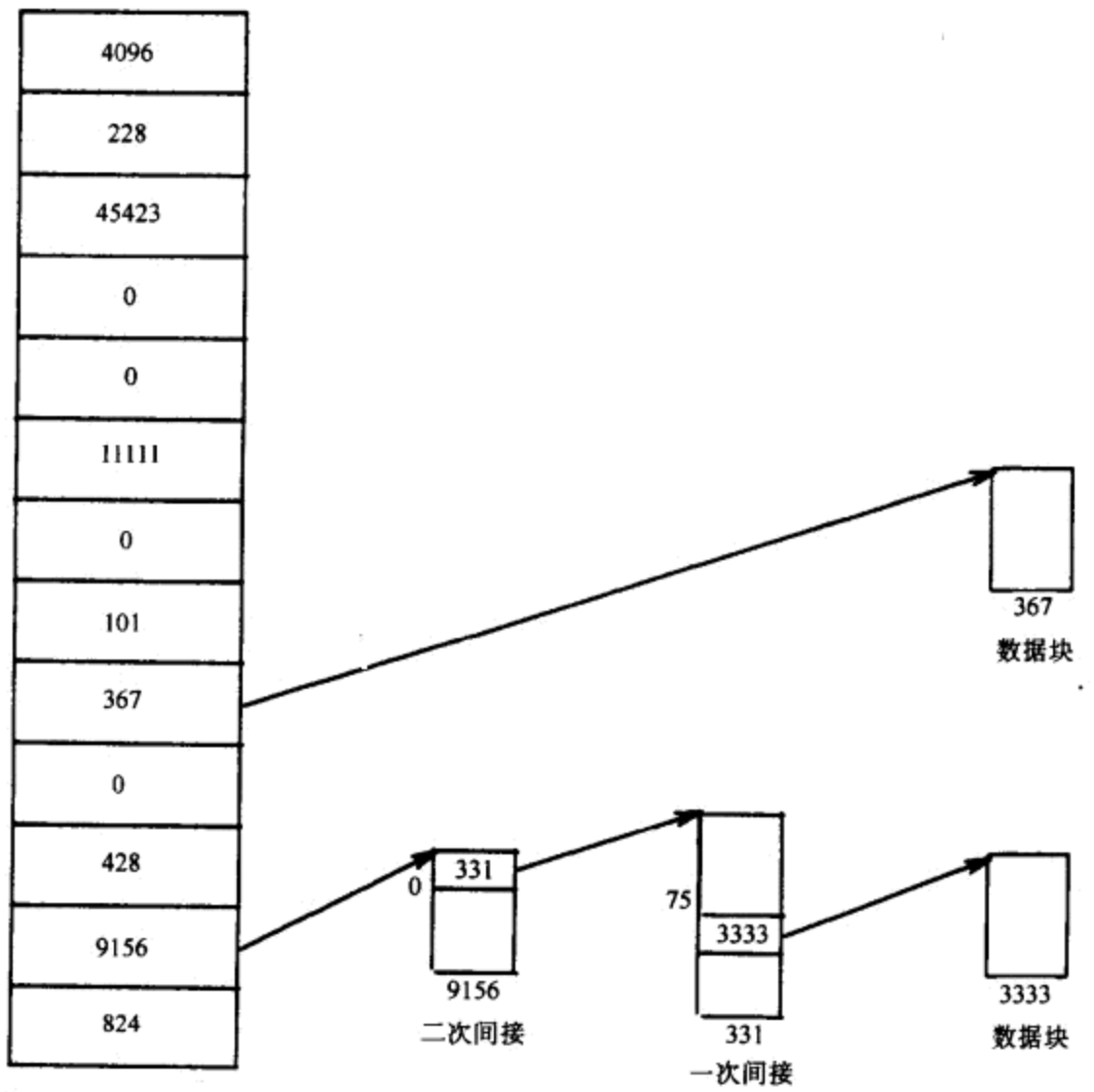


图 4-9　示例文件的磁盘块布局及其索引节点

更仔细地考察图 4-9，会看到索引节点中某些块登记项为 0，这意味着这些逻辑块登记项不含数据。如果从未有进程将数据往这些块对应的任意字节偏移量处写入该文件时，就会发生这一情况，从而这些块登记项中保留着它们的初始值 0。这些块没有造成磁盘空间的浪费。正如在下一章就要描述的那样，进程能通过使用系统调用 lseek 和 write 造成这种文件的块布局。下一章还描述内核怎样处理对这些块进行存取的系统调用 read。

对一个大的字节偏移量，特别是经由三次间接块引用的字节偏移量，进行转换是一个艰苦的过程。它要求内核除了存取索引节点与数据块之外，还要存取三个磁盘块。即使内核在高速缓冲中能找到这些磁盘块，但由于内核必须多次请求高速缓冲，并且会不得不等候上了锁的缓冲区，所以操作仍然是代价很高的。那么，该算法在实际上的有效性如何？这要依赖于如何使用这样的系统，以及用户团体和作业组合是使内核更经常地存取大文件，还是更经常地存取小文件。然而，已经进行的观察[Mullender 84]表明，UNIX 系统中的大多数文件都少于 10K 字节，很多甚至少于 1K 字节[⊖]！由于一个文件的 10K 字节被存储在直接块中，所以大多数文件数据可以通过一次磁盘存取而得到。所以，尽管存取大文件确实是高代价的操作，但存取一般大小的文件仍是快速的。

对刚刚描述的索引节点结构有两个发展。这两个发展主要考虑的是有利于文件大小的特性。4.2BSD 文件系统实现[Mckusick 84]中的主要原则是：内核在一次操作中在一个磁盘块上所能存取到的数据越多，则文件存取就越快。这就是主张把磁盘块搞得更大些。Berkeley 的实现允许 4K 或 8K 字节的逻辑磁盘块。但是，增大磁盘块长度会增大磁盘块碎片，使磁盘空间中很大部分处于未用的状态。比如，若逻辑块长为 8K 字节，那么，一个长度为 12K 字节的文件使用一个完整块，而第二块只用一半。第二块的另一半（4K 字节）被浪费了。没有其他文件能利用这部分空间存储数据。如果文件长度都象这样，即一个文件的最后一块的字节数都这样分布，则平均浪费的空间为每个文件半块；对于逻辑块长为 4K 字节的文件系统[Mckusick 84]来说，所浪费的磁盘空间可高达 45%。Berkeley 的实现通过把文件的最后的那些数据放到磁盘块分段中去的方法补救了这一情况。一个磁盘块可以包含属于若干文件的分段。第 5 章的一个练习考察了这一实现的某些细节。

对此处所描述的索引节点结构的第二个发展是把文件数据存储到索引节点中(见[Mullender 84])。通过扩展索引节点使之占据整个磁盘块，在很多情况下可使磁盘块的一个小部分用于存放索引节点结构本身，而磁盘块中的余下部分能存储整个文件；在余下部分不能存储整个文件的情况下，则存储一个文件的尾部。其优点是，如果文件都填入索引节点块中，那么，为要得到这个索引节点及其数据，只需一次磁盘存取。

4.3 目录

让我们回顾一下第 1 章，那里说到，目录是使文件系统具有树形结构的那些文件；目录在文件名到索引节点号的转换中扮演了重要角色。目录是文件，只是它的数据是一系列目录表项，每个目录表项由一个索引节点号和一个包含在这个目录中的文件名组成。一个路径名是被斜杠("/")字符分割成一个个独立分量的以空白符终结的字符串。除了最后一个分量外，每个分量都必须是一个目录名，但最后一个分量可以是一个非目录文件。UNIX 系统 V 把分量名限制为 14 个字符，索引节点号占两个字节。一个目录项长度为 16 个字节。

图 4-10 描绘了目录“etc”的布局。每个目录都包含“.”和“..”的文件名，其索引节点号分别是本目录和它的父目录的索引节点号。在“/etc”中的“.”的索引节点号在该文件中字节偏移量为 0 的位置上，其值为 83 。“..”的索引节点号在字节偏移量为 16 的位置

⊖ Mullender 和 Tannenbaum 对 19978 个文件取样并进行统计分析，发现其中 85% 的文件都小于 8K 字节，48% 的文件小于 1K 字节。虽然从一个装置到另一个装置这里的百分比数字会发生变化，但它们代表了很多 UNIX 系统。

上，其值为2。目录表项可以为空，由取值为0的索引节点号指示。例如，在“/etc”中的第224字节上的目录表项就为空，虽然它曾经含有过名为“crash”的文件的目录表项。程序mkfs初始化一个文件系统，以使根目录的“.”和“..”具有该文件系统的根索引节点号。

目录中的字节偏移量	索引结点号（2字节）	文件名
0	83	
16	2	..
32	1798	init
48	1276	fsck
64	85	clri
80	1268	motd
96	1799	mount
112	88	mknod
128	2114	passwd
144	1717	umount
160	1851	checklist
176	92	fsdblb
192	84	config
208	1432	getty
224	0	crash
240	95	mkfs
256	188	inittab

图 4-10 /etc的目录布局

内核就像为普通文件存储数据那样来为目录存储数据，也使用索引节点结构和直接块级、间接块级。进程可以按它们读正规文件的方式读目录，但内核保留写目录的独占权，因此能保证它的结构正确。一个目录的存取许可权具有如下含义：在一个目录上的读许可权表示允许进程读这个目录；写许可权表示允许进程（通过creat，mknod，link及unlink等系统调用）创建新目录表项或撤销老目录表项，从而变更目录内容；执行许可权表示允许进程为寻找一个文件名而搜索这个目录（显然，执行一个目录是无意义的）。习题4.6考察了读目录与搜索目录之间的差别。

4.4 路径名到索引节点的转换

最初对一个文件的存取是通过路径名进行的，正如在系统调用open，chdir（改变目录）或link中那样。因为内核在内部是用索引节点工作而不是用路径名工作，所以内核把路径名转换成索引节点号，以存取文件。算法namei每次分析路径名的一个分量，根据这个名字及正在搜索的目录，把每个分量转换成一个索引节点，并且最终返回输入路径名的索引节点（图4-11）。

回顾一下第2章，每个进程都与一个当前目录相联系。u区包含一个指向当前目录索引节点的指针。系统中第一个进程——0进程的当前目录是根目录。其他每个进程的当前目录都是从该进程被创建时它的父进程的当前目录出发的（见5.10节）。各个进程通过执行系统

```
算法 namei    /*  把路径名转换为索引节点  */
输入:路径名
输出:上了锁的索引节点
    {
    if (路径名从根开始)
        工作索引节点=根索引节点(算法 iget);
    else
        工作索引节点=当前目录索引节点(算法 iget);
    while (还有路径名)
    {
        从输入读下一个路径名分量;
        证实工作索引节点确是目录,存取许可权 OK;
        if (工作索引节点为根且分量为"..")
            continue;    /*  循环回到 while  */
        通过重复使用算法 bmap、bread 和 brelse 来读目录(工作索引节点);
        if (分量与目录(工作索引节点)中的一个目录表项匹配)
        {
            得到匹配分量的索引节点号;
            释放工作索引节点(算法 iput);
            工作索引节点=匹配分量的索引节点(算法 iget);
        }
        else    /*  目录中无此分量  */
            return(无索引节点);
    }
    return(工作节点);
}
```

图 4-11 从路径名到索引节点的转换算法

调用 chdir(改变目录)来改变它们的当前目录。所有的路径名搜索都是从那个进程的当前目录开始的,除非该路径名是以斜杠字符开始的——斜杠字符意味着应该从根目录开始搜索。无论在哪种情况下,内核都能容易地找到从那里开始搜索路径名的索引节点;当前目录存储在进程的 u 区,而系统根索引节点被存储在一个全局变量中[⊖]。

namei 在分析路径名时使用称之为工作索引节点的中间索引节点。搜索从哪儿开始,哪儿就是第一个工作索引节点。在 namei 的每次循环期间,内核都要证实工作节点确实是目录。不然,系统就违反了"非目录文件仅能是文件系统树的树叶节点"的要求。进程还必须具有搜索目录的许可权(读许可权是不充分的)。进程的用户 ID 必须与那个文件的所有者 ID 或用户组 ID 相匹配,并且必须被授予执行许可权,或者允许所有的用户搜索那个文件。否则搜索失败。

内核在与工作索引节点相联系的目录文件上进行线性搜索,试图使路径名分量与目录表项名字相匹配。从字节偏移量 0 开始,按照算法 bmap 把在目录中的字节偏移量转换成适当的磁盘块,并且使用算法 bread 把该磁盘块读出。它把块上的内容作为目录表项序列,为该路

⊖ 进程能执行系统调用 chroot 以改变它的文件系统根的符号。被改变了的根存储在 u 区中。

径名分量搜索该块，如果它发现匹配了，则它把被匹配的目录表项的索引节点号记录下来，释放该块(算法 brelse)及释放旧的工作索引节点(算法 iput)，并且分配匹配了的分量的索引节点(算法 iget)。新索引节点变为工作索引节点。如果在该块上没有发现任何名字与路径名相匹配，则内核释放该块，根据每块中字节数来调整字节偏移量，把新的字节偏移量转换成磁盘块号(算法 bmap)，并且读下一块。内核重复这一过程直至它把路径名分量与一个目录登记项名匹配时为止，或者直至到达该目录的尾部时为止。

举例来说，假设一个进程想要打开文件"/etc/passwd"。当内核开始分析这个文件名时，它遇到"/"，并获得了系统根索引节点。在把根作为它的当前工作索引节点后，内核读入字符串"etc"。在核对了当前索引节点确实是目录索引节点("/")，以及核对了该进程具有必要的搜索许可权之后，内核对根进行搜索，以找到名为"etc"的文件：它一块接一块地存取根目录中的数据，并且每次一个目录表项地对每一块进行搜索，直至它找到"etc"这一目录表项的位置。在找到这一登记项之后，内核释放根索引节点(算法 iput)，并且根据刚刚找到的目录表项的索引节点号，为"etc"分配索引节点(算法 iget)。在查明"etc"确实是一个目录、并查明它具有必要的搜索许可权之后，内核一块接一块地搜索"etc"，其目的是要找到文件"passwd"的目录结构登记项。参照图 4-10，它能发现：目录的第九个目录表项就是"passwd"的目录表项。在找到之后，核心释放"etc"的索引节点，为"passwd"分配索引节点，并且，由于路径名分析完了，所以返回这一索引节点。

很自然地会提出如下问题：为找到一个路径名分量而对目录进行线性搜索，其效率如何？Ritchie 指出(见[Ritchie 78b]的第 1968 页)线性搜索是有效的，因为它受目录长度的绑定。此外，早期的 UNIX 系统实现不是在具有大容量存储空间的机器上运行的，所以强调像线性搜索方案这样的简单算法。较复杂的搜索方案会要求一种不同的、较复杂的目录结构，而且在小目录上运行时或许比线性搜索方案更慢。

4.5　超级块

到目前为止，本章已描述了文件的结构，同时假设索引节点已经预先绑定到一个文件上，并且已经分配了磁盘块以容纳文件数据。下一节来谈谈内核怎样分配索引节点和磁盘块。为了理解那些算法，让我们考察超级块的结构。

超级块由如下字段组成：

- 文件系统的规模。
- 文件系统中空闲块的数目。
- 在文件系统上可用的空闲块表。
- 空闲块表中下一个空闲块的下标。
- 索引节点表的大小。
- 文件系统中空闲索引节点数目。
- 文件系统中的空闲索引节点表。
- 空闲索引节点表中下一个空闲索引节点的下标。
- 空闲块表的锁字段和空闲索引节点表的锁字段。
- 用来指示出超级块已经被修改了的标志。

本章的其余部分将解释数组、索引和锁的使用。如果超级块被修改，则内核定期地把超级块

写到磁盘上，以便它与文件系统中的数据保持一致性。

4.6　为新文件分配索引节点

内核使用算法 iget 分配一个已知的索引节点——它的（文件系统和）索引节点号是预先确定了的。比如，在算法 namei 中，内核通过把一个路径名分量与目录中的一个名字匹配来决定索引节点号。另一个算法 ialloc 则把一个磁盘索引节点分配给一个新建立的文件。

正如第 2 章中提到的那样，文件系统包含一个索引节点线性表。如果它的类型字段为 0，则说明这个索引节点是空闲的。当一个进程需要一个新的索引节点时，从理论上说，内核能搜索索引节点表，以寻找一个空闲节点。然而，像这样的搜索代价太高了，至少对每个索引节点都需要一个读操作（可能从磁盘读）。为改善性能，文件系统超级块包含一个数组，以便把文件系统中空闲的索引节点号缓存起来。

图 4-12 给出了分配新索引节点的算法 ialloc。由于随后要列举的原因，内核首先验证无其他进程锁住了对超级块空闲索引节点表的存取。如果超级块中的索引节点号表非空，则内核分配下一个索引节点号，使用算法 iget 为新分配的磁盘索引节点分配一个空闲的内存索引节点(如果需要的话，得从磁盘读出索引节点)，把磁盘索引节点拷贝到内存拷贝中，将索引节点的各字段初始化，并返回这一上了锁的索引节点。它修改磁盘索引节点，以指示出现在该索引节点正在使用中：一个非零文件类型字段指示出该磁盘索引节点已经分配出去了。在最简单的情况下，内核有一个好的索引节点，但存在多个竞争条件，需要进一步检查，对此稍后将做解释。按非严格的定义，当若干进程改变共享数据结构时，尽管所有进程都遵守加锁协议，但所产生的计算结果依赖于进程执行的次序，在这种情况下竞争条件就出现了。例如，这意味着一个进程可能会得到一个正在被使用的索引节点。竞争条件与第 2 章中定义的互斥条件有关，那里，加锁方案解决了互斥问题，但靠它们自己可能解决不了所有的竞争条件。

如果超级块空闲索引节点表为空，则内核搜索磁盘，并把尽可能多的空闲索引节点号放到超级块中。内核一块接一块地读磁盘上的索引节点表，并且填入超级块索引节点号表中直至满额。同时记住它所找到的最高序号的索引节点，称之为“铭记”的索引节点，这是保存在超级块中的最后一个索引节点。下次核心搜索磁盘上的空闲索引节点时，就从铭记索引节点开始，这可以确保它不浪费时间去读那些已不含空闲索引节点的磁盘块。在收集了一批新的空闲索引节点号之后，它再从头开始索引节点分配算法。无论何时内核分配一个磁盘索引节点，它都要将超级块中记录着的空闲索引节点计数值减 1。

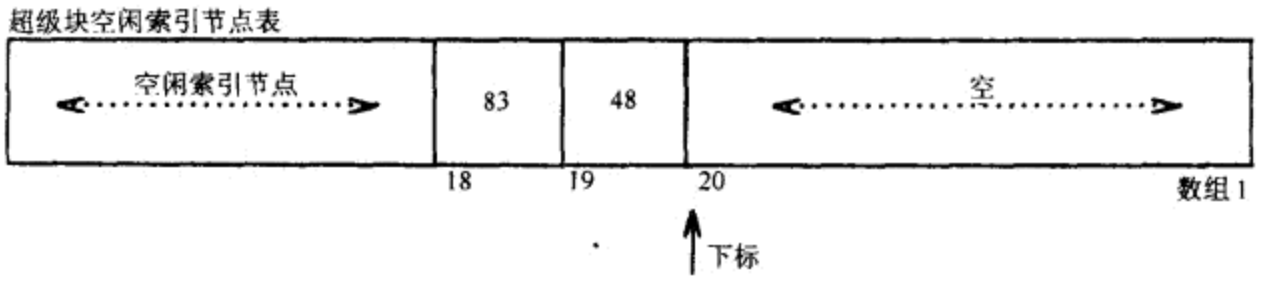
考虑图 4-13 中的空闲索引节点号的两对数组。当内核分配一个索引节点时，如果超级块中的空闲索引节点表看起来像图 4-13a 中的第一个数组，它把下标值减至 19，以指向下一个有效索引节点号，并且取走索引节点号 48。当超级块中的空闲索引节点表看起来像图 4-13b中第一个数组时，它将注意到该数组是空的，于是从铭记索引节点——第 470 号索引节点开始搜索磁盘，以便得到空闲索引节点。当内核把超级块空闲表填满时，它铭记住最后的索引节点，作为下一次搜索磁盘时的起始点。内核把它刚从磁盘上取来的索引节点（图中的第 471 号）分配出去，并且继续做它正在做的事。

释放一个索引节点的算法 ifree 要简单得多。在把文件系统中可用的索引节点总数加 1 之后，内核检查超级块的锁。如果超级块已被锁住，则它立即返回，以避免竞争条件：该索引节点不放到超级块中，但在磁盘上可以找到它，并且可以对它进行再分配。如果该表没有

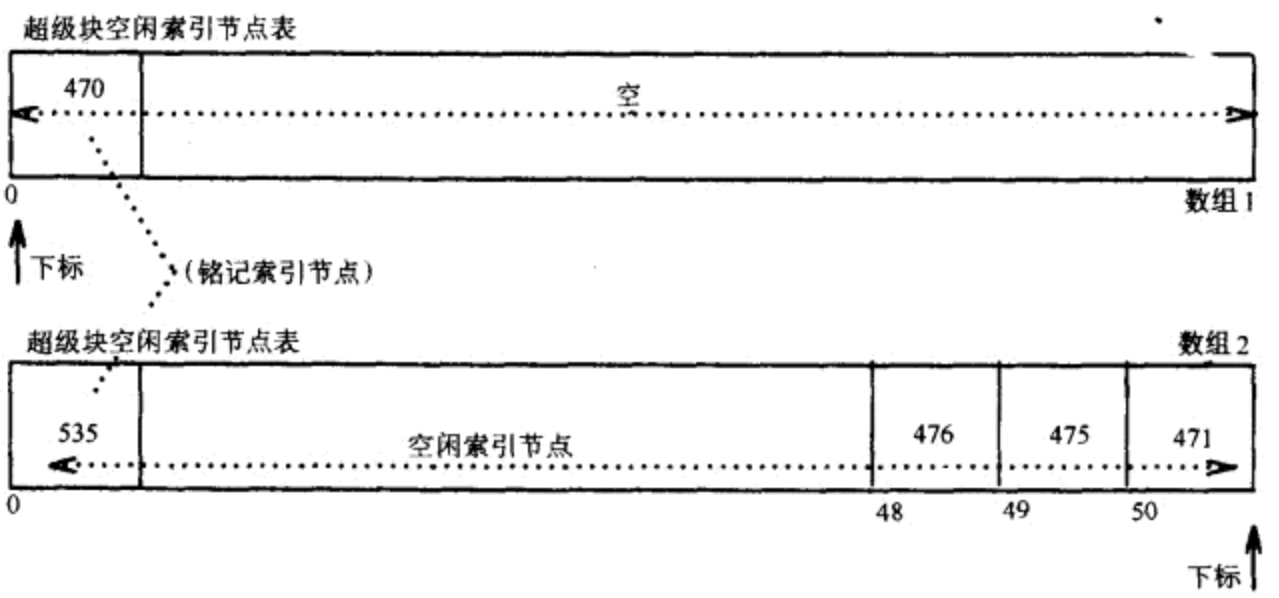
```
算法 ialloc    /*  分配索引节点  */
输入:文件系统
输出:上了锁的索引节点
{
    while (未完)
    {
        if (超级块上了锁)
        {
            sleep(等待"超级块变为空闲"事件);
            continue;    /*  while 循环  */
        }
        if (超级块中索引节点表为空)
        {
            将超级块上锁;
            为搜索空闲索引节点取铭记索引节点;
            搜索磁盘,将空闲索引节点登入超级块的索引节点表中,直至超
            级块满或再也找不到空闲索引节点(算法 bread 与 brelse);
            为超级块解锁;
            wake up("超级块变为空闲"事件);
            if (磁盘上没找到空闲索引节点)
                return(无索引节点);
            设置铭记索引节点以便为下一次搜索空闲索引节点做准备;
        }
        /*  超级块索引节点表中有索引节点  */
        从超级块索引节点表中得到索引节点号;
        得到索引节点(算法 iget);
        if (不是空闲的索引节点)    /*  !!!    */
        {
            把索引节点写到磁盘上;
            释放索引节点(算法 iput);
            continue;    /*    while 循环  */
        }
        /*  索引节点空闲  */
        将索引节点初始化;
        将索引节点写到磁盘上;
        文件系统空闲索引节点计数值减 1;
        return(索引节点);
    }
}
```

图 4-12　分配新索引节点算法

上锁，则内核检查它还有没有存放索引节点的空表项。如果有，则把该索引节点号放到表中，并且返回；如果该表已满，则内核不把新释放的索引节点保留在表中，而是把被释放的索引节点号与铭记索引节点号比较：如果被释放的索引节点号比铭记索引节点号小，则它"铭记住"新释放的索引节点号，把老的铭记索引节点号从超级块中丢掉。该索引节点并没



a)



b)

图 4-13 两个空闲索引节点号数组

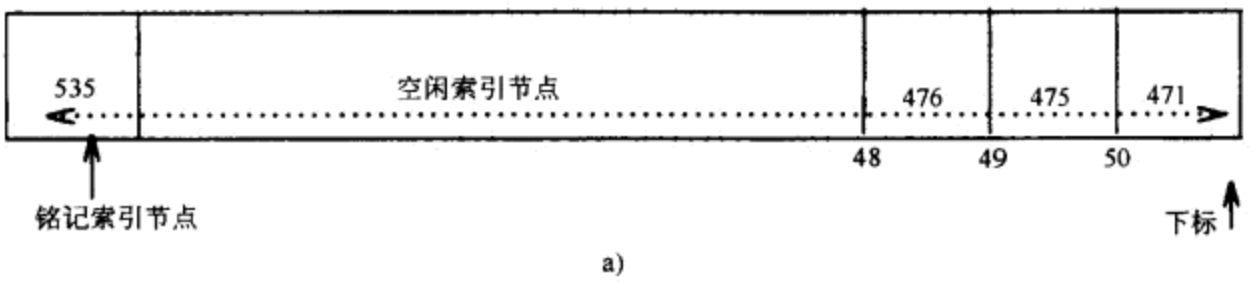
a) 从表的当中分配空闲索引节点 b) 分配空闲索引节点——超级块表空

有丢失，因为内核能搜索磁盘上的索引节点表而找到它。内核维护超级块表使得它从该表上分配出去的最后的索引节点是铭记索引节点。从理论上说，永远也不应该有其索引节点号比铭记索引节点号小的空闲节点，但例外是有的。

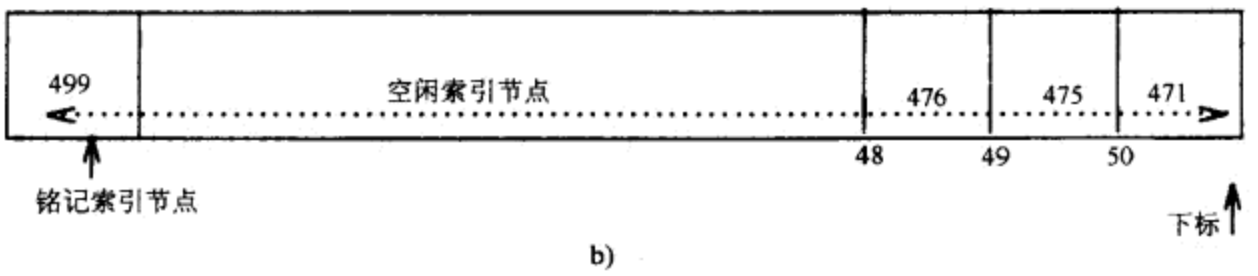
考虑释放索引节点的两个例子。如果像图 4-13a 那样，空闲索引节点的超级块表有空表项用来存放空闲索引节点号，则内核把这个被释放的索引节点号放到空闲表中，将下标增值以指向下一个空闲索引节点，并继续执行。但是，如果像图 4-15 那样，空闲索引节点表满了，则内核把它释放的索引节点号与下一次将开始搜索磁盘的铭记索引节点号比较。假如是从图 4-15a 中的空闲索引节点表开始的，如果内核释放了第 499 个索引节点，则它使 499 成为铭记索引节点，而把第 535 号从空闲表中驱逐出去。然后，如果内核释放第 601 号，则它不改变空闲表的内容。以后，当它用光了超级块空闲表中的索引节点时，它将从第 499 号索引节点开始搜索磁盘，以寻找空闲索引节点，并且会再次找到第 535 和 601 号索引节点。

```
算法　ifree　　/*　释放索引节点　*/
输入:文件系统索引节点号
输出:无
{
    文件系统空闲索引节点计数值增 1;
    if (超级块被锁住)
        return;
    if (索引节点表满)
    {
        if (索引节点号小于用于搜索的铭记索引节点号)
            置用于搜索的铭记索引节点号=输入索引节点号;
    }
    else
        把输入索引节点号存储到索引节点表中;
    return;
}
```

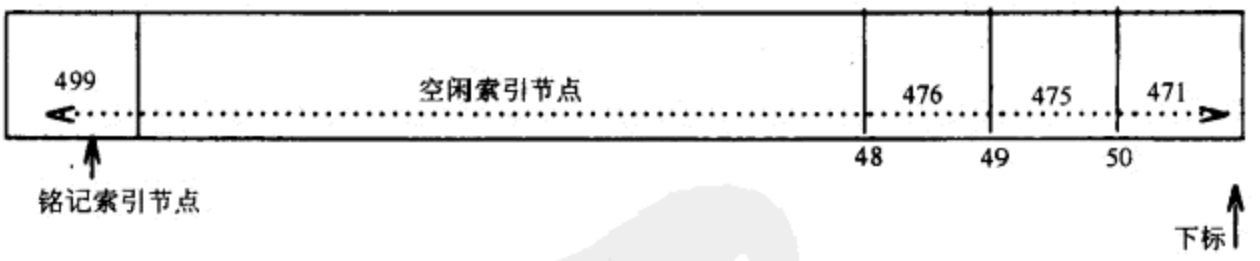
图 4-14　释放索引节点算法



a)



b)



c)

图 4-15　将空闲索引节点号放到超级块中

a）原超级块空闲索引节点表　b）空闲索引节点 499　c）空闲索引节点 601

前一段描述了这一算法的简单情况。现在考虑内核分配一个新索引节点，并随后为该索引节点分配内存索引节点拷贝的情况。这一算法暗含着内核能发现该索引节点已被分配出去的意思。下面的情况虽然极少见，但给出了这样的一个例子（参见图 4-16 与图 4-17）。考虑三个进程 A，B 和 C，并且假设代表进程 A[⊖]而动作着的内核分配了索引节点 I，但在内核把磁盘索引节点拷贝到内存索引节点之前，它去睡眠了。算法 iget（被 ialloc 调用）和 bread（被 iget 调用）给进程 A 以足够的机会去睡眠。当进程 A 正在睡眠时，假设进程 B 试图分配一个新索引节点，但发现超级块的空闲索引节点表空了。进程 B 搜索磁盘块以找到空闲索引节点，并假设它是从比进程 A 正在分配的索引节点号低的索引节点号开始搜索空闲索引节点的。由于进程 A 仍处于睡眠状态，所以可能进程 B 会发现在磁盘上索引节点 I 是空闲的，并且内核不知道这个节点即将被分配出去。进程 B 没有认识到这一危险，完成了它的磁盘搜索，并且把空闲索引节点填入超级块表（假设这样），分配一个索引节点，并离开这里。然而，节点 I 在超级块空闲索引节点号表中。当进程 A 醒来时，它才完成索引节点 I 的分配。现在，假设进程 C 接着请求一个索引节点，并且碰巧从超级块空闲表中提取了索引节点 I。当它得到该索引节点的内存索引节点拷贝时，它将发现它的文件类型被置位了，这意味着该索引节点已经分配出去了。内核检查这一竞争条件，并且由于发现该索引节点已被

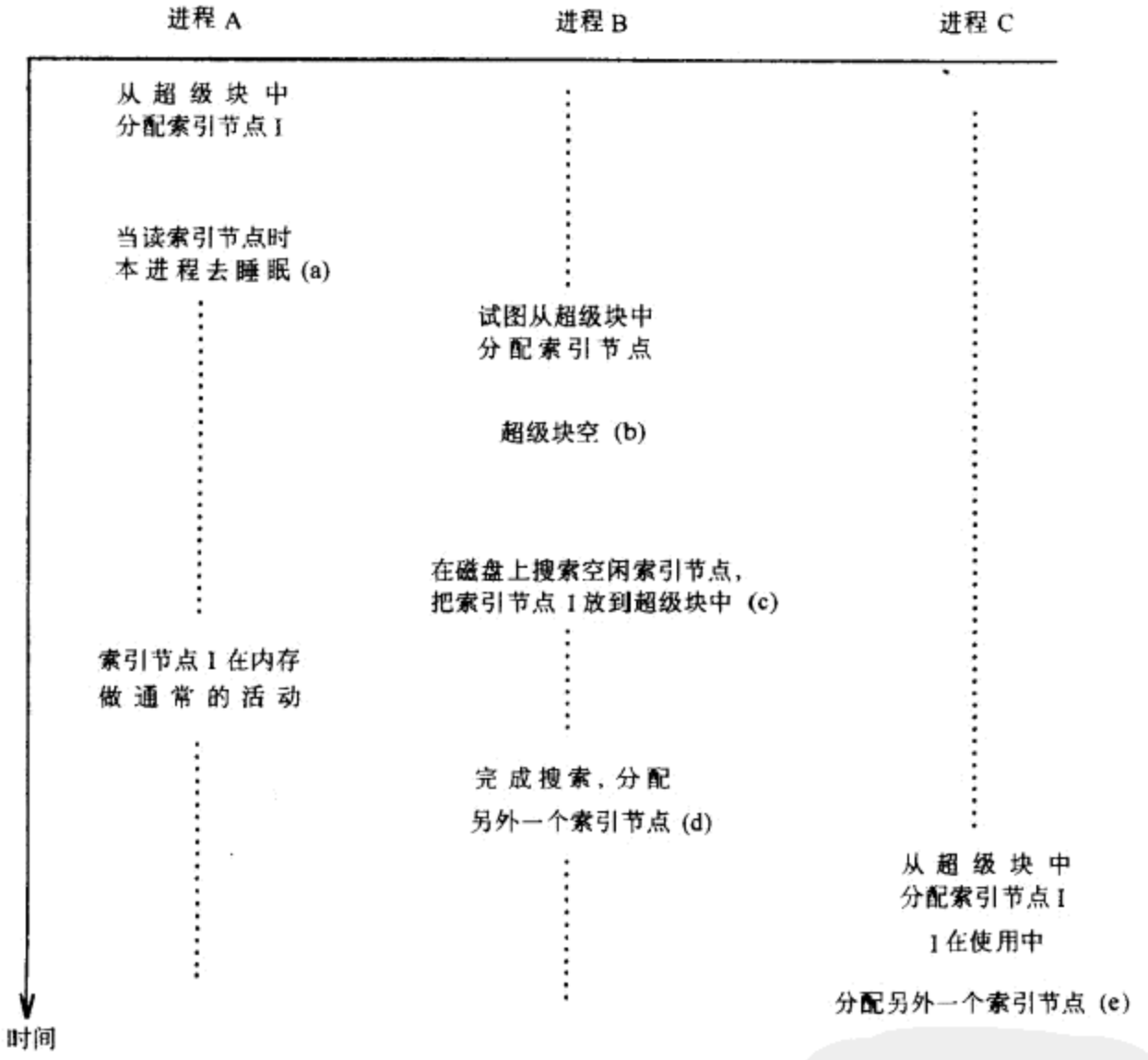


图 4-16 分配索引节点中的竞争条件

⊖ 与上一章一样，此处“进程”这一术语将意味着“为一个进程而动作着的核心。”

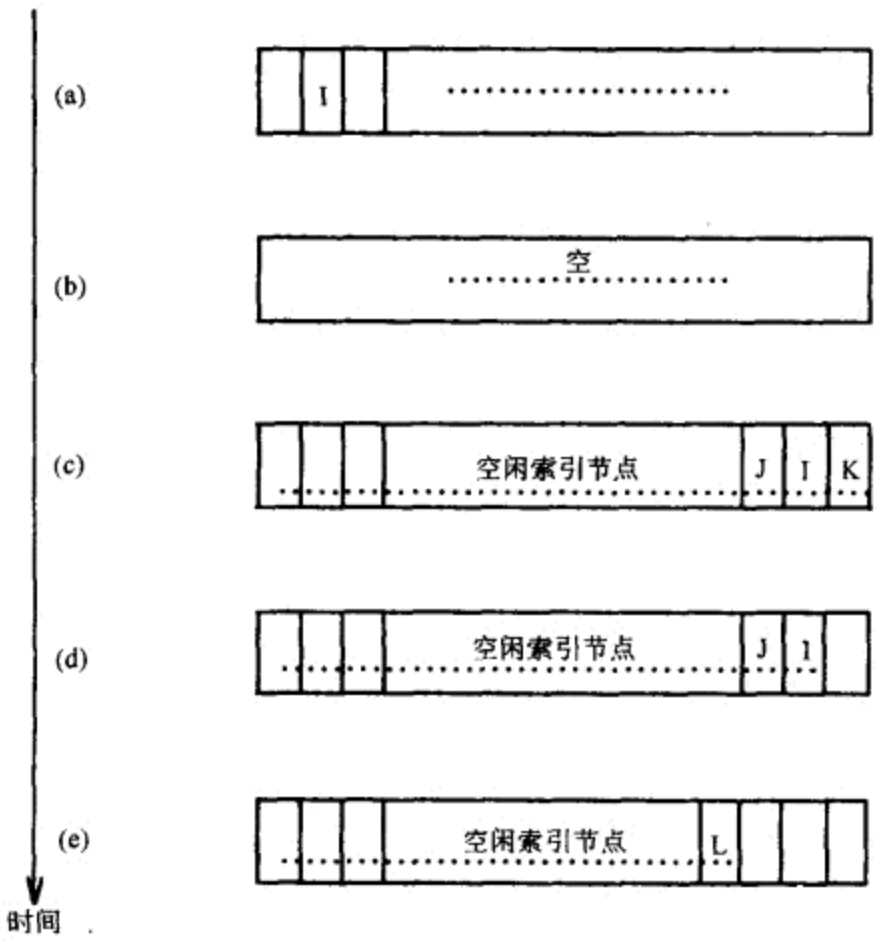


图 4-17 分配索引节点中的竞争条件（续）

分配出去，而试图分配一个新索引节点。显然，在算法 ialloc 中，把索引节点分配出去之后，立即把被修改了的索引节点写回磁盘，会使竞争机会减少，因为文件类型字段将标记出该索引节点已被使用。

当从磁盘读入一组新的空闲索引节点时，为超级块索引节点表加锁也会防止其他竞争条件。如果超级块表没有上锁，则一个进程可能会发现它为空，并试图从磁盘上往它那儿搬空闲索引节点，而且偶尔会因等候 I/O 完成而睡眠。假设第二个进程也试图分配一个新索引节点，并且发现该表为空，则它也会试图从磁盘往超级块表上移空闲索引节点。最好的情况是，两个进程都重复了它们的努力，并且浪费了 CPU 时间。最糟的情况是，上一段描述的那类竞争条件发生得更经常。类似地，如果一个释放索引节点的进程不检查该表上锁，则它会在别的进程正在从磁盘移的过程中，重写已经在空闲表中的索引节点。同样，上面描述的竞争条件会发生得更经常。虽然内核完满地处理了它们，但系统性能变坏。在超级块空闲表上使用锁会防止这样的竞争条件。

4.7 磁盘块的分配

当进程往文件上写数据时，内核必须从文件系统中分配磁盘块，以用作直接数据块，当然有时也用作间接数据块。文件系统超级块包含一个用来把文件系统中的空闲磁盘块高速缓冲起来的数组。实用程序 mkfs（文件系统生成）把一个文件系统的数据块组织到一张链表中，表中的每个链是一个磁盘块，块中包含的是一个数组，数组的分量是空闲磁盘块号，并且，数组中有一个分量是链表上的下一块的块号。图 4-18 给出了该链表的一个例子，此处第一块是超级块空闲表，而链表上的位于后面的块包含更多的空闲块号。

当内核想要从文件系统中分配一块（算法 alloc，图 4-19）的时候，它把超级块表中下

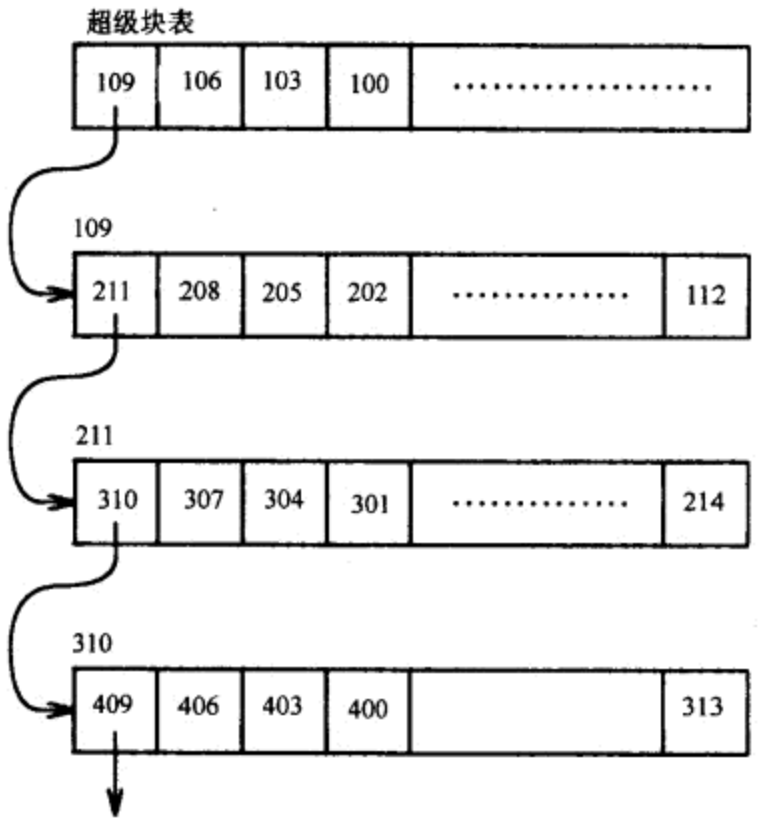


图 4-18 空闲磁盘块号链接表

一个可用的块分配出去。一旦分配出去，则直到该块变为空闲时为止，不能对它再分配。当所分配的是超级块高速缓冲中的最后一个可用块时，内核把它作为指向包含一张空闲块表的块的指针看待。它读该块，把新的块号表填充超级块数组，然后使用原来的块号。它为该块分配一个缓冲区，并且清除该缓冲区数据(将它清为0)。现在，磁盘块被能分配了，并且内核有一个工作缓冲区。如果文件系统不包含空闲块了，则调用者进程收到一个错误指示。

```
算法 alloc /* 文件系统块分配 */
输入:文件系统号
输出:用于新块的缓冲区
{
    while (超级块被锁住)
        sleep(事件"超级块解锁");
    从超级块空闲表中摘下该块;
    if (摘下的是空闲表中最后一块)
    {
        为超级块上锁;
        把刚从空闲表中取出的那块从磁盘上读出(算法 bread);
        把块中的所有块号都拷贝到超级块中;
        释放块缓冲区(算法 brelse);
        为超级块解锁;
        wake up(事件"超级块解锁");
    }
    为从超级块表中摘下的块获得一个缓冲区(算法 getblk);
    清除缓冲区内容;
    空闲块总计数值减 1;
    标记超级块被修改过了;
    return(缓冲区);
}
```

图 4-19 分配磁盘块算法

如果进程往文件上写大量数据，它会重复地向系统要求分配磁盘块，以存储数据，但内核每次只能分配一块。程序 mkfs 试图组织好空闲块号的原始链接表，使得分配给一个文件的那些块号互相靠近。这样一来，当一个进程顺序地读一个文件时，就能减少磁盘寻找时间和等待时间，所以有益于提高性能。图 4-18 的块号是按一种基于磁盘旋转速度的正常形式画出的。不幸的是，当进程由于写文件及删除文件而频繁使用磁盘块时，空闲块链接表上的块号次序就被打乱了，因为块号是随机地进入与离开空闲表的。内核没有打算给空闲表上的块号排序。

释放磁盘块的算法 free 与分配磁盘块的算法刚好相反。如果超级块表未满，新释放的块的块号就被放到超级块表中。然而，如果超级块表满了，则新释放的块就变为一个链接块；内核把超级块表写到这块上；并且把这块写到磁盘上。然后它把新释放的块的块号放到超级块表中：那个块号就成了表中的唯一成员。

图 4-20 示出在超级块空闲表上的一个表项上开始的 alloc 与 free 操作序列。首先，内核

图 4-20　请求与释放磁盘块

a)原始配置　b)在释放了第 949 块之后　c)在分配出第 949 块之后　d)在分配第 109 块之后填满超级块空闲表

释放第 949 块,把这个块号放到空闲表上。随后它分配一块,即从空闲表上取走第 949 块,最后,它分配一块即从空闲表上取走第 109 块。因为现在超级块空闲表为空了,所以内核把第 109 块,即链接表上的下一个链的内容拷贝到超级块空闲表中,从而把该表重新填满。图 4-20d 给出了满的超级块表,以及下一个链块——第 211 块。

分配和释放索引节点算法与分配和释放磁盘块算法是类似的,内核都使用超级块作为含有空闲资源——块号或索引节点号的索引的高速缓冲。它维护一个块号链接表,而文件系统中的每个空闲块号都作为链接表的某个元素出现在该表中,但对于空闲索引节点却不存在这样的链表。所以要这样区别对待的原因有如下三个:

(1)内核能通过观察来确定一个索引节点是否空闲:如果文件类型字段被清除了,则索引节点空闲。内核不需要另外的机制描述空闲索引节点。然而,内核却不能只是看一下磁盘块就确定出这个磁盘块是否空闲,因为它不能区别出一个位型究竟是用来指示出磁盘块空闲呢,还是数据刚好是这个位型的值。因此,内核需要外部的办法来标识出空闲块,而传统的实现已经使用了一个链接表的方法。

(2)可以借用磁盘块作链接表:一个磁盘块很容易容纳大的空闲块号表。但是索引节点却没有那么合适的地方,用来成批地存储大的空闲索引节点号表。

(3)用户们消耗磁盘块资源比它们消耗索引节点要快。因此,当为得到空闲索引节点而搜索磁盘时,在性能上的下降不像为得到空闲磁盘块而搜索磁盘时显得那么至关重要。

4.8 其他文件类型

UNIX 系统还支持另外两种文件类型:管道文件和特殊文件。管道文件也称为 fifo(即"先进先出"的意思),它的数据是短暂的,在这点上它不同于正规文件。一旦数据从管道上读出,就不再能从管道上读它了。还有,从管道上读出数据的次序与往管道上写入数据时的次序相同,系统不允许偏离那个次序。内核往管道中存储数据时与它往一个普通文件中存储数据时的方法相同,只不过它仅使用直接块而不使用间接块。下一章将考察管道的实现。

UNIX 系统中最后一个文件类型是特殊文件,包括块设备特殊文件和字符设备特殊文件。两种文件都指明了设备,因此文件索引节点不引用任何数据,而是含有两个称为主与次的设备号。主设备号指出像终端或磁盘这样的设备类型,次设备号指出这类设备的装置号。第 10 章将详细考察特殊设备。

4.9 本章小结

索引节点是描述文件属性的数据结构,其中包括文件数据在磁盘上的布局。索引节点有两个版本:磁盘拷贝与内存拷贝。磁盘拷贝存储着文件处于未使用状态时的索引节点信息;内存拷贝记录着关于活跃文件的信息。在系统调用 creat,mknod,pipe 和 unlink 期间,算法 ialloc 和 ifree 控制着给文件分配磁盘索引节点,当进程存取文件时,算法 iget 和 iput 则控制着内存索引节点的分配。算法 bmap 根据预先提供的在文件内的字节偏移量,来确定文件的磁盘块的位置。目录也是文件,是用来把文件名分量关联到索引节点号上去的文件。算法 namei 把被进程操纵的文件名,转换成由内核内部使用的索引节点。最后,内核使用算法 alloc 和 free 控制给文件分配新磁盘块。

本章描述的数据结构由链接表、散列队列和线性数组组成,因此,操纵这些数据结构的算

法是简单的。由于算法之间的交互作用而引起的竞争条件,导致了复杂情况的出现。在本章的正文中已经指出了某些与时间有关的问题,然而算法并不复杂,而且阐明了系统设计的简明性。

此处解释的数据结构与算法属于内核内部,对用户是不可见的。就整个系统的体系结构(图 2-1)来说,本章中所描述的算法占据的是文件子系统的下半部。下一章考察系统调用。系统调用提供了对文件系统的用户接口,它描述了文件子系统的上半部,而上半部引用此处所描述的内部算法。

4.10 习题

1. 按照 C 语言惯例,数组下标是从 0 开始计算的。为什么索引节点号是从 1 而不是从 0 开始的?

2. 若一个进程,因发现高速缓冲区索引节点为上锁状态,而在算法 iget 中睡眠,为什么它醒来之后必须重新开始循环?

3. 描述一个算法,该算法把内存索引节点作为输入,并修改相应的磁盘索引结点。

4. 算法 iget 与 iput 没有要求提高处理机执行级以封锁中断。这意味着什么?

5. 怎样才能高效地为 bmap 中对于间接块的循环进行编码?

6. 执行图 4-21 的 shell 命令文件。它建立了目录"junk",并在该目录中建立了五个文件。在做了几次控制命令 ls 之后,命令 chmod 关闭了对该目录的读许可权。现在,当不同的 ls 命令被执行时,会发生什么? 在变更目录到"junk"后会怎样? 在恢复了读许可权但从"junk"中撤销了执行(搜索)许可权之后,重复该实验,会发生什么? 内核中发生什么才会引起这个效应?

```
mkdir junk
for i in 1 2 3 4 5
do
echo hello > junk/$ i
done
ls -ld junk
ls -l junk
chmod -r junk
ls -ld junk
ls junk
ls -l junk
cd junk
pwd
ls -l
echo
cd ..
chmod +r junk
chmod -x junk
ls junk
ls -l junk
cd junk
chmod +x junk
```

图 4-21 对目录的读许可权与对目录的搜索许可权之间的区别

7. 给出系统Ⅴ上的一个目录项的当前结构。一个文件系统能包含的文件数目的最大值是什么?

8. UNIX 系统Ⅴ允许一个路径名分量最长达 14 个字符。namei 把一个分量中多余的字节截掉。应该怎样设计文件系统和相应的算法以允许任意长度的分量名?

9. 假设某用户有一个私人版本的 UNIX 系统,它对 UNIX 系统做了改变,使一个路径名分量能由 30 个字符组成。该操作系统的私人版本按照与标准操作系统相同的方式存储目录项,只是有一点不同,即目录项长为 32 个字节而不是 16 个字节。如果该用户把这个私人文件系统安装到标准系统上,当进程存取私人文件系统上的文件时算法 namei 中会发生什么?

*10. 考虑把路径名转换成索引节点的算法 namei。当搜索在进行时,内核检查出当前工作索引节点是目录索引节点。其他进程有可能删除(用系统调用 unlink)该目录吗? 内核怎样才能阻止这一点? 下一章将回到这一问题上来。

*11. 设计一个通过避免线性搜索来改进搜索路径名的效率的目录结构。考虑两个技术：散列和 n 元树。

*12. 设计一个通过把常用文件名高速缓冲起来以减少为查到文件名而搜索目录的次数的方案。

*13. 从理论上说，文件系统永远不会含有其索引节点号比被 ialloc 使用的“铭记”索引节点号小的空闲索引节点。为什么这一断言有可能不成立呢？

*14. 正如本章所描述的那样，超级块是一个磁盘块，并且除了包含空闲表之外还包含其他信息。因此，超级块空闲表不能包含象在空闲磁盘块的链接表上的一个磁盘块中可能储存的那么多的空闲块号。链接表上一个磁盘块中应存储多少个空闲块号才是最适宜的？

*15. 讨论用位图而不是块的链接表来记录空闲磁盘块的系统实现。这一方案的优点和缺点是什么？

第 5 章　文件系统的系统调用

上一章描述了文件系统的内部数据结构和对这些数据结构进行操作的算法。本章将用上一章给出的概念，讨论文件系统的系统调用。我们先讨论存取已存在的文件的系统调用，如 open，read，write，lseek 和 close，然后介绍创建新文件的系统调用，即 creat 和 mkmod，其后给出管理索引节点和文件系统的系统调用，即 chdir，chroot，chown，chmod，stat 和 fstat。本章还研究了更高级的系统调用 pipe 和 dup，它们对实现 shell 中的管道是很重要的。系统调用 mount 和 umount 扩充了对用户可见的文件系统树；link 和 unlink 修改文件系统层次的结构。本章还给出文件系统的抽象表示，这种表示允许支持各种文件系统，只要它们与标准接口一致。本章最后一节介绍文件系统的维护。本章引入了三种内核数据结构：文件表、用户文件描述符表和安装表（mount table)。系统中每个打开的文件在文件表中都占有一项；对某进程已知的每个文件描述符在用户文件描述符表中分配有一项；而安装表中含有每个活动的文件系统的信息。

图 5-1 给出这些系统调用与前章所描述的算法之间的关系。该图将系统调用分成几类，但有些系统调用可出现在多个分类里：

- 返回文件描述符的系统调用，这些文件描述符要在其他系统调用中使用。
- 使用算法 namei 来分析路径名的系统调用。
- 使用算法 ialloc 和 ifree 来分配和释放索引节点的系统调用。
- 设置或修改文件属性的系统调用。
- 使用算法 alloc，free 和缓冲区分配算法，对进程进行 I/O 操作的系统调用。
- 改变文件系统结构的系统调用。
- 允许一个进程改变其对文件系统树的看法的系统调用。

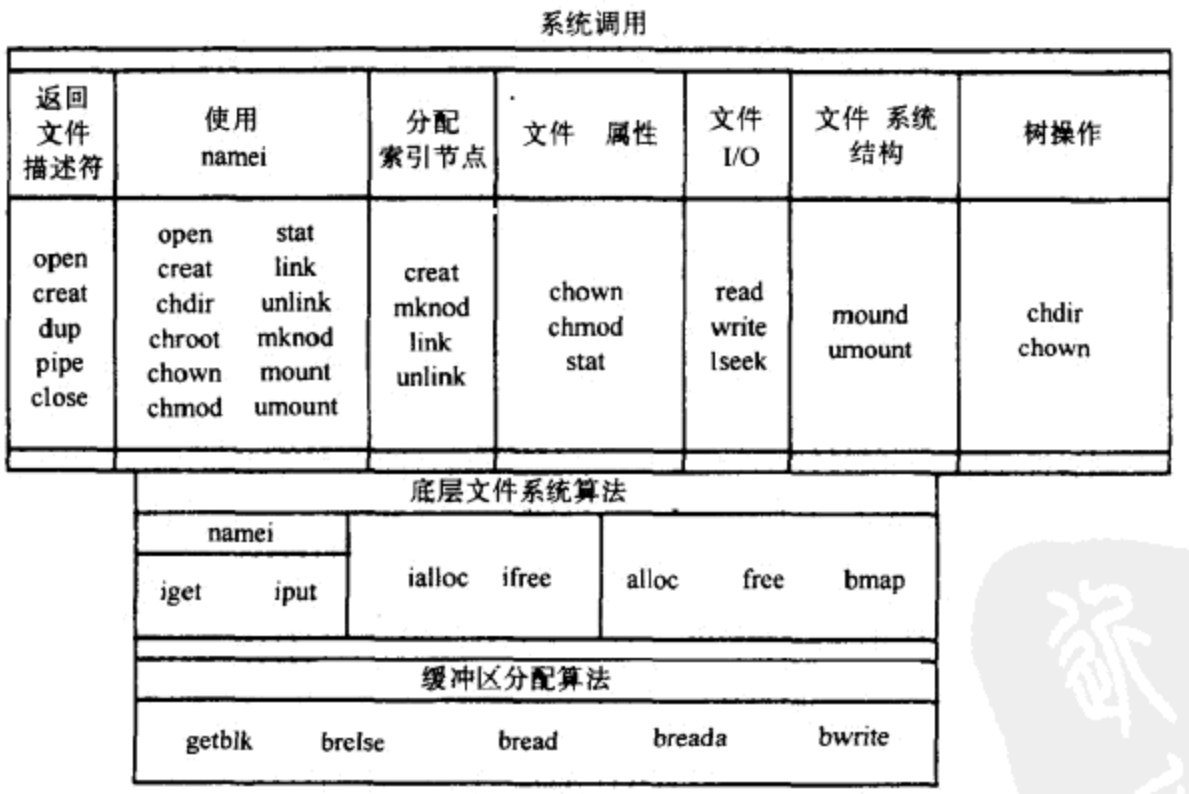


图 5-1　文件系统调用以及与其它算法的关系

5.1　系统调用 open

系统调用 open 是进程要存取一个文件中的数据所必须采取的第一步。系统调用 open 的语法格式是：

```
fd=open (pathname, flags, modes);
```

这里，pathname 是文件名；flags 指示打开的类型（如读或写）；modes 给出文件的许可权（如果文件正在被建立）。系统调用 open 返回一个称为文件描述符的整数[⊖]。其他系统调用，如读、写、定位和拷贝文件描述符、设置文件的 I/O 参数、确定文件状态和关闭文件等，都要使用系统调用 open 返回的文件描述符。

内核用算法 namei（见图 5-2）在文件系统中查找文件名参数。在内核找到内存中的索引节点后，它检查打开文件的许可权，然后为该文件在文件表中分配一个表项。文件表表项中有一个指针，指向被打开文件的索引节点，还有一个字段，指示文件中的偏移量，也就是内核预期下次读/写操作要开始的地方。在 open 调用期间，内核将这个偏移量置为 0，这意味着它默认最初的读、写操作是从文件头开始。另外一种方法是，一个进程以追加写（wirte-append）方式打开一个文件。在这种情况下，内核将偏移量置为文件的大小。内核还要在进程 u 区中的一个私用表中分配一个表项并记下该表项的索引。这个表叫做用户文件描述符表。表项的索引就是返回给用户的文件描述符。用户文件描述符表中的表项指向对应的全局文件表中的表项。

```
算法　open
输入：文件名
      打开类型
      文件许可权方式（对以创建方式打开而言）
输出：文件描述符
{
    将文件名转换为索引节点（算法 namei）;
    if　（文件不存在或不允许存取）
        return（错）;
    为索引节点分配文件表项，设置引用数和偏移量;
    分配用户文件描述符表项，将指针指向文件表项;
    if　（打开的类型规定清文件）
        释放所有文件块（算法 free）;
    解锁（索引节点）;　/*　在上面的 namei 算法中上锁　*/
    return（用户文件描述符）;
}
```

图 5-2　打开文件的算法

假定一个进程执行下列代码：

⊖ 所有的系统调用当失败时都返回 -1。今后讨论系统调用的语法时，我们不再显式地提到返回值 -1。

```
fd1 = open ( "/etc/passwd", O_RDONLY)⊖;
fd2 = open ( "local", O_RDWR);
fd3 = open ( "/etc/passwd", O_WRONLY);
```

该进程打开文件“/etc/passwd”两次，一次只读，一次只写；还以读写方式打开文件“local”一次。图 5-3 给出索引节点表数据结构、文件表数据结构和用户文件描述符数据结构之间的关系。每个 open 返回给进程一个文件描述符，它在用户文件描述符表中对应的表项指向文件表中唯一的表项，即使同一文件（“/etc/passwd”）被打开两次。一个被打开文件的所有实例所对应的那些文件表项都指向内存索引节点表中的同一表项。上述进程能够读写文件“/etc/passwd”，但只能通过图中的文件描述符 3 和 5。内核在 open 调用所分配的文件表项中记下读/写能力。

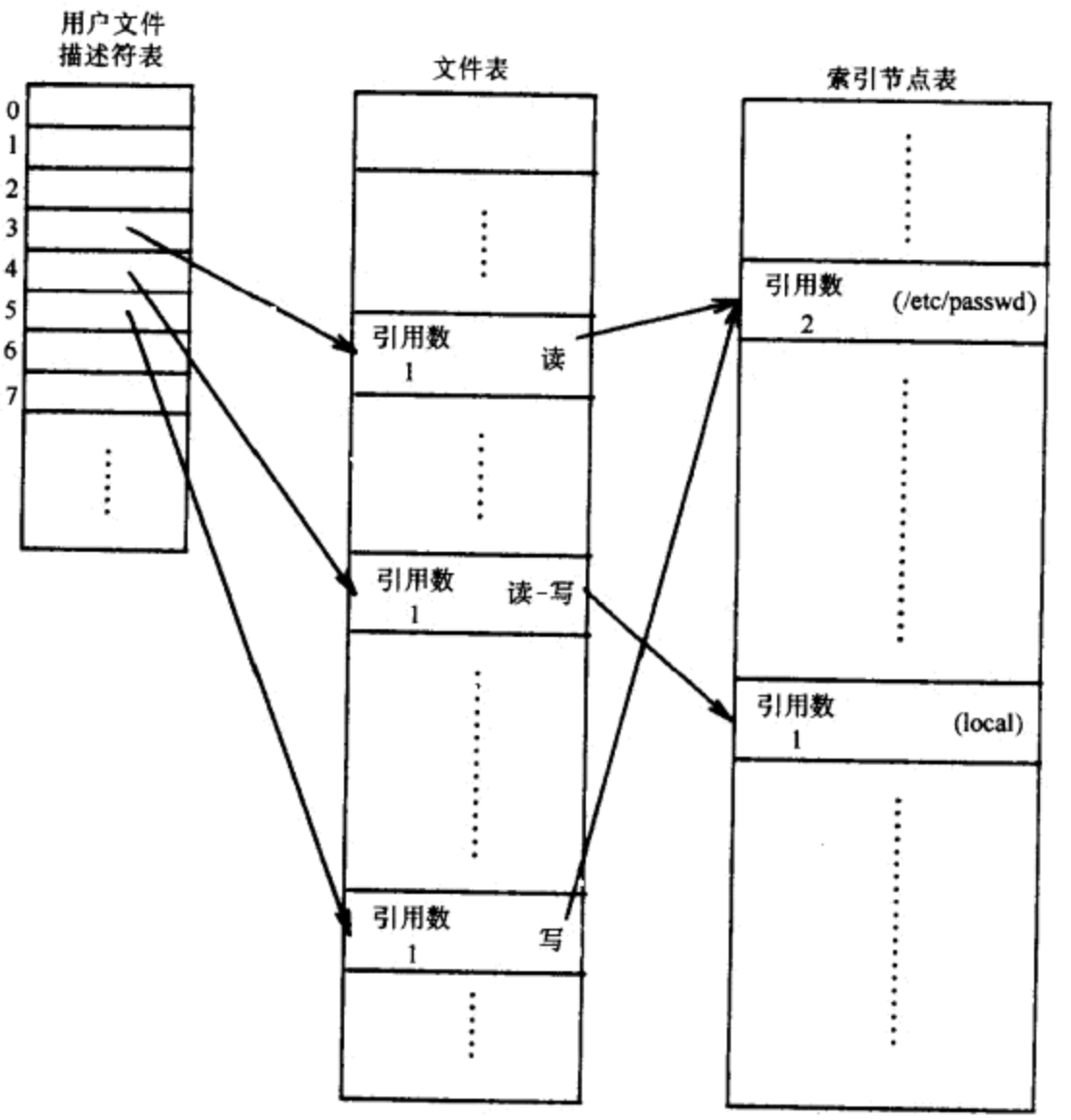


图 5-3　打开文件后的数据结构

假定第二个进程执行下列代码：

```
fd1 = open ( "/etc/passwd", O_RDONLY);
```

⊖ 系统调用 open 的定义要求三个参数（第三个参数用作 open 的创建方式），但程序员通常只用头两个。C 编译器不检查参数个数的正确性。典型的实现方式是传送头两个参数和第三个“废”参数（刚好在栈上的随便什么东西）给内核。内核不检查第三个参数，除非第二个参数要求它必须检查。这样就允许程序员编程时只用两个参数。

```
fd2 = open ( "private", O_RDONLY);
```

图 5-4 给出在两个进程打开文件期间（没有其他进程），相应的各数据结构之间的关系。再强调一遍，每个 open 调用都导致在用户描述符表和内核文件表中分配一个唯一表项。但在内核的内存索引节点表中，对每个文件只有一个表项。

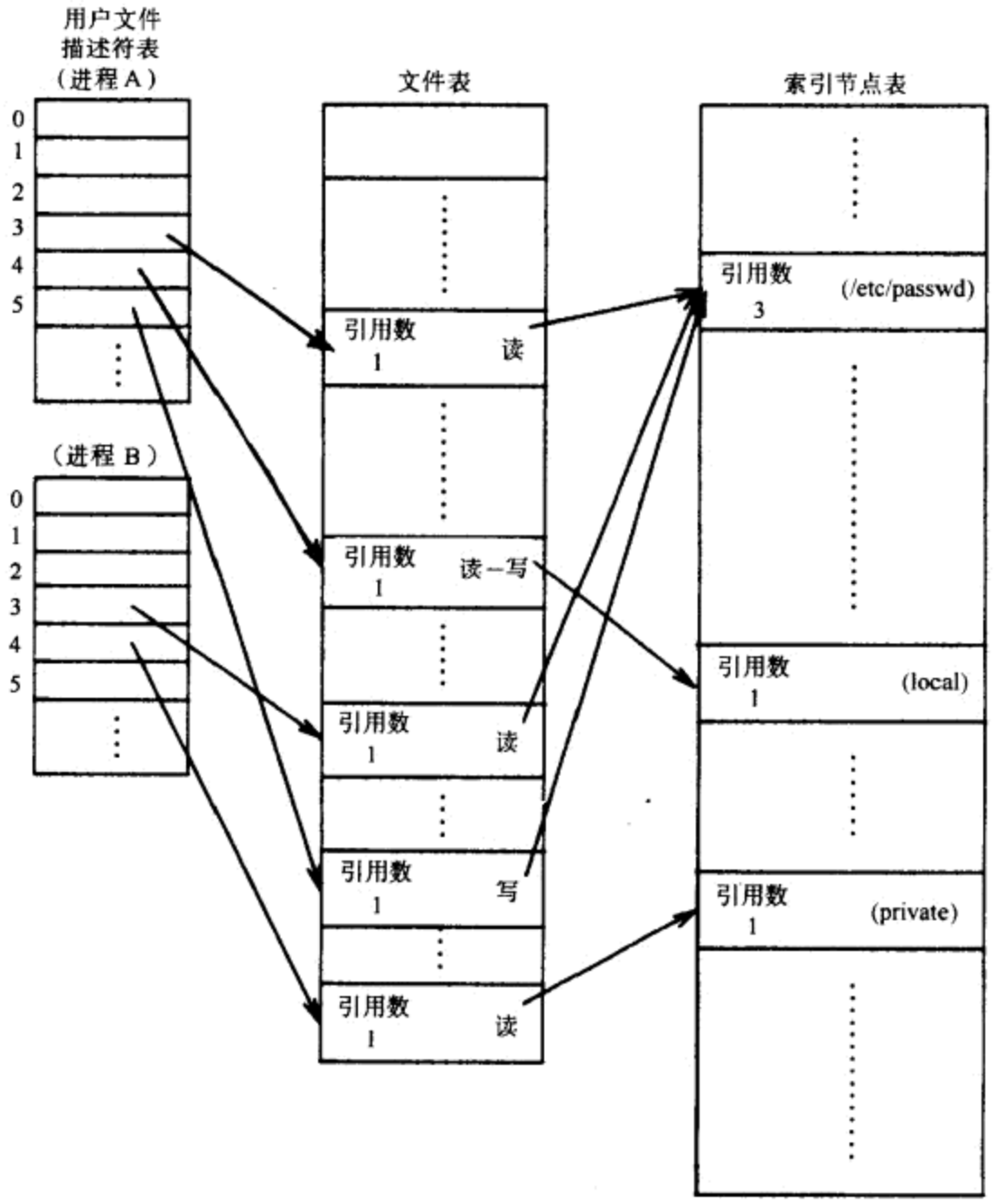


图 5-4　两个进程打开文件后的数据结构

可以想象，本来用户文件描述符表的表项中，可以含有下次对文件进行 I/O 操作的位置偏移量和直接指向对应文件的内存索引节点表项的指针，这样就不需要一个单独的内核文件表。在上面给出的例子里，用户文件描述符表项和内核文件表项之间也恰好是一对一的关系。然而，Thompson 指出，他把文件表作为一个独立的数据结构来实现，是为了在若干用户文件描述符之间能够共享偏移量指针（见［Thompson 78］第 1943 页）。在 5.13 节和 7.1 节中介绍的系统调用 dup 和 fork 就是通过管理这些数据结构来允许这种共享的。

前三个用户文件描述符（0，1 和 2）分别叫做标准输入、标准输出和标准错误文件描述符。UNIX 系统中的进程习惯上使用标准输入描述符读输入数据，用标准输出描述符写输出

数据，用标准错误描述符写出错数据（信息）。操作系统并没有假定这些文件描述符有任何特殊性。某一组用户可以有自己的习惯，如将文件描述符 4，6 和 11 定为特殊文件描述符。但是从 0 开始计算显得更自然些（在 C 中）。我们将在第 7 章看到，若所有的用户程序都遵守上述惯例，会使他们更容易地通过管道（pipe）进行通讯。一般地说，控制终端（见第 10 章）被用作标准输入、标准输出和标准错误文件。

5.2　系统调用 read

系统调用 read 的语法格式是：

```
number = read (fd, buffer, count);
```

这里，fd 是由 open 返回的文件描述符，buffer 是用户进程中的一个数据结构的地址。在该调用成功地结束时，该地址中将存放所读的数据。count 是用户要读的字节数，number 是实际读的字节数。图 5-5 给出读一个正规文件的算法 read。内核先从图 5-3 的指针得到对应于该用户文件描述符的文件表表项。然后它设置u区中的几个I/O参数(图5-6)，这样就不

```
算法　read
输入：用户文件描述符
      用户进程中的缓冲区地址
      要读的字节数
输出：拷贝到用户区的字节数
{
      由用户文件描述符得到文件表项；
      检查文件的可存取性；
      在 u 区中设置用户地址，字节计数，输入/出到用户的参数；
      从文件表中得到索引节点；
      索引节点上锁；
      用文件表中的偏移量设置 u 区中的字节偏移量；
      while　（字节数不满足）
      {
         将文件偏移量转换为磁盘块号（算法 bmap）；
         计算块中的偏移量和要读的字节数；
         if　（要读的字节数为 0）
            /*　企图读文件尾　*/
            break;　/*　出循环　*/
         读块（如果要超前读；用算法 breada，否则用算法 bread）；
         将数据从系统缓冲区拷贝到用户地址；
         修改 u 区中的文件字节偏移量、读计数、再写的用户空间地址字段；
         释放缓冲区；　/*　在 bread 中上锁　*/
      }
      解锁索引节点；
      修改文件表中的偏移量，用作下次读；
      return（已读的总字节数）；
}
```

图 5-5　读文件的算法

方式	指示读或写
计数	读或写的字节数
偏移量	文件中的字节偏移量
地址	拷贝数据的目的地址，在用户或内核存储空间
标志	指出地址是在用户存储空间还是在内核存储空间

图 5-6　在 u 区中保存的 I/O 参数

必将它们作为参数来传递。更确切地说，它设置 I/O 方式来表明它正在做“读”；设置标志表示该 I/O 将进入用户地址空间；设置表明要读的字节数的计数字段、用户数据缓冲区的目标地址以及来自文件表的偏移量字段——指示 I/O 操作在文件中开始的字节偏移量。内核设置了 u 区中的 I/O 参数后，它由文件表项的指针找到索引节点，并在它读该文件之前将该索引节点上锁。

这时，该算法进入一个循环，直到该 read 被满足。内核使用算法 bmap 将文件的字节偏移量变为磁盘块号，并记下在该块中 I/O 开始的字节偏移量以及它在该块中应该读多少字节。在将该块读入缓冲区后（也许使用下面将要说明的提前读算法 bread 和 breada），内核将数据从该块中拷贝到用户地址空间。然后，它根据刚读的字节数，修改 u 区中的 I/O 参数，增大文件字节偏移量和用户进程中的地址，使之成为下一次数据将要存放的地址。同时，内核减少它尚需读的字节数，以便满足用户的读请求。如果该用户的读请求还没满足，内核将重复整个循环——将文件的字节偏移量变为磁盘块号；从磁盘将该块读入系统缓冲区；再将数据从该缓冲区拷贝到用户进程；释放缓冲区，最后更新 u 区中的 I/O 参数。该循环在下列情况下结束：内核完全地满足了用户的请求；文件中不再含有数据；或内核在从磁盘中读数据或将数据拷贝到用户空间时遇到错误。循环结束后，内核根据它实际读的字节数更新文件表中的偏移量，这样，对文件的下次读操作将按顺序给出该文件的数据。系统调用 lseek（见 5.6 节）能够修改文件表中的偏移量值，从而改变一个进程读或写文件中的数据的次序。

```
#include  <fcntl.h>
main ()
{
    int fd;
    char lilbuf [20], bigbuf [1024];

    fd=open ( "/etc/passwd", O_RDONLY);
    read (fd, lilbuf, 20);
    read (fd, bigbuf, 1024);
    read (fd, lilbuf, 20);
}
```

图 5-7　读一个文件的程序例

考虑图 5-7 中的程序。其中，用户将 open 返回的文件描述符赋给变量 fd，并在其后的 read 调用中使用它。在系统调用 read 中，内核证实文件描述符参数是合法的，而且该进程已经先打开了要读的文件。内核将 lilbuf、20 和 0 这三个值存在 u 区，它们分别是用户缓冲区地址、用户要读的字节数和在读文件中的起始字节偏移量。内核计算出字节偏移量 0 是在该文件中的第 0 块并检索对应的索引节点中的第 0 块表项。假定这一块存在，内核将大小为 1024 字节的一整块读入系统缓冲区，但只将其中的头 20 个字节拷贝到用户地址 lilbuf 中去。

内核将u区中的字节偏移量增加到20并将要读的字节数减少到0。由于read已被满足，内核将文件表中的字节偏移量置为20，这样，下次将从文件中的第20字节处开始读数据。

对程序中的第二次read调用，内核还要证实文件描述符的合法性以及该进程是否已打开了要读的文件，因为内核没办法知道用户目前的读请求所涉及的文件就是上次读写中已被证实是合法的同一文件。然后，内核将用户地址bigbuf、进程要读的字节数1024及从文件表中得到的文件的起始偏移量20存到u区中。内核按照上面讲的同样方法，将文件字节偏移量转换成正确的磁盘块并读这一磁盘块。如果上次和这次读操作之间的间隔很短，就会遇到这一磁盘块仍在高速缓存中的好运气。但内核还不能用这一缓冲区就完全满足这次读请求，因为这一缓冲区的1024字节中只剩1004字节是给这次请求的。因此，内核从缓冲区中将后1004字节拷贝到用户数据结构bigbuf中，并修改u区的参数。修改后的参数表明：读循环的下次重复是从该文件的1024字节开始；应将数据拷贝到bigbuf中1004字节的位置上；为满足本次读请求，还要再读20个字节。

现在，内核又回到算法read的开始处。它将字节偏移量1024变为逻辑块偏移量1，查找索引节点中的第二个直接块号，并找到要读的正确磁盘块。它先从高速缓存中读该块，如果该块不在高速缓存中，则从磁盘上读入该块。最后，它从系统缓冲区中将20字节数据拷贝到用户进程的正确地址上。在从系统调用返回之前，内核将文件表项中的偏移量字段设置为1044，即下次访问该文件的偏移量。内核对本例子中的最后一个read调用的处理与第一个read调用的处理过程一样，只不过它这次是从文件中的1044字节开始读。

上面的例子说明，从文件系统块的边界开始的、而且大小为块的整数倍的I/O操作具有很多优点。这样做能使内核避免额外地重复算法read中的循环。这种重复是以访问索引节点以便找到数据的正确块号，以及与其他进程竞争对缓冲池的访问为代价的。标准I/O库的设计对用户隐瞒了内核缓冲区的大小。使用标准I/O库避免了一种不良性能，这种不良性能是那些低效地一点点啃文件系统的进程所固有的（见习题5.4）。

当内核通过read循环时，它决定一个文件是否需要提前读：如果一个进程顺序地读两个块，内核就假定所有后继的读都将是顺序的，直到证明不是顺序的。在循环的每次重复期间，内核将下一次逻辑块号保存在内存索引节点中，在下次循环期间，内核将当前逻辑块号和以前保存的逻辑块号相比较。如果它们相等，内核就为提前读计算物理块号并将其值保存在u区，算法breada将使用它。当然，如果一个进程根本不用读一个完整的块，内核就不必为下一块调用提前读。

回忆图4-9，一个索引节点或间接块中的某些块号可以有零值，即使其后的若干块有非零值。如果一个进程企图从这样的块中读数据，内核将在read循环中分配任意一个缓冲区，将缓冲区的内容清为零，然后把该缓冲区拷贝到用户地址，以此来满足用户的请求。这和进程遇到文件尾时的情况不一样。进程遇到文件尾意味着不会再有数据被写到用户地址的当前位置以后的任何单元。所以，当遇到文件尾时，内核不给进程返回数据（见习题5.1）。

当一个进程请求系统调用read时，内核在该调用期间锁住所涉及的索引节点。然后，进程进入睡眠，等待从缓冲区中读数据或该索引节点的间接块。如果允许另外一个进程在第一个进程睡眠期间修改文件，read将给出不一致的数据。例如，一个进程可能要读一个文件的若干块。假如在读第一块期间，该进程进入睡眠，而第二个进程要写这个文件的另外一些块，那么返回的数据就会是新、旧数据的混合。因此，索引节点在read调用期间一直是锁

住的，从而提供给进程一个和该调用开始时一致的文件视图。

在用户态中，内核能够在系统调用之间抢先一个读进程并调度其他进程运行。既然索引节点在一个系统调用结束之时已被解锁，那么就没有什么能阻止其他进程存取那个文件并修改其内容。如果从一个进程打开一个文件开始，直到它关闭该文件为止的一段时间里，系统一直锁住对应的索引节点，那是很不公平的。因为某个进程可能使一个文件老是处于打开状态，从而使其他进程永远也使用不了该文件。如果这个文件是注册进程用来检查用户口令的“/etc/passwd”文件，那么，一个居心不良的用户（或仅仅是出于无意）就能不让所有其他用户注册。为了避免这样的问题，内核在每个使用索引节点的系统调用的结尾，解锁该索引节点。如果在第一个进程的两个 read 调用之间，另一个进程修改了文件，那么第一个进程可能读到意想不到的数据，但内核数据结构是一致的。

例如，假定内核并发地执行图 5-8 中的两个进程。假定这两个进程中的任意一个在开始 read 或 write 调用之前，都完成了各自的系统调用 open。内核可能按六种顺序中的任意一种顺序执行系统调用 read 和 write：read1，read2，write1，write2；read1，write1，read2，write2；或 read1，write1，write2，read2 等等。进程 A 读的数据依赖于系统执行这两个进程的系统调用的次序。系统不能保证在打开文件以后，文件中的数据保持不变。利用文件和记录上锁的方法（见 5.4 节），可以允许一个进程在保持一个文件打开期间，保证文件的一致性。

```
#include  <fcntl.h>
/*  进程 A */
main ()
{
    int fd;
    char buf [512];
    fd = open ( "/etc/passwd", O_RDONLY);
    read (fd, buf, sizeof (buf));      /* read1 */
    read (fd, buf, sizeof (buf))       /* read2 */
}
/* 进程 B */
main ()
{
    int fd, i;
    char buf [512];
    for (i = 0; i<sizeof (buf); i++)
        buf [i] = 'a';
    fd = open ( "/etc/passwd", O_WRONLY);
    write (fd, buf, sizeof (buf));     /* write1 */
    write (fd, buf, sizeof (buf));     /* write2 */
}
```

图 5-8　一个读进程和一个写进程

最后，图 5-9 的程序说明了一个进程怎样才能不只一次地打开一个文件，并通过不同的文件描述符读取该文件。内核分别地管理与这两个文件描述符有关的文件表偏移量。因此，如果没有其他进程同时写“/etc/passwd”，那么当该进程结束时，数组 buf1 和 buf2 中的内容应该是一样的。

```
#include <fcntl.h>
main ()
{
    int fd1, fd2;
    char buf1 [512], buf2 [512];

    fd1 = open ( "/etc/passwd", O_RDONLY);
    fd2 = open ( "/etc/passwd", O_RDONLY);
    read (fd1, buf1, sizeof (buf1));
    read (fd2, buf2, sizeof (buf2));
}
```

图 5-9 用两个文件描述符读一个文件

5.3 系统调用 write

系统调用 write 的语法格式为：

```
number = write (fd, buffer, count);
```

这里，变量 fd，buffer，count 和 number 与系统调用 read 中的意义一样。写一个正规文件的算法和读一个正规文件的算法类似，然而，如果文件中还没有要写的字节偏移量所对应的块，内核要用算法 alloc 分配一个新块并将该块号存放在索引节点内容表中的正确位置上。如果这个字节偏移量是一个间接块中的偏移量，内核可能需要分配几个块，用作间接块和数据块。对应的索引节点在系统调用 write 期间是上锁的。因为内核在分配新块时可能会修改该索引节点。如果允许其他进程存取该文件，那么当几个进程同时为相同的字节偏移量分配块时，就会破坏该索引节点。当写操作结束时，如果该文件变大了，内核将修改索引节点中表示文件大小的字段。

举例来说，假定一个进程要写到一个文件的字节号为 10 240 的地方，这是目前已经写到的该文件的最高字节编号。当用算法 bmap 访问该字节时，内核将发现，不仅该文件中没有对应那个字节的块，而且也没有必要的间接块。这时，内核将分配一个磁盘块作为间接块，并将该块号写到内存索引节点中去。然后，内核为数据块分配一个磁盘块，并将它的块号写到新分配的间接块中的第一个位置上。

像在算法 read 中一样，内核在每次内层循环期间向磁盘写一块。在每次循环期间，内核要决定是写整个块还是只写块中的一部分。如果是后一种情况，内核必须先从磁盘上把该块读进来，以防止改写仍需保持不变的那些部分。但如果是写整块，内核就不必读该块，因为它总是要覆盖掉该块先前的内容。写过程一块一块地进行，但内核采用延迟写（见 3.4 节）的方法，将数据写到磁盘上，也就是说，先把数据放到高速缓存，万一另一个进程要读或写这一块，就可以避免额外的磁盘操作。将延迟写用于管道大概是最有效的，因为另一个进程要读管道并移走其中的数据（见 5.12 节）。但是，即使对正规文件，在文件只是被暂时建立并很快就要读的情况下，延迟写也能提高效率。例如，许多程序（如编辑程序和邮件程序）要在目录“/tmp”中建立临时文件并很快取走它们。使用延迟写能够减少向磁盘上写临时文件的操作次数。

5.4　文件和记录的上锁

由 Thompson 和 Ritchie 开发的早期 UNIX 系统不具有能使进程保证互斥地存取文件的内部机制。当时，正如 Ritchie 所说，因为“我们面对的不是大的、由许多独立的进程所维护的单文件数据库（见［Ritchie 81］）”，所以不必要考虑加锁机制。为了使 UNIX 系统对带有数据库应用系统的商业用户更具吸引力，现在，系统 V 已具有文件和记录的加锁机制了。文件的加锁指的是防止其他进程读或写整个文件的任何一部分的能力。记录的加锁是指防止其他进程读或写特定记录（指在文件中两个指定字节之间的部分）的能力。习题 5.9 详细探讨了如何实现文件和记录的加锁。

5.5　文件的输入/输出位置的调整——lseek

系统调用 read 和 write 的一般使用方法，提供了对文件的顺序访问的手段。但是，进程可以使用系统调用 lseek 来指定 I/O 的位置，从而实现文件的随机存取。lseek 的语法格式为：

```
position = lseek (fd, offset, reference);
```

这里，fd 是标识文件的文件描述符。offset 是一个字节偏移量。reference 指出该字节偏移量是从哪儿开始：可以从文件的头、文件的当前读/写偏移量位置或文件尾开始。position 是返回值，即下次开始读或写的字节偏移量。图 5-10 给出了一个例子。一个进程打开一个文件，读一个字节，然后调用 lseek 使文件前进了 1023 个字节（用 reference 为 1），下面便进入循环。这样，该进程读的是文件的每隔 1024 的倍数的那个字节。如果 reference 的值为 0，内核则从文件头开始，按偏移量定位；如果 reference 的值为 2，内核从文件尾部开始定位。系统调用 lseek 与让一个磁盘臂定位在一个特定磁盘扇区上的查找(seek)操作无任何关系。为实

```
#include  <fcntl.h>
main (argc, argv)
    int argc;
    char * argv [];
{
    int fd, skval;
    char c;

    if (argc! =2)
        exit ();
    fd = open (argv [1], O_RDONLY);
    if (fd = = -1)
        exit ();
    while ( (skval = read (fd, &c, 1)) = =1)
     {
        printf ( "char %c \ n", c);
        skval = lseek (fd, 1023L, 1);
        printf ( "new seek val %d \ n", skval);
    }
}
```

图 5-10　带有系统调用 lseek 的程序

现 lseek，系统要做的只是调整文件表中的字节偏移量值。这之后的系统调用 read 和 write 的操作，仍将使用文件表中的字节偏移量作为它们的起始字节偏移量。

5.6　系统调用 close

当系统不再使用一个打开的文件时，就关闭该文件。系统调用 close 的语法格式是：

```
close (fd);
```

其中，fd 是一个已打开文件的文件描述符。内核对文件描述符、对应的文件表项和索引节点表项进行相应的处理，来完成关闭文件的操作。如果文件表项的引用数由于系统调用 dup 或 fork 的结果而大于 1，那么就意味着还有其他用户文件描述符引用这个文件表项（这一点马上就会谈到）。这时，内核将引用数减 1，关闭操作便完成了。如果文件表项的引用数为 1，内核则释放该表项，使它重新可用，并释放在系统调用 open 中分配的内存索引节点（算法 iput）。如果其他进程还引用该索引节点，内核则使索引节点引用数减 1，并仍然保持它和其他进程的联系；否则，内核归还该索引节点以便再次分配，因为其引用数已经为零。当系统调用 close 结束时，对应的用户文件描述符表项为空。如果该进程还企图使用这个文件描述符，将导致出错，除非该文件描述符由于其他系统调用的结果而被重新赋值。当一个进程退出时，内核检查它的活动的用户文件描述符并在内部关闭它们。因此，没有任何进程在终止运行之后还能保持一个文件打开着。

举例来说，图 5-11 给出图 5-4 中的有关表项在第二个进程关闭它的两个文件之后的情况。在第二个进程的用户文件描述符表中，文件描述符 3 和 4 所对应的表项为空。对应的文

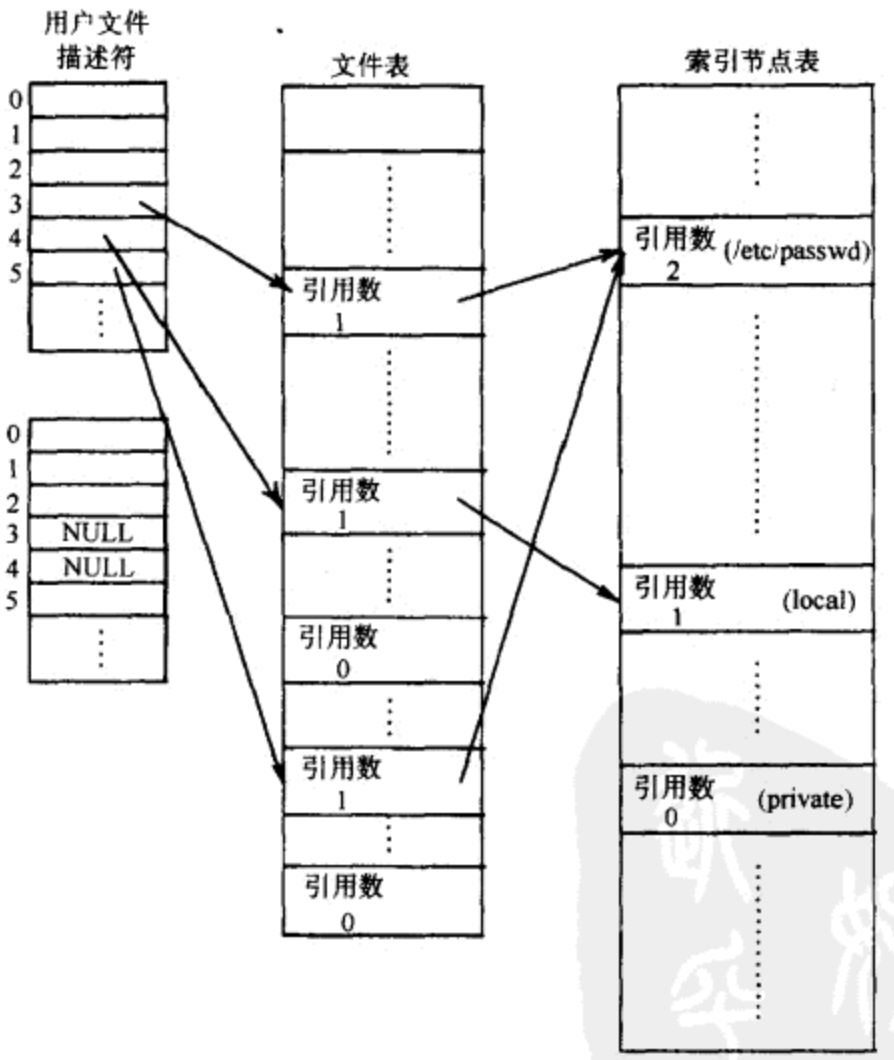


图 5-11　关闭文件后的各表

件表项的引用数为 0，表项内容也为空。文件“/etc/passwd”和“private”的索引节点引用数都被减 1。这时，文件“private”的索引节点表项在自由链表中，因为它的引用数为 0，但它的项不为空。如果另一个进程在该索引节点仍在自由链表上时，访问文件“private”，那么内核将重新使用该索引节点，这一点已在 4.1.2 节中解释了。

5.7　文件的建立

系统调用 open 给出了存取一个已存在的文件的过程，而系统调用 creat 则在系统中创建一个新文件。系统调用 creat 的语法格式为：

```
fd = creat (pathname, modes);
```

其中，变量 pathname，modes 和 fd 的意义和在系统调用 open 中的意义一样。如果以前不存在这个文件，内核就以指定的文件名和许可权方式创建一个新文件；如果该文件已经存在，内核就清该文件（释放所有已存在的数据块并将文件大小置为 0），但这要受适当的文件存取许可权方式的限制[⊖]。图 5-12 是建立文件的算法。

内核首先用算法 namei 分析路径名。在分析路径名期间，内核要重复地使用该算法。然

```
算法　creat
输入：文件名
      许可权方式
输出：文件描述符
{
    取对应文件名的索引节点（算法 namei）;
    if  （文件已经存在）
    {
        if  （不允许访问）
        {
            释放索引节点（算法 iput）;
            return（错）;
        }
    }
    else  /*  文件还不存在  */
    {
        从文件系统中分配一个空闲索引节点（算法 ialloc）;
        在父目录中建立新目录项：包括新文件名和新分配的索引节点号;
    }
    为索引节点分配文件表项，初始化引用数;
    if  （文件在创建时已存在）
        释放所有文件块（算法 free）;
    解锁（索引节点）;
    return（用户文件描述符）;
}
```

图 5-12　建立文件的算法

⊖ 系统调用 open 也规定了两种类型标志，O_CREAT（创建）和 O_TRUNC（截零）：如果一个进程在 open 中规定了类型 O_CREAT，而且，对应的文件不存在，内核将创建该文件。如果文件已经存在，则内核不清该文件，除非指定了类型 O_TRUNC。

而，当 namei 达到路径名的最后一个分量，即内核将要创建的文件名时，namei 记下其目录中的第一个空目录槽的字节偏移量，并将该偏移量保存在 u 区中。如果内核在目录中没有找到该路径名分量，那么它最终会将这个文件名写到刚刚找到的空槽中。如果该目录中已没有空槽，内核则记下目录尾的偏移量，并在那儿建立一个新槽。内核还要在 u 区中记下被查找的目录的索引节点，并锁住该索引节点。当前的这个目录将成为新文件的父目录。此时，内核还不把新文件名写入该目录，这是为了在以后发生错误事件时，可以少做一点儿恢复工作。内核要检查该目录是否允许该进程写：因为一个进程，由于系统调用 creat 的结果，将做写目录操作。对一个目录允许写，意味着允许进程在该目录中创建文件。

假如以前不存在给定名字的那个文件，内核则用算法 ialloc（4.6 节）给新文件分配一个索引节点。然后，内核按保存在 u 区中的字节偏移量，把新文件名和新分配的索引节点号写到父目录中。这之后，它释放父目录的索引节点，因为从内核搜索该目录来查找文件名时起，它一直占用着该结点。此时，父目录中已含有新文件名和它的索引节点号。内核在将含有新名字的目录写到磁盘之前，要先将新分配的索引节点写到磁盘上（算法 bwrite）。如果系统在对索引节点和父目录的两次写操作之间突然失效，那么系统中将存在一个不被任何路径名所引用的索引节点，但系统还会正常工作。另一方面，如果在写新分配的索引节点之前，先把父目录写到磁盘，且系统在两次写之间垮台，那么文件系统中就会有个引用坏索引节点的路径名（详见 5.16 节）。

如果给定的文件在系统调用 creat 之前就已存在，那么内核在查找该文件名期间会找到它的索引节点。这个老文件必须要求该进程有写许可权，这样该进程才能以同样的名字建立一个“新”文件。这是因为，在系统调用 creat 期间，内核要改变文件的内容：它清该文件，用算法 free 释放其所有数据块，这样该文件看上去就像新建的文件一样。然而，新文件的所有者和许可权方式仍和原来的文件一样，内核并不将所有权重新赋为该进程的所有者，同时也忽略该进程指定的许可权方式。最后，内核不检查已存在的文件的父目录是否有写许可权，因为它并不改变目录的内容。

接着，系统调用 creat，按与系统调用 open 同样的方法进行操作。内核为创建的文件在文件表中分配一个表项，从而进程能够往该文件写。内核还在用户文件表中分配一个表项，最后返回这一表项的索引作为用户文件描述符。

5.8　特殊文件的建立

系统调用 mknod 用来在文件系统中建立一些特殊文件，包括有名管道、设备文件和目录。该系统调用与 creat 的类似之处是，内核分配一个索引节点。系统调用 mknod 的语法格式是：

```
mknod (pathname, type and permissions, dev);
```

其中，pathname 是要建立的节点名，type 及 permissions 给出结点的类型（如目录）和要被建立的新文件的访问许可权，dev 为块特殊文件和字符特殊文件（第 10 章）规定主设备号和次设备号。图 5-13 描述了建立一个新节点的算法 mknod。

内核首先在文件系统中查找要建立的文件名。如果该文件还不存在，内核则在磁盘上分配一个新的索引节点，并将新文件名和索引节点号写入父目录。内核在索引节点中设置文件

类型字段，来表示该文件的类型是管道、目录或特殊文件。最后，如果是块设备特殊文件或字符设备特殊文件，内核还要将主设备和次设备号写入该索引节点。如果 mknod 正在创建的是一个目录节点，在系统调用完成之后，该节点会存在，但其内容的格式是不对的（没有“.”和“..”目录表项）。习题 5.33 考虑将一个目录变成正确格式所需要的其他一些步骤。

```
算法  mknod
输入：节点（文件名）
      文件类型
      许可权方式
      主设备号和次设备号（用于块和字特殊文件）
输出：无
{
    if  （新结点不是有名管道而且用户不是超级用户）
        return（错）；
    取新节点的父索引节点（算法 namei）；
    if  （新节点已经存在）
    {
        释放父索引节点（算法 iput）；
        return（错）；
    }
    从文件系统中为新节点分配一个自由索引结点，（算法 ialloc）；
    在父目录中建立新目录表项：
            包括新节点名和新分配的索引节点号；
    释放父目录的索引节点（算法 iput）；
    if  （新节点为块或字符特殊文件）
        将主、次设备号写入索引节点结构；
    释放新索引节点（算法 iput）；
}
```

图 5-13 创建新节点的算法

5.9 改变目录及根

当系统初启时，进程 0 在初始化期间使文件系统的根成为它的当前目录。它在这个根索引节点上执行算法 iget，将该索引节点保存在 u 区作为它的当前目录，然后释放该索引节点的锁。当通过系统调用 fork 创建一个新进程时，新进程则继承老进程在 u 区中的当前目录，内核将该索引节点的引用数加 1。

算法 chdir（图 5-14）改变一个进程的当前目录。chdir 的语法格式为：

```
chdir（pathname）；
```

其中，pathname 是要成为进程的新的当前目录的目录名。内核用算法 namei 分析这一目录名，并检查这一文件是否是目录、进程的所有者是否具有对该目录的存取权。然后，内核释放新索引节点的锁，但仍保持该索引节点是已分配的，并使其引用数加 1，释放保存在 u 区中的老的当前目录（算法 iput），并将新的索引节点存放在 u 区中。在进程改变其当前目录

后，算法 namei 将使用该节点作为起始目录，来查找不从根目录开始的所有路径名。执行了系统调用 chdir 之后，新目录的索引节点的引用数至少为 1，而原来的当前目录的索引节点引用数可能为 0。在这一点上，chdir 类似于系统调用 open，因为二者都访问一个文件并使其索引节点成为已分配的。在系统调用 chdir 期间分配的索引节点，只有当进程执行另外一个 chdir 或当该进程退出时，才能被释放。

```
算法　chdir
输入：新目录名
输出：无
{
    取新目录名的索引节点（算法 namei）;
    if　（索引节点不是目录的索引节点或不允许进程存取该文件）
    {
        释放该索引节点（算法 iput）;
        return（错）;
    }
    解锁索引节点;
    释放“老”的当前目录的索引节点（算法 iput）;
    将新索引节点放到 u 区中当前目录域中;
}
```

图 5-14　改变当前目录的算法

一个进程通常对所有以“/”开始的路径名使用全局文件系统的根。内核中有一个全局变量，指向全局根的索引节点。这个全局变量是系统初启时由 iget 分配的。进程能够用系统调用 chroot 改变它们对文件系统的根的概念。如果一个用户想模仿一般的文件系统层次结构并在那里运行进程，用 chroot 是很有用的。该调用的语法格式是：

```
chroot（pathname）;
```

其中，pathname 是一个目录，内核在该操作之后，将把它当作进程的根目录。内核执行系统调用 chroot 时所用的算法基本上和改变当前目录的算法相同。它将新的根索引节点存放在 u 区中，在完成该系统调用时，对该索引节点解锁。然而，由于内核的省缺根存放在全局变量中，内核并不自动地释放老的根的索引节点，除非内核或一个祖先进程曾执行过系统调用 chroot。现在，新索引节点是该进程及它的所有后代进程的文件系统逻辑根，这意味着在算法 namei 中，所有从根(“/”)开始的路径名的查找都将从这个索引节点开始，试图在根目录上使用“..”将使进程的工作目录保持在新的根下。就象一个进程能将它的当前目录传给其新的子进程一样，它也将它改变了的根传给其新的子进程。

5.10　改变所有者及许可权方式

改变一个文件的所有者或许可权方式是对文件的索引节点的操作，而不是对文件本身。这两个系统调用的语法格式为：

```
chown（pathname，owner，group）
```

```
chmod (pathname, mode)
```

为了改变一个文件的所有者，内核先通过算法 namei 将文件的名字转换为一个索引节点。进程的所有者必须是超级用户或者是该文件的所有者（一个进程不能将不属于它的东西送掉）。然后，内核赋予该文件新的所有者和新的用户组，清除文件存取权方式中的 setuid 位和 setgid位（见 7.6 节），并用 iput 释放该索引节点。在改变属主关系之后，老的所有者就失去对该文件的“所有者”存取权。为改变一个文件的许可权方式，内核按类似的过程，改变索引节点中的方式标志而不是所有者号。

5.11　系统调用 stat 和 fstat

系统调用 stat 和 fstat 允许进程查询文件的状态，它们返回诸如文件类型、文件所有者、存取许可权、文件大小、联结的数目、索引节点号、文件的访问时间等信息。这两个系统调用的语法格式是：

```
stat (pathname, statbuffer);
fstat (fd, statbuffer);
```

其中，pathname 是一个文件名；fd 是以前的系统调用 open 返回的文件描述符；stabuffer 是用户进程中的一个数据结构的地址，它将含有系统调用完成时所返回的文件状态信息。这两个系统调用只是简单地将对应的索引节点中的若干字段写到 statbuffer 中。图 5-33 中的程序将说明 stat 和 fstat 的使用。

5.12　管道

管道允许在进程之间按先进先出的方式传送数据，管道也使进程能同步执行。管道的实现使进程之间能够通信，尽管它们不知道管道的另一端是什么进程。管道的传统实现方法是采用文件系统作为数据存储。有两种类型的管道：有名管道和无名管道，对后者找不到更好的名字了。除了进程最初存取它们的方式不同以外，这两种管道是一样的。进程对有名管道使用系统调用 open，但使用系统调用 pipe 来建立无名管道。然后，在使用管道时，进程用正规文件的系统调用，如 read，write 和 close。只有相关的进程，即发出 pipe 调用的进程的后代，才能共享对无名管道的存取。例如，图 5-15 中，如果进程 B 建立了一个管道，然后派生出进程 D 和 E，那么 B、D、E 这三个进程都可以存取这个管道，而进程 A 和 C 却不能。然而，所有的进程都能按通常的文件许可权，存取有名管道，而不管它们之间的关系是什么。因为无名管道使用得更普遍，我们先讨论它。

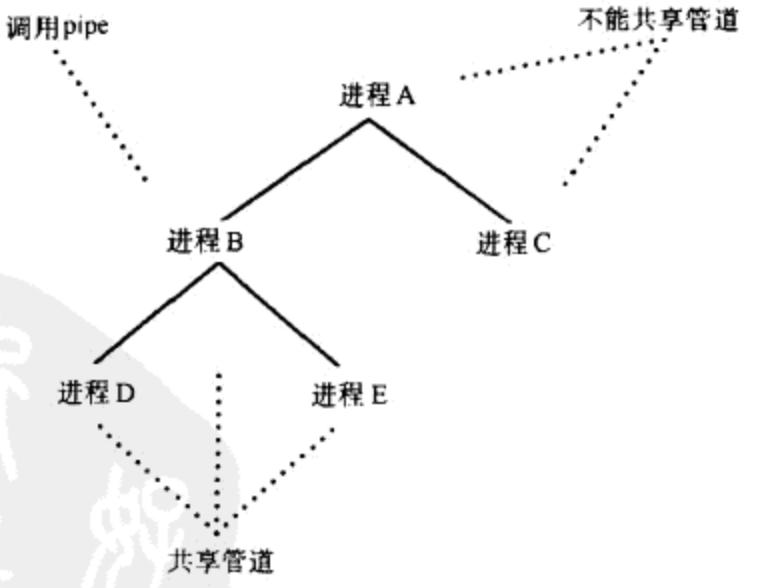


图 5-15　进程树和共享管道

5.12.1　系统调用 pipe

系统调用 pipe 的语法格式为：

```
pipe (fdptr);
```

这里，fdptr 是指向一个整型数组的指针。这个整型数组将含有读、写管道用的两个文件描述符。因为内核是在文件系统中实现管道的，而且在使用之前，管道并不存在，所以内核在创建管道时必须为管道分配一个索引节点。它还要为管道分配一对用户文件描述符及相应的文件表项：一个用来从管道读，一个用来向管道写。为了使管道的 read，write 以及其他系统调用的接口与正规文件的这些系统调用的接口一致，内核使用文件表。因此，进程不必知道它们读或写的是正规文件还是一个管道。

图 5-16 给出建立无名管道的算法。内核首先使用算法 ialloc，从一个叫做管道设备的文件系统中为管道分配一个索引节点。管道设备就是一个文件系统，内核能够从中为管道分配索引节点和数据块。在系统生成时，系统管理员要指定管道设备。它可以是另一文件系统，但在管道活动期间，内核不能将该管道的索引节点和数据块再分配给另一文件。

```
算法　pipe
输入：无
输出：读文件描述符和写文件描述符
{
    由管道设备中分配一个新索引节点（算法 ialloc）；
    分配用于读和写的两个文件表项；
    初始化文件表项，使它们指向新索引节点；
    分配两个用于读和写的用户文件描述符；
    初始化用户文件表项，使它们分别指向对应的文件表项；
    将索引节点引用数置为 2；
    将索引节点读计数和写计数分别置为 1；
}
```

图 5-16　创建（无名）管道的算法

然后，内核分别为读文件描述符和写文件描述符分配两个文件表项，并修改内存索引节点中的管理信息。每个文件表项记录该管道有多少用于读或写的打开实例（每个表项的初值为 1），而索引节点引用数则表示该管道被“打开”的次数，其初值为 2——相应于每个表项一个。最后，索引节点还记录下次对该管道的读或写在管道中的起始字节偏移量。在索引节点中保持字节偏移量对管道数据的先进先出方式的存取更为方便。这一点不同于正规文件，正规文件的偏移量是在文件表中保持的。进程不能通过 lseek 修改管道的字节偏移量，因而对管道中的数据是不可能随机存取的。

5.12.2　有名管道的打开

一个有名管道是一个文件，语义上和无名管道一样。但这个文件有目录项并可通过路径名来存取。进程以与打开正规文件同样的方式打开有名管道，因此，关系不密切的进程也可

以相互通信。有名管道在文件系统树中永久地存在（可用系统调用 unlink 来清除），而无名管道是临时性的：当所有的进程都结束使用某个管道时，内核便收回它的索引节点。

打开一个有名管道的算法和打开一个正规文件的算法基本相同。然而，在完成系统调用之前，内核增加索引节点中的读计数或写计数，它表示为了读或写而打开该管道的进程数。为读而打开有名管道的进程将进入睡眠，直到另一进程以写方式打开该管道，反之亦然。对于一个为了读而打开的管道，如果没有希望收到数据（即没有写管道的进程），那么这种打开是无意义的。反之，对于写也是一样。根据进程是为读还是为写而打开管道，内核唤醒某些睡眠进程，这些睡眠进程正分别等待着该有名管道上的写进程或读进程。

如果一个进程为读而打开一个有名管道，而该管道上的写进程已存在，那么这个 open 调用也就完成了。或者，如果一个进程用“无延迟”选择打开一个有名管道，那么，即使没有写进程，该 open 也立即返回。但如果上述两个条件都不存在，该进程则进入睡眠状态直到有一个写进程打开该管道。对于为了写而打开一个管道的进程，也有同样的规则。

5.12.3　管道的读和写

应该把管道看作好象是某些进程写入该管道的一端，而另外一些进程从另一端读该管道。如上所述，进程以先进先出的方式从管道中存取数据。读管道数据的进程的数目不必一定和写管道的进程的数目相等。如果读或写进程的数目大于 1，则这些进程必须以其他机制来协调对管道的使用。内核存取管道数据的方式和它存取正规文件的数据的方式完全一样。内核在系统调用 wirte 期间，将数据放在管道设备上，并按需要将磁盘块分配给管道。为管道分配存储空间和为正规文件分配存储空间的不同之处是，管道只使用索引节点的直接块以获得较高的效率，尽管这会限制管道一次所能容纳的数据量。内核将索引节点的直接块作为一个循环队列来管理，内部地修改读写指针，从而保持先进先出的顺序（图 5-17）。

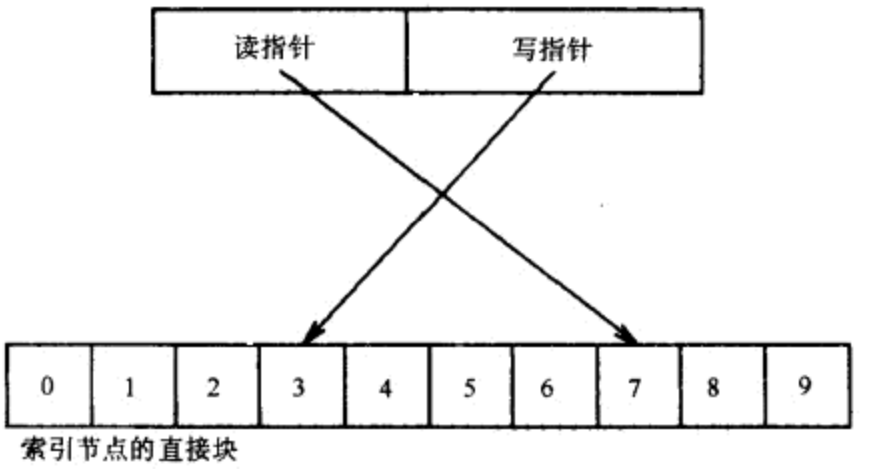


图 5-17　读写管道的逻辑图

考虑四种读写管道的情况：写管道，管道中尚有空间存放欲写的数据；读管道，管道中有足够的数据来满足读操作；读管道，管道中没有足够的数据来满足读操作；最后，写管道，管道中没有足够的空间存放要写的数据。

先考虑第一种情况。一个进程正在写管道，假定管道尚有空间存放要写的数据：正在写的字节数和管道中已有的字节数之和小于或等于管道的容量。内核用与写正规文件一样的算法写管道。不同的是在每次 wirte 之后，内核自动地增加管道的大小，因为按照定义，管道的数据量随着每次 write 而增大。这一点与正规文件的增长不同。对于正规文件，仅当进程在当前文件尾之后写数据时，进程才增加文件的大小。如果管道中的下个字节偏移量需要使用一个间接块时，内核则修改 u 区中的文件偏移量值，使其指向管道的开始位置（字节偏移量 0）。内核从不覆盖管道中的数据，它之所以能够将字节偏移量重置为零，是因为它已经确定了数据不会超出管道的容量。当写进程已经将其全部数据写入管道时，为了使下一个写

管道的进程从上一个写进程停止的地方接着写，内核要修改管道的（索引节点）写指针。然后，内核唤醒所有等待从该管道读数据的睡眠进程。

当一个进程读管道时，内核先要检查管道空否。如果管道中有数据，内核按读正规文件的算法 read 从管道中读数据，就像管道是个正规文件。然而，read 的初始偏移量是存在索引节点中的读指针，它表示前次 read 所达到的地方。在每读一块之后，内核根据它读的字节数减少管道的大小，如果必要的话，调整 u 区的偏移量值，使其绕回到管道的开始。当系统调用 read 结束时，内核唤醒所有睡眠的写进程并将当前读偏移量保存在索引节点中（而不是文件表项中）。

如果进程要读的数据比管道中的数据多，read 将返回管道中当前所有的数据，成功地结束，即使没有满足用户要求的数据量。如果管道为空，一般地说，进程将进入睡眠，直到另一个进程将数据写入管道。此时，所有等待该管道数据的进程都会醒来，并竞争读管道。然而，如果一个进程用“无延迟”的选择打开有名管道，而且管道中又没有数据，那么该进程会从 read 中立即返回。读写管道的语义类似读写终端设备的语义（见第 10 章），能够允许程序忽略它们正处理的文件类型。

如果一个进程写管道，而该管道不能容纳所有的数据，内核将对该索引节点作标记。然后，进程进入睡眠，等待数据从管道排出。当另一个进程随后从该管道读时，内核将通知那些等待数据从管道中排出的睡眠进程，并唤醒它们。上述情况的一个例外是，当进程写的数据量大于管道的容量（即，能够被存到索引节点直接块中的数据量）时，内核将尽可能多的数据写到管道中，然后使进程睡眠，直到获得更多的空间。因此，如果另外的进程在这个进程继续向管道写之前，将它们的数据写入管道，则该进程写的数据有可能是不连在一起的。

通过分析管道的实现可知，进程与管道的接口和进程与正规文件的接口是一致的。不同的是，在实现上，内核将读写偏移量放在索引节点中而不是文件表中。内核必须将偏移量放在有名管道的索引节点中，这样，多个进程才能共享这些偏移量值。它们不能共享文件表项中的值，因为对每一次 open 调用，进程得到的是一个新的文件表项，其实，在实现有名管道以前就有共享索引节点中的读写偏称量的情形了：存取无名管道的进程，通过共同的文件表项共享对管道的存取，因此，可以想象它们是将读写偏移量存放在文件表项中，就象对正规文件一样。但事实上并没有这样实现，因为内核中的低层子程序不再存取文件表项。进程共享存储在索引节点中的偏移量使代码更简单了。

5.12.4 管道的关闭

关闭管道的过程和关闭正规文件的过程一样，只不过内核在释放管道的索引节点之前要做一些特殊处理。内核根据文件描述符的类型，减少管道的读者或写者的数目。如果写进程的数目降为零，同时还有等待读该管道数据的睡眠进程，内核则唤醒它们。这些进程从它们的读调用中空手返回，没有读到任何数据。如果读进程的数目降为零，而且有等待向该管道写数据的进程，内核则唤醒它们，并发给它们一个软中断信号（见第 7 章），指出一个错误情况。在上述两种情况中，管道的状态已没有希望再发生变化。这时，继续让这些进程睡眠是没有意义的。举例来说，如果一个进程在等待读一个无名管道，并且不再有写进程，那么就永远不会再有写进程了。尽管对于有名管道来说，是有可能得到新的读进程或新的写进程的，但内核将它们与无名管道同等地对待。如果没有读进程或写进程存取管道，内核则释放

它的所有数据块并修改索引节点来指明管道为空。当内核释放一个普通管道的索引节点时，它释放该索引节点的磁盘拷贝，以便进行重新分配。

5.12.5　例

图 5-18 给出一个说明管道的使用的程序。图中的进程创建一个管道，然后进入一个无限循环，将字符串“hello”写到管道，并从管道中将它读出。内核并不知道、也不关心写管道的进程和读管道的进程是否是同一个进程。

一个执行图 5-19 中的程序的进程创建了一个叫做 fifo 的有名管道节点。如果以一个哑参数作为第二个参数来调用该程序，该程序将连续地将字符串“hello”写入管道；如果没有第二个参数，它则读管道。这两个进程使用的是同一程序，并且暗地里同意通过有名管道“fifo”来通讯，但这两个进程不必有任何联系。其他用户也能执行该程序并加入（或干预）它们的谈话。

```
char string [] = "hello";
main ()
{
    char buf [1024];
    char * cp1, * cp2;
    int fds [2];

    cp1 = string;
    cp2 = buf;
    while (*cp1)
        *cp2++ = *cp1++;
    pipe (fds);
    for (;;)
    {
        write (fds [1], buf, 6);
        read (fds [0], buf, 6);
    }
}
```

图 5-18　读写一个管道

```
#include <fcntl. h>
char string [] = "hello";
main (argc, argv)
    int argc;
    char *argv [];
{
    int fd;
    char buf [256];

    /* 以读/写许可权的方式为所有用户创建有名管道 */
    mknod ("fifo", 010777, 0);
    if (argc = =2)
        fd = open ("fifo", O_WRONLY);
    else
        fd = open ("fifo", O_RDONLY);
    for (;;)
        if (argc = =2)
            write (fd, string, 6);
        else
            read (fd, buf, 6);
}
```

图 5-19　读写一个有名管道

5.13　系统调用 dup

系统调用 dup 将一个文件描述符拷贝到该用户文件描述符表中的第一个空槽中，给用户返回一个新的文件描述符。dup 适用于所有文件类型。该调用的语法格式为：

```
newfd=dup (fd);
```

其中，fd 是要复制的文件描述符，newfd 是引用该文件的新的文件描述符。因为 dup 复制了文件描述符，所以它使对应的文件表项的引用数值加 1。这时，这个文件表项多了一个指向

它的文件描述符表项。以图 5-20 的数据结构为例。假定某进程作了下列系统调用：它先打开文件“/etc/passwd”（文件描述符 3），然后再一次打开文件“/etc/passwd”（文件描述符 4），这之后又打开文件“local”（文件描述符 5），最后复制文件描述符 3，返回的是文件描述符 6。

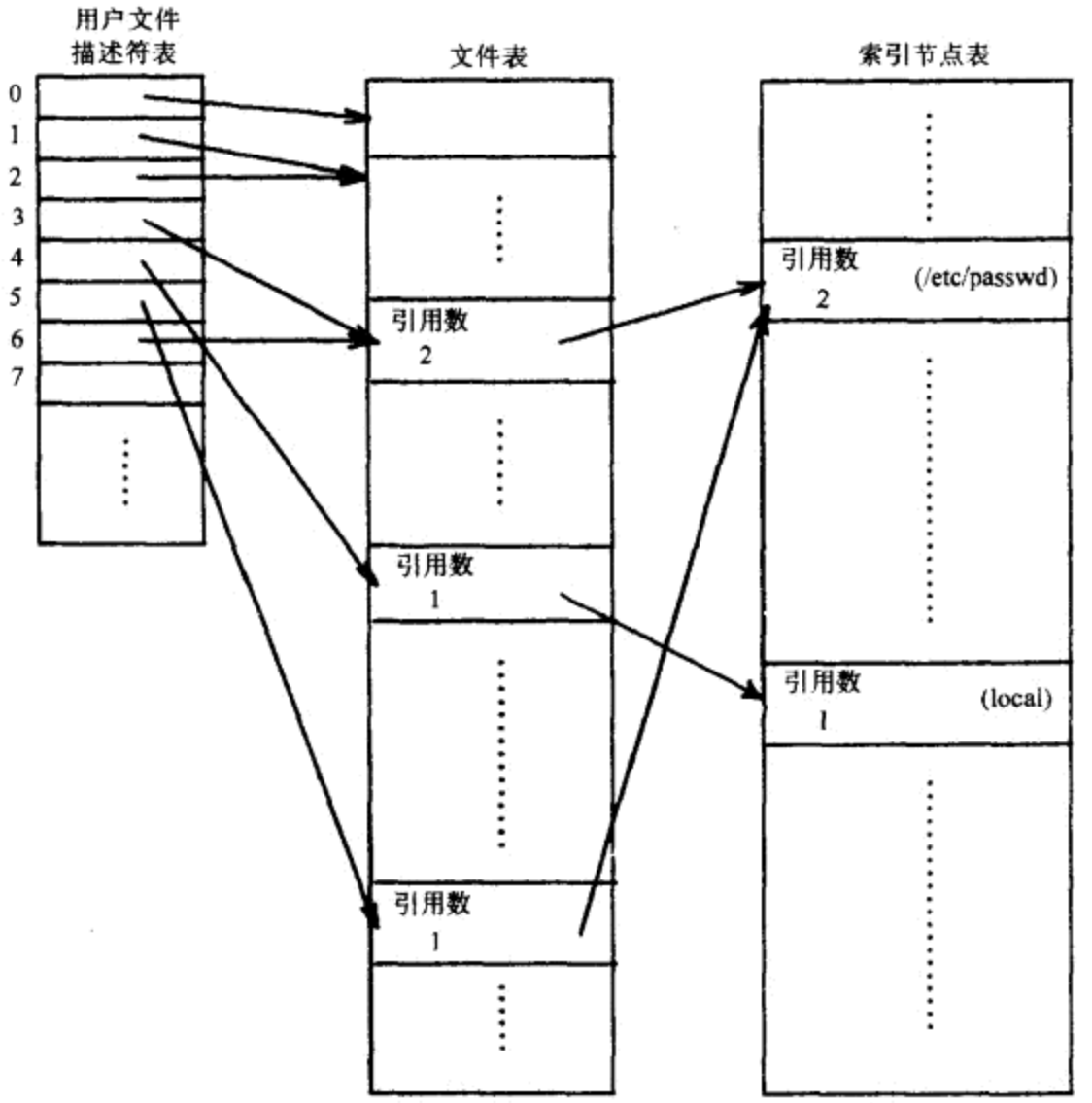


图 5-20　系统调用 dup 之后的数据结构

dup 可算作是一个聪明的系统调用，因为它假定用户知道系统将返回用户文件描述符表中最低编号的空表项。当由简单的标准构件程序构造复杂的程序时，dup 起着重要的作用。例如在构造 shell 管道线（pipeline）的典型例子中（见第 7 章）。

考虑图 5-21 中的程序。变量 i 中存放的是系统调用 open 返回的文件描述符，即打开文件“/etc/passwd”的结果。变量 j 中含有系统调用 dup 所返回的文件描述符，即复制文件描述符 i 的结果。在该进程的 u 区中，以变量 i 和 j 所代表的两个文件描述符表项指向同一个文件表项，因此它们使用相同的文件偏移量。这样，进程中的头两个 read 按顺序读数据，而两个缓冲区 buf1 和 buf2 并不含有相同的数据。这和一个进程对同一文件打开两次、读两次同样的数据的情况不一样（见 5.2 节）。如果一个进程想关闭两个文件描述符中的任意一个，它可以关闭，但是 I/O 仍在另一个文件描述符上正常继续，图 5-21 的例子说明了这一点。特别要指出的是，一个进程能够关闭它的标准输出文件描述符（文件描述符 1），复制

另一个文件描述符，使其成为文件描述符 1，把该文件描述符所对应的文件作为标准输出来对待。第 7 章在描述 shell 的实现时，将给出一个更实际的使用 pipe 和 dup 的例子。

```
#include <fcntl. h>
main ()
{
    int i, j;
    char buf1 [512], buf2 [512];

    i = open ( "/etc/passwd", O_RDONLY);
    j = dup (i);
    read (i, buf1, sizeof (buf1));
    read (j, buf2, sizeof (buf2));
    close (i);
    read (j, buf2, sizeof (buf2));
}
```

图 5-21　说明系统调用 dup 的 C 程序

5.14　文件系统的安装和拆卸

一个物理的磁盘设备是由一些被磁盘驱动程序划分的逻辑段（disk section）组成的。每个逻辑段有一个设备文件名，通过打开适当的设备文件名，然后读写该“文件”，进程就能存取一个段中的数据。进程将这个“文件”看作是一个磁盘块序列。第 10 章将给出关于这个接口的细节。磁盘的一个段可以含有一个逻辑的文件系统，由第 2 章所描述过的引导块、超级块、索引节点表和数据块组成。系统调用 mount 将在一个磁盘的指定段中的文件系统连到一个已存在的文件系统目录树中，而系统调用 umount 将一个文件系统从该文件系统目录树中拆卸下来。因此，系统调用 mount 允许用户以文件系统的方式，存取磁盘段中的数据，而不是按磁盘块序列的方式存取数据。系统调用 mount 的语法形式是：

```
mount (special pathname, directory pathname, options);
```

这里，special pathname 是磁盘段的设备特殊文件名，该磁盘段中含有要安装的文件系统。directory pathname 是已存在的文件系统目录树中的目录，即要安装的文件系统将被安装的地方，称为安装点（mount point）。options 指出所安装的文件系统是否应被安装成“只读”的（这时，像 write 和 creat 这样的用来写文件系统的系统调用将会失败）。例如，如果一个进程发出如下系统调用：

```
mount ( "/dev/dsk1", "/usr", 0);
```

内核则将磁盘段中叫做“/dev/dsk1”的文件系统连到已存在的、文件系统树中被称为“/usr”的目录中（图 5-22）。文件“/dev/dsk1”是个块特殊文件，这意味着它是一个块设备的名字，典型地，是磁盘的一部分。内核还假定，给定的这部分磁盘中，含有一个带超级块、索引节点表和根索引节点的文件系统。在完成系统调用 mount 之后，被安装的文件系统的根

就可以通过“/usr”来存取。进程可以在被安装的文件系统上存取文件，并常常不知道这个系统是可拆卸的这一事实。只有系统调用 link 才检查一个文件的文件系统，因为系统V不允许文件的联结扩展到多个文件系统（见 5.15 节）。

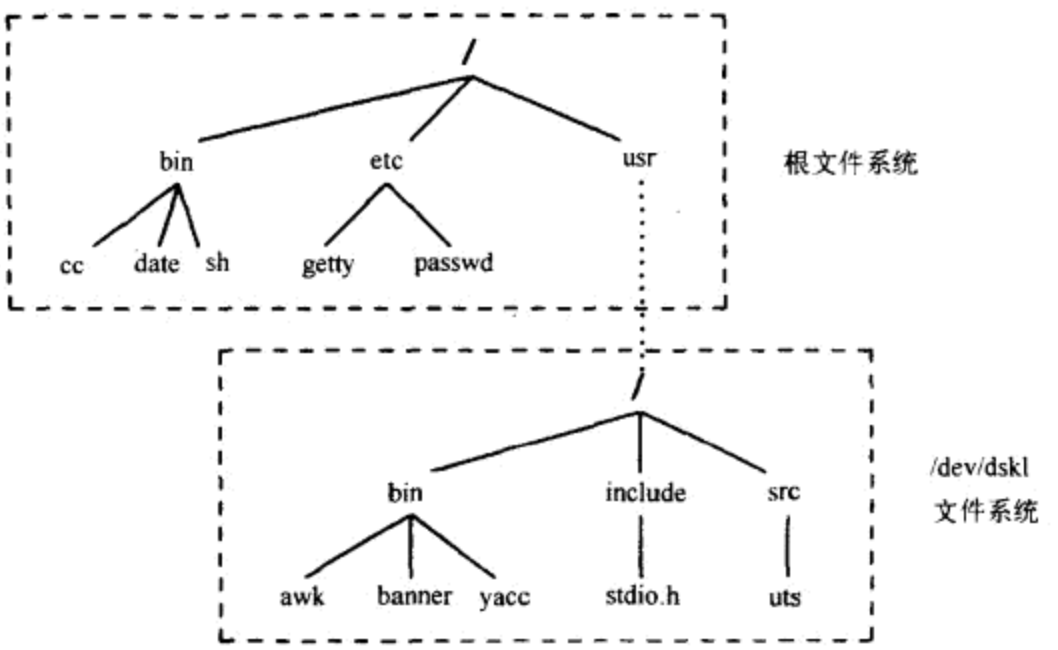


图 5-22　安装前和安装后的文件系统目录树

内核中有个安装表，每个被安装的文件系统在该表中都占有一个表项。每个安装表项含有如下字段：

- 设备号，用来标识被安装的文件系统（即前述的逻辑文件系统号）。
- 指向含有被安装的文件系统超级块的缓冲区的指针。
- 指向被安装的文件系统的根索引节点（在图 5-22 中，是文件系统“/dev/dsk1”的根“/”）的指针。
- 指向安装点的目录的索引节点（在图 5-22 中，是根文件系统的“usr”）的指针。

安装点的索引节点和被安装的文件系统的根索引节点之间的联系是在系统调用 mount 期间建立的。这种联系使内核能很好地遍历文件系统树，而不需要特别的用户知识。

图 5-23 给出了安装文件系统的算法。内核只允许属于超级用户的那些进程安装或拆卸文件系统。若将安装和拆卸的许可权给予全体用户，将会使居心不良的用户严重破坏文件系统。超级用户仅在偶然的情况下会破坏文件系统。

内核查找特殊文件的索引节点，它代表了要被安装的文件系统。然后，它取出标识相应磁盘段的主设备号和次设备号，再找出文件系统要安装到的目录索引节点。由于存在潜在的危险的副作用（见习题 5.27），该目录索引节点的引用数不能大于 1（必须至少为 1——为什么?）。内核在安装表中分配一个自由槽，标记该槽为已使用，并将安装表中的设备号字段赋值。上述赋值需立即完成，因为调用进程在继而发生的打开设备过程中或在读文件系统超级块时，可能进入睡眠，而其他进程可能企图安装一个文件系统。内核通过将对应的安装表项置上使用标志，就防止了两个系统调用 mount 使用同一表项；通过记下要被安装的设备号，就防止了其他进程再次安装同一文件系统。如果允许重复安装，则会发生莫明其妙的事情（见习题 5.26）。

```
算法　mount
输入：块特殊文件的文件名
      安装点的目录名
      选择项（只读）
输出：无
{
    if　（非超级用户）
        return（错）;
    取块特殊文件的索引节点（算法 namei）;
    作合法性检查;
    取安装点目录名的索引节点（算法 namei）;
    if　（不是目录，或引用数大于 1）
    {
        释放索引节点（算法 iput）;
        return　（错）;
    }
    查找安装表中的空槽;
    调用块设备驱动程序的 open 子程序;
    从高速缓存中取空闲缓冲区;
    将超级块读入空闲缓冲区;
    初设超级块中的各字段;
    取被安装设备的根索引节点，并保存在安装表中;
    标记安装点的目录索引节点为安装点;
    释放特殊文件的索引节点（算法 iput）;
    解锁安装点目录的索引节点;
}
```

图 5-23　安装文件系统的算法

内核对含有被安装的文件系统的块设备调用过程 open 时，所用的方法和它直接地打开块设备时一样（见第 10 章）。一般来说，设备的打开过程（device open procedure）检查设备是否合法，有时要初始化驱动程序的数据结构，并向硬件发送初始化命令。然后，内核从缓冲池中分配一个自由缓冲区（算法 getblk 的变体）来安装被安装的文件系统的超级块，并用算法 read 的变体来读这个超级块。内核保存一个指针，该指针指向原文件系统目录树在安装点的目录索引节点。我们将会看到，这会允许含有“..”的文件路径名遍历安装点。内核找出被安装的文件系统的根索引节点，并将指向该索引节点的指针保存在安装表中。对用户而言，安装点的目录和被安装的文件系统的根在逻辑上是等价的。内核通过使它们在安装表中同时存在，来建立它们的等价关系，而进程再也不能存取安装点目录的索引节点了。

内核初始化文件系统超级块中的各个字段，即清除自由块链表的锁字段及自由索引节点的锁字段，并将超级块中的自由索引节点数置为零。这样做的目的是减少文件系统遭受破坏的危险，因为在系统的一次垮台之后安装文件系统会有这种危险。而这样的初始化使内核认为在超级块中没有自由索引节点，从而强迫算法 ialloc 到磁盘上去查找自由索引节点。不幸的是，如果磁盘自由块的链表被破坏，内核不会内部地修复该链表（见 5.17 节的文件系统维护）。如果安装文件系统的用户用了只读选择，禁止所有对该文件系统的写操作，那么内核将在超级块中设置一个标志。最后，内核为安装点的索引节点置上“安装点”标记，这样，其他进程以后就会将它识别出来。图 5-24 给出了在系统调用 mount 结束之后的各种数据结构。

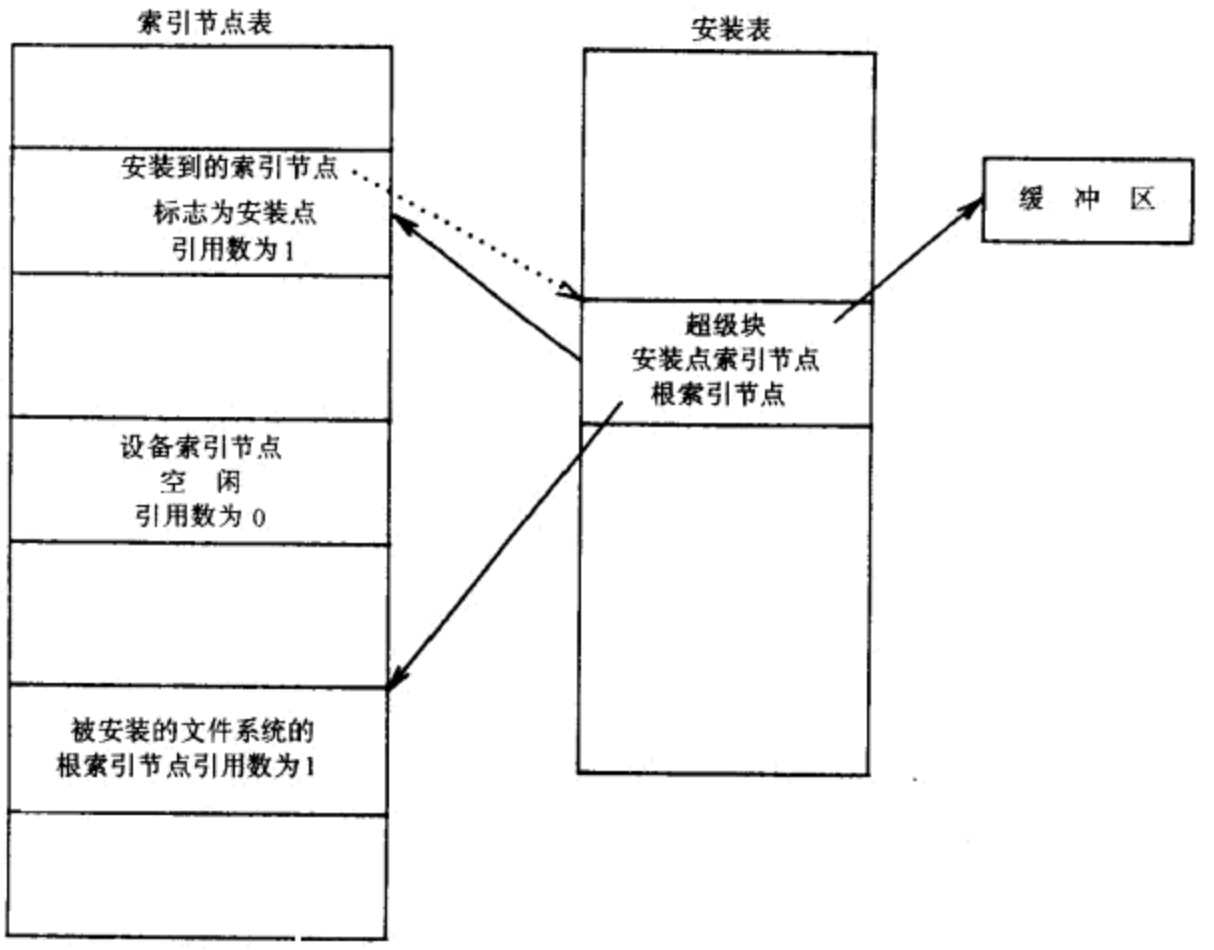


图 5-24 mount 之后的数据结构

5.14.1 在文件路径名中跨越安装点

本节针对一个路径名跨越安装点的情况，重新考虑算法 namei 和 iget。跨越安装点的情况有两种，一种是从原安装点的文件系统跨越到被安装的文件系统（方向是从整个系统的根到某个叶结点）；另一种是从被安装的文件系统跨越到安装点的文件系统。下列 shell 命令说明了这两种情形：

```
mount /dev/dsk1 /usr
cd /usr/src/uts
cd .. /.. /..
```

命令 mount 在作了一致性检查之后，调用系统调用 mount，并将磁盘段“/dev/dsk1”上的文件系统安装到目录“/usr”上。第一个 cd（转目录）命令使 shell 执行系统调用 chdir，内核分析路径名，在“/usr”处跨越安装点。第二个 cd 命令使内核分析路径名，并在第三个“..”处跨越安装点。

对于从安装点的文件系统跨越到被安装的文件系统的情况，请参见图 5-25 中修改过的算法 iget。该算法与图 4-3 类似，只不过它要检查索引节点是否是安装点：如果被检查的索引节点有“安装点”标志，内核便知道它是一个安装点。该算法在安装表中查找这个安装点的索引节点所对应的安装表表项，并记下被安装的文件系统的设备号。然后，利用该设备号和根的索引节点号（所有的文件系统都一样），存取被安装设备的根索引节点并返回那个索引节点。以上述例子的第一个 cd 命令为例，内核首先在安装点的文件系统中取“/usr”的索引节点，发现它有“安装点标志”，就在安装表中找出被安装的文件系统的根索引节点，

并存取该文件系统的根索引节点。

```
算法　iget
输入：文件系统的索引节点号
输出：上锁的索引节点
{
    while　（没做完）
    {
        if　（索引节点在索引节点高速缓存中）
        {
            if　（索引节点被锁）
            {
                sleep（索引节点解锁事件）;
                continue;　/*　循环　*/
            }
            /*　对安装点的特殊处理　*/
            if　（索引节点是安装点）
            {
                查安装表，找安装点；
                从安装表中取新文件系统号；
                在查找中用新的根索引节点号；
                continue;　/*　循环　*/
            }
            if　（索引节点在索引节点空闲链表中）
                从空闲链表中取出索引节点；
            索引节点引用数加 1；
            return（索引节点）;
        }

        /*　索引节点不在索引节点高速缓存中　*/
        从空闲链表中取新索引节点；
        设置索引节点号和文件系统；
        从老的散列队列中取走索引节点，放入新的索引节点；
        从磁盘读索引节点（算法 bread）;
        初始化索引节点（如，设置引用数为 1）;
        reutrn（索引节点）;
    }
}
```

图 5-25　修改了的存取索引节点算法

对于第二种情况，即从被安装的文件系统跨越到安装点的文件系统，请见图 5-26 修改了的算法 namei。该算法类似于图 4-11。不同的是，在找出目录中一个路径名分量的索引节点号之后，内核要检查该索引节点号是否为一个文件系统的根节点。如果是，而且当前工作索引节点也是根，路径名分量又是点点（“..”），那么内核便识别出该节点是个安装点。该算法在安装表中查找设备号等于最后找出的索引节点的设备号的表项，取安装点目录的索引节点，并以安装点目录的索引节点作为当前工作索引节点，继续查找“..”。但是，在文件系统的根处，“..”便是根本身。

```
算法　namei　/*　将路径名转换为索引节点　*/
输入：路径名
输出：上锁的索引节点
{
    if　(路径名从根开始)
        工作索引节点=根索引节点（算法 iget);
    else
        工作索引节点=当前目录的索引节点（算法 iget);
    while　(还有路径名分量)
    {
        从输入中读下一个路径名分量;
        验证索引节点是目录，验证许可权;
        if (索引节点是改变了的根目录而且分量名为“..”)
            continue;　/*　循环　*/
  component search:　/*　查找分量　*/
        读索引节点（目录）(算法 bmap，bread，brelse);
        if　(路径名分量与一个目录项匹配)
        {
            取被匹配的分量的索引节点号;
            if　(找到的索引节点是根，工作索引节点是根而且分量名为“..”);
            {
                /*　跨越安装点　*/
                取工作索引节点的安装表表项;
                释放工作索引节点（算法 iput);
                工作索引节点=安装点索引节点;
                锁安装点的索引节点;
                工作索引节点引用数加 1;
                go to component search（找“..”);
            }
            释放工作索引节点（算法 iput);
            工作索引节点=新索引节点号的索引节点（算法 iget);
        }
        else　/*　路径名分量不是目录　*/
            return（错);
    }
    return（工作索引节点);
}
```

图 5-26　修改了的分析路径名算法

在上述例子（cd .. /.. /..）中，假定进程的起始当前目录是“/usr/src/uts”。在namei 分析路径名时，起始工作索引节点是当前目录。在分析了路径名中的第一个“..”之后，内核的工作节点便转到“/usr/src”的索引节点。然后，内核分析第二个“..”，找到以前安装的文件系统的根索引节点“usr”，并使它成为 namei 的工作节点。最后，内核分析第三个“..”：它发现这个“..”的索引节点号是根索引节点号，它的工作索引节点是该根的索引节点，“..”是当前路径名分量。内核在安装表中查找安装点“usr”所对应的表项，释

放当前工作索引节点（“usr”文件系统的根），并指定安装点索引节点作为新的工作索引节点（即根文件系统中目录“usr”的索引节点）。这之后，内核在安装点“/usr”的目录结构中查找“..”，这样便找到文件系统的根的索引节点号(“/”)。至此，系统调用 chdir 便正常结束；调用进程忘记了它跨越了一个安装点这一事实。

5.14.2　文件系统的拆卸

系统调用 umount 的语法格式是：

```
umount (special filename);
```

这里特殊文件 special filename 是指要被拆卸的文件系统。在拆卸一个文件系统时（图 5-27），内核取要被拆卸的设备的索引节点，查找特殊文件的设备号，释放对应的索引节点（算法 iput），并在安装表中查找设备号等于该特殊文件的设备号的表项。在内核真正拆卸一个文件系统之前，它要在索引节点表中查找设备号等于被拆卸的文件系统的设备号的所有文件，以便确认在该文件系统中没有正在使用的文件。活动文件的引用计数值大于零，其中包括那些是某些进程的当前目录的文件、带有共享正文的正在被执行的文件（见第 7 章）以及打开而未关闭的文件。如果该文件系统中有任何活动的文件，系统调用umount便失败。如果不

```
算法　umount
输入：要拆卸的文件系统的特殊文件名
输出：无
{
    if　(不是超级用户)
      return (错);
    取特殊文件的索引节点 (算法 namei);
    取出要被拆卸的主、次设备号;
    根据主、次设备号取要拆卸的文件系统的安装表项;
    释放特殊文件的索引节点 (算法 iput);
    从区表中清除属于该文件系统的共享正文表项;
    更新超级块、索引节点，腾空缓冲区;
    if　(该文件系统中仍有正在使用的文件)
      return (错);
    从安装表中取要拆卸的文件系统的根索引节点;
    锁该索引节点;
    释放该索引节点 (算法 iput);　/*　在 mount 中是 iget　*/
    调用该特殊设备的关闭子程序;
    使被拆卸的文件系统在缓冲池中的缓冲区无效;
    从安装表中取安装点的索引节点;
    锁该索引节点;
    清除其安装点标志;
    释放该索引节点 (算法 iput);　/*　在 mount 中是 iget　*/
    释放超级块所用的缓冲区;
    释放安装表项;
}
```

图 5-27　拆卸文件系统的算法

按失败来处理的话，活动的文件就会是不可存取的了。

缓冲区池中可能还有一些尚未写到磁盘上的“延迟写”块，因此内核要把它们从缓冲区池中腾出去；内核清除区表中不可操作的共享正文表项（详见第7章）；将所有最新修改过的超级块写到磁盘上，并更新所有需要更新的索引节点拷贝。其实，内核只要对被拆卸的文件系统更新磁盘块、超级块和索引节点就够了，但由于历史的原因，内核对所有的文件系统都做这些工作。然后，内核释放被安装的文件系统的根索引节点，该索引节点自从在系统调用 mount 期间第一次被访问以后，一直被占用着。内核调用相应的设备驱动程序来关闭含有被拆卸的文件系统的设备。做完这些事情之后，内核检查高速缓存中的各缓冲区，使那些装有被拆卸的文件系统数据块的缓冲区无效，因为已经没有必要在高速缓存中存放这些数据了。内核将这些无效的缓冲区移到缓冲区自由链表的表头，使高速缓存中的其他有效块能在高速缓存中呆的时间更长一些。这之后，内核在安装点的索引节点中，清除系统调用 mount 所设置的“安装点”标志并释放该索引节点。在为对应的安装表表项设置了“可用”标志之后，系统调用 umount 便结束。

5.15　系统调用 link

系统调用 link 在文件系统结构中将一个文件联结到另一个新名字上，从而为一个已存在的索引节点创建一个新的目录项。该系统调用的语法是：

```
link（source file name，target file name）;
```

其中，source file name 是源文件名，即一个已存在的文件的名字；target file name 是在完成系统调用之后，源文件所具有的新的（附加的）名字。文件系统中对该文件的每个联结（link）都有个路径名。进程可以通过其中的任意一个路径名存取该文件。内核并不知道哪个名字是最初的文件名，所以对任何文件名都不特殊对待。例如，在执行了系统调用

```
link（“/usr/src/uts/sys”，“/usr/include/sys”）;
link（“/usr/include/realfile.h”，“/usr/src/uts/sys/testfile.h”）;
```

之后，以下三个路径名指的是同一文件：“/usr/src/uts/sys/testfile.h”，“/usr/include/sys/testfile.h”和“/usr/include/realfile.h”（见图 5-28）。

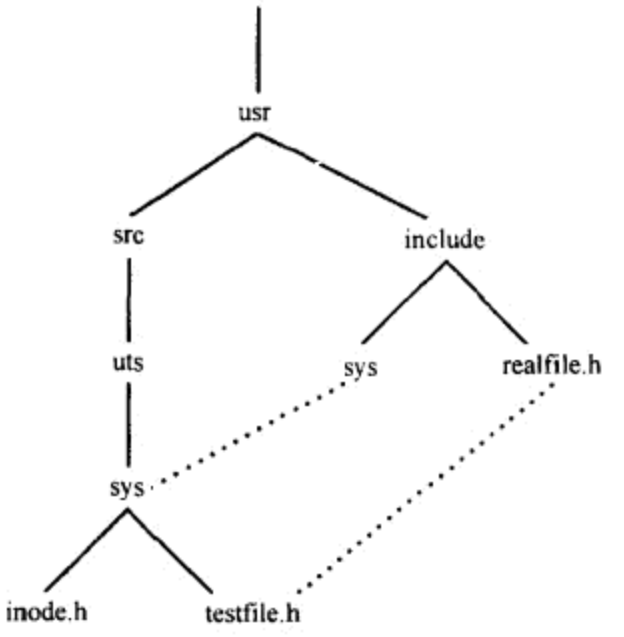


图 5-28　文件系统树中联结的文件

为简化遍历文件系统树的代码，内核仅允许超级用户联结一个目录。如果允许任意的用户联结目录，用来遍历文件系统树的程序必须考虑陷入无限循环的情况，例如，当用户要将一个目录联结到层次结构中在它下面的节点名上时。在做这样的联结时，超级用户总会更加小心。但在 UNIX 的早期版本中，必须有支持联结目录的能力。因为创建新目录的命令 mkdir 的实现要依赖联结目录的功能。在增加了系统调用 mkdir 以后，就不要求有联结目录的功能了。

图 5-29 是 link 的算法。内核先用算法 namei 取出源文件的索引节点，将其联结数加 1，更新该索引节点的磁盘拷贝（为一致性起见），并为该索引节点解锁。然后，内核查找目的文件；如果该文件已存在，这个 link 调用便失败，内核要使刚刚加 1 的联结数减 1。如果该文件不存在，它便记下目的文件父目录中的空槽位置，将目的文件名和源文件的索引节点号写到该槽中，并用 iput 释放目的文件的父目录的索引节点。因为目的文件最初并不存在，所以没有其他索引节点要释放。该算法最后要做的是释放源文件的索引节点。源文件的联结数现在比系统调用 link 开始之时大 1，这时，在该文件系统中，允许用另外一个名字存取它。联结数记录了引用该文件的目录表项数，因而它不同于索引节点的引用数。如果在 link 调用结束时，没有其它进程存取该文件，那么该文件的索引节点引用数是零，但它的联结数至少为 2。

```
算法　link
输入：已存在的文件名
      新文件名
输出：无
{
    取已存在的文件的索引结点；
    if　（文件的联结数过多或无超级用户许可权而联结目录）
    {
        释放索引节点（算法 iput）；
        return（错）；
    }
    索引节点的联结数加 1；
    修改索引节点的磁盘拷贝；
    解锁索引节点；
    取将含有新文件名的父目录的索引节点（算法 namei）；
    if　（新文件名已存在或已存在的文件与新文件在不同的文件系统中）
    {
        使上述更新操作作废；
        return（错）
    }
    在新文件名的父目录中建立新目录表项：
        填入新文件名、源文件名的索引节点号；
    释放父目录的索引节点（算法 iput）；
    释放源文件的索引节点（算法 iput）；
}
```

图 5-29　联结文件的算法

例如，当执行

```
link（“source”，“dir/target”）；
```

时，内核找到文件“source”的索引节点，使其联结数加 1，记下它的索引节点号，假定是 74，然后解锁该索引节点。这之后，内核查出“target”的父目录“dir”的索引节点，在“dir”中找一个空目录项，并将文件名“target”和索引节点号 74 写入该空目录项。最后，它用算法 iput 释放“source”的索引节点。如果原来“source”的联结数是 1，现在便是 2。

值得注意的是，存在两种死锁的可能性，两者都与为什么进程在增加源文件的联结数之

后，要解锁它的索引节点有关。如果内核不解锁索引节点，同时执行下列系统调用的两个进程就会引起死锁；

```
进程 A：link（“a/b/c/d”，“e/f/g”）；
进程 B：link（“e/f”，“a/b/c/d/e”）；
```

假定进程 A 在找到文件“a/b/c/d”的索引节点的同时，进程 B 找到了文件“e/f”的索引节点。“同时”这个词意味着系统达到了每个进程都已找到了它的索引节点这一状态。图 5-30 说明了一种可能的执行情况。这时，当进程 A 企图找目录“e/f”的索引节点时，它将进入睡眠，

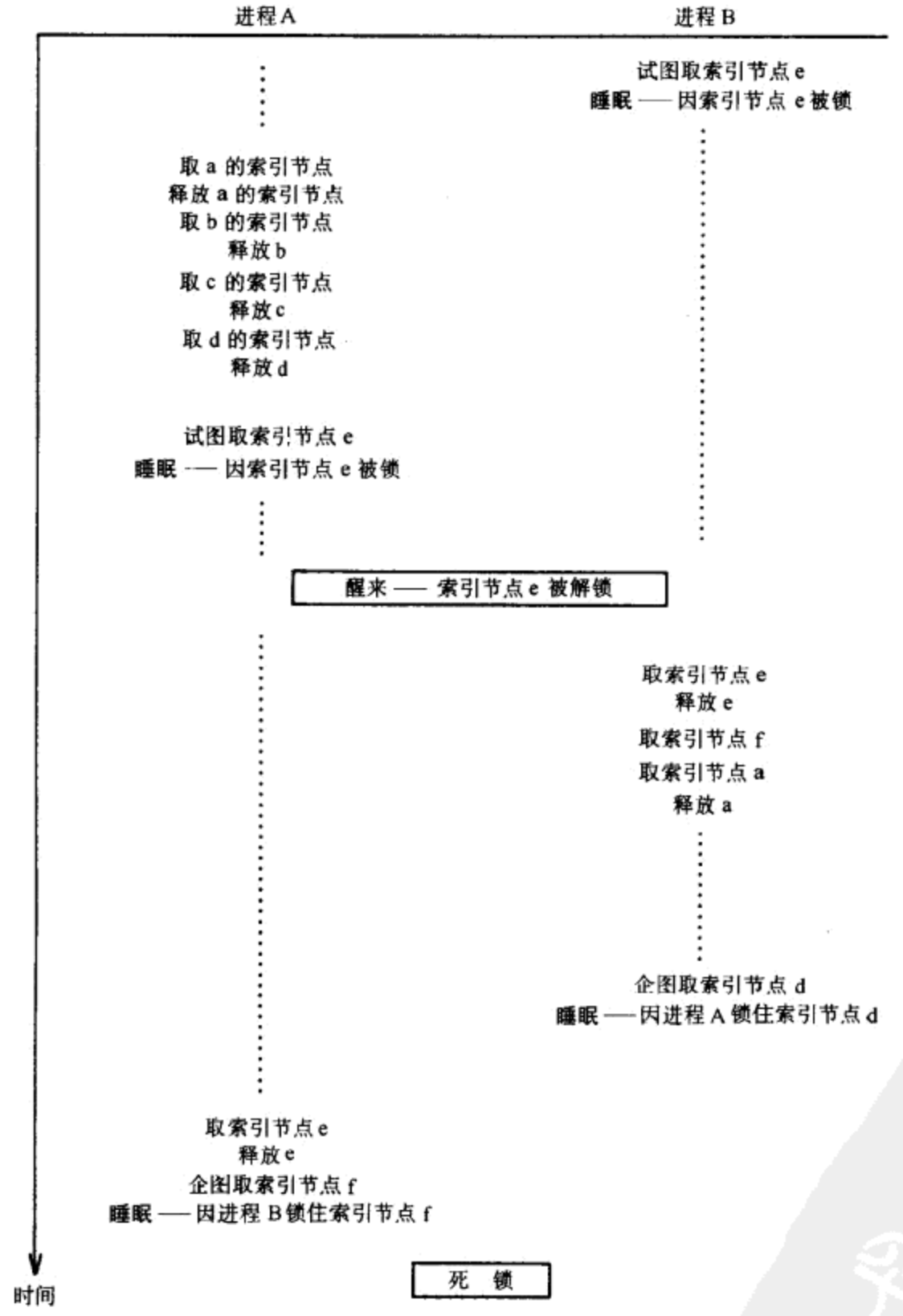


图 5-30　link 的死锁情况

等待着索引节点“f”被释放这一事件。但当进程 B 要找目录“a/b/c/d”的索引节点时，它也将进入睡眠，等待索引节点“d”被释放。这样，进程 A 占有了一个进程 B 想要的上锁索引节点，而进程 B 又占有了一个进程 A 想要的上锁节点。为避免这种典型的死锁情况，内核在使源文件的索引节点联结数加 1 之后，要释放源文件的索引节点。当访问下一个资源的时候，第一个资源（索引节点）已是可用的，因此不会发生死锁。

上面的例子说明，如果不释放源文件的索引节点，两个进程是如何发生死锁的。但即使是单个进程自己，也有可能发生死锁。例如，进程执行

```
link ( "a/b/c", "a/b/c/d");
```

在算法的头一部分，它找到了文件“c”的索引节点；如果内核不释放该索引节点，那么，在查找文件“d”时，就会因遇到被锁的索引节点“c”而发生死锁。若有两个进程，甚至一个进程，由于死锁而不能继续运行，对系统有什么影响呢？由于索引节点是有限的可分配的资源，因此，收到一个软中断信号并不能将进程从其睡眠中唤醒（见第 7 章）。因此，若不重新启动系统就不会中止死锁。如果其他进程不存取使进程发生死锁的那些文件，那么这些进程就不会受到影响。但是，任何访问这些文件的进程，或通过这些锁住的目录存取其他文件的进程，都会发生死锁。这样，当死锁的文件是“/bin”或“/usr/bin”（命令的典型存放处）或“/bin/sh”时，对系统来说，将是一个大灾难。

5.16　系统调用 unlink

系统调用 unlink 清除文件的一个目录表项。该调用的语法格式是：

```
unlink (pathname);
```

这里，pathname 是要从目录树中拆掉（unlink）的文件名。如果一个进程拆掉一个给定文件，就不能通过该文件名再存取这个文件了，除非以该文件名为文件建立另一个目录表项。举例来说，在下面的代码段中，系统调用 open 不会成功：

```
unlink ( "myfile");
fd = open ( "myfile", O_RDONLY);
```

因为当前目录里不再有文件“myfile”。如果被拆除的文件是该文件的最后一个联结，那么，内核最终将释放它的数据块。但是，如果该文件有多个联结，通过它的其他名字，仍能存取到该文件。

图 5-31 给出了 unlink 的算法。内核用算法 namei 的另一种形式找出它要拆除的文件名，但这种 namei 返回的不是该文件的索引节点，而是其父目录的索引节点。然后，内核用算法 iget 取要被拆除的文件的内存索引节点（拆除文件“.”的特殊情况在一个习题中讨论）。在检查了错误条件，以及，对那些可执行文件，清除了区表中不活动的共享正文表项之后（见第 7 章），内核从父目录表项中清除该文件名，即把索引节点号的值置为零。然后，内核向磁盘做该目录的同步写，以确保该文件用已清除的文件名取不到。内核使文件的联结数减 1，并用算法 iput 释放父目录的内存索引节点和被拆除的文件的索引节点。

在用 iput 释放被拆除的文件的内存索引节点时，如果引用数降为零，联结数也为零，

内核则收回由该文件占有的磁盘块。这之后，不再有任何文件名引用该索引节点，而且该索引节点不活动。为收回磁盘块，内核在该索引节点的内容表中循环一遍，立即释放所有直接块（根据算法 free)。对于间接块，内核递归地释放在各个间接层中出现的所有块，首先释放更直接的块。内核将该索引节点内容表中的块号置为零，将文件大小字段也置为零。然后，内核清除索引节点的文件类型域，表明该索引节点是自由的，并用算法 ifree 释放该索引节点。内核还要修改该索引节点的磁盘拷贝，因为在它的磁盘拷贝中，该索引节点仍然是“在使用”状态。现在，这个索引节点是自由的，可以分配给其他文件了。

```
算法　unlink
输入：文件名
输出：无
{
    取要拆除的文件的父索引节点（算法 namei);
    /*　如果拆除当前目录…　*/
    if （文件名的最后分量是“.”）
        父索引节点的引用数加 1;
    else
        取要被拆除的文件的索引节点（算法 iget);
    if　（拆除的文件是目录，但用户不是超级用户）
    {
        释放索引节点（算法 iput);
        return（错);
    }
    if （是共享正文文件，而且联结数为 1）
        从区表中清除;
    写父目录：将拆除文件的索引节点号置 0;
    释放父目录的索引节点（算法 iput);
    文件联结数减 1;
    释放文件的索引节点（算法 iput);
        /*　iput 检查联结数是否为零：如果是，
        *　　释放文件块（算法 free)
        *　　和归还索引节点（算法 ifree);
        */
}
```

图 5-31　拆除文件的算法

5.16.1　文件系统的一致性

为了当系统发生故障时，使文件系统的破坏达到最小程度，内核要按某种次序来进行写磁盘操作。例如，当内核从一个父目录中清除一个文件名时，它要在清除该文件的内容及释放其索引节点之前，同步地将父目录写到磁盘上去。如果系统在该文件的内容被清除之前发生故障，对文件系统造成的损坏将是最小的：会有一个其联结数比实际存取它的目录表项数大 1 的索引节点，但所有其它通向该文件的路径仍然是合法的。如果写目录不是同步进行，那么，在系统发生故障之后，磁盘上的该目录表项就有可能指向了一个已归还（或者已重新分配）的索

引节点。这样，在文件系统中，引用这个索引节点的目录表项数就会大于该索引节点的联结数。特别要指出的是，如果这个文件名是这个文件的最后一个联结，它就会引用一个未分配的索引结点。很显然，在第一种情况下，系统的破坏较轻，比较容易改正（见 5.18 节）。

举例来说，假定一个文件有两个联结，路径名分别为“a”和“b”，一个进程用系统调用 unlink 拆除 a。如果内核按上述第一种情况的次序作写磁盘操作，结果是将“a”的目录表项清零并把它写到磁盘。如果在写磁盘之后，系统发生故障，那么文件“b”的联接数是 2，但文件“a”并不存在，因为在系统垮掉之前，它的老目录表项已被清零。文件“b”有个多余的联接，但当系统重新启动后，系统仍会正常工作。

现在，假定内核按与上述情况相反的次序做写磁盘操作。也就是说，内核先使“b”的联结数减 1，将该索引节点写到磁盘，然后，在它将文件 a 的目录表项清零之前，系统发生了故障。当重新启动系统时，文件“a”和“b”的目录表项分别存在于各自的目录中，但它们引用的文件的联结数却是 1。这之后，如果有一个进程拆除文件“a”，那么，该文件的联结数减为零，即使文件“b”还引用着该索引节点。这时，如果系统中有一个系统调用 creat 的请求，那么内核将重新分配该索引节点，新文件的联接数为 1，但有两个路径名引用它。系统不能自动改正这种情况，除非使用一些维护程序（如 fsck，见 5.18 节）。这些维护程序通过块或原始接口（raw interface）存取文件系统。

同理，内核也应按特别的次序释放索引节点和磁盘块，来减少系统发生故障时所造成的破坏。当清除一个文件的内容和清它的索引节点时，有可能先释放含有文件的数据的磁盘块，也有可能先释放索引节点并将索引节点写回磁盘。正常情况下，这两种情况的结果是相同的。但如果其间系统发生故障，结果就不同了。假定内核先释放文件的磁盘块，然后发生了故障，当系统重新启动时，该文件的索引节点仍然引用老的磁盘块，而这些磁盘块中可能不再含有和该文件有关的数据了。内核看到的是外表上完好的文件，而存取该文件的用户却发现文件被破坏了，也有可能这些磁盘块被分配给了其他文件。用 fsck 程序来清理文件系统是很费力费时的。相反，如果系统先将索引节点写回磁盘，然后系统垮台，那么当系统重新启动时，用户不会发现文件系统有错。以前属于该文件的那些数据块对系统来讲是不可存取的，但用户不会发现问题。用 fsck 程序收回这些无链的磁盘块，会比第一种情况下它所做的清理工作容易一些。

5.16.2　竞争条件

在系统调用 unlink 中，存在许多竞争条件（race condition），特别是在拆除目录时。命令 rmdir 在证实一个目录不含任何文件之后，清除该目录（该命令读目录，检查所有目录表项的索引节点号是否为零）。但是，由于 rmdir 是在用户级进行，证实一个目录是否为空并清除该目录的一系列操作不是原子的（atomic）。系统在执行系统调用 read 和 unlink 之间可能进行上下文切换。因此，另一个进程可能在 rmdir 确定该目录为空之后，又在该目录中创建一个文件。用户只有靠利用文件和记录上锁功能，才能避免这种情况。然而，一旦一个进程执行 unlink，其他进程就不能存取正被拆除的文件，因为该文件及其父目录的索引节点都是上锁的。

请回顾系统调用 link 的算法以及内核如何在完成该系统调用之前解锁索引节点。如果另一个进程在索引节点解锁期间要拆除该文件，它只能使该索引节点的联结数减 1；因为在

拆除该索引节点之前，联结数曾被加1，所以此时联结数仍然大于0。因此，该文件不会被清除，系统是安全的。这一条件等价于在link调用之后立即发生unlink的情况。

另一种竞争条件发生在：一个进程正在用算法namei将一个文件的路径名转变为一个索引节点时，另一个进程正在清除该路径名中的一个目录。假定进程A正在分析路径名“a/b/c/d”，并睡眠等待为“c”分配一个内存索引节点。睡眠可能发生在正要为该索引节点上锁期间，也可能发生在读取该索引节点所在的磁盘块期间（见算法iget和bread）。如果进程B要拆除目录“c”，它可能因和进程A相同的原因而进入睡眠。假定过了一会儿，内核在调度A之前，先调度B去运行。进程B可能在进程A再运行之前，就拆除了目录“c”，并清除了它的内容（假定为最后的联结）而结束运行。再过一会儿，当进程A恢复运行时，它将试图存取一个已被删除的非法内存索引节点。因此，算法namei在进行操作之前，要检查联结数是否为0，如果为0，则要报告出错。

然而，仅作这个检查是不够的。可以想象，另一个进程可能在该文件系统的某个地方创建一个新目录，并使用以前用于“c”的索引节点。进程A上当受骗，认为它存取的是正确的索引节点(见图5-32)。尽管如此，系统还维护着它的完整性。可能发生的最坏情况是，

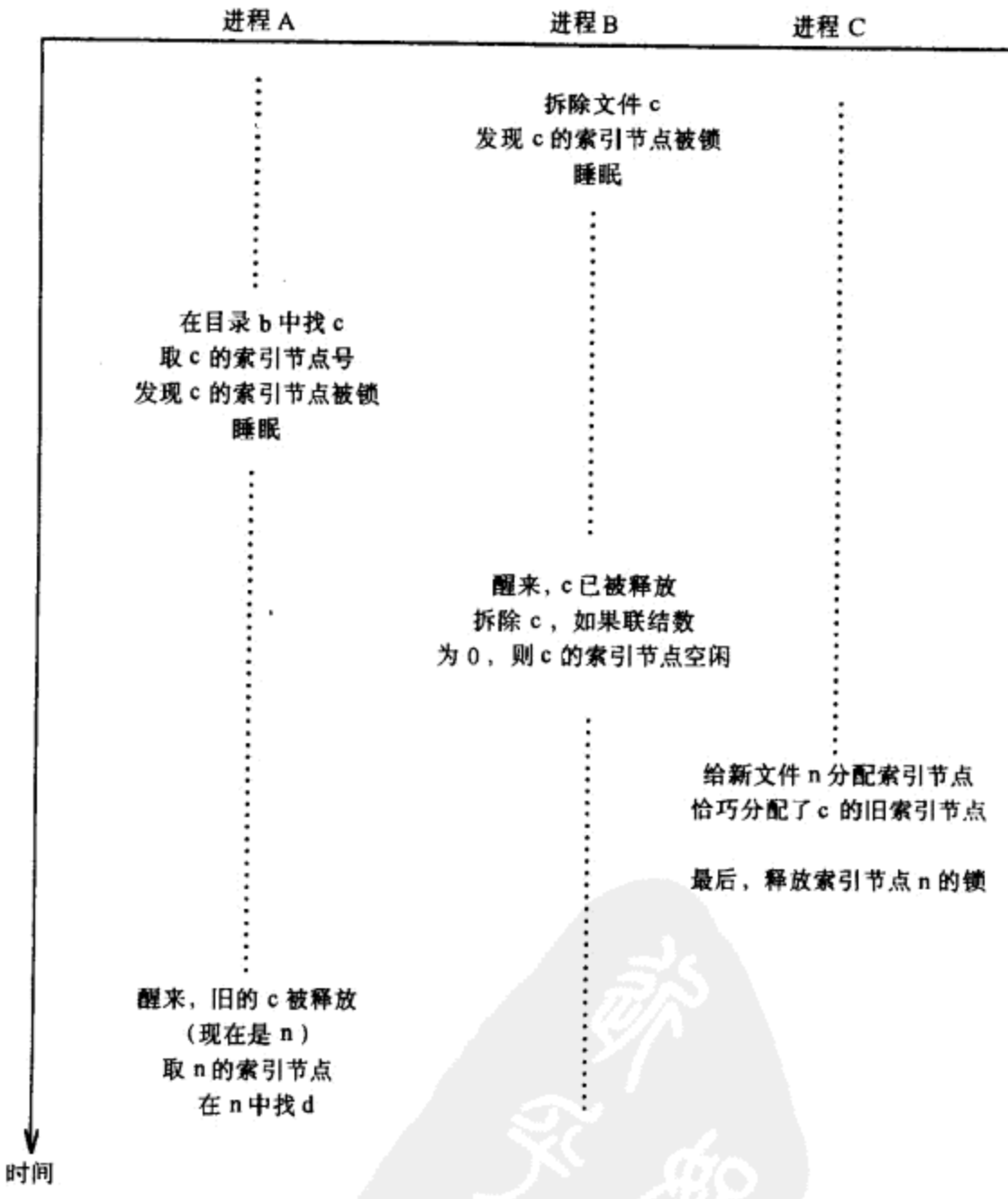


图 5-32　unlink的竞争条件

进程取错了文件，因而可能破坏了保密性——但是，这种竞争条件在实际中很少发生。

当一个进程使一个文件处于打开状态时，另一个进程可能在该文件打开期间拆除该文件（甚至做这件事的进程本身就是发出系统调用 open 的进程）。因为内核在 open 调用结束时，解锁索引节点，所以系统调用 unlink 会成功，内核将成功地执行 unlink，就好象这个文件没被打开一样。它将清除该文件对应的目录表项，其他进程都不能再存取这个已被拆除的文件。但是，由于系统调用 open 已使该索引节点的引用数加了 1，所以，当内核在 unlink 结尾处执行算法 iput 时，并不清除该文件的内容。这样，打开文件的进程仍能用它的文件描述符进行所有正常的文件操作，包括读、写文件。但是，当它关闭该文件时，在算法 iput 中，该索引节点的引用数减为零，内核便清除该文件的内容。简言之，打开文件的进程继续运行，好象 unlink 没有发生过，而 unlink 也正常操作，好象该文件没被打开过。执行过该 open 调用的进程中的其他系统调用也会继续为该进程服务。

图 5-33 给出了一个进程打开一个文件，然后又拆除它的程序。系统调用 stat 不会成功，因

```
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>

main (argc, argv)
        int atgc;
        char * argv [];
{
        int fd;
        char buf [1024];
        struct stat statbuf;

if (argc! =2)           /* 需要一个参数 */
        exit ();
fd=open (argv [1], O_RDONLY);
if (fd= = -1)           /* open 失败 */
    exit ();
if (unlink (argv [1]) = = -1)       /* 拆除刚刚打开的文件 */
    exit ();
if (stat (argv [1], &statbuf) = = -1)       /* 用文件名查状态 */
    printf ( "stat %s fails as it should \n", argv [1]);
else
    printf ( "stat %s succeeded!!!! \n", argv [1]);
if (fstat (fd, &statbuf) = = -1)            /* 用 fd 查文件状态 */
    printf ( "fstat %s fails!!! \n", argv [1]);
else
    printf ( "fstat %s succeeds as it should \n", argv [1]);
while (read (fd,, buf, sizeof (buf)) >0)    /* 读打开而且已拆除的文件 */
    printf ( "%1024s", buf);        /* 打印 1K 字节的域 */
}
```

图 5-33　拆除一个打开的文件

为原来的路径名在 unlink 之后不能再用来引用该文件了（当然要假定没有其他进程同时又创建了叫这个名字的文件），但是 fstat 将会成功返回，因为它用该文件的文件描述符得到了该文件的索引节点。然后，该进程进入循环，每次读该文件中的 1024 个字节，并将该文件打印到标准输出上。当系统调用 read 遇到文件尾时，该进程退出（exit）：在系统调用 exit 中，做了 close 之后，该文件不再存在。利用这个特点，一些进程通常创建一些临时文件并立即拆除它们。这些进程仍能继续读写这些文件，但文件名在目录结构中不再出现，如果某个进程由于某种原因失败，它不会留下临时文件的痕迹。

5.17　文件系统的抽象

Weinberger 引进了文件系统类型（file system types）来支持他的网络文件系统（见[Killian 84] 对这个机制的简要描述），而最新的系统 V 版本支持他的方案的一个分支。文件系统类型允许内核同时支持多种文件系统，例如网络文件系统（见第 13 章），甚至其他操作系统的文件系统。进程用通常的 UNIX 系统调用存取这些文件，内核再将文件操作的通用集映射到每个文件系统中专用操作。

索引节点是抽象文件系统和特殊文件系统之间的接口。一个通用的内存索引节点中所含的数据与特定的文件系统无关，它指向一个文件系统专用的索引节点，而这个索引节点含有该文件系统专用的数据，像存取权限和数据块布局这样的信息。通用索引节点含有设备号、索引节点号、文件类型、大小、所有者及引用数等信息。其他的文件系统的专用数据是超级块和目录结构。图 5-34 给出了通用索引节点表和两个文件系统的专用索引节点表，这两个文件系统的专用索引节点表，一个用于系统 V 的文件系统结构，另一个用于远程（网络）索引节点。远程索引节点大致含有足以识别一个远程系统中的文件的信息。一个文件系统可能没有类似于索引节点这样的结构；但是，可以由该文件系统的专有代码，构造一个满足 UNIX文件系统语义的实体，并在内核分配通用索引节点时，分配该文件系统的专用“索引节点”。

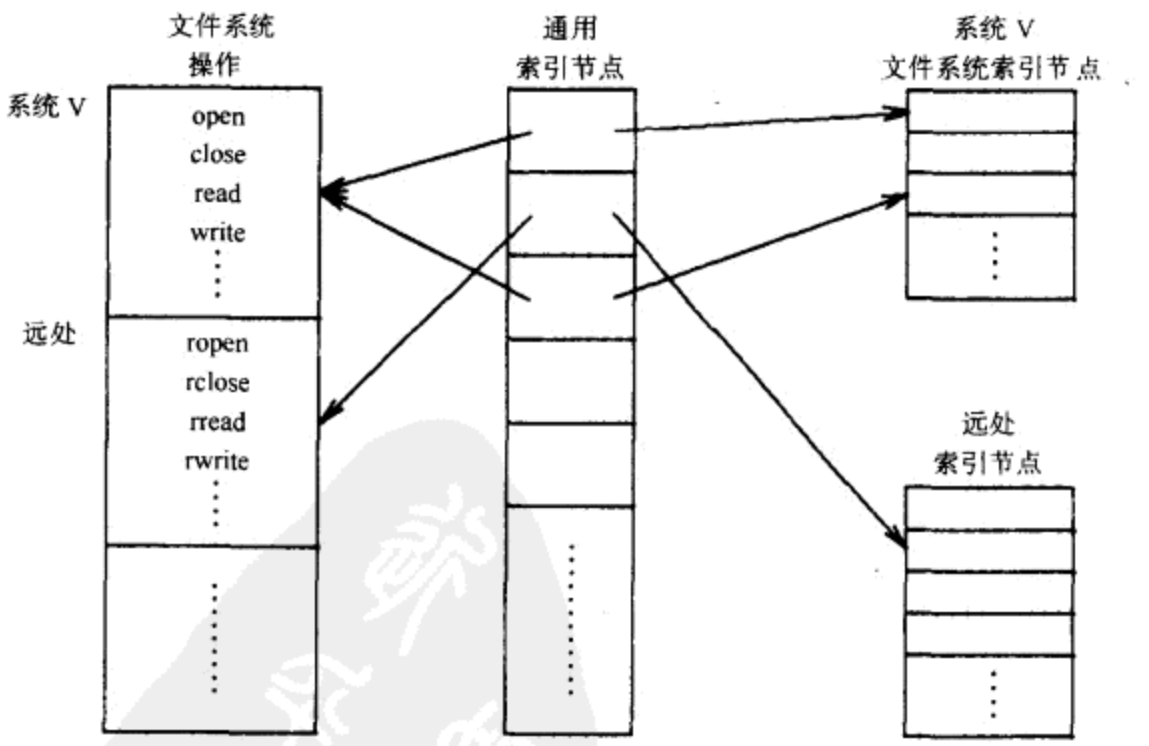


图 5-34　文件系统类型的索引节点

每种文件系统类型都有一个含有完成抽象操作的函数的地址的结构。当内核要存取一个文件时，它根据文件系统的类型和操作，做一次间接的函数调用（见图 5-34）。抽象操作包括打开文件、关闭文件、读写数据、返回一个文件名分量的索引节点（类似于 namei 和 iget）、释放一个索引节点（类似于 iput）、更新索引节点、检查存取权限、设置文件属性（存取权限）及安装和拆卸文件系统等。我们将在第 13 章介绍分布式文件系统时，说明文件系统抽象的使用。

5.18　文件系统维护

在正常的操作中，内核能保持文件系统的一致性。但是，由于像电源故障之类的异常情况可能引起系统的垮台，使文件系统处于不一致状态——文件系统中的大部分数据是可用的，但有些可能是不一致的。命令 fsck 检查这种不一致性，并在必要时加以修正。它通过磁盘块或原始接口存取文件系统（见第 10 章），并绕过正规的文件存取方法。本节将介绍几种由 fsck 检查的不一致性。

一个磁盘块可能属于一个以上的索引节点，或属于自由块链表和一个索引节点。当一个文件系统最初建立时，所有的磁盘块都在自由链表上。当某个磁盘块被分配使用时，内核将它从自由链表中取出并分给一个索引节点。内核不可以将没有归还到自由链表中的磁盘块重新分配给另一个索引节点。因此，正常情况下，一个磁盘块或者在自由链表上，或者被分配给一个索引节点。但是，存在这样一种可能性：内核释放了一个文件中的磁盘块，将该块的块号返回给超级块的内存拷贝，又将该磁盘块分配给另一个新文件。如果内核已将新文件的索引节点和磁盘块写到了磁盘上，但在修改老文件在磁盘上的索引节点拷贝之前，系统垮台了，那么就会有两个索引节点对相同的磁盘块号寻址。类似地，如果内核已将超级块和它的自由链表写回磁盘上，但在将老的索引节点写出之前，系统垮台，那么该磁盘块既在自由链表上，又属于老的索引节点。

如果一个块号既不在块的自由链表上，又不在任何文件中，文件系统则是不一致的，因为如上所述，所有的磁盘块必须出现在某个地方。这种情况发生在将一个磁盘块从一个文件移走，并被放到超级块的自由链表上时：老的文件已被写回磁盘，但在将超级块写到磁盘上之前，系统垮台了。这样，在磁盘上的任何链表中都不会有这个磁盘块。

一个索引节点可能有非零联结数，但它的索引节点号却不在文件系统中的任何目录中。除（无名）管道以外的所有文件都必须在文件系统树中。如果在创建了一个管道或创建了一个文件之后，在创建它的目录之前，系统发生故障，那么其索引节点的联结数字段已被设置，即使该索引节点并未出现在文件系统中。如果在确保一个目录中的所有文件都被拆除之前，就拆除该目录，也会发生这种情况。

如果一个索引节点的格式不正确，比如，文件类型字段的值是无定义的值，那么也意味着发生了什么不对劲儿的事。如果一个系统管理员安装了一个没有正确格式化的文件系统，就会发生这种情况。内核存取它认为是装有索引节点的磁盘块，但实际上，该磁盘块是数据块。

如果一个索引节点号出现在目录表项中，但同时它又是自由可用的，那么文件系统也是不一致的，因为出现在一个目录表项中的索引节点号应该是一个已分配的索引节点。这种情况发生在内核正在创建一个新文件时：它已将目录表项写回磁盘，但因系统垮台，没来得及将该索引节点写到磁盘上。这种情形还可能发生在一个进程拆除了一个文件，并将自由的索

引节点写到了磁盘上，但因系统垮台，没来得及将目录表项写到磁盘上时。以适当的次序进行操作写，可以避免这些情况。

如果保存在超级块中的自由块数或自由节点数与磁盘上存在的数目不一致，该文件系统就是不一致的。超级块中的总计信息必须总是与文件系统的状态相一致。

5.19　本章小结

本章结束了本书的第一部分——文件系统概述。本章介绍了三种内核表，它们是用户文件描述符表，系统文件表和安装表。本章还描述了许多有关文件系统的系统调用算法及它们之间的相互联系。这里还介绍了文件系统的抽象，它允许 UNIX 系统支持各种文件系统类型。最后，本章描述了 fsck 如何检查文件系统的一致性。

5.20　习题

1. 考虑图 5-35 中的程序。三个 read 的返回值是什么？缓冲区的内容是什么？请描述在每次 read 期间，内核中发生的事情。

```
#include <fcntl.h>
main ()
{
    int fd;
    char buf [1024];
    fd = creat ( "junk", 0666);
    lseek (fd, 2000L, 2);          /* 定位到字节 2000 处 */
    write (fd, "hello", 5);
    close (fd);

    fd = open ( "junk", O_RDONLY);
    read (fd, buf, 1024);          /* 读零 */
    read (fd, buf, 1024);          /* 取到某些内容 */
    read (fd, buf, 1024);
}
```

图 5-35　读零和文件尾

2. 重新考虑图 5-35 中的程序，但这里假定语句

```
lseek (fd, 9000L, 0);
```

放在第一个 read 之前。这时，进程看到什么？内核中发生什么情况？

3. 一个进程可以“追加写”方式打开一个文件，这意味着每次写操作都是从标识当前文件尾的字节偏移量处开始。因此，两个进程可以以追加写方式打开同一个文件并写该文件而不会覆盖数据。如果一个进程以追加写方式打开一个文件，并定位于文件头，会发生什么情况呢？

4. 标准 I/O 库通过在库中缓冲数据，使用户能更有效地进行读和写，因此暗地里节省了用户必须作的系统调用数。你如何实现库函数 fread 和 fwrite？库函数 fopen 和 fclose 应该

做些什么？

5. 如果进程从一个文件中连续地读数据，内核则在内存索引节点中记录提前读块的值。如果有若干进程从同一个文件连续地读，会发生什么情况？

6. 考虑图 5-36 中的程序。程序执行的时候会发生什么情况？如果 buf 的说明被夹在另外两个大小为 1024 字节的数组的说明之间，会发生什么情况？内核如何判断出 buf 对 read 来说是太小了？

```
#include <fcntl.h>
main ()
{
    int fd;
    char buf [256];

    fd=open ( "/etc/passwd", O_RDONLY);
    if (read (fd, buf, 1024) <0)
        printf ( "read fails\n");
}
```

图 5-36　小缓冲区时的读

*7. 根据下面给出的规则，BSD 文件系统允许按需要将文件的最后一块分割成分段。

- 以类似于超级块的结构，记住自由分段。
- 内核并不保存一个预分配的自由分段池，而是在必要的时候，将一个自由块分割成若干分段。
- 内核仅对一个文件的最后一块分配分段。
- 如果一个块被分割为若干分段，内核可以将它们分给不同的文件。
- 每个文件系统中，一个块所含的分段的数目是固定的。
- 内核在系统调用 write 中分配分段。

请设计一个将分段分配给一个文件的算法。为允许分割分段，索引节点需作哪些改变？对于使用间接块的那些文件，使用分段，从性能的角度讲有什么优点？若在系统调用 close 中，而不是在 write 中分配分段，会更好些吗？

*8. 回顾第 4 章中关于将数据放到一个文件的索引节点中的讨论。如果索引节点的大小等于一个磁盘块的大小，请设计一个算法，使一个文件的最后数据写到索引节点块中（如果装得下的话）。请将这个方法和第 7 题中讨论的方法作一比较。

*9. 系统 V 用系统调用 fcntl 来实现文件和记录的锁操作：

```
fcntl (fd, cmd, arg);
```

这里，fd 是文件描述符，cmb 指定锁操作的类型，arg 规定各种参数，例如锁类型（读或写）和字节偏移量（见附录）。锁操作包括：

- 测试属于其他进程的锁并立即返回，指明是否发现其他的锁。
- 设置一个锁并进入睡眠直到成功。
- 设置一个锁，如果不成功，也立即返回。

内核在关闭（close）文件时，自动释放由某个进程设置的锁。请描述一个实现这种文件和记

录锁操作的算法。如果锁是强制性的，应该禁止其他进程存取该文件。系统调用 read 和 write 需作哪些修改？

*10. 如果一个进程在等待某个文件的锁被释放期间进入睡眠，则存在着死锁的可能：进程 A 可能将文件“1”上锁并企图将文件“2”上锁，而进程 B 可能将文件“2”上锁并试图锁文件“1”。两个进程都处于不能继续运行的状态。请扩充上一题的算法，使内核能够在这种情况即将发生时，检测出这种情况，并使系统调用失败。内核是不是检测死锁的合适的地方？

11. 在文件锁操作的系统调用出现之前，通过执行具有原子特性的系统调用，用户可以使协同操作的进程实现锁机制。请问可以利用本章描述过的哪些系统调用？使用这种方法的内在危险是什么？

12. Ritchie 认为（见［Ritchie 81］），文件上锁不足以防止像编辑程序这样的程序所引起的混乱：编辑程序在编辑文件时，先作一个文件拷贝，编辑完之后，再写到原文件上。请解释他的意思并加以评论。

13. 请考虑另外一种预防破坏性修改文件的上锁方法。假定索引节点中有一个新的许可权设置，它每次只允许一个进程为写而打开文件，但允许多个进程为读而打开文件。请给出一种实现方法。

*14. 考虑图 5-37 中的程序。该程序以错误的格式创建了一个目录结点（没有“.”和“..”的目录表项）。请在这个新目录上试几个像 ls -l，ls -ld 或 cd 这样的命令。会发生什么情况？

```
main (argc. argv)
    int argc;
    char * argv [];
{
    if (argc! =2)
    {
        printf ( "try: command directory name \ n");
        exit ();
    }

    /* 方式：目录 (04) 对所有用户可读可写可执行 */
    /* 只有超级用户可做 */
    if (mknod (argv [1], 040777, 0) = = -1)
        printf ( "mknod fails \ n");
}
```

图 5-37　一个错误格式的目录

15. 写一个程序，它打印出文件所有者、文件类型、存取权限和文件的存取时间，文件名是提供的参数。如果是目录文件，该程序应该读（read）该目录，并对该目录中的所有文件，打印出上述信息。

16. 假定一个目录对一个用户有读许可权，但没有执行许可权。当该目录被用作 ls -i 的参数时，会发生什么情况？对 ls-l 呢？请对答案作出解释。试就对该目录有执行许可权，但

没有读许可权的情况，回答上述问题。

17. 请比较一个进程对下列操作应有的存取权限并加以评论：

- 创建一个新文件要求对一个目录有写许可权；
- 创建一个已存在的文件要求对该文件有写许可权；
- 拆除一个文件要求对其父目录有写许可权，但不要求对该文件本身有许可权。

*18. 写一个程序，从当前目录开始，访问每个目录。它应如何在目录树中处理循环？

19. 执行图 5-38 中的程序，并描述内核中发生的情况。（提示：当程序运行完时，执行一下 pwd。）

```
main (argc, argv)
    int argc;
    char * argv [];
{
    if (argc! =2)
    {
        printf ( "need 1 dir arg \ n");
        exit ();
    }

    if (chdir (argv [1] = = -1)
        printf ( "%s not a directory \ n", argv [1]);
}
```

图 5-38　使用系统调用 chdir 的程序

20. 请写一个程序，将该程序本身的根改变到一个指定的目录，并探讨一下该程序能访问的目录树。

21. 为什么一个进程不能使前一次系统调用 chroot 无效？请修改实现方法，使进程能将它的根改回到以前的某个根。具有这种特性后的优缺点是什么？

22. 请看图 5-19 中简单的管道例子。其中，一个进程将字符串“hello”写入管道，然后读该字符串。如果写入管道的是 1024 个字节，而不是 6 个字节，会发生什么情况（读的数据量仍是 6）？如果系统调用 read 和 write 的次序颠倒一下，会发生什么情况？

23. 在说明有名管道的使用的程序中（图 5-19），如果 mknod 发现那个有名管道已经存在，会发生什么情况？内核应如何实现这一点？如果有许多进程都想通过这个有名管道进行通信，而不是像文中暗示的只有一个读进程和一个写进程，那么会发生什么情况？多个进程如何保证只有一个读进程和一个写进程在通信？

24. 当为读而打开一个有名管道时，一个进程将睡眠直到另一个进程为写而打开该管道。请问为什么？难道进程不能由系统调用 open 中成功地返回，继续处理，直到要读该管道的时候，再在系统调用 read 中进入睡眠吗？

25. 你如何用语法格式

```
dup2 (oldfd, newfd);
```

来实现 dup2（第 7 版）？这里，oldfd 是要被复制的文件描述符，newfd 是复制后的文件描述符。如果 newfd 已经指向一个打开的文件，会发生什么情况？

*26. 如果内核允许两个进程同时在两个安装点上安装同一个文件系统会发生什么怪现象?

27. 假定一个进程将其当前目录转到“/mnt/a/b/c”，然后，第二个进程将一个文件系统安装到“/mnt”。这个系统调用 mount 会成功吗? 如果第一个进程执行 pwd，会发生什么情况? 如果“/mnt”的索引节点引用数大于 1，内核不允许 mount 成功。请加以解释。

28. 在跨越安装点的算法中，当识别出文件路径名中的“..”时，内核检查三个条件，看它是不是安装点：所找到的索引节点是不是具有根索引节点号；工作索引节点是不是文件系统的根；路径名分量是不是“..”。为什么要检查所有这三个条件? 请指出为什么其中任意两个条件都不足以允许进程跨越安装点?

29. 如果一个用户安装一个“只读”文件系统，内核就在超级块中设置一个标志。在 write，creat，link，unlink，chown 和 chmod 这些系统调用中，内核如何防止写操作? 上述这些系统调用都对文件系统做了什么写操作?

*30. 假定在一个进程要拆卸一个文件系统的同时，另一个进程要在这个文件系统中创建一个新文件。系统调用 unlink 和 creat 中只有一个能成功，请找出竞争条件。

*31. 当系统调用 umount 检查一个文件系统中不再有活动的文件时，遇到这样的问题：在系统调用 mount 期间，通过 iget 分配了文件系统的根索引节点，因此，该索引节点的引用数大于零。umount 如何确认没有活动的文件并考虑文件系统的根? 考虑如下两种情况：

- 在检查活动索引节点之前，系统调用 umount 通过算法 iput 释放根索引节点。(如果还有活动的文件，内核如何恢复根索引节点?)
- 在释放根索引节点之前，umount 检查活动文件，但允许根索引节点保持活动着。(根索引节点能如何活动?)

32. 当在一目录上执行命令 ls-ld 时，请注意目录的联结数总不会是 1，为什么?

33. 命令 mkdir（建新目录）是如何工作的?（提示：当 mkdir 结束时，“.”和“..”的索引节点号是什么?)

*34. 符号联结指的是在不同的文件系统中联结文件的能力。用一个新的类型指示符指定一个符号联结文件；符号联结文件中的数据是要联结到的文件的路径名。请给出一个实现符号联结的算法。

*35. 当一个进程执行

```
unlink(".");
```

时，会发生什么情况? 该进程的当前目录是什么?（假定具有超级用户许可权。)

36. 设计一个系统调用，该系统调用将一个已存在的文件截为任意大小（为所提供的一个参数)，并说明实现方法。实现一个系统调用，它允许用户清除指定字节偏移量之间的一段文件，压缩文件的大小。在没有这样的系统调用情况下，设计一个提供这种功能的程序。

37. 描述一个索引节点的引用数可能大于 1 的所有条件。

38. 在文件系统抽象中，每一个文件系统是否都应支持一个由通用代码调用的私有锁操作? 一个通用的锁操作是否满足需要?

第6章 进程结构

第2章曾系统地阐述了进程的一些高层次特性。本章将更严格地论述这些观点，定义进程上下文，并描述内核是如何标识和查找一个进程的。6.1节定义UNIX系统的进程状态模型及状态集合的转换。内核内部有一个进程表，进程表的表项描述了系统中每个活动进程的状态。更多的控制进程操作的信息，存放在一个称为u区的数据结构中，进程表表项及u区是进程的上下文组成部分。显然，使一个进程上下文不同于另一个进程上下文的最明显的地方，是它的地址空间的内容。6.2节描述了进程和内核的存储管理原理，以及操作系统和硬件如何通过协作来完成虚地址的转换。6.3节详细给出进程上下文的组成部分。6.4节说明，在发生中断、系统调用或上下文切换时，内核是如何保存一个进程的上下文，以后又如何恢复执行被挂起的进程的。6.5节给出第7章要描述的系统调用所采用的各种算法，这些算法管理着进程的地址空间。最后，6.6节是关于使进程睡眠和唤醒进程的算法。

6.1 进程的状态和状态的转换

如第2章所述，一个进程的生命期从概念上可分为一组状态，这些状态刻划了进程。下面列出了进程状态的完整集合。

(1) 进程在用户态下正在执行。

(2) 进程在核心态下正在执行。

(3) 进程未被执行，但处于就绪状态，只要内核调度到它，即可执行。

(4) 进程正在睡眠并驻留在主存中。

(5) 进程处于就绪状态，但对换进程（swapper）（进程0）必须把它换入主存，内核才能调度它去执行。

(6) 进程正在睡眠，而且对换进程已把它换到二级存储器，从而为其他进程腾出空间。

(7) 进程正从核心态返回用户态。但内核抢先于它，并做了上下文切换，以调度另一个进程。我们一会儿再说明这个状态和状态3（“就绪状态”）的区别。

(8) 进程刚被创建，处于变迁状态。该进程存在，但还没有就绪，也不在睡眠。这个状态是除进程0以外的所有进程的初始状态。

(9) 进程执行了系统调用exit，处于僵死（zombie）状态。该进程不再存在，但它留下一个记录，该记录可由其父进程收集，其中含有出口码及一些计时统计信息。僵死状态是进程的最后状态。

图6-1给出了完整的进程状态转换图。下面让我们看一个典型的进程经历这个状态转换模型的过程。这里所描述的事件是人为设置的，进程并不总是要经历这些事件，但这些事件说明了各种可能的状态转换。首先，当父进程执行系统调用fork时，其子进程进入状态模型中的“创建”状态，并最终会移到“就绪状态”（3或5）。为简单起见，假定该进程进入“在内存中就绪”状态。进程调度程序最终将选取这个进程去执行，这时，它便进入“核心态运行”状态。在这个状态下，它完成它的fork部分。

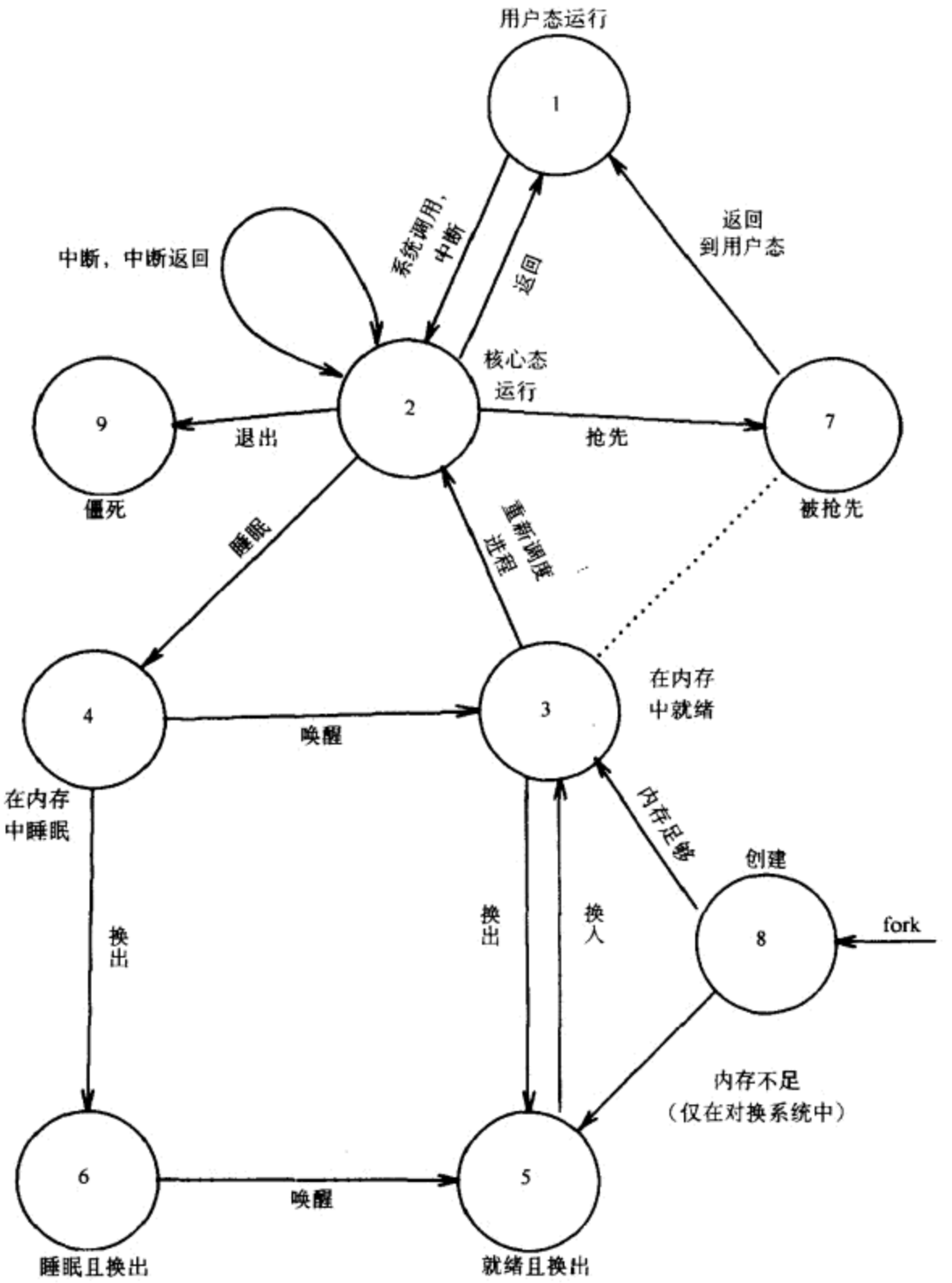


图 6-1 进程的状态转换图

当该进程完成系统调用时，它可能进入“用户态运行”状态，此时它在用户态下运行。一段时间后，系统可能中断处理机，进程则再次进入“核心态运行”状态。当时钟中断处理程序结束了中断服务时，内核可能决定调度另一个进程运行。这样，头一个进程进入“被抢先”状态（prompted state），而后者开始了它的运行。“被抢先”状态实际上和“在内存中就绪”状态一样（图中用两个状态间的点划线强调它们的等价性）。分别画出这两种状态是为了强调：在核心态运行的进程，只有在它即将返回用户态时，才可被抢先。因而，内核在必要时，可能从“被抢先”状态中对换一个进程。最后，调度程序将选取我们例子中的进程去执行，它便回到“用户态运行”状态，再次在用户态下运行。

当一个进程执行系统调用时，它便离开“用户态运行”状态，进入“核心态运行”状

态。假定这个系统调用是请求磁盘输入/输出操作，则该进程需等待输入/输出的完成。因此它进入“在内存中睡眠”状态，一直睡到被告知输入/输出已经完成。当输入/输出完成时，硬件便中断 CPU，中断处理程序唤醒该进程，使它进入“在内存中就绪”状态。

假定核心正在执行多个进程，但它们不能同时都装入主存。进程 0，即对换进程，换出我们的进程，以便为另一个处于“就绪且换出（swapped）”状态的进程腾出空间。当进程被从主存中驱逐出去后，它进入“就绪且换出”状态。最后，对换进程选择我们的进程作为最应换入主存的进程，这样，它便重新进入“在内存中就绪”状态。调度进程最终将选取该进程运行，使它进入“核心态运行”状态。当该进程完成时，它发出系统调用 exit，进入“核心态运行”状态，最后，进入“僵死”状态。

进程可以在用户级对某些状态的转换加以控制。首先，一个进程可以创建另一个进程。然而，进程从“创建”状态所发生的状态转换（即到“在内存中就绪”或到“就绪且换出”状态）却取决于内核：进程不能控制这些状态的转换。其次，一个进程可以发出系统调用，实现从“用户态运行”状态到“核心态运行”状态的状态转换。但是，进程不能控制何时从内核返回。有些事件可能强迫它永不返回而进入“僵死”状态（见 7.2 节关于软中断信号的讨论）。最后，一个进程能按自己的意愿退出（exit）。但如前所述，外部事件可能强迫它退出，而不用它明显地发出系统调用 exit。所有其他状态的转换都严格遵循内核中已编码的模型，并根据本章和以后章节中详细说明的规则，以可预言的方式对事件作出反应。某些规则已经作了介绍，例如，任何进程都不能抢先另一在内核中运行的进程。

描述进程状态的内核数据结构是：进程表项及 u 区。进程表中的域对内核来说必须总是可存取的。而 u 区中的域只能由正在运行的进程来存取。因此，只有当创建一个进程时，内核才为其分配 u 区空间。那些不与任何进程相联系的进程表项是不需要 u 区的。

进程表中的字段有：

- 状态字段，标识进程的状态。
- 进程表项中有一些字段，这些字段允许内核找出对应的进程和其 u 区在内存或在二级存储器中的位置。当进程从“在内存中就绪”状态转换到“核心态运行”状态，或从“被抢先”状态到“用户态运行”状态时，内核使用这些信息对进程作上下文切换（context switch）。此外，当进程被换入或换出主存（或换页）时，内核也要用到这些信息。进程表项中还有一个字段，给出进程大小，使内核知道应为该进程分配多少空间。
- 若干用户标识号（user ID，简称用户 ID 或 UID），决定进程的各种特权。例如，用户 ID 字段描述了一组相互之间可以发送软中断信号的进程。下一章将详细给出其含义。
- 若干进程标识号（process ID，简称进程 ID 或 PID），说明进程相互间的关系。这些 PID 字段是当进程由于系统调用 fork 而进入“创建”状态时而被设置的。
- 在进程处于睡眠状态时，进程表项中有一个事件描述符字段。本章将在 sleep 和 wakeup 算法中详细讨论它的用处。
- 调度参数，内核用它们决定若干个进程转换到核心态和用户态的次序。
- 一个软中断信号字段，记录发向一个进程的所有未处理的软中断信号（见 7.2 节）。
- 各种计时字段，给出进程执行的时间和内核资源的利用情况。这些信息用于为进程记帐和计算进程调度优先权。其中有一个字段是由用户设置的计时器，用来向进程发送闹钟信号（见 8.3 节）。

u区中的各个字段进一步刻画了进程状态的特性。下面给出的是u区中的域。前一章曾讨论了其中后7个字段，为完整起见，这里又简单地描述了它们。

- 一个指向进程表的指针，标识了对应于该u区的进程表项。
- 真正用户标识号（real user ID）及有效用户标识号（effective user ID），决定了进程的各种特权，如文件存取权限（见7.6节）。
- 计时器字段，记录进程（及其后代）在用户态和核心态运行时所用的时间。
- 一个表示进程希望如何对软中断信号作出反应的数组。
- 控制终端字段，标识与进程相关的“注册终端”（如果存在的话）。
- 一个出错字段，记录在一个系统调用过程中遇到的错误。
- 一个含有系统调用的结果的返回值字段。
- 一些输入/输出参数，描述要传送的数据量、在用户空间的源（或目的）数据的数组地址、文件的输入/输出偏移量等。
- 当前目录和当前根，描述进程的文件系统环境。
- 用户文件描述符表，记录该进程已打开的文件。
- 限制字段，限制了一个进程的大小及它能“写”的一个文件的大小。
- 对该进程创建的所有文件设置的一个许可权方式字段的屏蔽模式。

本节在逻辑级上描述了进程状态的转换。每个状态还有其物理特性。这些物理特性，尤其是进程的虚地址空间，由内核来管理。下一节将给出存储管理模型，其后的各节在物理级上描述进程的状态和状态的转换，主要讨论“用户态运行”状态、“核心态运行”状态、“被抢先”状态及“睡眠”(在内存)”状态。第7章将描述“创建”状态和“僵死”状态。第8章讨论的是“在内存中就绪”状态，而关于两种“对换”状态和请求调页问题，则是第9章的内容。

6.2 系统存储方案

假定机器的物理存储器是可编址的，由偏移量0字节开始，直到某个等于机器存储容量的偏移量。如第2章中所述，一个UNIX系统上的进程由三个逻辑段组成，它们是正文段、数据段和栈（为了目前讨论方便，第11章中讨论的共享存储器应该被看作是数据段的一部分)。正文段含有机器为一个进程所执行的指令集合。正文段的地址包括正文地址（用于分支和子程序调用)、数据地址（用于存取全局数据变量）或栈地址（为了存取局部于子程序的数据结构)。若机器将生成的地址看作物理存储器中的地址，那么，两个进程在它们生成的地址集合重叠的情况下，就不可能并发地运行了。虽然编译程序可以产生在程序之间不重叠的地址，但这样的程序对于通用计算机是行不通的，因为一种机器上的存储容量是有限的，而所有可能被编译的程序的集合是无限的。即使编译程序采用试探法，力图避免生成地址的不必要的重叠，其实现也是很不灵活的，因此是不合要求的。

因此，编译程序是在一给定范围的虚地址空间（virtual address space）上生成地址。机器的存储管理部件将编译生成的虚地址，转换成物理存储器中的地址单元。编译程序不必知道内核以后会把程序装入内存的什么地方去执行。事实上，程序的若干拷贝还可以在内存中同时存在：它们都在相同的虚地址上运行，但引用的却是不同的物理地址。内核的若干子系统和硬件共同操作，把虚地址转换成物理地址。存储管理（memory management）子系统就

是由这些部分组成的。

6.2.1 区

系统 V 的内核把一个进程的虚地址空间分成若干逻辑区（logical region）。区是进程虚地址空间上的一段连续区域，可把这段区域看作是可被共享和保护的独立实体。因此，正文、数据及栈通常形成一个进程的几个独立区。若干进程可以共享一个区。例如，几个进程可以执行同一个程序，自然它们会共享一个正文区。类似地，几个进程可以合作，共享一个共同的共享存储区。

内核中有一个区表，每个在系统中活动的区对应于表中的一个表项。6.5 节将更详细地描述区表的各个字段和区操作。这里，先假定区表中的信息用来决定区的内容放在物理存储器的什么地方。每个进程都有一个私有的本进程区表，简称为 pregion 表。pregion 的表项可以放在进程表、u 区或独立分配的存储区域中，这取决于实现方式。为简单起见，我们假定它们是进程表项的一部分。每个 pregion 表项指向一个区表项，并含有该区在进程中的起始虚地址。共享区在每个进程中可能有不同的虚地址。pregion 表项还含有一个许可权域，指出了对对应的进程所允许的存取类型：只读、读/写或读/执行。本进程区表和区结构类似于文件系统中的文件表和索引节点结构：若干进程能通过一个区，共享它们地址空间的某些部分，就像它们能通过一个索引节点共享对文件的存取；每个进程通过私有 pregion 表的表项存取一个区，就像它通过用户文件描述符表中的私有表项及内核文件表存取一个索引节点。

图 6-2 给出两个进程 A 和 B 以及它们的区、pregion 表和区所在的虚地址。两个进程共享正文区 a，相应的虚地址分别是 8K 和 4K。如果进程 A 读位于 8K 的存储单元，进程 B 读位于 4K 的存储单元，实际上它们读的是区 a 中的同一个存储单元。两个进程的数据区和栈区是各自私有的。

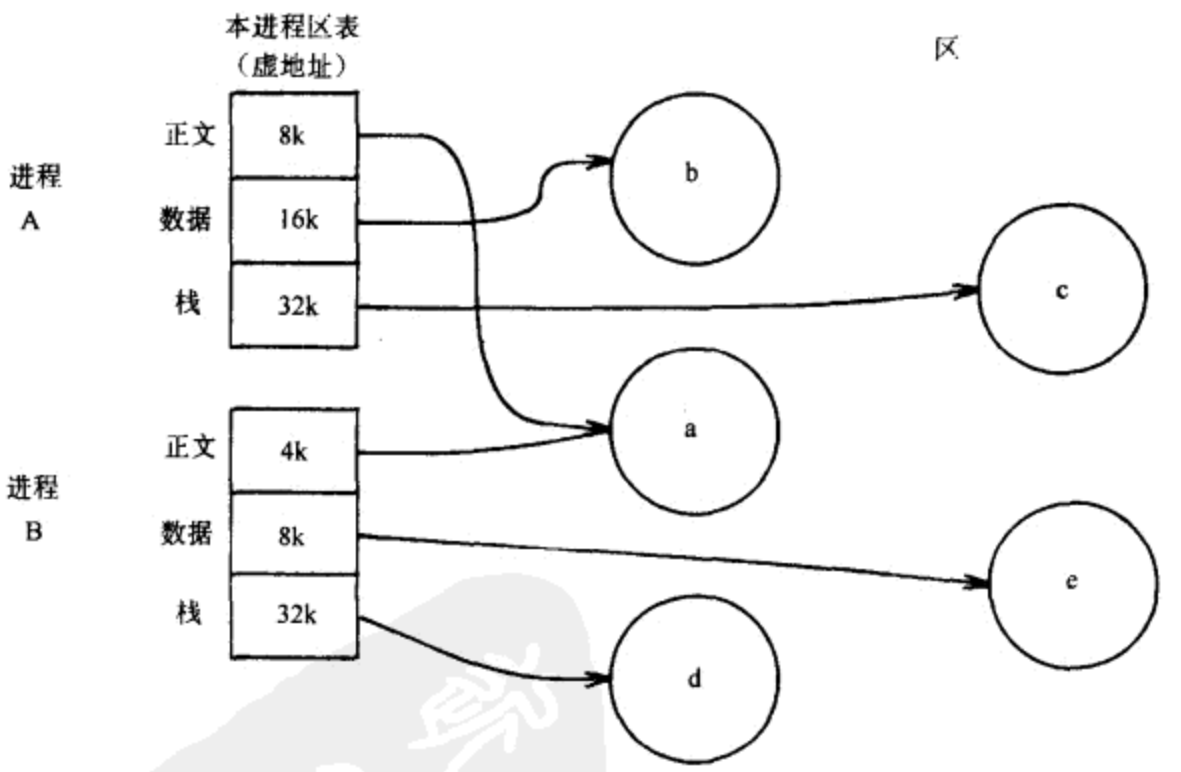


图 6-2 进程和区

区的概念独立于存储管理策略。存储管理策略是由操作系统实施的。存储管理策略是指内核为确保进程公平地共享存储资源所采取的行动。例如，第9章中讨论的两种存储策略——对换和请求调页。区的概念还独立于存储管理的实现，如，存储器是分页的还是分段的。为了给第9章中描述的请求调页算法打下基础，在目前的讨论中，我们假定存储结构是基于请求调页算法的。

6.2.2　页和页表

本节定义本书自始至终都将采用的存储模型，但这种存储模型并不是只针对UNIX系统的。在基于页（page）的存储管理体系结构中，存储管理的硬件将物理存储器分为一些大小相等的块，称为页。典型的页面大小是从512字节到4K字节，具体多大由硬件决定。页中含有存储器中的每一个可寻址单元。因此，可由一个数对（页号，页内字节偏移量）来寻址内存的每个单元。例如，如果一个机器的物理存储器是2^{32}个字节，且一页的大小为1K字节，那么，它就有2^{22}个物理存储页。可将每个32位地址看成由一个22位的页号和一个10位的页内偏移量组成（见图6-3）。

十六进制地址	58432	
二进制	0101 1000 0100 0011 0010	
页号，页内偏移量	01 0110 0001	00 0011 0010
十六进制形式	161	32

图6-3　按页寻址的物理存储器

当内核向一个区分配存储器的物理页时，不必一定分配相邻的页，也不必按特殊的次序。将存储器分页的目的，是为了在分配物理存储器时有较大的灵活性。这一点类似于在文件系统中将磁盘块分配给文件，增加了灵活性，减少了由块的分割所造成的不可用空间。内核将存储器按页分给区，也是为了这个目的。

内核将区中的逻辑页号映射为机器上的物理页号，从而使区的虚地址和它们的物理机器地址联系起来（图6-4）。在一个程序中，由于区是连续的地址空间，所以逻辑页号就自然地成为一个物理页号数组的数组下标。区表项中有一个指针，指向一个物理页号表，称作页表。页表中的项也可以含有依赖于机器的信息，如许可位，表示对该页允许的读/写操作。像其他内核数据结构一样，内核将页表存放在内存，以便存取使用它们。

逻辑页号	物理页号
0	177
1	54
2	209
3	17

图6-4　逻辑页号到物理页号的映射

图 6-5 给出一个将进程映射到物理存储的例子。图中假定一页的大小为 1K 字节，并假定该进程要存取 68，432 这个虚地址。本进程区表的表项表明，该虚地址在栈区中，栈区的起始地址是 64K（十进制的 65，536）。假定栈是朝高地址方向增长，那么由 68，432 减去 65，536，可知地址 68，432 在栈区中的偏移量是 2896 字节。因每页大小为 1K 字节，则该地址是在栈区的第 2 页中（从 0 算起），页中偏移量为 848 字节。该页的物理地址是 986K。第 6.5.5 节（区的装入）将说明页表中标有“empty”的项的意思。

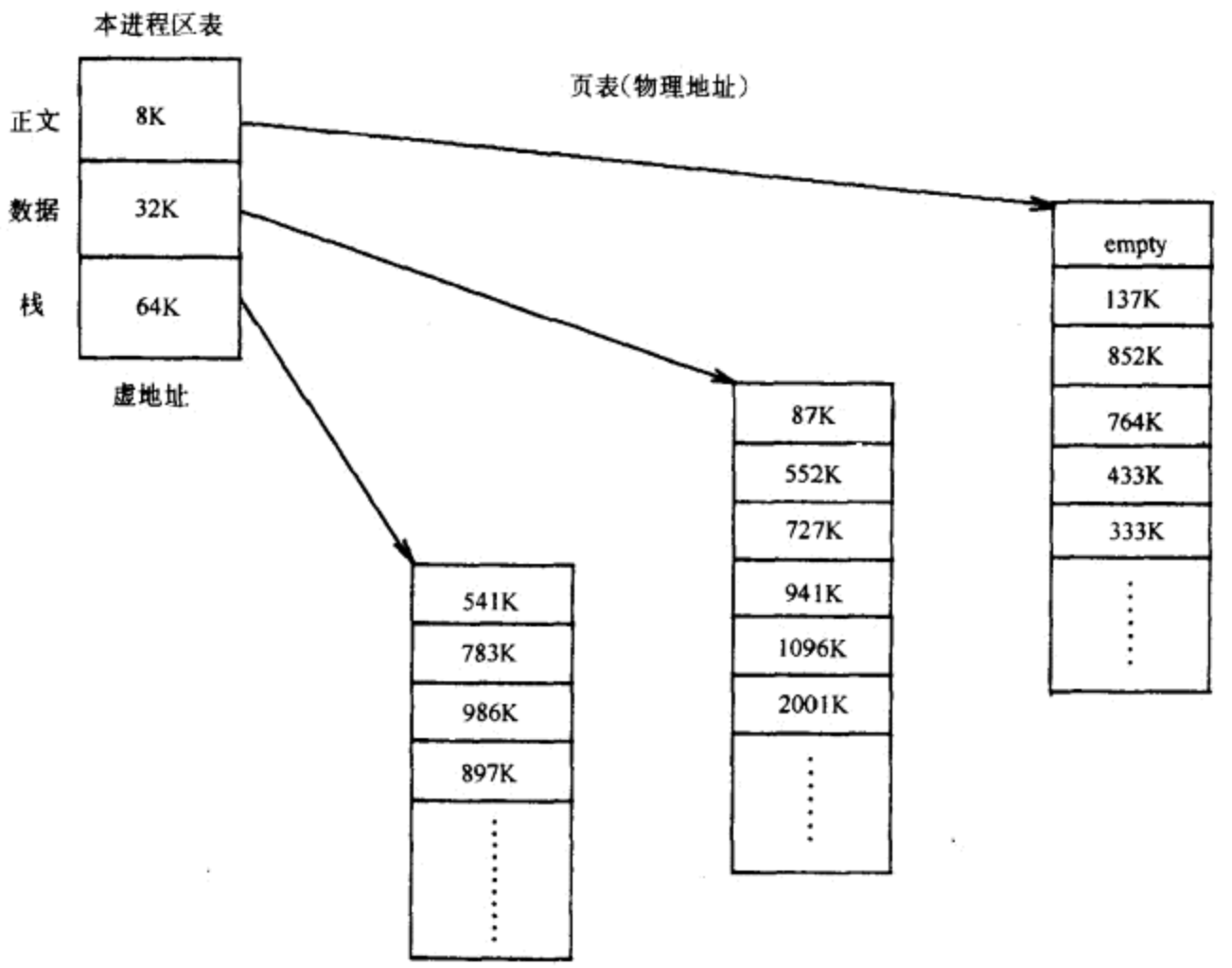


图 6-5 虚地址到物理地址的映射

先进的机器采用各种硬件寄存器和高速缓存，来加速上述地址转换过程。否则，存储器的访问和地址的计算将是很慢的。当恢复一个进程的运行时，内核要填写适当的寄存器，来告诉存储管理硬件关于页表和该进程的物理存储是在何处。由于这些操作与机器有关，并因不同的实现而异，这里不再讨论它们。本章后的习题列举了一些特殊的机器结构。

在讨论存储管理时，我们采用下述简单的存储模型。存储器按大小为 1K 字节的页来组织，通过上面描述的页表来存取。这种系统含有一些存储管理寄存器三元组（memory management register triple）（假定足够多）。寄存器三元组中的第一个寄存器存放一个页表在物理存储器中的地址，第二个寄存器存放控制信息，如页表中的页数及页存取的许可权（只读，读/写）。这种模型对应于刚刚讨论过的区模型。当内核为一个进程的运行作准备时，它将存放在本进程区表表项中的数据装入对应的存储管理寄存器三元组中。

如果进程寻址的存储单元超出它的地址空间，硬件就产生一个例外条件。如果图 6-5 中的正文区大小为 16K 字节，而一个进程存取的虚地址为 26K，那么，硬件将产生一个例外

条件，由操作系统来处理。类似地，如果一个进程企图在不具备适当的许可权的情况下存取存储器，例如，在它的写保护正文区内写一个地址，硬件将产生一个例外条件。在这两个例子中，进程通常都会退出。下章将详细讨论之。

6.2.3　内核的安排

尽管内核是在某个进程的上下文（context）中执行，但与内核相联系的虚存映射却与所有进程都无关。内核的代码和数据永久地驻留在系统中，并被所有的进程共享。

当系统初启开始服务时，它将内核代码装入内存，并设置必要的表和寄存器来将它的虚地址映射到物理存储地址。内核的页表类似于进程的页表，所采用的转换内核虚地址的方法也类似于转换用户虚地址所采用的方法。在许多机器中，一个进程的虚地址空间分为几种，包括系统的和用户的。每一种有其自己的页表。当在核心态执行时，系统允许存取内核地址，但在用户态执行时，就禁止存取内核地址。因此，当中断或系统调用引起进程从用户态到核心态的状态转换时，操作系统和硬件相互合作，允许访问内核地址。当返回到用户态时，操作系统和硬件就禁止这种访问。在另外一些机器中，当在核心态运行时，通过填写特殊寄存器来改变虚地址的转换。

图 6-6 给出内核虚地址和一个进程的虚地址的例子。图中，内核虚地址空间的范围是从

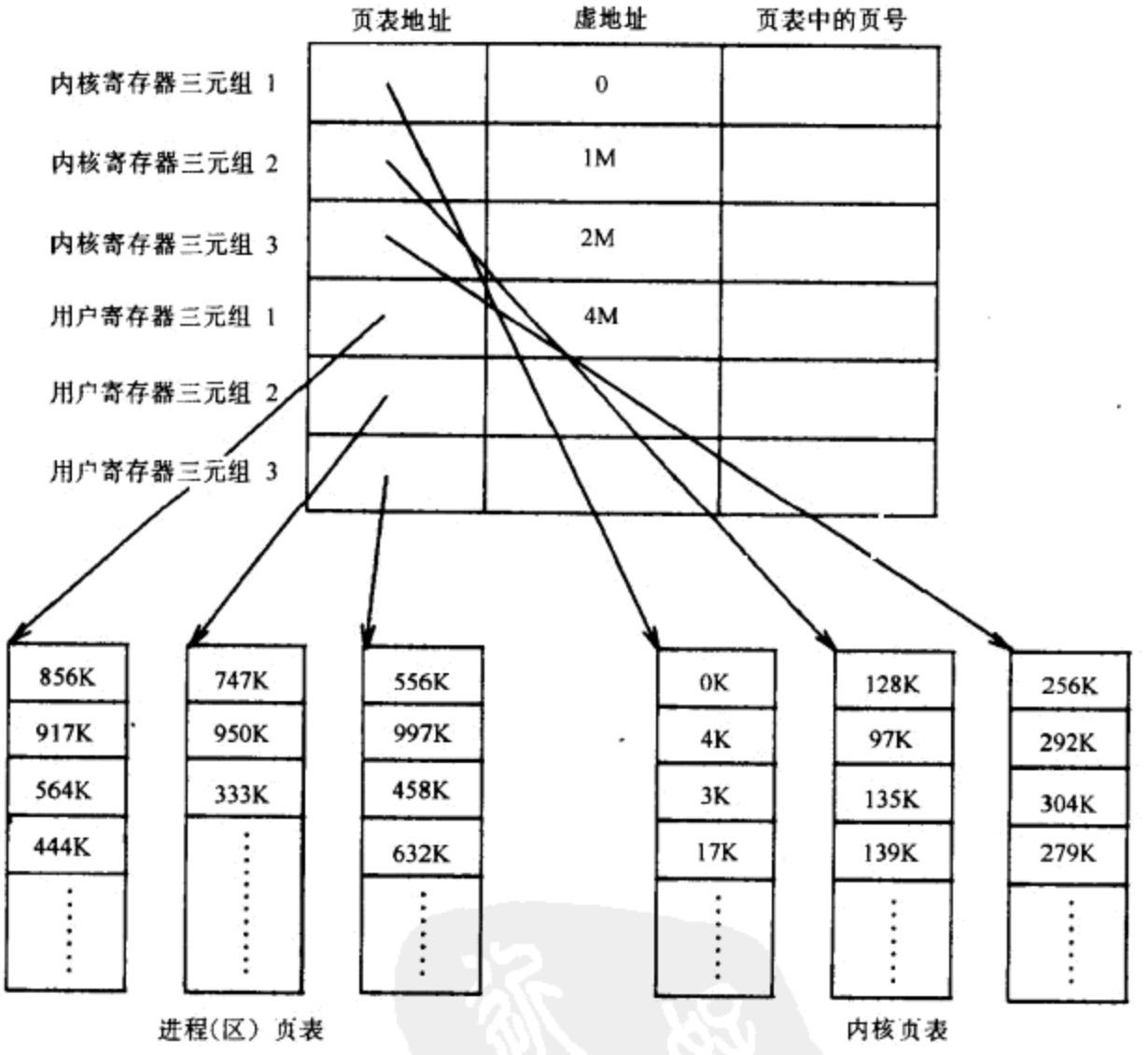


图 6-6　从用户态到核心态的状态变换

0到4M－1，用户虚地址空间是从4M开始往上。这里有两套存储管理寄存器三元组，分别用于内核地址和用户地址。每个寄存器三元组指向一个页表。页表中有对应于虚页地址的物理页号。只有在核心态时，系统才允许通过内核寄存器三元组访问地址；因此，在核心态和用户态之间的切换只是要求系统允许或拒绝用内核寄存器三元组来访问地址空间。

有些系统的实现方法是将内核装入内存，使大部分内核虚地址空间和它们的物理地址一致，这样，这部分地址的虚地址到物理地址的映射就变成一个相等函数。然而，在内核中，对u区的处理仍要求作虚地址到物理地址的映射。

6.2.4 u区

每个进程都有一个私有u区，然而，内核访问u区时，就好像系统中仅有一个u区，即正在运行的进程的u区。根据运行的进程，内核改变其虚地址转换的映射，以存取正确的u区。当编译操作系统时，装配模块给变量u赋一个值，即u区的名字，它是一个固定的虚地址。内核的其他部分也知道这个虚地址，尤其是做上下文切换的模块（见6.4.3节）。内核知道在存储管理表中的什么地方来完成对这个u区的虚地址转换。它能动态地改变u区的虚地址映射，使u区的地址映射到另一个物理地址上。两个不同的物理地址代表两个不同的进程的u区，但是内核用相同的虚地址存取它们。

一个进程在核心态运行时能够访问它的u区，但在用户态时却不能。由于内核用u区的虚地址，一次只能存取一个u区，因此，u区在某种程度上决定了在系统中运行的进程的上下文。当内核调度一个进程运行时，它在物理存储器中找到相应的u区，并使u区的虚地址空间成为可被存取的。

以图6-7为例。图中假定u区的大小为4K，在内核的虚地址的2M。这里，头两个寄存器三元组是关于内核正文和数据的（这里没有示出地址和指针），第三个寄存器三元组是关

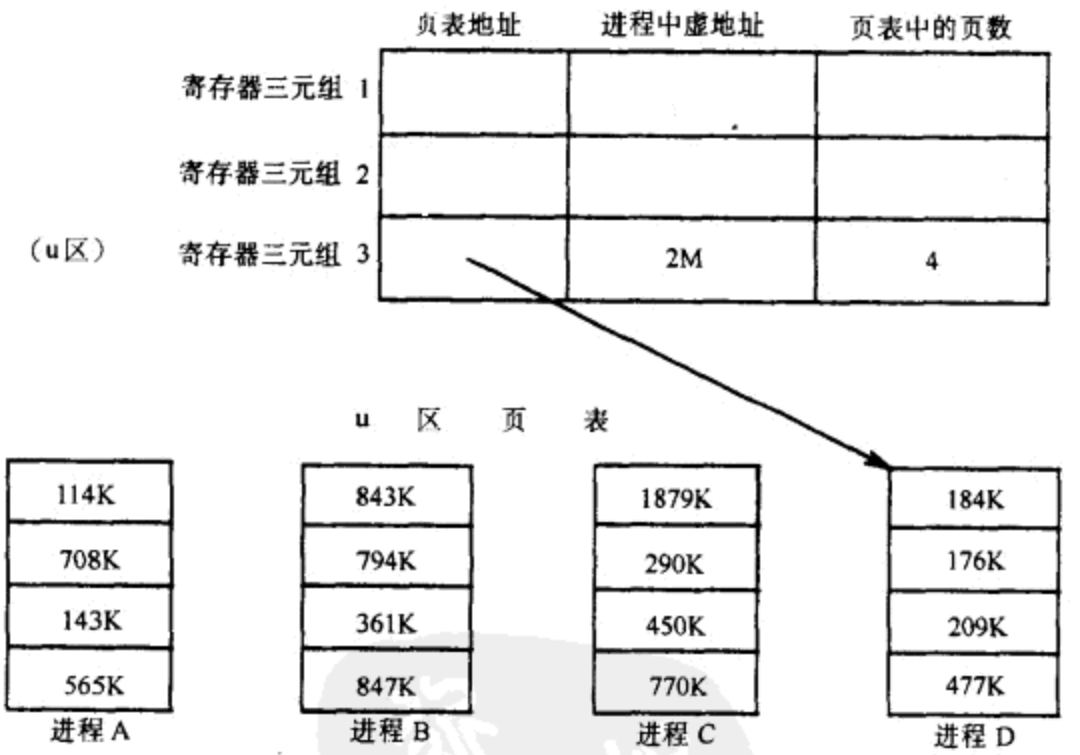


图6-7 u区在内核中的存储映射

于进程D的u区。如果内核要访问进程A的u区，它就为A的u区拷贝适当的页表信息，放到第三个寄存器三元组中。在任何时刻，第三个内核寄存器三元组总是指向目前正在运行的进程的u区。内核只要通过用一个新地址重写u区页表的地址项，就能引用另一个进程的u区。对内核来讲，寄存器三元组1和三元组2的表项是不变的，因为内核的正文和数据是由所有进程所共享的。

6.3 进程的上下文

进程的上下文是由其用户地址空间的内容、硬件寄存器的内容以及与该进程有关的内核数据结构组成。更严格地说，进程的上下文是由它的用户级上下文（user-level-context）、寄存器上下文（register context）以及系统级上下文（system-level context）组成[⊖]。

用户级上下文是由进程的正文、数据、用户栈和共享存储区组成。它们占据了该进程的虚地址空间。由于对换操作或调页，进程虚地址空间的某些部分不总是驻留在内存中，但它们仍然是用户级上下文的组成部分。

寄存器上下文由下列成分组成：

• 程序计数器（program counter），指出CPU将要执行的下条指令的地址；该地址是内核中或用户存储空间中的虚地址。

• 处理机状态寄存器（processor status register，简称PS），给出机器与该进程相关联时的硬件状态。例如，一般情况下，PS中有一些子域，指出最近一次计算的结果产生了0、正数或负数，或者某个寄存器溢出和进位位被置1等等。对一个特定的进程所作的操作导致了PS的设置，因此，PS中就含有机器与该进程关联时的硬件状态。特别是，在PS中可以找到另外一些重要子域，它们指出处理机的当前运行级（为中断而设）和当前以及最近的运行方式（如用户态或核心态）。指出运行方式的子域，决定着一个进程能否执行特权指令，能否访问内核地址空间。

• 栈指针，含有栈中下一项的当前地址。该地址是在核心栈中，还是在用户栈中，由运行方式来决定。机器的结构规定了栈指针指向的是栈中下一个自由项，还是栈中最后使用的项。类似地，机器的结构还决定了栈的增长方向，即根据数字，是向高地址方向增长还是向低地址方向增长。这样的问题对目前的讨论是不重要的。

• 通用寄存器，其中的数据是进程在其运行期间产生的。为简化下面的讨论，我们识别两个通用寄存器，寄存器0和寄存器1。在进程和内核之间传送信息时要用到它们。

进程的系统级上下文由静态部分和动态部分组成。下面列出的组成部分中，前3项是静态的，后3项是动态的。进程在其整个生命周期中，只有一个静态部分，但可以有可变数目的动态部分。可以把系统级上下文的动态部分看作由上下文层（context layer）组成的栈。当各种事件发生时，系统在其上进行压入和弹出操作。系统级上下文由下列成分组成：

• 一个进程的进程表表项，它定义了该进程的状态，并含有内核总能取到的控制信息。6.1节已对此作了讨论。

• 一个进程的u区，其中含有进程的控制信息，这些信息只需要在该进程的上下文中被存取。一般的控制参数，如进程优先权等，被存放在进程表中，因为它们要在进程的外部被

⊖ 本节所用的术语：用户级上下文、寄存器上下文和系统级上下文是本书作者所用的术语。

存取。

• 本进程区表表项、区表及页表。它们定义了从虚地址到物理地址的映射，因而决定了进程的正文区、数据区、栈区和其他的区。如果若干进程共享一些公用区，那么就把它们也看作是每个进程的上下文的组成部分，因为每个进程独立地对这些区进行操作。一部分存储管理任务要指明进程虚地址空间的哪些部分没驻留在主存中。

• 核心栈（kernel stack）。当一个进程在核心态执行时，它含有内核过程的栈结构。尽管所有的进程执行相同的内核代码，但它们各有一份私有的核心栈拷贝。这个拷贝指出了它们对内核函数的特殊调用。例如，某个进程可能调用 creat，然后进入睡眠，等待内核分配一个新索引节点。另一个进程可能调用 read，然后进入睡眠，等待把数据从磁盘传送到内存。这两个进程都执行了内核函数，但它们的栈是分开的。栈中装有它们各自的函数调用序列。内核必须能恢复核心栈的内容及核心栈指针的位置，从而使一个进程能重新在核心态下运行。在系统的实现中，通常是把核心栈放在进程的 u 区。但从逻辑上讲，它是独立的，可以把它存放在独立分配的存储区。当进程在用户态下执行时，核心栈是空的。

• 进程的系统级上下文的动态部分由一些“层”组成。可以把这些层想象为一个后进先出的栈。每个系统级上下文层，含有恢复前一层所必要的信息，其中包括前一层的寄存器上下文。

在发生中断时，或一个进程发生系统调用时，或进程进行上下文切换时，内核就压入一个上下文层。当内核从处理中断中返回，或一个进程在完成其系统调用后返回用户态，或一个进程进行上下文切换时，内核就弹出一个上下文层（context layer）。因此，上下文的切换总会引起一层系统级上下文的压入或弹出：内核压入老的进程的上下文层，弹出新的进程的上下文层。进程表表项存放着当前上下文层所必要的信息。

图 6-8 给出了形成进程上下文的各个部分。图的左面，表示上下文的静态部分，它由用户级上下文和系统级上下文的静态部分所组成。用户级上下文包括进程的正文（指令）、数据、栈和共享存储区（如果有的话）。系统级上下文的静态部分包括进程表项，u 区以及本进程区表表项（是用户级上下文中的虚地址映射信息）。图的右面，是上下文的动态部分。它由几个栈结构组成。其中每个栈结构中有保存的前一层寄存器上下文以及当内核在该层执行时的核心栈。内核的上下文层 0 是虚设层，它表示用户级上下文；这里，栈是向用户地址空间方向增长，核心栈是空的。图中的箭头从系统级上下文层的静态部分，指向动态部分的顶层，这表示进程表项的逻辑信息能使内核恢复进程的当前上下文层。

一个进程在它的上下文中运行，更精确地说，是在它的当前上下文层中运行。机器所能支持的中断级的数目，限制了上下文层的数目。例如，如果一个机器对软件中断、终端、磁盘、其他的全部外设和时钟中断支持不同的中断级，那么它就能支持 5 种中断级。因此，一个进程至多可有 7 个上下文层：每个中断级一个，系统调用一个，用户级一个。即使中断按最“坏”的次序发生，这 7 个上下文层也足以保存所有的上下文层。这是因为，对给定级别的中断，如果内核正在处理该级别的中断或更高级别的中断，则该级别的中断是被屏蔽的（即 CPU 令其延迟）。

尽管内核总是在某个进程的上下文中执行，但它执行的逻辑功能不必属于那个进程。例如，如果一个磁盘机由于送回数据而中断机器，也就是中断了正在运行的进程，内核则在一个新的系统级上下文层中执行中断处理程序，即使送回的数据属于另外一个进程。中断处理

程序一般不访问或修改进程上下文的静态部分，因为静态部分与中断无关。

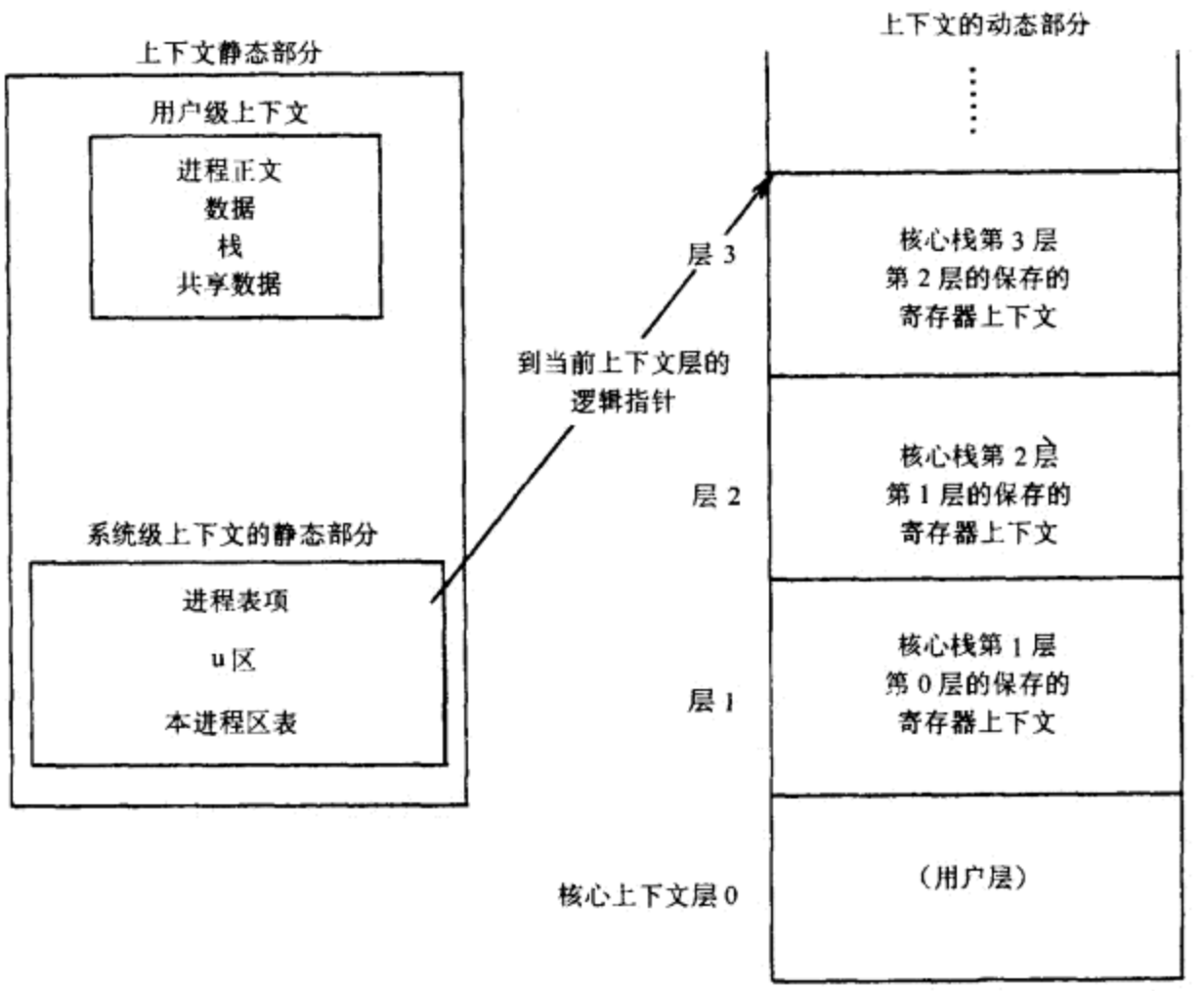


图 6-8　进程上下文的组成

6.4　进程上下文的保存

如前所述，每当内核压入一个新的系统上下文层时，它就要保存一个进程的上下文。特别是当系统收到一个中断，或一个进程执行系统调用，或当内核做上下文切换时，就要对进程的上下文进行保存。本节将详细讨论每种情况。

6.4.1　中断和例外

无论是硬件中断（如来自时钟和外设）、可编程中断（programmed interrupt）（执行引起“软件中断”（software interrupt）的指令），还是例外中断（如页面错），都由系统负责处理。当发生一个中断时，如果 CPU 正在比该中断级低的处理机运行级上运行，它就在解码下条指令之前，接受该中断，并提高处理机执行级。这样，在它处理当前的中断时，就不会响应该级别或更低级别的中断，从而维护了内核数据结构的完整性（见 2.2.2 节）。内核处理中断的操作顺序是：

（1）对正在执行的进程，保存其当前寄存器上下文并创建（压入）一个新上下文层。

（2）确定中断源，识别中断类型（如时钟或磁盘）。若可以的话，还识别中断的单元号（如哪个磁盘机引起中断）。当系统接受一个中断时，它从机器中得到一个数，系统把这个数用作查表的偏移量。这个表通常被称为中断向量（interrupt vector）。中断向量的内容因机器

而异。但一般都含有每种中断源的中断处理程序的地址以及中断处理程序取得参数的方式。图 6-9 给出了一个中断处理程序表。如果一个终端中断了系统，内核就由硬件得到一个中断号 2，并调用中断处理程序 ttyintr。

中断号	中断处理程序
0	clockintr
1	diskintr
2	ttyintr
3	devintr
4	softintr
5	otherintr

图 6-9 中断向量

(3) 内核调用中断处理程序。从逻辑上讲，新上下文层的核心栈不同于前一上下文层的核心栈。在实现方法上，有些是用正在运行的进程的核心栈存放中断处理程序的栈结构，另一些则是使用全局中断栈存放中断处理程序的栈结构，后者能保证中断处理程序不用进行上下文切换就能返回。

(4) 中断处理程序工作完毕前返回。内核执行一系列特殊机器指令。这些指令恢复前一上下文层的寄存器上下文和核心栈，使它们和中断发生时的情况一样，并恢复该上下文层的运行。相应的进程的工作情况可能会受到中断处理程序的影响，因为中断处理程序可能已修改过内核的全局数据结构，并唤醒过睡眠的进程。但在一般情况下，进程会像从来未发生过中断一样继续运行。

图 6-10 总结了内核是如何处理中断的。有些机器并不全用软件来完成这些操作，而是用硬件或微码完成该操作序列的某些部分，这样会获得更好的性能。但这要根据有多少上下文层要保存以及硬件指令完成保存工作的速度来权衡。因此，在一个 UNIX 系统的实现中，所要求的专门的操作是与机器有关的。

```
算法 inthand  /* 处理中断 */
输入：无
输出：无
{
    保存（压入）当前上下文层；
    确定中断源；
    查找中断向量；
    调中断处理程序；
    恢复（弹出）前一上下文层；
}
```

图 6-10 处理中断的算法

图 6-11 给出了一个例子。一个进程发出一个系统调用，在执行该系统调用期间，它收到一个磁盘中断。在执行磁盘中断处理程序时，系统收到一个时钟中断，并执行时钟中断处

理程序。每当系统收到一个中断（或作一个系统调用）时，它创建一个新的上下文层并保存前一层的寄存器上下文。

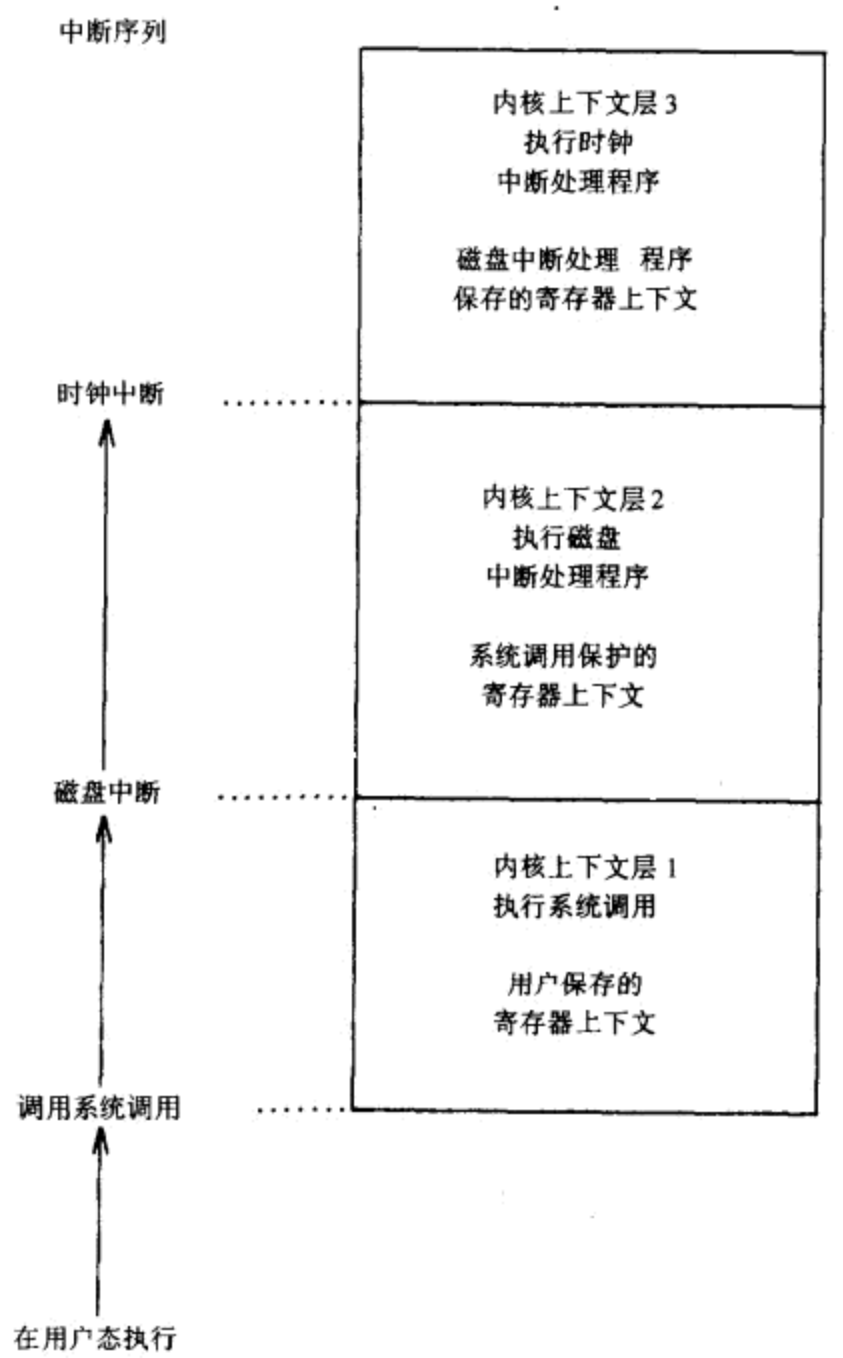


图 6-11　中断的例子

6.4.2　系统调用的接口

前面的章节中曾讨论过系统调用与内核的接口。系统调用好像是个一般的函数调用。显然，一般的函数调用序列不能引起从用户态到核心态的状态变化。C 编译程序采用一个预定义的函数库（C 子程序库），其中的函数具有系统调用的名字，从而解决了在用户程序中请求系统调用的问题。否则，这些系统调用的名字会是无定义的。这些库函数一般都执行一条指令，该指令将进程执行方式变为核心态，然后，使内核开始为系统调用执行代码。在以后的讨论中，我们称这个指令为操作系统陷入（operating system trap）。库函数是在用户态下执行，但是，简单地讲，系统调用的接口是一个中断处理程序的特例。库函数以与机器有关的方式传送给内核一个唯一的号（每种系统调用一个），该号或者作为给操作系统陷入的参数，或者存放在特殊的寄存器中，或者在栈中。这样，内核就可以确定用户所请求的系统调用了。

在处理操作系统陷入时，内核根据系统调用号查表，找出相应的内核子程序的地址。该地址是系统调用的入口点。内核还要确定该系统调用所要求的参数个数（见图 6-12）。然后，内核计算第一个参数的（用户）地址，即对用户栈指针加（或减，取决于栈的增长方向）一个偏移量，该偏移量对应于该系统调用所要求的参数数目。最后，内核将用户参数拷贝到 u 区，并调用相应的系统调用子程序。在执行完该系统调用之后，内核要确定是否出错。如果有错，它就修改所保存的用户寄存器上下文中的寄存器单元，最典型的操作是将 PS 寄存器的进位位置 1，并将错误号拷贝到寄存器 0 的存储单元中。如果没出错，内核将 PS 寄存器的进位位置 0，并将相应的系统调用返回值拷贝到寄存器 0 和 1 的单元。当内核从操作系统陷入返回到用户态时，它是返回到陷入指令之后的库函数指令。库函数解释内核返回的值，并给用户程序返回一个值。

```
算法 syscall  /* 系统调用的算法 */
输入：系统调用号
输出：系统调用的结果
{
    在系统调用表中找出对应于系统调用号的表项；
    确定系统调用的参数数目；
    将参数从用户地址空间拷贝到 u 区；
    为废弃返回而保存当前上下文(见第 6.4.4 节)；
    调用内核中的系统调用代码；
    if(在系统调用的执行中有错)
    {
        将用户保存的寄存器上下文中的寄存器 0 设置为错误号；
        将用户保存的寄存器上下文中的 PS 寄存器的进位位打开；
    }
    else
    将在用户保存的寄存器上下文中的寄存器 0,1 设置为系统调用返回值；
}
```

图 6-12 系统调用接口的算法

例如，在图 6-13 的上部，一个程序以所有用户可读、可写方式（0666）创建一个文件。图中的另一部分给出该程序在 Motorala 68000 系统上进行编译和反汇编以后所产生的输出的一部分。对这部分我们已做了编辑工作。

图 6-14 描述了在系统调用期间栈的结构。编译程序产生的代码将两个参数压入用户栈，压入的第一个参数是许可权方式设置——0666，第二个参数是变量 name[⊖]。然后，从地址 64 处，进程调用为系统调用 creat 设计的库函数（地址为 7a）。这个库函数的返回地址是 6a，进程也将这个数压入栈。为 creat 设计的库函数将常数 8 移到寄存器 0，然后执行 trap 指令。该指令引起进程从用户态转到核心态并处理系统调用。这时，内核知道用户正在作系统调用，并从寄存器 0 中取出 8 这个数，从而知道当前的系统调用是 creat。通过查内部表，内

⊖ 编译程序计算和压入函数参数的次序与实现方式有关。

```
char name[ ]="file";
main( )
{
        int fd;
        fd=creat(name,0666);
}
```

Portions of Generated Motorola 68000 Assembler Code

```
Addr   Instruction
        .
        .
#code for main
        .
58:     mov     &0x1b6,(%sp)        #将 0666 移到栈上
5e:     mov     &0x204,-(%sp)       #移栈指针并把变量 name
                                    #移到栈上
64:     jsr     0x7a                #调用 creat 的 C 库程序
        .
        .
#library code for creat
7a:     movq    &0x8,%d0            #将数据值 8 移到数据寄存器 0
7c:     trap    &0x0                #操作系统陷入
7e:     bcc     &0x6<86>            #如果进位位为零,则转到地址 86
80:     jmp     0x13c               #转到地址 13c
86:     rts                         #从子程序返回
        .
        .
        .
# library code for errors in system call
13c:    mov     %d0,&0x20e          #将数据寄存器 0 移到 20e 单元(错误号)
142:    movq    &-0x1,%d0           #将常量-1 移到数据寄存器 0
144:    mova    %d0,%a0
146:    rts                         #从子程序返回
```

图 6-13　系统调用 creat 和为 Motorola 68000 产生的代码

核得知 creat 有两个参数；通过恢复前一个上下文层的寄存器，它将两个参数从用户空间拷贝到 u 区。需要这两个参数的内核子程序可以在 u 区中指定的存储单元中找到它们。当内核执行完 creat 的代码后，便返回系统调用处理程序，该程序检查 u 区的出错域是否被设置。若已设置，则意味着在系统调用的过程中有错。如果有错，则处理程序将 PS 寄存器中的进位位置 1，将错误代码放到寄存器 0 中，然后返回。如果没错，内核将系统返回码放到寄存器 0 和 1 中。当系统调用程序返回到用户态时，C 库函数在地址 7e 处检查 PS 寄存器的进位位。如果进位位已置 1，进程则跳转到地址 13c，从寄存器 0 中取出错误代码并把它存到地址为 20e 的全局变量 errno 中。然后，进程将寄存器 0 的内容置为 -1，返回到发出系统调用（地址 64）后的下条指令。库函数的返回码为 -1，说明在系统调用中出错。当从核心态返回到用户态时，如果 PS 寄存器的进位位为 0，则进程从地址 7e 转到地址 86，并返回到调用者（地址 64），这时，寄存器 0 中存放着系统调用的返回值。

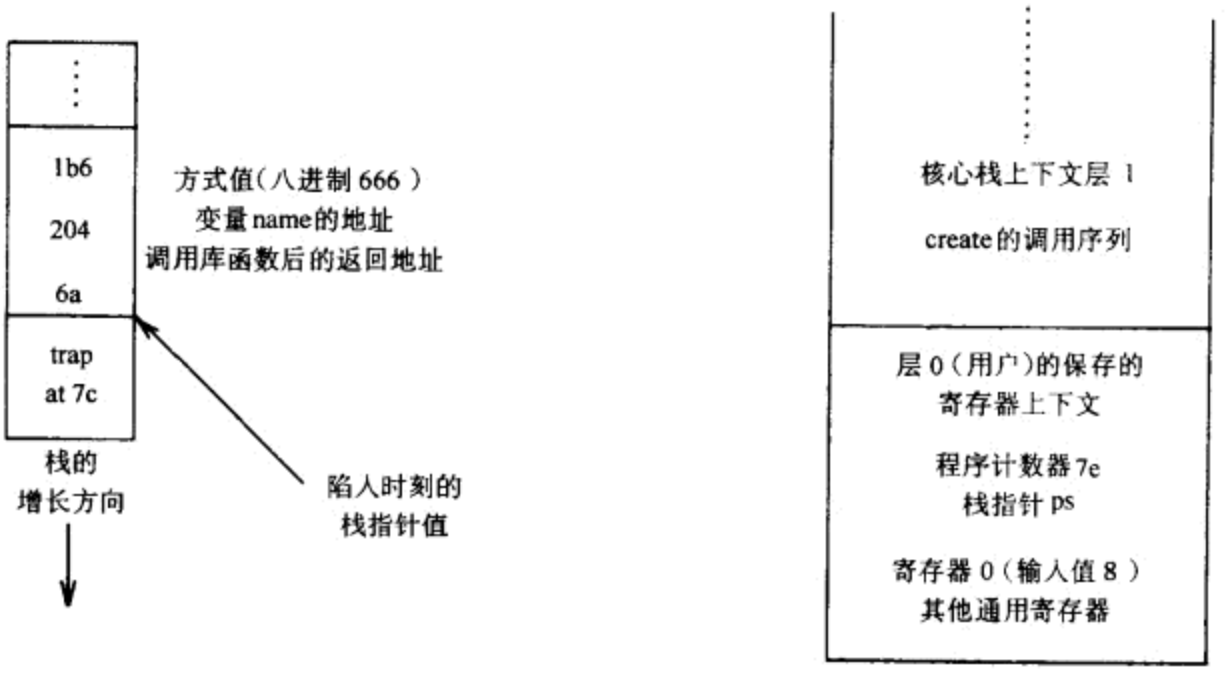


图 6-14 系统调用 creat 的栈结构

若干个库函数可以映射到一个系统调用入口点。系统调用入口点对每个系统调用定义其真正的语法和语义，但库函数通常则提供一个更方便的接口。例如，系统调用 exec 有几种不同的调用方式，如 execl 和 execle，它们为同一系统调用提供不同的接口。对这些调用，它们的库函数对它们各自的参数加以处理，来实现各自的特点，但最终地，这些库函数都映射到一个内核入口点。

6.4.3 上下文切换

从图 6-1 的进程状态图中可以看出，内核在四种情况下允许发生上下文切换：当进程使自己进入睡眠时；当它从一个系统调用返回用户态但不是最有资格运行的进程时；当它在内核完成中断处理后返回用户态，但不是最有资格运行的进程时；当它退出（exit）时。第 2 章中曾经指出过，内核是通过禁止任意的上下文切换来保证内部数据结构的完整性和一致性的。在内核做上下文切换之前，它要确保其数据结构的状态是一致的。也就是说，所有该做的更新都已完成；各种队列都已正确地链接；已设置了合适的锁以防止其他进程的打扰；应解锁的数据结构已被解锁等等。例如，如果内核分配一个缓冲区，从一个文件读一个块，然后睡眠以等待来自磁盘的 I/O 传输的完成，那么它就要保持它的缓冲区上了锁，这样，其他任何进程都不能破坏该缓冲区。但是，如果一个进程执行系统调用 link，内核就要在给第二个索引节点上锁之前为第一个索引节点解锁，从而避免死锁。

内核在系统调用 exit 结束时，必须做上下文切换，如若不然，它将无事可做。类似地，当一个进程进入睡眠状态时，内核也要做上下文切换，因为可能要经过相当长的一段时间后，该进程才会醒来，而在这段时间里，可以运行其他程序。为了保证公平的进程调度，当一个进程不是最有资格运行时，内核允许做上下文切换：如果一个进程完成了一个系统调用或从中断中返回，同时又有另外一个具有更高优先权的进程等待运行，那么，让具有更高优先权的进程继续等待是不公平的。

做上下文切换的过程与处理中断和处理系统调用的过程类似。不同的是，内核恢复的是一个不同进程的上下文层，而不是同一进程的前一个上下文层。这与做本次上下文切换的原

因无关。类似地，选择下次调度哪个进程是一种决策，所采用的策略并不影响做上下文切换所采用的方法。见图 6-15。

(1) 决定是否做上下文切换以及是否允许做上下文切换。
(2) 保存“老”进程的上下文。
(3) 用第 8 章中的进程调度算法，找“最好”的进程调度执行。
(4) 恢复它的上下文。

图 6-15　上下文切换的步骤

通常，UNIX 中实现上下文切换的代码是操作系统中最难理解的地方，因为在某些情况下，函数调用表现为没有返回，在另外一些情况下，表现为不知从哪儿实现的。产生这种情况的原因是，在许多实现中，内核在其代码中的某一点保存进程上下文，但又在这个“老”进程的上下文中继续执行上下文切换和调度算法。当它后来恢复这个进程的上下文时，它是根据前一次保存的上下文来恢复执行。为了区别内核重新开始一个新进程的上下文和内核在保存一个进程的上下文后，继续在老的上下文中执行这两种情况，一些临界函数的返回值可能变化，或者核心运行点的程序计数器会被人为地设置。

```
if(save _ context())     /* 保存运行进程的上下文 */
{
    /* 取另一个进程运行 */
    .
    .
    .
    resume _ context(new _ process);
    /* 不会达到这里！ */
}
/* 恢复的进程从这里开始执行 */
```

图 6-16　上下文切换的伪码算法

图 6-16 给出做上下文切换的伪码算法。函数 save _ context 保存关于正在运行的进程的上下文信息，并返回值 1。在内核保存的其他一些信息当中，有当前程序计数器的值（保留在函数 save _ context 中）和值 0，值 0 用作以后从 save _ context 返回时，寄存器 0 中的返回值。内核继续在老进程（进程 A）的上下文中执行，选取另外一个进程（进程 B）运行，并调用 resume _ context 来恢复 B 的上下文。在新的（B）上下文恢复后，系统开始执行 B，不再执行老进程 A，只是把 A 的已保存的上下文放在一边。（注释中以“不会达到这里”来表示。）过了一会儿，如刚刚所描述的一样，另一个进程做上下文切换，这时，内核将会再次选取进程 A 继续运行（当然，exit 的情况除外）。当进程 A 的上下文被恢复时，内核将把程序计数器的值设置为进程 A 先前曾保留在函数 save _ context 中的值，同时，内核还将保存

的、用作返回值的值0放到寄存器0中。内核在save_context的内部恢复执行A，即使实际上进程A在它的上下文切换之前已经执行到调用resume_context。最后，进程A以值0（在寄存器0中）从函数save_context中返回，从注释语句“恢复的进程从这里开始执行”之后开始执行。

6.4.4 为废弃返回（abortive return）而保存上下文

这种情况发生在内核必须中止它当前的执行顺序，立即从先前保存的上下文执行时。后面介绍sleep和signal的各节中，将描述若干进程必须突然改变其上下文的情况。这一节我们将说明执行一个以前的上下文的方法。保存一个上下文的算法叫setjmp，恢复这个上下文的算法叫做longjmp[⊖]，所采用的方法和前一节所描述的save_context的方法类似。不同的是，save_context压入一个新的上下文层，而setjmp将保存的上下文存入u区。并继续在老的上下文层中执行。当内核希望恢复它在setjmp中曾保存的上下文时，它做longjmp，从u区中恢复上下文，并从setjmp中返回值1。

6.4.5 在系统和用户地址空间之间拷贝数据

到目前为止，我们所说的进程或者在核心态运行，或者在用户态运行，没有两种状态的重叠。然而，前一章所讨论的许多系统调用，要在内核和用户空间之间传送数据。例如，将系统调用参数从用户空间拷贝到内核空间，或者，在系统调用read中，将I/O缓冲区的数据拷贝到用户空间。许多机器还允许内核直接地访问用户空间的地址。内核必须知道正在读或写的用户空间的地址是可存取的，就好像它一直在用户态下运行；否则，它可能破坏普通的写保护机制，无意地读、写用户地址空间（address space）之外的地址（有可能是内核数据结构）。因此，在内核空间和用户空间之间拷贝数据是一件开销很大的工作，不仅仅只需要一条指令。

以图6-17中的一段VAX上的代码为例，这段代码把一个字符从用户地址空间移到内核地址空间。其中，指令prober检查在用户态下（方式3），在参数指针寄存器（argument pointer register）加4(*4(ap))的地址中的一个字节是否可读。如果不能读，内核转到eret，在寄存器0中存入-1后返回；如果能读，内核则把该用户地址中的一个字节传送到寄存器0中，并将该值返回给调用者。这个过程开销很高，为了传送一个字符就需要5条指令（以及对函数fubyte的调用）。

```
fubyte:                                    #从用户空间移字节
        prober   $3,$1,*4(ap)              #字节可存取吗?
        beql     eret                      #否
        movzbl   *4(ap),r0
eret:
        mnegl    $1,r0                     #错误返回(-1)
        ret
```

图6-17 在VAX上从用户空间到系统空间的数据传送

⊖ 不要将这两个算法和有同样名字的库函数相混淆。后者，用户可以直接从他们的程序中调用（见[SVID 85]）。然而，它们的功能是类似的。

6.5 进程地址空间的管理

到目前为止，本章已经讨论了内核如何在进程之间切换上下文，如何弹出和压入上下文层，将用户级上下文看作在其进程上下文的恢复期间不变的静态对象。然而，在下章中我们将会看到，各种系统调用管理一个进程的虚地址空间，根据对区的明确定义的操作来完成这些工作。本节只定义区数据结构及区上的操作。下章再讨论使用这些区操作的系统调用。

区表表项中含有描述一个区的必要信息，特别要指出的是下列字段：

- 指向文件的索引节点的指针。该文件的内容是一开始被装入该区的数据。
- 区的类型（正文，共享存储区，私有数据或栈）。
- 区的大小。
- 区的物理存储器中的位置。
- 区的状态，可以是如下状态的组合。
  - ——锁住。
  - ——在请求中。
  - ——正在被装入内存的过程中。
  - ——有效，已被装入内存。
- 引用数，给出引用该区的进程数。

管理区的操作有：锁区、解锁区、分配区；将区附接（attach）到某个进程的存储空间；释放一个区；使一个区和某个进程的存储空间断接（detach）；复制区的内容。例如，系统调用 exec（其功能是用一个可执行文件的内容，覆盖用户的地址空间）所做的操作是：断接一些老区，如果它们不是共享区，则释放这些区；分配一些新区，使它们与进程附接起来，并将文件的内容装入这些区。本节的其余部分将更详细地给出区的操作。这里，假定存储管理模型是已描述过的页表和硬件寄存器三元组，并假定已知分配页表和分配物理存储器的页面的算法（见第 9 章）。

6.5.1 区的上锁和解锁

内核对区的上锁和解锁操作与分配和释放一个区的操作无关，就像文件系统中有对索引节点的上锁－解锁、分配－释放操作一样（算法 iget 和 iput）。这样，内核能上锁和分配一个区，而后又解锁该区但不释放它。类似地，如果它想使用一个已分配的区，它能锁住该区以防止其他进程的存取，而后再解锁该区。

6.5.2 区的分配

在系统调用 fork，exec 和 shmget（共享存储区）期间，内核将分配一个新区（算法 allocreg，见图 6-18）。内核中有一个区表，其表项或在一个自由链表中，或在一个活动链表中。当内核要分配一个区表项时，它从自由链表中取出第一个可用表项，并把它放到活动表中，为该区上锁并标识它的类型（共享的或私有的）。因为以前执行的 exec 调用，几乎每个进程（很少有例外）都与一个可执行文件有关，allocreg 设置该区表项中的索引节点字段，使其指向该可执行文件。对内核来说，索引节点标识了该区，这样，如果需要的话，其他进程可以共享该区。为了防止其他进程在解除与该索引节点的联结时（unlink）清除该区的内

容，内核要对索引节点的引用计数加 1。这一点将在 7.5 节中说明。allocreg 返回一个上锁的、已分配的区。

```
算法 allocreg /* 分配一个区数据结构 */
输入：(1)索引节点指针
      (2)区类型
{
    从自由链表上取一个区；
    赋值区类型；
    赋值区索引节点指针；
    if(索引节点指针非空)
        索引节点引用数加 1
    将区放到活动链表上；
    return(上锁的区)；
}
```

图 6-18 分配区的算法

6.5.3 区附接到进程

在系统调用 fork，exec 和 shmat 期间，内核要使一个区和一个进程的地址空间联系起来，称为附接（attach）。图 6-19 给出附接区的算法 attachreg。附接的区可以是一个新分配的区，也可以是该进程要与其他进程共享的、已存在的区。内核分配一个空闲的 pregion 表

```
算法 attachreg /* 将区附接到进程 */
输入：(1)指向所附接的(上锁的)区指针
      (2)区将被附接到的进程
      (3)要附接的区在进程中的虚地址
      (4)区类型
输出：本进程区表表项
{
    为进程分配本进程区表表项；
    初始化本进程区表表项：
        设置指向所附接的区的指针；
        设置区类型；
        设置虚地址字段；
    检查虚地址及区大小的合法性；
    区引用数加 1；
    根据所附接的区的大小，增加进程的大小；
    为进程初始化新的硬件寄存器三元组；
    return(本进程区表表项)；
}
```

图 6-19 算法 attachreg

项，将其类型字段设置为正文、数据、共享存储区或栈，并记下该区在该进程地址空间中的虚地址。进程不能超出系统强加的最高虚地址限制，新区的虚地址也不能与现存的区的地址重叠。例如，如果系统限制一个进程的最高地址空间是8兆字节，那么，将1兆字节大小的区附接到虚地址7.5兆处是非法的。如果将一个区附接到某进程是合法的，内核就根据该区的大小修改该进程表项中表示区的大小的字段，使它增加一个值，并给该区的引用数加1。

然后，算法attachreg为该进程设置一组新的存储管理寄存器三元组：如果该区还没被附接到另一个进程，内核则接着调用算法growreg（见下节），为该区分配页表；否则，就使用它的现存页表。最后，算法attachreg返回一个指针，该指针指向这一新附接的区的pregion表项。例如，假定内核想把一个大小为7K字节的、已经存在的共享正文区联系到一个进程的虚地址0上（图6-20）：内核将分配一个新的存储管理寄存器三元组，用该区的区页表地址、该进程的虚地址（0）和页表的大小（9页）来初始化寄存器三元组。

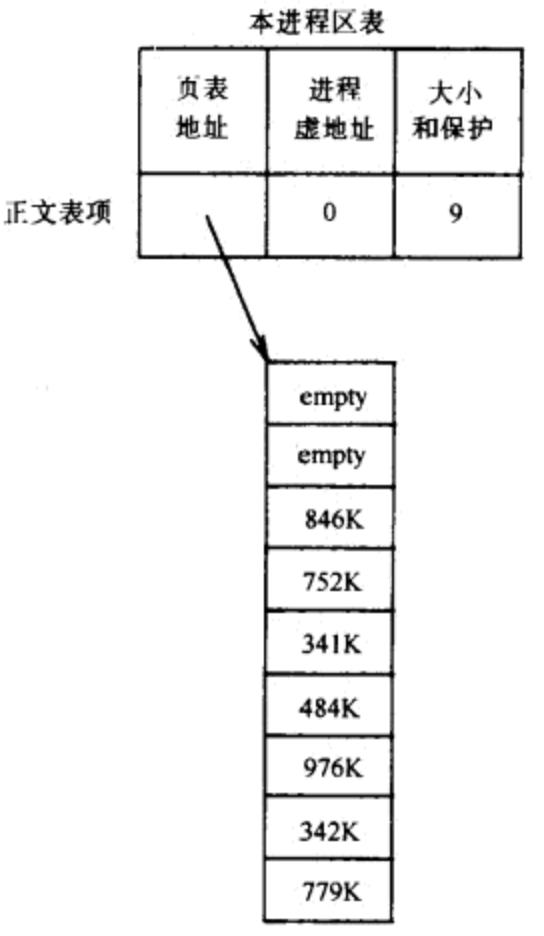


图6-20　将已存在的正文区附接到一个进程

6.5.4　区大小的改变

进程可以用系统调用sbrk来扩展或压缩其虚地址空间。类似地，根据过程调用的嵌套深度，进程的栈可以自动地扩展（即，进程不用做显示的系统调用）。内核在内部调用算法growreg来改变区的大小（图6-21）。当扩展区时，内核要保证扩展的区的虚地址不与另一个区的虚地址重叠，并且，区的增大不应引起进程的大小超过所允许的最大虚存空间。内核决不用算法growreg去增加已附接到若干进程的共享区的大小；因此，内核不用担心为某个进程增加区的大小会引起另一个进程超出系统规定的进程大小限制。内核将growreg用于已存在的区有两种情况：一种是用户栈的自动增长；一种是系统调用sbrk，它用于一个进程的数据区。这两种区都是私有区。正文区和共享存储区在初始化以后不能再被扩展。在第7章中，会更明显地看到这些情形。

内核分配页表（或扩展现存的页表）来适应更大的区。在不支持请求调页的系统上，内核要分配物理存储空间。当分配物理存储空间时，内核在调用growreg之前，要确定这部分存储空间是可用的。如果存储空间不可用，它要求助于其他方法增加区的大小（见第9章）。如果进程要压缩区，内核只要简单地释放分配给该区的存储空间即可。在这两种情况下，内核都要调整该进程和区的大小，并重新设置该进程的本进程区表表项和存储管理寄存器三元组，从而适应新的映射。

例如，假定一个进程的栈区是从虚地址128K开始，目前的大小是6K字节，内核要将该区扩展1K字节（1页）。如果该进程的大小是可以接受的，而且虚地址134K到135K－1不属于另一个附接到该进程的区，内核就扩展该栈区的大小。内核扩展页表，分配1页存储空间并初始化新的页表表项。图6-22给出了这个例子。

```
算法  growreg  /* 修改区的大小 */
输入:(1)指向本进程区表表项的指针
     (2)区大小的变化(或正或负)
输出:无
{
    if(增加区的大小)
    {
        检查新区大小的合法性;
        分配辅助表(页表);
        if(是不支持请求调页的系统)
        {
          分配物理存储空间;
          按需要初始化辅助表;
        }
    }
    else  /* 减少区的大小 */
    {
        释放物理存储空间;
        释放辅助表;
    }
    按需要作辅助表的其他初始化;
    在进程表中设置大小字段;
}
```

图 6-21 改变区大小的算法 growreg

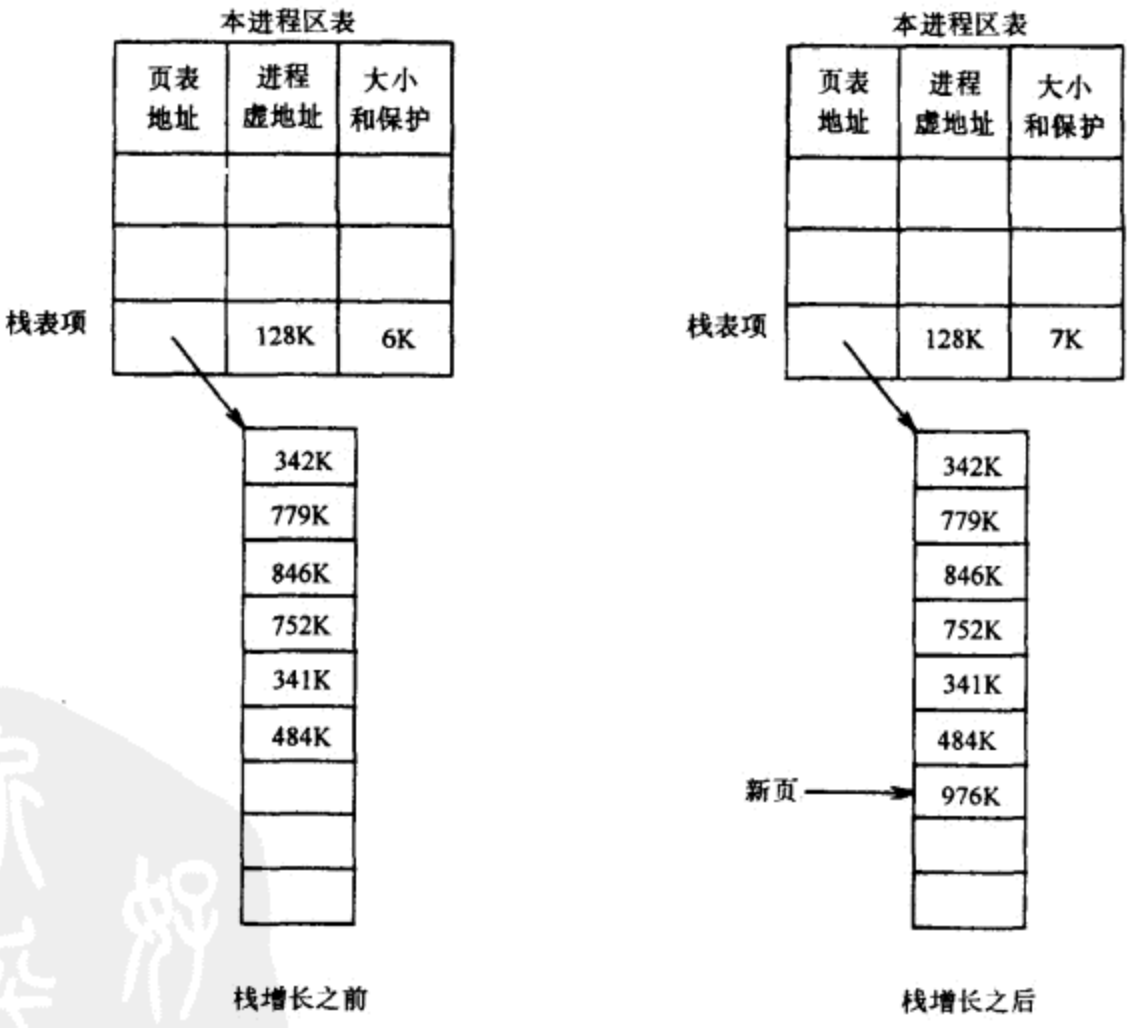


图 6-22 栈区增长 1K 字节

6.5.5　区的装入

在支持请求调页的系统中，内核能够在系统调用 exec 期间将一个文件“映射”到进程地址空间，其后按要求安排读入每个物理页，这一点将在第 9 章中说明。如果内核不支持请求调页，它必须将可执行文件拷贝到存储器中，将进程的区装到在可执行文件中指定的虚地址上。内核可以将区附接到一个虚地址上，这个虚地址不同于文件装入的实际地址，这就造成了页表中的间隙（回忆图 6-20 中的“empty”）。当用户程序非法地存取地址 0 时，上述特性将引起存储器错。有时，带有指针变量的程序对值为零的指针变量不做检查，错误地把它们当作指针来引用，这是非法的。适当地保护含地址 0 的页，可导致错误地存取地址 0 的那些进程出错，并终止其运行。这样，可以让程序员尽早发现这类的错误。

我们已经知道，一个区被附接到一个进程的虚地址上，在这个虚地址和该区数据的起始虚地址之间有时有一段间隙。为了将一个文件装入一个区，算法 loadreg（图 6-23）计算这个间隙的大小，并根据该区要求的存储量来扩展该区。然后，内核置该区的状态为“正被装入内存”，用系统调用 read 算法的内部形式将该区数据从文件读入内存。

```
算法　loadreg　/* 将文件的一部分装入一个区 */
输入：(1)指向本进程区表表项的指针
      (2)区要装入的虚地址
      (3)要装入该区的文件的索引节点
      (4)文件中的字节偏移量,用作区的起始
      (5)字节计数,表示要装入的数据量
输出：无
{
    根据区的最终大小增加区的尺寸(算法 growreg);
    设置区的状态为“正被装入内存”;
    解锁该区;
    设置读文件用的 u 区参数:
        数据要读入的目的虚地址,
        该文件的起始偏移量,
        从文件中要读的字节数;
    将文件读入该区(算法 read 的内部形式);
    上锁该区;
    设置区的状态为“有效,已在内存”;
    唤醒所有等待该区被装入的进程;
}
```

图 6-23　算法 loadreg

如果内核正在装入一个能被若干进程所共享的正文区，那么有可能在它的内容被完全装入之前，另外一个进程要查找并使用该区，这是因为头一个进程在读入该文件期间可能进入睡眠。为什么会发生这种情况以及为什么不能用锁？这一问题将在第 7 章和第 9 章的系统调用 exec 中详细讨论。为了避免发生问题，内核检查区的状态标志，确定该区是否已完全装入。如果该区尚未被装入，则对应的进程便进入睡眠。在算法 loadreg 结束之前，内核要唤

醒那些一直在等待该区被装入的进程，并将该区的状态改为有效和已在内存。

例如，假定内核要把大小为7K的正文装入一个区，该区被附接到某个进程的虚地址0上，而且，内核要在该区的开头留1K字节的间隙（图6-24）。至此，内核已使用算法allocreg和attachreg分配了一个区表表项，并将该区附接到地址0上。然后，它调用loadreg，该算法两次调用growreg，第一次计算该区开始的1K字节间隙，第二次为该区的内容分配存储空间。growreg还为该区分配了一个页表。下一步，为了读入文件，核收设置u区中的一些字段，然后，从该文件中指定的字节偏移量（由内核提供的参数）开始，读7K字节到该进程的虚地址1K处。

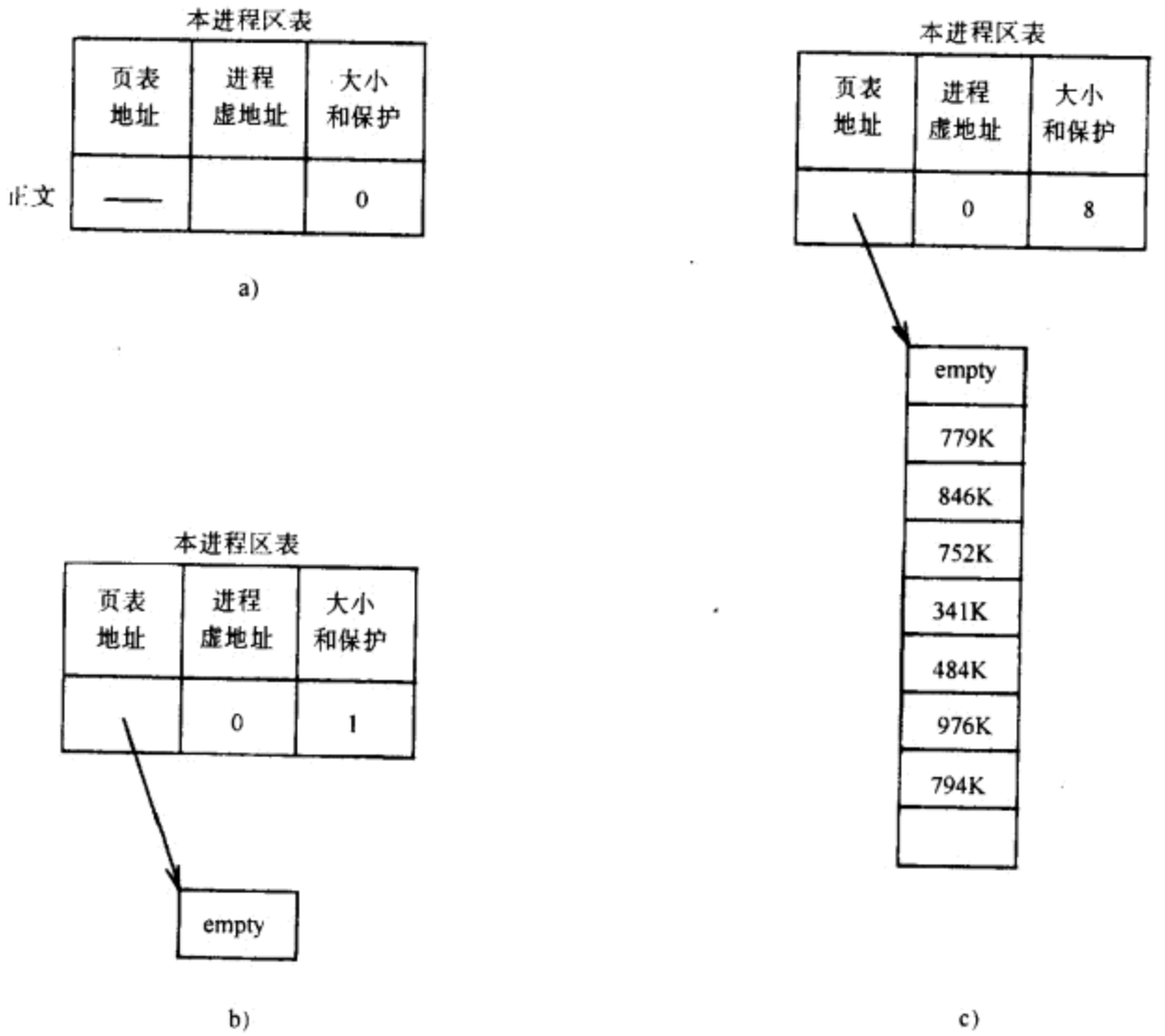


图6-24 装入一个正文区

a）最初的区表表项 b）第一次用growreg为1K间隙分配一个页表表项的页表以后 c）第二次growreg以后

6.5.6 区的释放

当一个区不再与任何进程相附接时，内核便释放该区并把它送回自由链表（图6-25）。如果该区和某个索引节点有关，内核就用算法iput释放该索引节点，这对应于算法allocreg中的索引节点引用数加1。内核释放与该区有关的物理资源，如页表和存储页。例如，假定内核要释放图6-22所示的栈区，假定该区的引用数是0，内核则释放7页的物理存储空间和该页表。

```
算法　freereg　/* 释放一个已分配的区 */
输入：指向(上锁的)区的指针
输出：无
{
    if (区的引用数非零)
    {
    /* 某些进程还在使用该区 */
        释放区锁;
        if (该区有相关的索引节点)
            释放索引节点锁;
        return;
    }
    if (该区有相关的索引节点)
      释放索引节点(算法 iput);
    释放仍与该区相联系的物理存储器;
    释放与该区相联系的辅助表;
    清除区的各字段;
    将区放回自由链表;
    解锁该区;
}
```

图 6-25　释放区的算法

6.5.7　区与进程的断接

在系统调用 exec，exit 和 shmdt（断接共享存储区）中，内核使有关的区与进程相断接。它更新本进程区表表项，使有关的存储管理寄存器三元组无效，从而切断区与物理存储的联系（图 6-26 算法 detachreg）。这样，地址翻译机制只是对某个特定的进程无效，而不是对该

```
算法　detachreg　/* 使区和进程断接 */
输入：指向本进程表表项的指针
输出：无
{
    取该进程的辅助存储管理表,合适的话便释放;
    减小进程的大小;
    减少区的引用数;
    if (区的引用数为零且驻留位未打开)
      释放区(算法 freereg);
    else　/* 或区的引用数不为零,或驻留位打开 */
    {
      释放索引节点锁(如果该区与索引节点有关);
      释放区锁;
    }
}
```

图 6-26　算法 detachreg

区无效（与算法 freereg 类似）。根据区的大小，内核减小进程表项的大小字段，并使该区的引用数减 1。如果该区的引用数降为零，而且没有任何理由要使该区保持不变（指该区不是带有驻留位（sticky bit）的共享存储区或正文区，见 7.5 节），那么，内核就用算法 freereg 释放该区，否则，内核释放该区的锁和索引节点锁，它们曾被锁住以防止竞争条件（见 7.5 节），但内核不收回该区和已分配的资源。

6.5.8 区的复制

系统调用 fork 要求内核复制一个进程的所有区。然而，如果某个区是共享的（共享正文或共享存储区），内核就不必物理地拷贝该区，而是增加该区的引用数，允许父进程和子进程共享该区。如果该区不是共享的，内核就必须物理地拷贝该区。它分配一个新的区表表项、页表以及该区的物理存储空间。图 6-27 中给出一个例子，其中进程 A 创建（fork）进程 B 并复制自身的区。进程 A 的正文区是共享的，因此，进程 B 可以和进程 A 共享它。但进程 A 的数据区和栈区是私有的，所以，进程 B 复制它们，将它们的内容拷贝到新分配的区。然而，即使是私有区，也不总是需要物理地拷贝，这一点将在第 9 章中说明。图 6-28 给出了 dupreg 的算法。

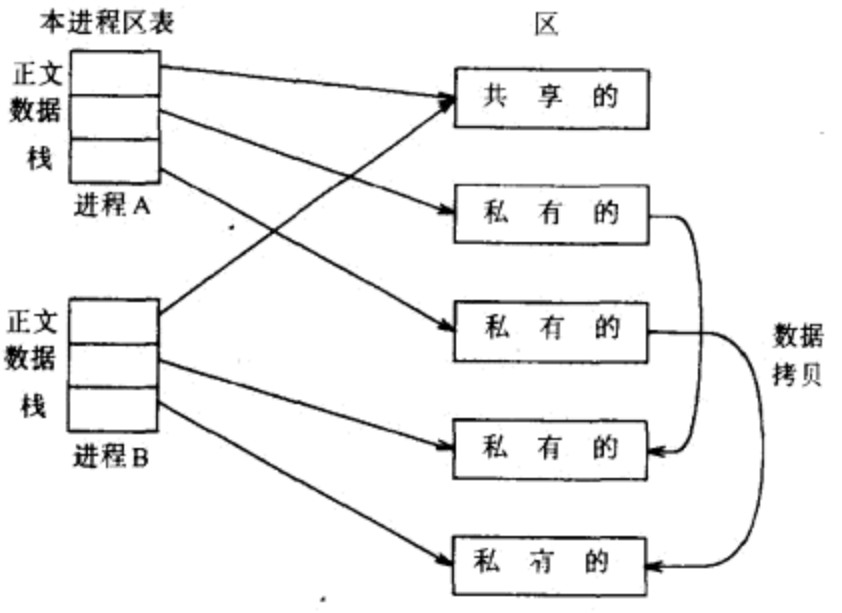


图 6-27 复制一个区

```
算法 dupreg /* 复制一个已存在的区 */
输入：指向区表表项的指针
输出：指向看上去和输入区相同的区的指针
{
    if (区类型为共享的)
        /* 调用者将用一个后继的 attachreg 调用
         * 来增加区的引用数
         */
        return(输入区的指针);
    分配一个新区(算法 allocreg);
    按当前在输入区现存的存储管理结构,设置辅助存储管理结构;
    为区的内容分配物理存储器;
    将区的内容从输入区"拷贝"到新分配的区;
    return(指向新分配的区的指针)
}
```

图 6-28 算法 dupreg

6.6　睡眠

到目前为止，本章已讨论了除使进程进入睡眠状态以外的所有低层次的功能，这些功能实现了进入“核心态运行”状态或脱离“核心态运行”状态的转换。本章将以介绍算法sleep和wakeup作为结束。sleep将进程的状态由“核心运行”变为“在内存中睡眠”，而wakeup将进程的状态由“睡眠”变为“在内存中就绪”或“就绪且换出”。

一般而言，进程是在执行一个系统调用期间进入睡眠的：该进程执行一个操作系统陷入，进入内核（上下文层1），然后进入睡眠等待某个资源。当该进程进入睡眠时，它做上下文切换，压入它的当前层，在内核上下文层2中运行（图6-29）。当进程由于存取尚未物理地装入的虚地址而导致页面错时，它们也进入睡眠；在内核读入这些页面的内容期间，进程睡眠等待。

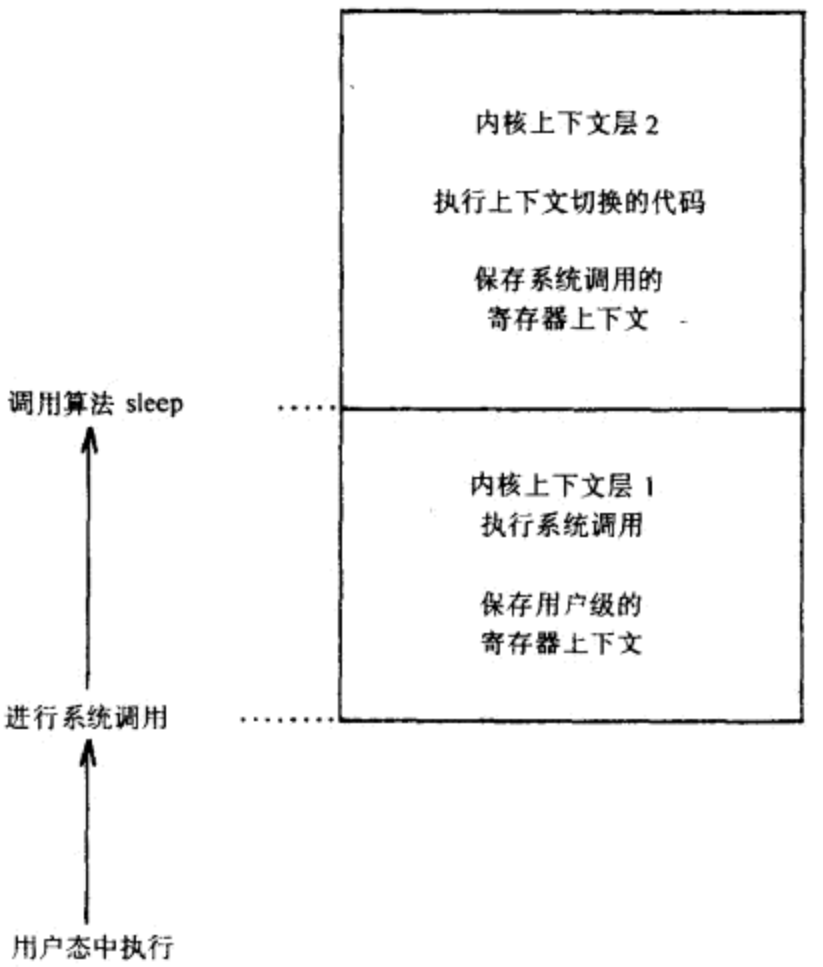


图6-29　睡眠进程的典型上下文

6.6.1　睡眠事件及其地址

在第2章中，我们说进程在一个事件上睡眠，意思是它们处于睡眠状态，直到该事件发生。当事件发生时，它们就醒来并进入“就绪”状态（在内存或换出）。尽管系统使用了“在一个事件上睡眠”这种抽象说法，实际上的实现方法是将一组事件映射到一组（内核）虚地址上。代表这些事件的地址被编码在内核中，它们唯一的意义是：内核希望将一个事件映射到一个特殊地址上。事件的抽象以及实现都不区别有多少进程在等待该事件。这样，就出现两种异常情况。首先，当一个事件发生，一个wakeup调用发向那些在该事件上等待的

进程时，这些进程都会醒来，从睡眠状态进入就绪状态。内核并不是一次唤醒一个进程，即使它们可能竞争同一个上锁的结构，因此它们中的许多进程可能在内核运行状态作暂短逗留之后就又回到睡眠状态。（请回忆第 2 章和第 3 章的讨论）。图 6-30 表示了几个睡眠在一些事件上的进程。

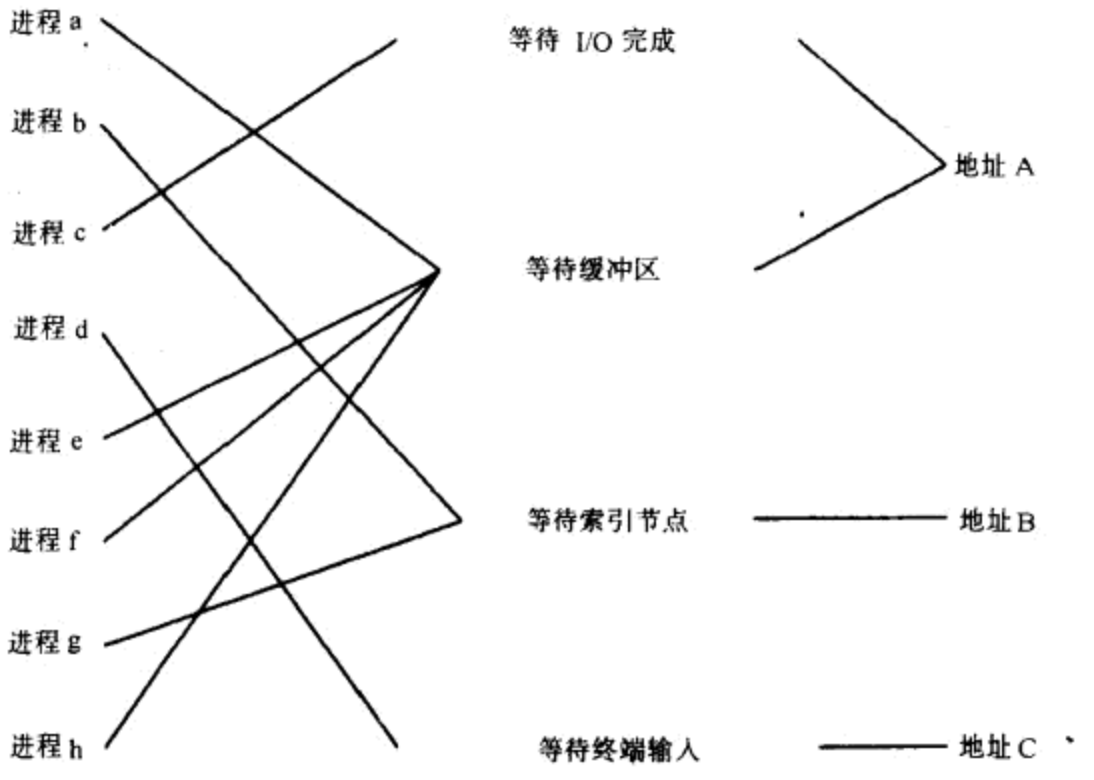


图 6-30 睡眠在事件上的进程及映射到地址上的事件

在实现中的第二个异常是，若干事件可能映射到一个地址上。例如，在图 6-30 中，事件“等待缓冲区可用”和事件“等待 I/O 完成”都映射到该缓冲区的地址上（“地址 A”）。当该缓冲区的 I/O 完成时，内核唤醒所有在这两个事件上睡眠的进程。由于某个等待 I/O 的进程使该缓冲区上锁，等待该缓冲区可用的其他进程，如果当它们执行时，该缓冲区还未解锁，就要回到睡眠状态。如果事件到地址的映射是一对一的，那么效率可能会更高一些。但是，实际上，由于多个事件映射到同一地址上的情况很少，以及由于运行的进程在其他进程被调度运行之前，通常要释放上锁的资源，所以这种多到一的映射并不影响性能。当然，从形式上看，如果映射是一对一的话，会使核心更容易理解。

6.6.2 算法 sleep 和 wakeup

图 6-31 给出了算法 sleep。内核首先提高处理机优先级来屏蔽所有中断，这样，当它处理睡眠队列时，就不会有竞争条件；内核还要保存老的处理机执行级，当进程后来被唤醒时，内核就可以恢复老的处理机执行级。然后，内核置该进程的状态为“睡眠”，将睡眠地址和优先级保存在进程表中，并把该进程放到睡眠进程的散列队列中。在简单的情况下（睡眠不能被中断），该进程做上下文切换，并安全地进入睡眠。当睡眠的进程醒来时，内核过一会儿会调度它运行：该进程从算法 sleep 中的上下文切换中返回，恢复处理机执行级为它进入该算法时的值，然后返回。

```
算法 sleep
输入:(1)睡眠地址
     (2)优先级
输出:如果进程由一个该进程捕俘的软中断信号唤醒,则为 1;
     如果进程由它不捕俘的软中断信号唤醒,则作 longjmp;
     否则为 0
{
    提高处理机执行级来屏蔽所有中断;
    设置进程状态为睡眠;
    根据睡眠地址将进程放到散列队列;
    将睡眠地址保存在进程表项中;
    设置进程优先级为输入优先级;
    if(进程的睡眠是不可中断的)
    {
        做上下文切换;
        /* 当进程醒来时它从此处恢复执行 */
        处理机优先级复位为进程进入睡眠时的值,从而允许中断;
        return(0);
    }
    /* 这里,进程的睡眠允许由软中断信号中断 */
    if(该进程没有待处理的软中断信号)
    {
        作上下文切换;
        /* 当进程醒来时它从此处恢复执行 */
        if(该进程没有待处理的软中断信号)
        {
            处理机优先级复位为进程进入睡眠时的值;
            return(0);
        }
    }
    将进程从睡眠队列中移出(如果它仍在那里);

    处理机的优先级复位为进程进入睡眠时的值;
    if(进程睡眠优先权设置为捕俘软中断信号)
       return(1);
    作算法 longjmp;
}
```

图 6-31 算法 sleep

为唤醒睡眠的进程，内核执行算法 wakeup（图 6-32）。内核或者在普通的系统调用算法中执行它，或者在处理中断时执行它。举例说来，算法 iput 释放一个上锁的索引节点，并唤醒所有等待释放该锁的进程。类似地，磁盘中断处理程序唤醒一个等待 I/O 完成的进程。核心在 wakeup 中提高处理机执行级来屏蔽中断。然后，对睡眠在输入的睡眠地址上的每个进程，将其状态置为“就绪”，把它们从睡眠进程的队列中移出，放到有资格被调度的进程

的队列中。这之后，内核要清除进程表中标志睡眠地址的字段。如果被唤醒的进程尚未装入内存，内核就唤醒对换进程，将该进程换入内存（假定系统不支持请求调页）；否则，如果唤醒的进程比正在执行的进程更有资格运行，那么内核就设置调度标志，因此，当该进程返回到用户态时，它会经历调度算法（第8章）。最后，内核恢复处理机执行级。再强调一遍，wakeup并不立即使一个进程被调度，它只是使该进程有资格被调度。

```
算法 wakeup /* 唤醒睡眠的进程 */
输入：睡眠地址
输出：无
{
    提高处理机执行级来屏蔽所有中断；
    查找睡眠散列队列或睡眠地址；
    for（每个在该地址上睡眠的进程）
    {
        将进程移出散列队列；
        置进程状态为"就绪"；
        将进程加入就绪进程的调度链表中；
        清除进程表中的睡眠地址域；
        if（进程尚未装入内存）
          唤醒对换进程(进程0)；
        else if（被唤醒的进程比当前正在运行的进程更有资格运行）
            设置调度标志；
    }
    将处理机的执行级恢复为原来的级别；
}
```

图 6-32 算法 wakeup

以上讨论的是算法sleep和wakeup的简单情况，因为我们假定进程睡眠，直到合适的事件发生。一般而言，进程在事件上睡眠，而这些事件"肯定"会发生，如等待某个上锁的资源（索引节点或缓冲区）时，或等待磁盘输入/输出的完成。因为这样的资源的使用总是被设计为临时的，所以睡眠的进程肯定会醒来。然而，一个进程有时可能睡眠在不一定发生的事件上。如果这样的话，必须有某种方法能重新获得控制并继续执行。对这种情况，内核通过给进程发送软中断信号来"中断"睡眠的进程。下章将详细解释软中断信号（signal）的含义。目前，假定内核能以软中断信号（可选择地）唤醒睡眠的进程，而且，该进程能够识别出曾给它发送过的软中断信号。

例如，如果某个进程向一个终端发出系统调用read，那么，在用户在那个终端上敲入数据之前，内核并不满足这个系统调用（见第10章）。然而，启动该进程的用户可能离开该终端，去开一个全天会议，使该终端老在睡眠并等待输入，而另一用户可能要用该终端。如果第二个用户采取了极端的措施（如关终端），那么内核需要有某种方法恢复被断开的进程：第一步，它必须以一个软中断信号把该进程从睡眠中唤醒。顺便提一下，进程睡眠时间很长并不是什么坏事。虽然睡眠的进程在进程表中占据一个表项，因而可能增加某些算法的查找时间，但它们不使用CPU时间，因此开销很小。

为了区分各种类型的睡眠，内核在进程进入睡眠时，根据其睡眠优先级参数以及某个事件是否会发生的知识，来设置该睡眠进程的调度优先级。内核用这个优先级值调用算法 sleep。如果优先级高于阈值，那么睡眠的进程在收到一个软中断信号时不会过早地醒来，而是一直睡到等待的事件发生。但是，如果优先级低于阈值，睡眠的进程会因收到软中断信号而立即醒来。㊀

当进程进入算法 sleep 时，如果已经给该进程设置了软中断信号，那么，上面讲的条件用来确定该进程是否真的要进入睡眠。例如，如果该进程的睡眠优先级高于阈值，进程则进入睡眠并等待显式的唤醒调用。反之，如果其睡眠优先级低于阈值，该进程则不进入睡眠，而是响应该软中断信号，就好象该软中断信号在它睡眠期间到达一样。如果在进入睡眠之前，内核不检查软中断信号，那么该信号可能不会再来，因而进程就会长睡不醒。

当一个进程由于软中断信号的作用而被唤醒时（或者进程由于已存在的软中断信号根本没进入睡眠），内核可能要作 longjmp。是否作 longjmp 取决于该进程最初进入睡眠的原因。如果内核无法完成它正在执行的系统调用，则要作 longjmp 来恢复一个以前保存的上下文。例如，如果一个终端的 read 调用由于用户关闭终端而被打断，这个 read 就不会完成，而应该返回一个错误指示。这一点适用于所有在进程睡眠期间可能被打断的系统调用。由于等待的睡眠事件没能满足，该进程被从睡眠中唤醒以后，不能正常地继续运行。因此，内核在大部分系统调用开始之时，都用 setjmp 来保存进程的上下文，为后来需要的 longjmp 做准备。

在有些情况下，内核想让进程在收到软中断信号时醒来，但不做 longjmp。内核用一个特殊的优先级参数调用算法 sleep，该参数制止 longjmp 的执行，并使 sleep 返回值 1。这样做比在 sleep 之前马上做 setjmp，然后又恢复该进程的上下文效率更高些。其目的是允许内核清除局部数据结构。例如，一个设备驱动程序可能分配一些私有数据结构，然后在一个可中断的优先级上进入睡眠；如果它由于一个软中断信号而醒来，它就应该释放已分配的数据结构。然后，如果必要的话，做 longjmp。用户不能控制一个进程是否做 longjmp；这取决于进程进入睡眠的原因，以及在该进程从系统调用返回之前，内核数据结构是否需要修改。

6.7　本章小结

本章定义了进程的上下文。UNIX 系统中的进程按明确定义的转换规则，在各种逻辑状态之间变换。状态信息被保存在其进程表和 u 区中。一个进程的上下文由它的用户级上下文和它的系统级上下文组成。用户级上下文包括进程的正文、数据、用户栈和共享存储区；系统级上下文由静态部分和动态部分组成。静态部分包括进程表项、u 区和存储器映射信息，动态部分则由核心栈和保存的前一层系统上下文层的寄存器组成。动态部分随着进程执行系统调用、处理中断和做上下文切换而被压入和弹出。进程的用户级上下文被分成若干独立的区，它们是由虚地址上连续的区域构成，并被看作是被保护和共享的可区分的实体。本章还定义了描述进程的虚地址布局的存储管理模型。该模型假定对进程的每个区使用一个页表。内核采用各种算法来管理这些区。最后，本章给出了 sleep 和 wakeup 的算法。以后的章节中将使用本章介绍的这些低层次的结构和算法，说明一些为了实现进程管理、进程调度和存储管理策略而设计的系统调用。

㊀ 术语“高于”和“低于”指的是通常的高优先级和低优先级的使用方法。但是，在核心的实现中，使用一些整数来衡量优先级值。较低的值意味着较高的优先级。

6.8 习题

1. 给定虚地址和 pregion 表项的地址，请设计将虚地址转换为物理地址的算法。

2. AT&T 3B2 计算机和 NSC 32000 系列使用一种两级（分段的）转换方法，把虚地址转换为物理地址。这种方法是，在系统中有一个指向页表指针表的指针，表中的每一项能够按其在表中的偏移量寻址某进程地址空间的一部分。请将这些机器上的虚地址转换算法与本书讨论的存储模型的算法做一比较。请考虑有关性能和辅助表所需要的空间问题。

3. 在 VAX-11 结构中有两组变址寄存器和上下界寄存器，用于用户的地址转换。这种方法和上一问题中的方法一样，只不过页表指针的数目为 2 个。假定进程有三个区，即正文区、数据区和栈区，将这些区映射到页表和使用这两组寄存器的好方法是什么？栈区应该是什么样子？（注意 VAX-11 结构中栈的增长方向是朝向较低的虚地址。）第 11 章将讨论用作共享存储区的另一种区，这种区应如何适应 VAX-11 的结构？

4. 请设计一种分配和释放存储页和页表的算法。什么样的数据结构能允许最好的性能和最简单的实现？

5. Motorola 68000 微处理机系列的 MC68541 存储管理部件允许存储段的分配。存储段的大小是 2 的幂，可以从 256 字节到 16 兆字节。每个存储管理部件有 32 个段描述块。请给出存储分配的有效方法。区的实现看上去应是什么样的？

6. 考虑图 6-5 中的虚地址映射。假定内核换出该进程（在一个对换系统中）或换出栈区中的许多页（在一个请求调页系统中）。如果该进程后来读 68342，它必须读物理存储器中与在对换或换页之前它曾读的地址相同的地址吗？如果较低层的存储管理是以页表实现的，这些页表一定要装入物理存储器中相同的位置吗？

*7. 有可能实现一种系统，在这种系统中，核心栈在用户栈上部增长。试讨论这种实现的优缺点。

8. 当将一个区附接到一个进程时，内核如何检查该区没有和已经附接到该进程的其他区的虚地址相重叠？

9. 考虑做上下文切换的算法。假定系统只含有一个就绪的进程。换言之，内核选取运行的进程是刚刚保存了它的上下文的进程。试说明会发生什么情况？

10. 假定一个进程进入睡眠，而系统中又没有就绪进程。当欲睡眠的进程做上下文切换时，会发生什么情况？

11. 假定在用户态下运行的某个进程用完了它的时间片，由于时钟中断的原因，内核调度了一个新的进程去运行。请给出发生在内核上下文第二层上的上下文切换。

12. 在调页系统中，在用户态运行的进程可能因为试图存取尚未装入内存的页而导致页面错。在为中断服务的过程中，内核从对换设备读这一页并进入睡眠。请给出在内核上下文第二层上发生的上下文切换（在睡眠期间）。

13. 在一个请求调页系统中，一个进程执行系统调用：

```
read (fd, buf, 1024);
```

假定内核执行算法 read，read 刚好达到这样一点上：它已经把数据读入系统缓冲区，但当它试图将数据拷贝到用户地址空间时，由于含有 buf 的页被换出而导致页面错。内核处理这种

中断的方法是将缺页读入内存。在每个内核上下文层中会发生什么情况？如果页面错处理程序在等待该页被写入内存期间进入睡眠，会发生什么情况？

14. 在图 6-17 中，当把数据从用户地址空间拷贝到内核时，如果用户提供的地址是非法的，会发生什么情况？

*15. 在算法 sleep 和 wakeup 中，内核靠提高处理机执行级来防止中断。如果内核不这样做，会发生什么恶果？（提示：内核经常要从中断处理程序中唤醒睡眠的进程。）

*16. 假定某个进程要在事件 A 上睡眠，但还没有执行到算法 sleep 中的屏蔽中断的代码。假定一个中断发生在进程提高处理机执行级之前（在算法 sleep 中），而中断处理程序要唤醒所有睡眠在事件 A 上的进程。对正要进入睡眠的那个进程会发生什么情况？是一个危险的情况吗？如果是这样，内核如何才能避免它？

17. 如果内核向所有睡眠在地址 A 上的进程发出 wakeup 调用，但此时此刻没有睡眠在该地址上的进程，请问会发生什么情况？

18. 许多进程可以睡眠在同一个地址上，但内核可能想唤醒那些被选定的、收到一个软中断信号的进程。假定软中断信号机制能够识别特殊的进程。请描述如何修改算法 wakeup，使它唤醒在睡眠地址上的某个进程，而不是全部。

19. Multics 系统中的算法 sleep 和 wakeup 具有如下语法形式：

```
sleep (event);
wakeup (event, priority);
```

也就是说，wakeup 将一个优先级赋给它正在唤醒的进程。请将这些调用和 UNIX 系统中的 sleep 和 wakeup 调用做一比较。

第7章 进程控制

上一章定义了进程上下文并解释了管理进程上下文的算法。本章将讨论控制进程上下文的系统调用的使用和实现。系统调用 fork 创建一个新进程；系统调用 exit 终止进程的执行；系统调用 wait 使父进程的执行与子进程的终止同步。软中断信号将异常事件通知给进程。因为内核是通过软中断信号来同步系统调用 exit 和 wait 的执行的，所以本章在讨论 exit 和 wait 之前，先讨论软中断信号。系统调用 exec 使一个进程调用执行一个“新”的程序，并用该文件的可执行映象覆盖自己的地址空间。系统调用 brk 允许进程动态地分配更多的空间；类似地，系统使用一个与 brk 一样的机制，允许在需要时通过分配更多的空间使用户栈动态地增长。本章的最后部分将概略地给出 shell 和 init 程序的主循环结构。

图 7-1 给出了本章所描述的系统调用与上一章中所描述的存储管理的算法之间的关系。几乎所有的系统调用都使用 sleep 和 wakeup，因而这两个算法没有在该表中出现，此外系统调用 exec 还与第 4 章和第 5 章中描述过的文件系统的算法交互。

与存储管理有关的系统调用			与同步有关的系统调用				其他	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

图 7-1 进程系统调用以及与其他算法的关系

7.1 进程的创建

在 UNIX 操作系统中，用户创建一个新进程的唯一方法就是调用系统调用 fork。调用 fork 的进程称为父进程，而新创建的进程叫做子进程。系统调用 fork 的语法格式是：

```
pid = fork();
```

在从系统调用 fork 中返回时，两个进程除了返回值 pid 不同外，具有完全一样的用户级上下文。在父进程中，pid 的值为其子进程的进程号。在子进程中，pid 的值为零。在系统初启时由内核内部地创建的进程 0 是唯一不通过系统调用 fork 而创建的进程。

内核为系统调用 fork 完成下列操作：

（1）为新进程在进程表中分配一个空槽。

（2）为子进程赋一个唯一的进程标识号（PID）。

（3）做一个父进程上下文的逻辑副本。由于进程的某些部分，如正文区，可能被几个进程所共享，所以内核有时只要增加某个区的引用数即可，而不用真的将该区拷贝到一个新的内存物理区。

（4）增加与该进程相关联的文件表和索引节点表的引用数。

（5）对父进程返回子进程的进程号；对子进程返回零。

理解系统调用 fork 的实现是十分重要的，因为子进程就像从天而降一样地开始它的执行序列。请求调页系统和对换系统中的 fork 算法稍有不同，这里的讨论是基于传统的对换系统的。但我们将指出为适应请求调页系统而需改动的地方。我们还假定系统有足够的可用主存空间来存放子进程。第 9 章将考虑没有足够空间来存放子进程的情形，并描述在请求调页系统中是如何实现系统调用 fork 的。

图 7-2 是系统调用 fork 的算法。内核首先确信有足够的资源来成功地完成 fork。在对换系统中，需要在内存或磁盘上有空间存放子进程；在请求调页系统中，内核需要为辅助表（如页表）分配存储空间。如果资源不满足要求，则系统调用 fork 失败。如果资源满足要求，内核在进程表中找一个空项，并开始构造子进程的上下文。

```
算法　fork
输入：无
输出：对父进程是子进程的 PID
      对子进程是 0
{
    检查可用的内核资源；
    取一个空闲的进程表项和唯一的 PID 号；
    检查用户没有过多的运行进程；
    将子进程的状态设为“创建”状态；
    将父进程的进程表项中的数据拷贝到子进程表项中；
    当前目录的索引节点和改变的根目录（如果可以）的引用数加 1；
    文件表中的打开文件的引用数加 1；
    在内存中作父进程上下文的拷贝（u 区，正文，数据，栈）；
    在子进程的系统级上下文中压入虚设系统级上下文层；
        /*  虚设上下文层中含有使子进程能
         *  识别自己的数据，并使子进程被调度时
         *  从这里开始运行
         */
    if （正在执行的进程是父进程）
    {
        将子进程的状态设置为“就绪”状态；
        return（子进程的 PID）；  /*  从系统到用户  */
    }
    else  /*  正在执行的进程是子进程  */
    {
      初始化 u 区的计时域；
      return（0）；  /*  到用户  */
    }
}
```

图 7-2　fork 算法

内核要肯定调用 fork 的用户没有过多的进程已在运行。内核还要为子进程选一个唯一的进程标识号，它比最近一次分配的进程标识号大 1。如果另一个进程已占用所选中的标识号，则内核就试着分配下一个更大的标识号。当标识号达到一个最大值时，则从 0 开始重新分配标识号。由于大多数进程只运行很短的时间，所以当重新开始分配时，大多数的标识号是未使用的。

系统对一个用户可以同时运行的进程数有一个限制（这个限制数是可装配的），因而任何用户都不能偷用过多的进程表项，不然的话，就会妨碍其他用户创建新进程。类似地，普通用户也不能创建会占用进程表中剩下的最后一个表项的进程，否则，系统事实上将死锁。也就是说，内核不能保证已存在的进程都将正常退出（exit），而且由于所有的进程表项都被占用，所以不可能再创建任何新的进程。另一方面，超级用户可按其意愿创建任意多个进程，其数目仅受限于进程表的大小。一个超级用户进程可以占据进程表中的最后一个空槽。可以推测，一个超级用户在需要时，可以采取断然措施，启动一个子进程来强迫其他进程退出（见 7.2.3 节，系统调用 kill）。

下一步，内核初始化子进程的进程表项，从父进程的表项中拷贝各个字段。例如，子进程“继承”了父进程的真正用户标识号和有效用户标识号、父进程的组号和用于计算调度优先权的父进程的 nice 值。下面各节将讨论这些字段的含义。内核填写子进程的父进程标识号字段，从而将子进程放入相应的进程树结构中。然后，内核初始化各种调度参数，如初始优先权值，初始 CPU 使用时间，以及其他计时字段。该进程的初始状态为“创建”状态（回忆图 6-1）。

这之后，内核调整文件的引用数，这些文件自动地与该子进程相关联。首先，子进程位于父进程的当前目录。当前存取该目录的进程数增加 1，相应地内核要增加该目录索引结点的引用数。其次，如果父进程或一个祖先进程曾因执行过系统调用 chroot 而改变了它的根目录，子进程也继承这一改变的根目录并增加其索引结点的引用数。最后内核查找父进程的用户文件描述符表，找出该父进程已知的打开文件，然后增加与每一打开文件相关联的全局文件表中的引用数。子进程不仅继承了打开文件的存取权限，而且还与父进程共享存取这些文件，这是因为两个进程管理相同的文件表项。系统调用 fork 的效果就像通过系统调用 dup 重复打开一个文件一样：用户文件描述符表中的一个新表项指向已打开文件的全局文件表项。所不同的是对于系统调用 dup，在用户文件表中的两项都是在同一进程内，而对于 fork，它们却在不同的进程内。

内核现在可以为子进程创建用户级上下文了。它为子进程的 u 区、区，及辅助页表分配内存；用算法 dupreg 复制父进程的所有区；并用算法 attachreg 将每个区附接到子进程。在对换系统中，内核拷贝所有非共享区的内容到一个新的内存区域。在 6.2.4 节中指出了 u 区中含有一个指向它的进程表表项的指针。除了这个字段外，子进程的 u 区被初始化为与其父进程 u 区相同的内容，但当 fork 调用完成之后它们就分道扬镳了。例如，父进程的 fork 调用之后打开一个新文件，而子进程并不能自动地被许可存取该文件。

至此，内核已为子进程创建了进程上下文的静态部分，现在，内核要为子进程创建动态部分。内核拷贝父进程上下文的第 1 层，该层含有用户保存的寄存器上下文和系统调用 fork 的核心栈。如果系统实现是将内核栈作为 u 区的一部分，则当内核创建子进程 u 区时，也就自动地创建了子进程的内核栈。否则，父进程必须将其内核栈拷贝到与子进程相关联的一个私有存储区上。不论是这两种情况中的哪种情况，父进程与子进程的内核栈完全一样。然后，内核为子进程创建一个虚设的上下文层（2），用以保存上下文层（1）的寄存器上下文。内核在该保存的寄存器上

下文中，设置程序计数器和其他寄存器，以便能“恢复”子进程的上下文，尽管子进程还从来没有执行过，同时，这也是为了使子进程运行时能够识别出自己是子进程。例如，内核代码通过测试寄存器 0 的值来决定本进程是子进程还是父进程，然后它在子进程的第 1 层上下文的保存的寄存器上下文中写入适当的值。这一机制类似于前一章中讨论的上下文切换。

当子进程上下文就绪后，父进程将子进程的状态改为“在内存中就绪”，并将子进程的进程标识号返回给用户，从而完成了它的那部分 fork 调用。内核以后将通过通常的调度算法来调度子进程的运行，那时，子进程将“完成”它的那部分 fork 调用。子进程的上下文是由父进程设置的；对于内核来说，子进程就像是在等待一个资源时被唤醒一样。根据内核从第 2 层上下文中保存的寄存器中恢复的程序计数器的值，子程序执行系统调用 fork 的最后部分，并将 0 作为系统调用的返回值。

图 7-3 给出了系统调用 fork 完成之后，父、子进程以及它们与其他内核数据结构之间的关系。可以看出，两个进程共享父进程在执行 fork 之前已打开的文件，并且，这些文件的文件表引用数也比以前大 1。同理，子进程具有与父进程相同的当前目录和改变的根目录（如果合适的话），这些目录的索引节点引用数也比 fork 之前大 1。父、子进程具有完全相同的正文和数据及（用户）栈区的副本；区的类型及系统实现决定二个进程是否可以共享正文区的一份物理拷贝。

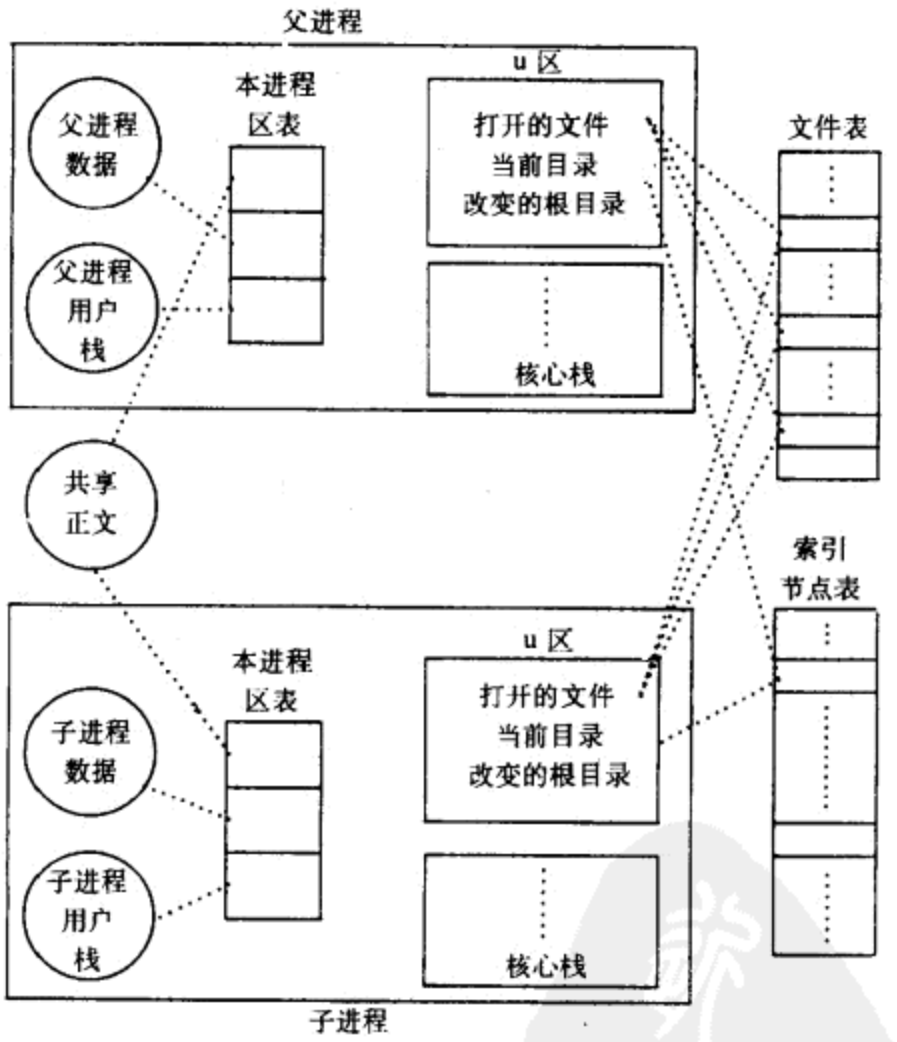


图 7-3　系统调用 fork ——创建一个新的进程上下文

考虑图 7-4 中的程序，该例子说明的是经过系统调用 fork 之后，对文件的共享存取。用户调用该程序时应有两个参数，一个是已有的文件名，另一个是要创建的新文件名。该进程

打开已有的文件，创建一个新文件，然后，假定没有遇到过错误，它调用 fork 来创建一个子进程。在内部，内核要为子进程做一个父进程上下文的拷贝，父进程和子进程在不同的地址空间上运行。每个进程都存取私有的全局变量 fdrd，fdwt 和 c 及私有的栈变量 argc 和 argv，但每个进程都不能存取另一进程的这些私有变量。然而，内核在 fork 调用时已将父进程的 u 区拷贝到子进程，因此，子进程通过使用相同的文件描述符而继承地存取父进程的文件（即父进程已打开和创建的文件）。

```
#include <fcntl.h>
int fdrd, fdwt;
char c;

main(argc, argv)
        int argc;
        char argv[];
{
        if(argc != 3)
                exit(1);
        if((fdrd=open (argv [1], O_RDONLY)) == -1)
                exit(1);
        if((fdwt=creat (argv [2], 0666)) == -1)
                exit(1);

        fork();
        /* 两个进程执行同样的代码 */
        rdwrt();
        exit(0);
}
rdwrt()
{
        for(;;)
        {
                if(read(fdrd, &c, 1) != 1)
                        return;
                write(fdwt, &c, 1);
        }
}
```

图 7-4 父子进程共享文件存取的进程

当然，父进程和子进程要分别独立地调用 rdwrt 函数，并执行一个循环，即从源文件中读一个字节，然后写一个字节到目标文件中去。当系统调用 read 遇到文件尾时，函数 rdwrt 立即返回。内核已增加了源文件和目标文件的文件表计数值，而且，两个进程的文件描述符都指向相同的文件表项，也就是说，两个进程的文件描述符 fdrd 都指向源文件的文件表项，fdwt 都指向目标文件的文件表项。这两个进程永远不会读或写到相同的文件偏移量，因为内核在每次 read 和 write 调用之后，都要增加文件的偏移量。尽管两个进程似乎是将源文件拷贝了两次，但因为它们分担了工作任务，因此，目标文件的内容依赖于内核调度二个进程的次序。如果内核这样调度两个进程：使它们交替地执行它们的系统调用，或甚至使它们交替地执行每对 read，write 调用，则目标文件的内容与源文件的内容完全一致。但考虑下述情况：两个进程正要读源文件中的两个连

续的字符“ab”。假定父进程读了字符“a”，这时，内核在父进程做写之前，做了上下文切换来执行子进程。如果子进程读到字符“b”并在父进程被调度到之前，将它写入目标文件，那么，目标文件将不再正确地含有字符串“ab”，而是含有“ba”了。内核并不保证进程执行的相对速率。

现在来看图 7-5 中的程序，子进程从父进程继承了文件描述符 0 和 1（即标准输入和标

```
#include <string.h>
char string[]="hello world";
main()
{
    int count, i;
    int to_par[2], to_chil[2];      /* 到父、子进程的管道 */
    char buf[256];
    pipe(to_par);
    pipe(to_chil);
    if (fork()==0)
    {
        /* 子进程在此执行 */
        close(0);           /* 关闭老的标准输入 */
        dup(to_chil[0]);    /* 将管道的读复制到标准输入 */
        close(1);           /* 关闭老的标准输出 */
        dup(to_par[1]);     /* 将管道的写复制到标准输出 */
        close(to_par[1]);   /* 关闭不必要的管道描述符 */
        close(to_chil[0]);
        close(to_par[0]);
        close(to_chil[1]);
        for (;;)
        {
            if ((count=read(0,buf,sizeof(buf)))==0)
                exit();
            write(1, buf, count);
        }
    }
    /* 父进程在此执行 */
    close(1);       /* 重新设置标准输入、输出 */
    dup(to_chil[1])
    close(0);
    dup(to_par[0]);
    close(to_chil[1]);
    close(to_par[0]);
    close(to_chil[0]);
    close(to_par[1]);
    for (i=0; i<15; i++)
    {
        write(1,string,strlen(string));
        read(0,buf,sizeof(buf));
    }
}
```

图 7-5　系统调用 pipe，dup 和 fork 的使用

准输出)。两次执行系统调用 pipe 分别在数组 to _ par 和 to _ chil 中分配了两个文件描述符。然后，该进程执行系统调用 fork，并拷贝进程上下文：像前一个例子一样，每个进程都存取自己的私有数据。父进程关闭它的标准输出文件（文件描述符 1），并复制（dup）从管道 to _ chil 返回的写文件描述符。因为在父进程文件描述符表中的第一个空槽是刚刚由关闭操作腾出来的，所以内核将管道写文件描述符拷贝到了文件描述符表中的第 1 项中，这样，标准输出文件描述符变成了管道 to _ chil 的写文件描述符。父进程以类似地操作将标准输入文件描述符替换为管道 to _ par 的读文件描述符。与此类似，子进程关闭它的标准输入文件（文件描述符 0），然后复制（dup）管道 to _ chil 的读文件描述符。由于文件描述符表的第一个空槽是原先的标准输入槽，所以子进程的标准输入变成了管道 to _ chil 的读文件描述符。子进程做一组类似的操作使它的标准输出变为管道 to _ par 的写文件描述符。然后，两个进程都关闭从 pipe 调用返回的文件描述符，这是良好的程序设计习惯，我们在后面将要解释这一点。上述操作的结果是，当父进程向标准输出写时，它实际上是写向 to _ chil ——向子进程发送数据，而子进程则从它的标准输入读管道。当子进程向它的标准输出写时，它实际上是写入管道 to _ par ——向父进程发送数据，而父进程则从它的标准输入接收来自管道的数据。两个进程通过两条管道交换消息。

不管两个进程执行相应的系统调用的顺序如何，这个例子的结果是不变的。也就是说，父进程在子进程之前还是在子进程之后从 fork 调用返回是没有关系的。类似地，不管两个进程在进入它们的主循环之前是以什么样的相对次序执行系统调用都没关系：内核的结构是完全一样的。如果子进程在其父进程进行写操作之前调用 read，它将进入睡眠，直到其父进程写管道并唤醒它。如果父进程在子进程读管道之前写管道的话，父进程的读标准输入的操作要等到子进程读它的标准输入并写它的标准输出后才能完成。从这时起，执行的顺序便定下来了：每一个进程在完成 read 和 write 系统调用之后，都要等到另一进程在完成其 read 和 write 之后，才能完成下一次 read 调用。父进程在 15 次循环之后退出（exit)。然后，子进程因管道线没有写进程而读到“文件尾”标志，并退出（exit)。

我们上面提到过，关闭不必要的文件描述符是一个良好的程序设计习惯，其理由有三条：第一，从系统规定了限制数的角度看，可以节省文件描述符。第二，我们将看到，如果子进程调用 exec，文件描述符在新的上下文中将保持已分配状态。在执行 exec 系统调用之前，关闭多余的文件，将使程序在一个干净、具有充足的空闲资源的环境下执行。它仅有打开的标准输入、标准输出和标准错误文件描述符。第三，仅当没有进程打开写管道时，管道的读操作才会返回文件尾。如果一个读进程保持管道的写文件描述符打开着，它就永远不会知道什么时候写进程关闭了它那头的管道。那样的话，上面这个例子就不能正确地工作了，除非子进程在进入它的循环之前关闭了它的写管道文件描述符。

7.2 软中断信号

软中断信号（signal，简称信号）通知进程发生了异步事件。进程之前可以通过系统调用 kill 相互发送软中断信号。内核也可以从内部发送软中断信号。在 UNIX 系统 V（版本 2）中有 19 个软中断信号，分为下列几类（见［SVID 85］中系统调用 signal 的描述)：

- 与进程终止相关的软中断信号。当进程退出时或当进程以子进程死（SIGCLD）为参

数调用系统调用 signal 时，发送这类软中断信号。

• 与进程例外事件相关的软中断信号。如进程访问一个在其虚地址空间以外的地址，企图写向一个只读内存区（如程序正文区），或执行一个特权指令及其他各种硬件错误。

• 与在系统调用期间遇到的不可恢复的条件相关的软中断信号。如在执行 exec 系统调用时，原来的资源被释放，而系统资源又被用光的情况。

• 由在执行一个系统调用时遇到的非预测错误条件所引起的软中断信号。例如，调用一个不存在的系统调用（即进程传递的系统调用号不对应于合法的系统调用）；向一个没有读进程的管道写数据或对系统调用 lseek 使用了一个非法的“访问”值。如果对这样的系统调用错误能返回一个错误的话，将会比产生一个软中断信号更一致些，但使用信号来中断有错误行为的进程却更实际些。[⊖]

• 由在用户态下的进程发出的软中断信号。例如一个进程想在一段时间后收到一个闹钟（alarm）信号，或进程用系统调用 kill 向其他进程发送任意的软中断信号。

• 和终端交互有关的软中断信号。例如当用户挂断一个终端时（或无论因何原因致使这条线上的“载波”信号消失时），或当一个用户按了终端键盘上的“break”键或“delete”键时所产生的软中断信号。

• 跟踪进程执行的软中断信号。

本章和以后的章节将解释使用各类软中断信号的场合。

软中断信号的处理有以下几个方面：内核如何向一个进程发软中断信号；进程如何接收软中断信号；以及进程如何控制自己对软中断信号的反应。为了给一个进程发送一个软中断信号，内核在该进程表中，按所要接收的信号类型设置软中断信号字段的某一位。如果该进程睡眠在一个可中断的优先级（interruptiable priority）上，内核就唤醒它。发送者（内核或进程）的任务就算完成了。一个进程可以记住不同类型的软中断信号。举例来说，如果某个进程收到了一个挂起（SIGHUP）信号和一个灭亡（SIGKILL）信号，它将在进程表的软中断信号字段中设置相应的两位，但是它没法告诉收到了多少这样的信号。

当一个进程即将从内核态返回到用户态时，或它要进入或离开一个适当的低调度优先级睡眠状态时，内核要检查它是否收到了一个软中断信号（见图 7-6）。内核仅当一个进程从内核态返回用户态时才处理软中断信号。因此，当一个进程在内核态下运行时，软中断信号并不立即起作用。如果一个进程正在用户态下运行，而且内核处理一个使一个软中断信号发送给该进程的中断，那么，当内核从中断返回后，它将识别和处理该软中断信号。这样，在处理未被处理的软中断信号之前，进程永远不会在用户态下执行。

图 7-7 给出了内核用以决定一个进程是否收到了一个软中断信号的算法。“子进程死”软中断信号的情况将在本章后面讨论。我们将看到，一个进程可以用系统调用 signal 来选择忽略软中断信号。在算法 issig 中，内核关闭该进程想要忽略的那些软中断信号指示位，而仅注意那些未被忽略的信号。

⊖ 在某些情况下使用软中断信号，可以揭露出一些不检查系统调用失败的程序中的错误（来自与 D.Ritchie 个人交流的看法）。

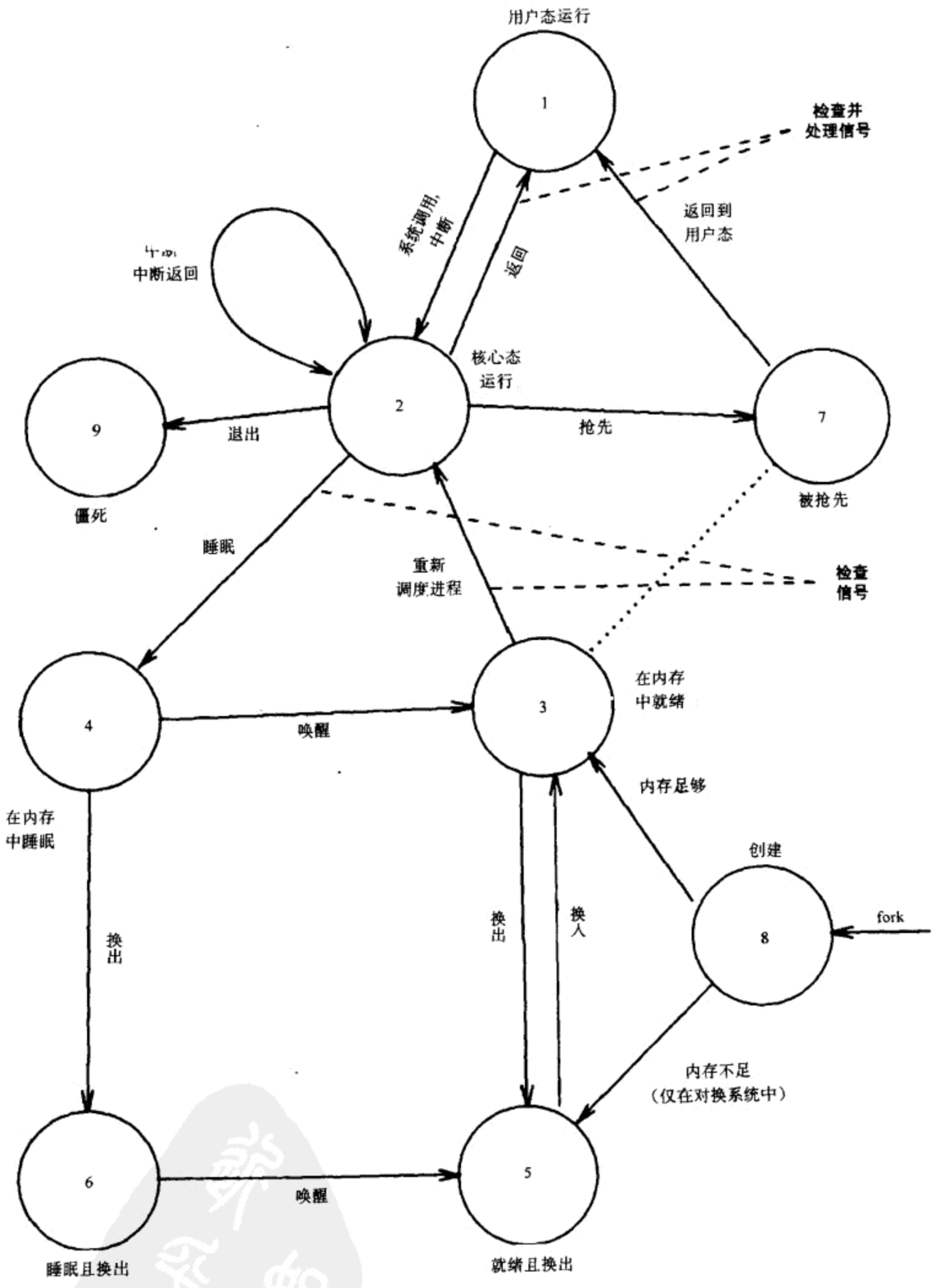


图 7-6 在进程状态图中软中断信号的检查和处理

```
算法 issg /* 测试软中断信号的接收 */
输入：无
输出：如果进程收到它不忽略的软中断信号，则真；
      否则，为假
{
    while (进程表项中收到信号的字段不为0)
    {
        找出发送给进程的一个软中断信号号；
        if (该信号是子进程死)
        {
            if (忽略子进程死信号)
                释放僵死子进程的进程表项；
            else if (捕俘子进程死信号)
                return(真)；
        }
        else if (不忽略信号)
            return(真)；
        关掉进程表项中收到信号字段的信号位；
    }
    return(假)；
}
```

图 7-7 识别软中断信号的算法

7.2.1 软中断信号的处理

内核在收到软中断信号的进程上下文中处理软中断信号，因此进程必须运行以便处理信号。处理软中断信号有三种情况：进程收到软中断信号后退出（exit）；进程忽略信号；或收到信号后执行一个特殊的（用户）函数。缺省动作是在内核态调用 exit，但是一个进程可以用系统调用 signal 来定义当收到某一信号时要做的特殊动作。

系统调用 signal 的语法格式为：

```
oldfunction=signal(signum, function);
```

其中 signum 为进程要为其定义动作的软中断信号号，function 为在收到该信号后，进程希望调用的函数的地址。返回值 oldfunction 为最近一次为信号 signum 所定义的函数的值。进程可以用 1 或 0 来代替一个函数的地址：如果参数值为 1，进程将忽略以后出现的该信号（7.4 节讨论忽略“子进程死”信号这一特殊情况）。如果参数值为 0（缺省值），收到信号后进程将在内核中退出（exit）。在 u 区中含有一个软中断处理程序字段的数组，每一个字段对应系统中定义的一个信号。内核将用户函数的地址存放于对应信号的字段中。定义一种信号的处理不影响对其他类型信号的处理。

在处理软中断信号时（图 7-8），内核首先确定信号的类型并关闭在进程表项中相应的信号位，即当进程收到该信号时所设置的位。如果软中断信号处理函数被置为缺省值时，内核对某些类型的信号在退出（exit）前要转储进程的“内存”映象（core）（见习题 7.7）。这

为程序人员提供了方便，允许他们确定转储的原因，并可以调试他们的程序。内核对意味着进程有错的那些软中断信号转储内存映象，例如，当进程执行了一个非法指令或存取一个在其虚地址范围以外的地址时。对那些不意味着程序错的信号，内核并不转储内存映象。例如，当收到用户在终端键盘上按“delete”键或“break”键时所发出的“interrupt”（SIGINT）软中断信号时，意味着用户要提前终止某个进程的执行；而收到一个挂起信号时，意味着注册终端不再是“连接着”的了。这些软中断信号并不意味着进程出现了错误。但软中断信号 SIGQUIT，尽管它在运行进程的外部引起，也要导致一个内存转储。这个“quit”信号通常是由在终端上敲“Ctrl—|”键所发出的，它使程序设计人员得到一个正在运行的进程的内存映象转储。这对一个进入无限循环的进程特别有用。

```
算法 psig /* 识别出软中断信号的存在之后处理信号 */
输入：无
输出：无
{
    取进程表项中设置的信号号；
    清进程表项中的信号号；
    if (用户已调用 signal 来忽略该信号)
        return; /* 完成 */
    if (用户指定了处理该信号的函数)
    {
        取 u 区中信号捕俘程序的虚地址；
        /* 下条语句有不好的副作用 */
        清 u 区中存放信号捕俘程序地址的项；
        修改用户级上下文：创建用户栈来模拟对信号捕俘程序的调用；
        修改系统级上下文：将信号捕俘程序的地址写入用户保存的寄存器上
            下文中的程序计数器域；
        return;
    }
    if (信号类型要求转储进程的内存映象)
    {
        在当前目录中创建“core”文件；
        将用户级上下文中的内容写入“core”文件中；
    }
    立即调用算法 exit；
}
```

图 7-8 处理软中断信号的算法

当进程收到一个它已决定要忽略的信号时，它就像没有收到该信号似地继续运行。因为内核并不重新设置 u 区中表示被忽略的信号的域，所以，如果该信号再发生的话，进程将继续忽略它。如果进程收到一个已决定要捕俘的信号，那么，当它返回用户态时，便立即执行用户定义的软中断处理函数。在执行前，内核要执行下列步骤：

(1) 内核存取用户保存的寄存器上下文，找出它为返回用户进程曾保存的程序计数器和

栈指针。

(2) 清除在u区中的软中断信号处理函数字段，将其置为缺省状态。

(3) 内核在用户栈上创建一个新的栈层，写入从用户保存的寄存器上下文中取出的程序计数器和栈指针的值，并在需要的情况下分配新的空间。用户栈看上去就像进程调用了一个用户层的函数（软中断信号捕俘程序)。调用点是它做系统调用或内核中断它的点（在识别信号之前)。

(4) 内核改变用户保存的寄存器上下文：它将程序计数器重置为软中断捕俘函数的地址，并将栈指针置为用户栈增长后的值。在从内核态返回用户态后，进程将执行软中断信号处理函数。当它从软中断处理函数返回时，进程回到系统调用或中断发生时的用户代码处，就像是从系统调用或中断返回一样。

```
#include <signal.h>
main()
{
        extern catcher();
        signal(SIGINT, catcher);
        kill(0, SIGINT);
}
catcher()
{
}
```

图 7-9 捕俘一个软中断信号的程序源代码

以图 7-9 为例。图中的程序捕俘 interrupt 软中断信号（SIGINT)，并向自己发一个 interrupt 软中断信号（这里是系统调用 kill 的结果)。图 7-10 是在 VAX 11/780 上的装入(load) 模块中与此有关部分的反汇编。系统执行该进程时，调用库子程序 kill 的地址是 ee(十六进制)。该库子程序在地址 10a 处执行 chmk（变为内核态）指令来调用系统调用 kill。系统调用的返回地址是 10c。在执行系统调用时，内核向该进程发送 interrupt 信号。内核在要返回用户态时注意到这一 interrupt 信号。内核从用户保存的寄存器上下文中取走地址 10c 并将其放入用户栈。内核取函数 catcher 的地址 104，将其放入用户保存的寄存器上下文中。图 7-11 显示了用户栈的状态及保存的寄存器上下文。

这里所描述的软中断信号处理算法中有几点值得注意的地方。第一个也是最重要的一个是：当一个进程处理软中断信号且在其返回用户态之前，内核要清除 u 区中含有用户软中断信号处理函数的地址的字段。如果进程要再次处理该信号，它必须再次调用系统调用 signal。这一点造成了不幸的后果：由于该软中断信号的第二次出现可能先于进程有机会再次调用 signal 之前，从而导致了竞争条件。由于进程正在用户态运行，内核可能进行上下文切换，这更增加了进程在重置信号捕俘函数之前收到该信号的可能性。

图 7-12 所示的程序给出了一个竞争条件。该进程调用 signal，安排捕俘 interrupt 软中断信号并执行函数 sigcatcher。然后，该进程创建子进程，调用系统调用 nice 来降低自己相对于其子进程的调度优先级（见第 8 章)，最后，该进程进入无限循环。子进程挂起 5 秒钟，

```
 * * * *          VAX 反汇编          * * * *
_main()
                e4;
                e6: pushab    0x18 (pc)
                ec: pushl     $0x2
   # 下行调用 signal
                ee: calls     $0x2, 0x23 (pc)
                f5: pushl     $0x2
                f7: clrl      -(sp)
   # 下行调用库子程序 kill
                 f9: calls    $0x2, 0x8 (pc)
                100: ret
                101: halt
                102: halt
                103: halt
_catcher()
                104:
                106: ret
                107: halt
_kill()
                108:
   # 下行陷入内核
                10a: chmk     $0x25
                10c: bgequ    0x6 <0x114>
                10e: jmp      0x14 (pc)
                114: clrl     r0
                116: ret
```

图 7-10　捕俘一个软中断信号的程序的反汇编

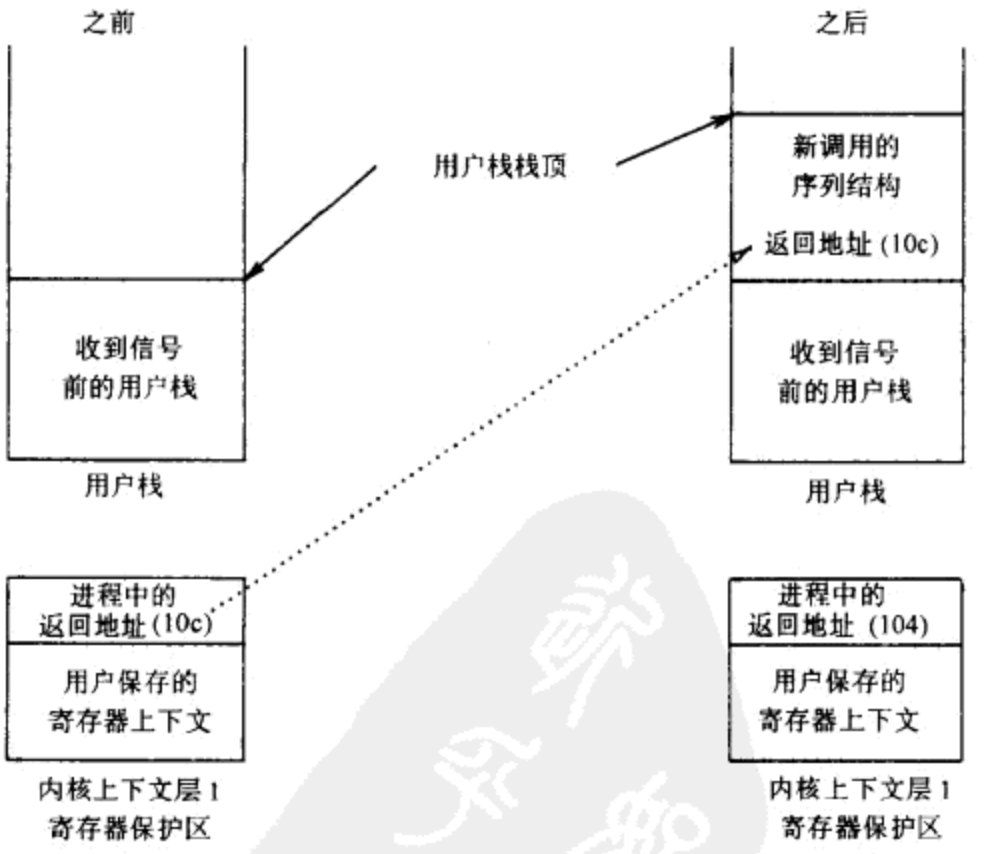


图 7-11　在收到软中断信号之前及之后的用户栈及内核保存区

```
#include <signal.h>
sigcatcher()
{
        printf("PID %d caught one \n", getpid());    /* 打印进程标识号 */
        signal(SIGINT,sigcatcher);
}

main()
{
        int ppid;

        signal(SIGINT, sigcatcher);

        if (fork() ==0)
        {
                /* 为两个进程留足够的时间做准备 */
                sleep(5);               /* 延迟5秒钟的库函数 */
                ppid=getppid();         /* 取父进程的标识号 */
                for (;;)
                        if (kill (ppid, SIGINT) ==-1)
                                exit();
        }

        /* 降低优先权，增大演示竞争的机会 */
        nice(10);
        for (;;)
                ;
}
```

图 7-12 演示在捕俘软中断信号时竞争条件的程序

给父进程以足够的处理时间来执行系统调用 nice 和降低其本身的优先权。子进程然后也进入循环，在每次循环中，子进程通过系统调用 kill 向父进程发 interrupt 软中断信号。如果系统调用 kill 由于出错而返回，很可能是因为父进程不再存在了，那么子进程便退出。该程序的思想是，父进程每次收到 interrupt 信号时，都应调用信号捕俘函数 sigcatcher。该函数打印一条消息，并再次调用 signal 以便捕俘下次 interrupt 信号的发生。然后，父进程继续执行无限循环。

然而，事件有可能按下列顺序发生：

(1) 子进程向父进程发一个 interrupt 信号。

(2) 父进程捕俘到这一信号并调用软中断捕俘函数，但内核在该进程再次执行 signal 之前，抢先该进程并切换上下文。

(3) 子进程再执行，又向父进程发一个 interrupt 信号。

(4) 父进程收到第二个 interrupt 信号，但是它却并没有安排捕俘这一信号。当父进程恢复运行后便退出。这样写这个程序是为促使这种情况的出现，因为父进程的 nice 系统调

用，导致内核更频繁地调用子进程。但究竟何时发生这一结果却是不确定的。

按照 Ritchie 的意见（私人通讯），软中断信号原被设计为对付致命的或忽略的事件，不必进行处理。所以在早期的版本中，并没有解决竞争条件。然而，对于要捕俘软中断信号的程序，这却造成了严重问题。如果收到软中断信号后不清除软中断信号字段，则可以解决上述问题。但随之而来的新问题是，如果信号不断地来并不断地被捕俘，用户栈将由于软中断捕俘函数的嵌套而溢出。另一种解决方法是，内核将软中断信号处理函数重新设置为缺省函数，直到用户定义如何处理该类信号。这一方案隐含着丢失信息的危险，因为进程无法得知它收到了多少个信号。然而，信息的丢失并不比进程在得到机会处理以前收到多个同类信号更严重。最后，BSD 系统用一个新的系统调用来阻止和开放进程对软中断信号的接收。当进程开放信号时，内核就发出当进程阻止信号时就已到来的待处理的信号。当进程接收到了一个信号，内核就自动地阻止进一步接收该类信号，直到完成对该信号的处理。这个方法模拟了内核对硬件中断的处理；在处理某个中断时，阻止接收新的中断。

处理软中断信号中要注意的第二点是捕俘发生在进程正在一个系统调用的执行中，并睡眠在可中断优先级上的软中断信号。这样的信号引起进程做一次 longjmp 跳出睡眠，返回用户态并调用软中断处理函数。当信号处理函数返回时，进程就像是从系统调用返回一样，但带回一个错误指示，指示该系统调用曾被中断。用户可以检查错误返回，并重新执行该系统调用。但有时如果内核自动地重新开始执行系统调用，就像在 BSD 系统中那样，将会更方便。

第三个要注意的地方是进程忽略一个信号的情况。如果一个进程睡眠于可被中断的优先级上，它收到这个软中断信号，该进程将被唤醒，但不做 longjmp。也就是说，内核只有在唤醒该进程并运行它时，才能认识到该进程忽略了这个软中断信号。更一致的策略是让进程保持睡眠。然而，内核把软中断信号处理函数的地址存放于 u 区中，但当发信号给该进程时，u 区却不一定可被存取。解决这个问题的一个可能的方案是将软中断处理函数的地址存放于进程表项中，当收到信号时，核心可以在进程表项中查看是否应当唤醒该进程。换句话说，如果发现进程不应被唤醒的话，进程可以立即在 sleep 算法中重新进入睡眠。不过，用户进程永远不会得知它被唤醒过，因为内核在算法 sleep 中（见第 2 章）有一个 while 循环，如果睡眠事件没有真的发生的话，它将使进程重新睡眠。

最后一点，内核不像处理其他软中断信号那样处理“子进程死”软中断信号。内核对“子进程死”信号进行特殊的处理。当进程识别出收到了一个“子进程死”软中断信号后，它关闭在进程表项中的信号指示，在缺省值的情况下，该进程就像没有发生该信号似地工作。“子进程死”信号的作用是唤醒一个睡眠在可被中断优先级上的进程。如果该进程捕俘到“子进程死”软中断信号，它像处理其他软中断信号一样也调用用户处理函数。如果进程忽略该信号，内核的操作将在 7.4 节中讨论。最后，如果一个进程用“子进程死”作为参数，调用 signal 系统调用，而且该进程有一个子进程处于僵死状态的话，内核将向进程发一个“子进程死”软中断信号。在 7.4 节中将讨论参数为“子进程死”的系统调用 signal 的基本工作原理。

7.2.2 进程组

尽管在 UNIX 中进程是以唯一的进程标识号所标识的，但系统有时必须用“组”来标识进程。例如，以一个注册 shell 进程作为共同祖先的所有进程通常是相关的，因此，当一个用户按“delete”键或“break”键时，或当终端挂起时，所有这样的进程都要收到信号。

内核用进程组标识号（process group ID）来标识一组相关的进程。这组进程对于某些事件应收到共同的信号。系统将组标识号放在进程表中，同一进程组中的进程具有相同的组标识号。

系统调用 setpgrp 初始化一个进程的进程组号，将其置为与该进程标识号相同的值。该系统调用的语法格式为：

```
grp=setpgrp();
```

其中，grp 是新的进程组号。在系统调用 fork 期间，子进程得到其父进程的进程组号。对于设置一个进程的控制终端，系统调用 setpgrp 也有着重要的作用（见 10.3.5 节）。

7.2.3　从进程发送软中断信号

进程使用系统调用 kill 来发送软中断信号。该系统调用的语法格式为：

```
kill(pid, signum)
```

其中，pid 标识了一个接收软中断信号的进程集合。signum 是要发送的软中断信号号。下面列出的是一组进程与 pid 值之间的对应关系。

- 如果 pid 是正值，内核将信号发送给进程号为 pid 的进程。
- 如果 pid 为 0，内核将信号发送给所有与发送进程同组的进程。
- 如果 pid 为 -1，内核将信号发给所有真正用户标识号等于发送进程的有效用户标识号的进程（7.6 节将定义进程的真正和有效用户标识号）。如果发送进程具有超级用户的有效用户标识号，则内核将信号发送给除进程 0 和进程 1 以外的所有进程。
- 如果 pid 为负数但非 -1，内核将信号发送给组号为 pid 绝对值的进程组中的所有进程。

在上述所有情况中，如果发送进程不具有超级用户的有效用户标识号，或者它的真正的或有效的用户标识号与接收进程的真正的或有效的用户标识号不匹配，则系统调用 kill 失败。

```
#include <signal.h>
main()
{
        register int i;

        setpgrp();
        for (i=0; i<10; i++)
        {
                if (fork()==0)
                {
                        /* 子进程 */
                        if (i & 1)
                                setpgrp();
                        printf("pid=%d pgrp=%d\n", getpid(), getpgrp());
                        pause();        /* 挂起执行的系统调用 */
                }
        }
        kill(0, SIGINT);
}
```

图 7-13　使用系统调用 setpgrp 的例子

在图 7-13 所示的例子中，进程重新设置了它的进程组号并创建了 10 个子进程。当创建子进程时，每一子进程都具有与父进程相同的进程组号，但奇数遍循环中所创建的子进程又重新设置了它们的进程组号。系统调用 getpid 和 getpgrp 分别返回执行进程的进程标识号和进程组标识号。系统调用 pause 挂起执行进程，直到它收到软中断信号。最后，父进程执行系统调用 kill 向所有同组的进程发 interrupt 信号。内核将信号发送给 5 个没有重置进程组号的“偶数”进程，而 5 个“奇数”进程却继续挂起。

7.3　进程的终止

UNIX 系统中的进程执行系统调用 exit 来终止运行。每个退出的进程进入僵死状态（见 6.1 节），释放它的资源，撤除进程上下文，但保留它的进程表项。系统调用 exit 的语法格式为：

```
exit(status);
```

其中 status 的值返回给父进程以便检查。进程可以显式地调用 exit 或在程序的结尾隐含地调用：与所有 C 程序连接的 startup 子程序，在从主函数（main，即所有程序的入口点）返回时，调用 exit。另一方面，像前面讨论的那样，内核在进程收到非捕俘的信号时，可以从内部调用 exit。如果是这样，则 status 的值是该软中断信号号。

系统对进程的执行时间没有限制。进程常常存在很长时间，例如进程 0（对换进程）和进程 1（init 进程）在系统的整个生命周期期间一直存在着。其他的例子有 getty 进程，它监视终端线路，等待着一个用户注册；还有一些特殊目的的管理进程。

图 7-14 给出了系统调用 exit 的算法。内核首先关闭进程的软中断信号的处理函数，因为

```
算法　exit
输入：给父进程的返回码
输出：无
{
        忽略所有软中断信号;
        if （是与控制终端关联的进程组组长）
        {
            向该进程组的所有组员发送挂起信号;
            将所有组员的进程组号置为 0;
        }
        关闭所有打开的文件（算法 close 的内部形式）;
        释放当前目录（算法 iput）;
        释放改变的根目录（如果存在的话）（算法 iput）;
        释放区及与该进程有关的内存（算法 freereg）;
        写记帐记录;
        使进程状态为僵死状态;
        将所有子进程的父进程置为 init 进程（1）;
            若任何子进程僵死，则向 init 发送子进程死信号;
        向父进程发送子进程死信号;
        上下文切换;
}
```

图 7-14　系统调用 exit 的算法

信号处理已不再有任何意义了。如果终止的进程是与某一控制终端相关联的进程组组长（见10.3.5节），则内核假定用户不再做任何有用的工作，并向所有同组的进程发“挂起”软中断信号。这样，如果一个用户在注册 shell 进程中按下“文件尾”键（Ctrl－D）时，如果尚有一些活动进程与该终端相关联的话，正在退出的进程将向这些进程发送一个“挂起”软中断信号。内核还将同组进程的进程组号重置为 0，因为以后另一个进程可能得到刚刚退出的那个进程的进程标识号，并且也为进程组的组长。属于老进程组的进程将不属于后来的这个进程组。内核然后扫描打开的文件描述符，用算法 close 内部地关闭每一个打开的文件，并用算法 iput 释放被内核作为当前目录和改变的根目录而存取的索引节点。

然后，内核用算法 detachreg 释放相应的区，从而释放了所有用户内存，并将进程的状态改为僵死状态。内核在进程表中保留退出的状态码以及该进程及其后代的用户和内核运行累计时间。7.4 节中关于算法 wait 的描述将给出一个进程如何得到其后代的计时数据。内核还要写一个记帐记录到一个全局的记帐文件中去。记录中含有各种运行时的统计信息，如用户号，CPU 及内存的使用情况，还有进程的输入输出量。用户层的程序以后可以读记帐文件，收集各种统计数据，以利于性能监控及制做顾客帐单。最后，内核使进程 1（init 进程）接受该退出进程的全部子进程，从而使它从它的进程树上断开。也就是说，进程 1 变成了退出进程所创建的所有还活着的子进程的合法父进程。如果有子进程处于僵死状态，退出进程则向 init 进程发一个“子进程死”的软中断信号，因此 init 进程就能将僵死的子进程从进程表中删除（见 7.9 节）。退出进程也要向其父进程发一个“子进程死”软中断信号。在典型的情况下，父进程要执行系统调用 wait 来与子进程的退出保持同步。当前僵死的进程做一个上下文切换，以便内核可以调度另一个进程运行。内核决不会调度一个僵死的进程去运行。

```
main()
{
    int child;

    if ((child=fork())==0)
    {
        printf("child PID %d\n",getpid());
        pause();      /* 挂起执行，直到收到信号 */
    }
    /* 父进程 */
    printf("child PID %d\n", child);
    exit(child);
}
```

图 7-15　exit 的例子

在图 7-15 的程序中，一个进程创建了一个子进程。子进程打印它的进程号，并执行系统调用 pause，挂起自己直到收到一个软中断信号。父进程打印子进程的进程号后退出，并返回子进程的 PID 作为它的状态码。如果 exit 没有显式地出现，startup 例程将在该程序从函数 main 返回时，调用 exit。尽管父进程已死亡，但由父进程所产生的子进程却继续下去

直到收到一个软中断信号为止。

7.4 等待进程的终止

一个进程可以通过系统调用 wait 使它的执行与子进程的终止同步。系统调用 wait 的语法格式为

```
pid=wait(stat_addr)
```

其中，pid 是僵死子进程的进程号；stat_addr 是一个整数在用户空间的地址，它将含有子进程的退出状态码。

图 7-16 给出了系统调用 wait 的算法。内核寻找该进程的某个僵死子进程。如果该进程没有子进程，则返回一个错误。如果找到一个僵死子进程，内核取该子进程的 PID 及子进程在 exit 调用中提供的参数，并从系统调用返回这些值。这样，一个退出的进程可以定义各种返回码来给出退出的原因。但是实际上，许多程序对返回码的设置并不一致。内核分别将子进程在内核态及用户态执行的累计时间加到父进程 u 区中的相应字段中，并释放被僵死子进程先前所占用的进程表项。该表项现在可用于一个新的进程。

```
算法 wait
输入：存放退出进程的状态的变量地址
输出：子进程标识号，子进程退出码
{
    if (等待进程没有子进程)
        return(错);
    for(;;) /* 该循环直到从循环内返回时结束 */
    {
        if (等待进程有僵死子进程)
        {
            取任一僵死子进程；
            将子进程的 CPU 使用量加到父进程；
            释放子进程的进程表项；
            return(子进程标识号，子进程退出码);
        }
        if (该进程没有子进程)
            return(错);
        睡眠于可中断的优先级上（事件：子进程退出);
    }
}
```

图 7-16 系统调用 wait 的算法

如果执行 wait 的进程有子进程，但没有僵死子进程，则该进程睡眠在可被中断的优先级上，直到出现一个软中断信号。内核对于一个睡眠于系统调用 wait 中的进程没有显式的唤醒调用：这样的进程只能在收到软中断信号时被唤醒。对除了“子进程死”之外的任一软中断信号，进程将像前面所描述的那样做出反应。但是，如果软中断信号是“子进程死”，该进程将采取不同的对策。

• 在缺省的情况下，进程将从系统调用 wait 的睡眠中醒来。算法 sleep 将调用算法 issig 来检查软中断信号。算法 issig(图 7-7)识别出“子进程死”软中断信号的特殊情况，并返回“假”，接下来内核并不为算法 sleep 做 longjmp，而是返回系统调用 wait。内核将重新开始 wait 算法中的循环，找出一个僵死的子进程——至少有一个僵死的子进程保证存在，释放子进程的进程表项，并从系统调用 wait 中返回。

• 如果进程捕俘“子进程死”软中断信号，内核将像处理其他信号一样，安排调用用户的软中断信号处理子程序。

• 如果进程忽略“子进程死”软中断信号，内核将重新开始 wait 中的循环，释放僵死子进程的进程表项，然后寻找其他的子进程。

```
#include <signal.h>
main(argc, argv)
    int argc;
    char * argv[] ;
{
    int i, ret_val, ret_code;

    if (argc>1)
        signal(SIGCLD, SIG_IGN);      /* 忽略子进程死 */
    for(i =0; i<15; i++)
        if (fork() ==0)
        {
            /* 子进程 */
            printf("child proc %x \n",getpid());
            exit(i);
        }
    ret_val=wait( &ret_code);
    printf("wait ret_val %x ret_code %x \n", ret_val, ret_code);
}
```

图 7-17　等待并忽略子进程死软中断信号的例子

例如，用一个或零个参数调用图 7-17 的程序，用户将得到不同的结果。首先考虑第一种情况，用户不用参数调用这一程序（argc 为 1，即程序名）。(父）进程创建 15 个子进程。子进程将以 i 作为返回码退出，i 是创建该子进程时的循环变量。内核在父进程执行系统调用 wait 时，寻找僵死子进程并返回它的进程标识号和退出码。至于找到哪个子进程是不确定的。系统调用 exit 的 C 库程序代码将退出码存放于变量 ret_code 的 8～15 位，并将子进程的标识号作为 wait 调用的返回值。因此，ret_code 等于 256 * i，它取决于子进程的 i 值。ret_val 等于子进程标识号的值。

如果用户用一个参数调用上述的程序（argc>1)，则（父）进程调用 signal 来忽略“子进程死”软中断信号。假定父进程在所有子进程退出前睡眠于系统调用 wait 中。当子进程退出时，它向父进程发一个“子进程死”软中断信号。父进程被唤醒，因为它在 wait 中睡眠在一个可中断的优先级上。当父进程又开始运行时，它发现出现的软中断信号是“子进程

死”；但因它忽略“子进程死”软中断信号，内核便从进程表中清除该僵死子进程的表项并继续执行系统调用 wait，就像没有发生过软中断信号一样。每当父进程收到一个“子进程死”软中断信号后，内核都要做上述的处理过程，直到内核执行完算法 wait 的循环，并发现父进程没有子进程了。然后，系统调用 wait 返回 -1。这两次调用之间的不同是，在第 1 次调用中，父进程等待任意一个子进程结束；在第 2 次调用中，父进程等待所有子进程结束。

在 UNIX 系统老一些的版本中，系统调用 exit 和 wait 的实现没有采用“子进程死”软中断信号。系统调用 exit 用唤醒父进程代替发送“子进程死”软中断信号。如果父进程正睡眠于系统调用 wait 中，它会被唤醒并找到一个僵死的子进程然后返回。如果它不是睡眠于系统调用 wait 中，则本次唤醒无效，在它下一次调用 wait 时，它将发现它的一个僵死子进程。类似地，如果退出进程想要将僵死子进程过继给 init 进程并唤醒 init 进程，则 init 进程也要睡眠于系统调用 wait 中。

这一实现的问题是，除非父进程执行系统调用 wait，否则不可能清除僵死子进程。如果某个进程创建了许多子进程，但不执行系统调用 wait，则当子进程退出后，进程表将被僵死子进程占满。例如，考虑图 7-18 中的“派遣者”程序。该进程读它的标准输入文件，直到文件结束。每一次读入后，它创建一个子进程。然而，父进程并不做系统调用 wait 来等待子进程的结束，因为它想尽快地派遣进程，并且，子进程可能要经过很长时间才退出。如果父进程使用系统调用 signal 来忽略“子进程死”软中断信号，内核将自动地释放僵死进程的进程表项。否则，僵死进程会逐渐地占满进程表所允许的所有表项。

```
#include <signal.h>
main(argc, argv)
{
    char buf[256];

    if (argc != 1)
        signal(SIGCLD, SIG_IGN);      /* 忽略子进程死 */
    while (read(0,buf,256))
        if (fork()==0)
        {
            /* 此处为典型的对 buf 进行操作的子进程 */
            exit(0);
        }
}
```

图 7-18　说明子进程死软中断信号的必要性的例子

7.5　对其他程序的引用

系统调用 exec 引用另一个程序，它用一个可执行文件的副本覆盖一个进程的存储空间。除了 exec 的参数之外，在 exec 调用之前存在的用户级上下文的内容，在 exec 调用之后就不能再存取了。内核将 exec 的参数从旧地址空间拷贝到新的地址空间。这一系统调用的语法

格式是：

```
execve(filename, argv, envp)
```

其中，filename 是要引用的可执行文件的文件名；argv 是一个字符指针数组的指针，这组字符指针是可执行程序的参数。envp 是另一个字符指针数组的指针，这一组字符指针是执行程序的环境。调用系统调用 exec 的库函数有好几个，如 execl，execv，execle 等等。这些库函数最终都调用 execve，所以，这里我们用它来说明系统调用 exec。当一个程序使用命令行参数时，就像

```
main(argc, argv)
```

中一样，数组 argv 是给系统调用 exec 的参数 argv 的副本。在环境中的字符串的形式为“名字=值”，它可以含有对程序十分有用的信息。如用户的主目录和查找可执行程序的路径名。进程可以通过一个由 C 语言的 startup 例程进行初始化的全局变量 environ 来存取它们的环境。

```
算法　exec
输入：(1) 文件名
      (2) 参数表
      (3) 环境变量
输出：无
{
    取文件的索引节点（算法 namei）；
    验证文件可被执行，用户有执行许可权；
    读文件头，检查它是否为装入模块；
    将 exec 参数从老地址空间拷贝到系统空间；
    for （与该进程相附接的每个区）
        解除全部老的区（算法 detachreg）；
    for （在装入模块中指定的每个区）
    {
        分配新区（算法 alloreg）；
        将新区附接到进程（算法 attachreg）；
        如果合适的话，将区装入内存（算法 loadreg）；
    }
    将 exec 的参数拷贝到新的用户栈区；
    对 setuid 程序和进程跟踪做特别处理；
    为返回用户态，初始化用户寄存器保存区；
    释放文件的索引节点（算法 iput）；
}
```

图 7-19　系统调用 exec 的算法

图 7-19 给出了系统调用 exec 的算法。exec 首先通过算法 namei 存取文件，判断它是否是一个可执行的正规（非目录）文件，并判断该用户是否有执行该程序的许可权。然后，内核读文件头，判断可执行文件的结构。

图 7-20 给出了在文件系统中一个可执行文件的逻辑格式。可执行文件一般由汇编程序或

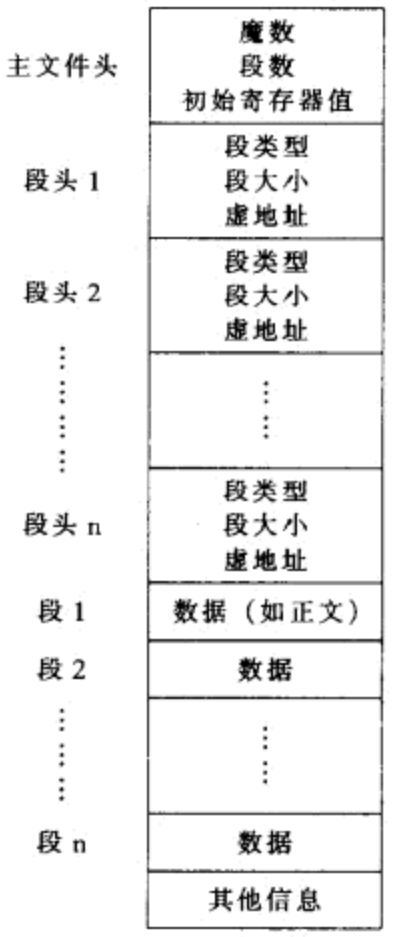


图 7-20　一个可执行文件的映象

装配程序（loader）所生成。它由四部分组成：

（1）主文件头，它描述了文件中有多少段、进程执行的起始地址和一个“魔数”（magic number）。魔数给出了可执行文件的类型。

（2）若干个段头（section header），段头描述了文件中的每个段，给出了段的大小、在系统中运行时该段所占据的虚地址及其他一些信息。

（3）含有“数据”的段，如正文，这些“数据”开始要被装入进程的地址空间。

（4）其他信息段，这些段可能含有符号表和其他数据。这些信息对于调试程序是很有用的。

多年来，文件定义的格式不断地变化，但所有可执行文件总是具有一个带有“魔数”的主文件头。

魔数是一个短整数，它标识文件是一个装入模块并使内核可以区别其运行时的特征。例如，在 PDP 11/70 上所用的特殊魔数告知内核该进程将使用 128K 的空间而不是 64K[⊖]。魔数在请求调页系统中也具有重要的作用，我们将在第 9 章看到这一点。

此时，内核取到了可执行文件的索引结点，并证实了它是可执行的。内核将释放目前形成用户级上下文的内存资源。但由于新程序的参数还在要被释放的存储空间中，内核先要将这些参数从老的存储空间拷贝到一个临时缓冲区，直到它为新的存储空间附接一些区。

因为系统调用 exec 的参数是一组字符串的地址，所以内核对每一字符串都要先拷贝字符

⊖ 魔数的值是 PDP 11 转移指令的值。系统的最初版本执行这些指令。程序计数器根据文件头的大小和要执行文件的类型，分别转移到不同的地址上去。自从系统用 C 语言改写后，这一特性就不再被使用了。

串的地址，然后再拷贝字符串到内核空间。可以选择几个不同的地方来存放这些字符串，这取决于实现的方式。最常用的地方是内核栈（在内核中的局部数组）；也可以将字符串放在被暂时借用的未分配的存储区域（如页面）中，或二级存储器，如对换设备。

在把 exec 的参数拷贝到内核中的保存区后，内核用算法 detachreg 使该进程与老的区断接。对于正文区的特殊处理将在本节后面讨论。此时，进程没有用户级上下文，所以从现在起遇到的任何由软中断信号引起的错误都导致进程的终结。这样的错误包括内核区表溢出、企图装入一个大小超出系统限制的程序、企图装入一个区重叠的程序及其他一些错误。内核为正文和数据分配区，附接区，并将可执行文件的内容装入到主存（算法 allocreg，attachreg，和 loadreg）。进程的数据区被（初始地）分为两部分：在编译时初始化的数据和在编译时没被初始化的数据（“bss”）。数据区的初始分配和附接是针对已初始化的数据的。然后，内核用算法 growreg 为存放“bss”数据而扩充数据区的大小，并将这部分存储器的值初始化为零。最后，它为进程栈分配一个区，把该区附接到进程，然后分配内存以存放 exec 的参数。如果内核将 exec 的参数存放在存储页中，它可以使用这些页作为栈。否则，内核要拷贝 exec 的参数到用户栈。

内核清除 u 区中的用户软中断处理函数的地址，因为这些地址在新的用户级上下文中是没有意义的。忽略的信号在新的上下文中还保持被忽略。然后，内核为用户态设置保存的寄存器上下文，特别是设置用户栈指针和程序计数器：装配程序已将初始程序计数器写在文件头中。内核对设置标识号的程序（简称 setuid 程序）和进程跟踪采取特殊的动作，这两点将在下一节和第 11 章中分别讨论。最后，内核调用算法 iput 释放在 exec 开始时，由算法 namei 所分配的那个节点。算法 namei 和 iput 的使用相应于它们在打开和关闭文件中的使用。在 exec 期间，文件的状态类似于一个打开的文件，但没有文件表项。当进程从系统调用 exec“返回”后，它执行新程序的代码。然而，它还是执行 exec 之前的那个进程；它的进程标识号没有变，它在进程家族树中的位置也没有变。变化的只是用户级的上下文。

例如，图 7-21 中的程序创建一个子进程。该子进程调用系统调用 exec。当父进程和子进程刚从系统调用 fork 返回时，它们执行这个程序的独立副本。当子进程要调用 exec 时，它的正文区含有本程序的指令；它的数据区有字符串“/bin/date”和“date”；它的栈含有进程为执行 exec 调用而压入的栈层。内核在文件系统中找到文件“/bin/date”，发现所有的用户都可以执行它，并判断它是一个可执行的装入模块。习惯上，exec 的 argv 参数表的第 1 个参数是可执行文件的路径名（的最后一个分量）。这样，进程可以在用户级得到程序名，有时这是有用的特性。[⊖]内核然后拷贝字符串“/bin/date”和“date”到一个内部的保存区，然后释放进程占用的正文、数据和栈区，将文件“/bin/date”的指令段拷贝到正文区，将文件的数据段拷贝到数据区。内核重新构造原来的参数表（在本例中是字符串“date”）并将其放入栈区。在系统调用 exec 之后，子进程不再执行老的程序，而是执行程序“date”。当“date”程序完成后，父进程从系统调用 wait 收到它的退出码。

⊖ 例如在系统 V 上，文件换名的标准程序 mv、拷贝文件的标准程序 cp 和联结文件的标准程序 ln 是一个可执行文件，因为它们执行类似的代码。进程查看用户调用它时所用的名字来决定它应做什么。

```
main()
{
        int status;
        if (fork() ==0)
                execl("/bin/date","date", 0);
        wait(&status);
}
```

图 7-21　系统调用 exec 的使用

到目前为止，我们一直假定进程的正文和数据分别占据可执行文件的独立的段，并因此而占据一个运行进程的独立的区。保持数据和正文的分离有两个好处：保护和共享。如果正文和数据在同一区内，则系统就不能防止一个进程改写它的指令，因为它无法知道哪些地址含有指令，哪些地址含有数据。相反，如果正文和数据在不同的区中，内核就可以打开硬件保护机制来防止进程改写它的正文空间。如果一个进程错误地企图写它的正文空间，它将导致一个保护错，而这一般总要导致进程的终止。

```
#include <signal.h>
main()
{
        int i, *ip;
        extern f(), sigcatch();

        ip=(int*)f;      /* ip 指向函数 f 的地址 */
        for(i =0; i<20; i++)
                signal(i, sigcatch);
        *ip=1;           /* 企图改写 f 的地址 */
        printf("after assign to ip \n");
        f();
}

f()
{
}

sigcatch(n)
        int n;
{
        printf("caught sig %d \n", n);
        exit(1);
}
```

图 7-22　改写自己正文区的程序例

例如，图 7-22 中的程序将函数 f 的地址赋给指针 ip，然后，安排捕俘软中断信号。如果程序被编译为正文和数据在分开的区中，那么，执行这一程序的进程在企图改写 ip 的内容时，

将导致一个保护错。这是因为它要改写它的写保护的正文区。在一台 AT&T 3B20 计算机上，内核向该进程发一个 SIGBUS 软中断信号，但在其他实现版本中可能发不同的软中断信号。进程捕捉到该信号，没有执行在主函数中的打印语句便退出。然而，如果程序被编译为正文和数据在同一区（数据区）中，内核将不能识别进程正在改写函数 f 的地址。函数 f 的地址中将含有值 1。进程执行了主函数中的打印语句，但当它调用 f 时执行的却是一个非法指令。内核将向进程发一个 SIGILL 软中断信号，然后，该进程退出。

将指令和数据放在不同的区有利于对地址错的检测。但是，因为早期 PDP 机对进程大小有个限制，所以那时的 UNIX 系统允许将正文和数据放在一个区内，如果这样做，程序就小一些并需要较少的“段”寄存器。目前的系统对进程大小没有这种严格的限制，因而将来的编译程序不再支持将正文和数据放在同一区内的选择。

将指令和数据分区存放的第二个好处是允许区的共享。如果一个进程不能写它的正文区，则其正文从内核将它当作可执行文件装入时起，便不再变化。因此，如果有几个进程执行同一个文件，那么它们可以共享一个正文区，以节省内存。这样，当内核在 exec 中为进程分配正文区时，它检查可执行文件是否允许它的正文被共享，这由它的魔数来指明。如果允许，内核使用算法 xalloc 来找该文件的已存在的正文区或分配一个新区（见图 7-23）。

在算法 xalloc 中，内核在活动的区表中查找该文件的正文区。如果表中有一个区的索引节点与可执行文件的索引节点相同，则找到该文件正文的一个活动区。如果不存在这样的区，内核则分配一个新区（算法 allocreg），将该区附接到进程上（算法 attachreg），将其装入内存（算法 loadreg），并将其保护方式改为只读。如果进程企图写正文区的话，最后的这步将导致保护错（memory protection fault）。如果在查找活动区表时，内核找到一个含有该文件的正文区，则内核要肯定该区已被装入内存（否则进入睡眠），并将其附接到进程上。内核在算法 xalloc 的最后要解锁该区，并且，当其在系统调用 exec 和 exit 中执行算法 detachreg 时要减少区引用数。

在传统的实现中，系统有一个正文表。内核就像刚刚描述的正文区的管理方法一样管理正文表。因此，正文区的集合可以被看作正文表的现代版本。

内核在系统调用 exec 一开始，就曾调用算法 namei，使索引节点的引用数加 1（在 iget 中），这之后，它还要在算法 altocreg 第一次分配一个区时，增加与该区相联的索引节点引用数（回忆 6.5.2 节）。由于在算法 exec 的最后，内核要调用 iput 使索引节点引用数减 1，这样，被执行的（共享正文）文件的索引节点引用数至少为 1。因此，如果某个进程对该文件做 unlink，则该文件的内容将保持不变。内核把文件装入内存之后就不再需要该文件了，但它需要在区表中有个指针，指向标识该区所对应的文件的内存索引节点。如果索引节点引用数减为零，内核就会将该内存索引节点重新分配给另一个文件，从而破坏了区表中该索引节点指针的意义：如果用户又调用 exec 执行一个新文件，内核将会错误地找到一个老文件的正文区。因此，内核在算法 allocreg 中增加索引节点的引用数，来防止内存索引节点的再分配，从而避免了这个问题。当进程在系统调用 exit 或 exec 中断接该正文区时，内核要在算法 freereg 中多减一次索引节点的引用数，除非该索引节点设置了驻留位。我们将在本章的后面讨论这种情况。

例如，考虑调用 exec 执行图 7-21 中的“/bin/date”。假定该文件具有分开的正文和数据段。该进程最初执行“/bin/date”时，内核为正文分配一个区表项（图 7-24），同时使索引节点引用数为 1（在调用 exec 完成后）。当“/bin/date”退出时，内核调用算法 detachreg

```
算法 xalloc    /* 分配初始化正文区 */
输入：可执行文件的索引节点
输出：无
{
    if（可执行文件没有独立的正文区）
        return;
    if（存在与索引节点的正文相联系的正文区）
    {
        /* 正文区已存在，将它附接到进程 */
        锁区；
        while（该区的内容还未装入）
        {
            /* 修改引用计数，防止完全清除该区 */
            区引用数加 1；
            解锁该区；
            睡眠（事件：区的内容被装入）；
            锁区；
            区的引用数减 1；
        }
        将该区附接到进程（算法 attachreg）；
        return;
    }
    /* 不存在这样的正文区，创建一个区 */
    分配正文区（算法 allocreg）；
    if（索引节点模式设置了驻留位）
        打开区的驻留位标志；
    将该区附接到由索引节点文件头指示的虚地址上（算法 attachreg）；
    if（文件被特别地格式化以用于请求调页系统）
        /* 第 9 章讨论此问题 */
    else  /* 没有为请求调页系统而格式化 */
        将文件正文读入该区（算法 loadreg）；
    将本进程区表中的区保护设置为只读；
    解锁该区；
}
```

图 7-23 分配正文区的算法

和 freereg，将索引节点的引用数减为 0。如果在“/bin/date”第一次被执行时，内核没有增加“/bin/date”索引节点的引用数的话，在程序运行时，索引节点的引用数将为 0 并且被放在空闲链表中。假定另一个进程调用 exec 执行文件“/bin/who”，内核将先前被“/bin/date”所有的内存索引节点分配给“/bin/who”。内核在区表中找到对应于“/bin/who”的索引节点，但实际上找到的是“/bin/date”的索引节点。该进程认为该区含的是文件“/bin/who”的正文，所以它将执行错误的程序。因此，对于运行的共享正文文件的索引节点，其引用数至少为 1。这样，内核就不能重新分配该索引节点了。

共享正文区能使内核通过使用驻留位减少执行一个程序的起动时间。系统管理员可以用系统调用（和命令）chmod 为经常使用的可执行文件设置驻留位文件类型。当一个进程执行一个设置有驻留位的文件时，内核在执行系统调用 exit 或 exec 来断接进程与正文区的联系时，并不释放分配给该正文的内存，尽管该区的引用数已降为 0。虽然该区已不再与任何进程

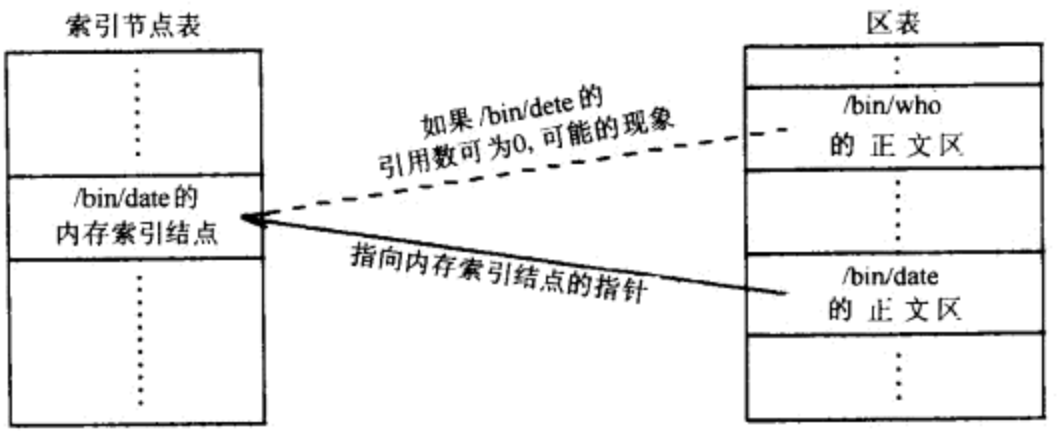


图 7-24　索引节点表和共享正文区表的关系

相关联，但内核保持正文区不变，其索引节点引用数仍为 1。当另一个进程执行该文件时，它找出对应该文件的正文的区表项。进程的起动时间是很短的，因为它并不从文件系统中读该文件；如果正文区还在内存，内核不为正文做任何 I/O 操作；如果内核已将该正文换到一个对换设备上，那么从对换设备上装入正文也要比从文件系统中装入正文快。我们将在第 9 章中看到这一点。

内核在下列情况下清除有驻留位的正文区表项：

(1) 如果一个进程以写方式打开该文件，写操作将改变文件的内容。这使区的内容失效。

(2) 如果一个进程改变文件的许可权方式（chmod），使驻留位不再存在，则文件不应继续留在区表内。

(3) 如果一个进程对文件执行 unlink，因为该文件已不存在于文件系统中，所以没有任何进程能再执行该文件，因而也就没有新进程存取该文件的区表项。因为已不需要该正文区，所以内核可清除它，以释放一些资源。

(4) 如果一个进程拆卸该文件系统，那么该文件就不再可存取了，并且也没有进程可以执行它。所以，在逻辑上应与上一种情况同样处理。

(5) 如果内核用光了对换设备上的空间，内核试图释放当前没有使用的、有驻留位的区来获得可用空间。尽管另一个进程可能很快就需要该正文区，但内核更急需空间。

在前两种情况下，驻留的正文区必须被清除，因为它不再反映文件的当前状态。在后三种情况下，内核清除驻留位表项是因为实用上的需要。当然，内核只有当没有进程正在使用该正文区时（其引用数为 0）才能释放它，否则系统调用 open，unlink 和 umount（情况，1，3 和 4）将失败。

如果一个进程调用 exec 执行它本身，则情况要稍微复杂一些。例如，一个用户键入：

```
sh script
```

shell 进程要创建子进程，子进程调用 exec 运行 shell 来执行文件“script”中的命令。如果一个进程调用 exec 执行它本身，并且允许共享正文区，那么内核必须避免索引节点和区锁上的死锁。也就是说，内核不能对“老”的正文区上锁，保持该锁，然后企图给“新”的区上锁，因为老区和新区是同一个区。内核只是简单地保持着老区与进程的联系，因为不管怎

样，该正文区将被重新使用。

进程通常在系统调用 fork 之后调用 exec；这样，子进程在 fork 中拷贝父进程的地址空间，又在 exec 中放弃该地址空间，去执行一个不同于父进程的程序映象。将这两个系统调用合并为一个系统调用，它调用一个新的程序作为一个新的进程来运行是否更自然一些呢？Ritchie 把将 exec 和 fork 作为两个不同的系统调用的原因归结为：当设计 UNIX 系统时，他和 Thompson 不用对已存在的内核代码做很大的改动就可以把系统调用 fork 加进去（见[Ritchie84a] 1584 页）。但系统调用 fork 和 exec 的分开从功能上来讲也很重要，因为进程能独立地管理它们的标准输入和标准输出文件描述符。这样可以比合并两个系统调用更精巧地建立管道。7.8 节中 shell 的例子将展示这一特性。

7.6 进程的用户标识号

内核将两个用户标识号与一个进程相关联。它们独立于进程标识号。这两个用户标识号分别称为真正用户标识号和有效用户标识号，后者也称为 setuid（设置的用户 ID）。真正用户标识号标识负责运行进程的用户。有效用户标识号用于给新创建的文件赋所有权、检查文件的存取权限和检查通过系统调用 kill 向进程发送软中断信号的许可权限。当进程用 exec 执行一个 setuid 程序时，或当进程显式地调用系统调用 setuid 时，内核允许进程改变它的有效用户标识号。

一个 setuid 程序是一个可执行文件，它的许可权方式字段中的 setuid 位被设置为 1。当一个进程执行一个 setuid 程序时，内核将进程表项中和 u 区中的有效用户标识号设置为文件所有者的标识号。为了区分这两个字段，我们把进程表中的字段称为保存的用户标识号（saved user ID），并将以一个例子说明这两个字段之间的不同之处。

系统调用 setuid 的语法格式是：

```
setuid(uid);
```

其中，uid 是新的用户标识号。它的结果取决于有效用户标识号的当前值。如果调用进程的有效用户标识号是超级用户，内核将使进程表中和 u 区中的真正用户标识号和有效用户标识号被置为 uid。如果调用进程的有效用户标识号不是超级用户，而且如果 uid 的值是真正用户标识号或是保存的用户标识号，则内核将 u 区中的有效用户标识号改为 uid。否则，系统调用返回一个错误。一般来讲，一个进程在系统调用 fork 时继承了它父进程的真正和有效有户标识号，并在系统调用 exec 之后维持不变。

图 7-25 中的程序演示了系统调用 setuid。假定编译该程序时产生的可执行文件的所有者是“maury”（用户标识号是 8319），它的 setuid 位为 1，并且所有用户都有权执行该文件。还假定用户“mjb”（用户标识号是 5088）和用户“maury”分别拥有文件 mjb 和 maury，并且这两个文件对它们的所有者具有只读许可权。用户“mjb”在执行这个程序时可以看到下列输出：

```
uid 5088 euid 8319
fdmjb -1 fdmaury 3
after setuid (5088): uid 5088 euid 5088
fdmjb 4 fdmaury -1
after setuid (8319): uid 5088 euid 8319
```

```
#include <fcntl.h>
main()
{
        int uid, euid, fdmjb, fdmaury;

        uid=getuid();           /* 取真正的 UID  */
        euid=geteuid();         /* 取有效的 UID  */
        printf("uid %d euid %d\n", uid, euid);

        fdmjb=open("mjb", O_RDONLY);
        fdmaury=open("maury", O_RDONLY);
        printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury);

        setuid(uid);
        printf("after setuid(%d):uid %d euid %d\n",uid,getuid(),geteuid());

        fdmjb=open("mjb", O_RDONLY);
        fdmaury=open("maury", O_RDONLY);
        printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury);

        setuid(euid);
        printf("after setuid(%d):uid %d euid %d\n",euid,getuid(),geteuid());
}
```

图 7-25　setuid 程序执行的例子

系统调用 getuid 和系统调用 geteuid 分别返回用户“mjb”的真正用户标识号 5088 和有效用户标识号 8319。所以该进程不能打开文件“mjb”，因为它的有效用户标识号 8319 不具有读该文件的许可权，但它可以打开文件“maury”。在调用了 setuid 重新将该进程的有效用户标识号设置为真正用户标识号（“mjb”）之后，第二个打印语句的输出值是 5088 和 5088，即“mjb”的用户标识号。现在该进程可以打开文件“mjb”了，因为它的有效用户标识号对该文件有读许可权，但它却不能打开文件“maury”。最后，在调用 setuid 重新将有效用户标识号设为该程序保存的有效用户标识号（8319）之后，第三个打印语句又打出 5088 和 8319。最后的这一步显示出一个进程可以执行一个“setuid”的程序，使它的有效用户标识号在它的真正用户标识号和它执行的有效用户标识号之间来回变化。

用户“maury”执行上述程序时，将看到下列输出：

```
uid 8319 euid 8319
fdmjb－1 fdmaury 3
after setuid (8319)：uid 8319 euid 8319
fdmjb－1 fdmaury 4
after setuid (8319)：uid 8319 euid 8319
```

真正用户标识号和有效用户标识号总是 8319：该进程决不可能打开文件“mjb”，但它却可以打开文件“maury”。存放在 u 区中的有效用户标识号由最近一次 setuid 系统调用或执行一个 setuid 程序来产生。它仅用于判定文件的许可权。在进程表中保存的用户标识号使进程通

过执行系统调用 setuid，又可以将有效用户标识号重新置为保存的用户标识号，从而恢复它原来的有效用户标识号。

用户在注册到系统上时所运行的 login 程序是一个典型的调用系统调用 setuid 的程序。login 程序被设置为 root（超级用户）的用户标识号，因此，运行时的有效用户标识号为 root。它询问用户各种问题如注册名、口令等。当一切正确时，它调用系统调用 setuid，将其真正的用户标识号和有效用户标识号置为试图注册的用户标识号（在文件“/etc/passwd”中的相应字段中找到）。login 程序最后执行 shell，该 shell 运动时具有相应用户的真正和有效用户标识号。

命令 mkdir 是一个典型的 setuid 程序。回忆 5.8 节，只有当有效用户标识号为超级用户的进程时才能创建一个目录。为了使普通用户也具有创建目录的能力，mkdir 是一个属于 root（超级用户许可权）的 setuid 程序。当执行 mkdir 时，进程具有超级用户的存取权限，通过算法 mknod 为用户创建一个目录，然后将该目录的所有者和存取权改为真正用户的存取权限。

7.7　改变进程的大小

通过系统调用 brk，一个进程可以增加或减小其数据区的大小。brk 的语法格式是：

```
brk (endds);
```

其中，endds 将成为进程数据区的最高虚地址值（称为 break 值）。另一种方法是调用 sbrk：

```
oldendds = sbrk (increment);
```

其中，increment 为以字节为单位对当前 break 值的改变量；oldendds 是调用前的 break 值。sbrk 是一个调用 brk 的 C 语言库函数。如果调用的结果是增加进程的数据空间，则新分配的数据空间虚拟地接在老数据空间之后，即进程的虚拟地址空间连续地扩展到新分配的数据空间上。内核要检查新的进程大小是否小于系统的最大值，以及新的数据区与原分配的虚地址空间是否没有重叠（图 7－26）。如果检查通过，则内核调用算法 growreg 分配额外的内存

```
算法　brk
输入：新 break 值
输出：老 break 值
{
    为进程的数据区上锁；
    if (增加区大小)
        if (新区大小是非法的)
        {
            解锁数据区；
            return (错)；
        }
    修改区的大小（算法 growreg)；
    清新数据区的地址空间；
    解锁进程的数据区；
}
```

图 7-26　系统调用 brk 的算法

(例如页表)给数据区，并增加进程的大小字段。在一个对换系统上，内核也是试图为新的空间分配内存，并将其内容清零。如果在内存中没有空间了，内核则将进程换出以获得新的空间（第9章将详细讨论)。如果进程调用brk来释放以前分配的空间，则内核要释放内存空间。如果进程存取释放了的页面中的虚地址，则导致一个存储器错。

图7-27给出了使用系统调用brk的程序及在一台AT&T 3B20计算机上运行时的输出。

```
#include<signal. h>
char * cp;
int callno;

main ()
{
    char * sbrk ();
    extern catcher ();

    signal (SIGSEGV, catcher);
    cp = sbrk (0);
    printf ("original brk value %u \ n", cp);
    for (;;)
        *cp++ =1;
}

catcher (signo)
    int signo;
{
    callno++;
    printf ("caught sig %d %dth call at addr %u \ n", signo, callno, cp);
    sbrk (256);
    signal (SIGSEGV, catcher);
}
```

```
original brk value 140924
caught sig 11 1th call at addr 141312
caught sig 11 2th call at addr 141312
caught sig 11 3th call at addr 143360
    … (same address printed out to 10th call)
caught sig 11 10th call at addr 143360
caught sig 11 11th call at addr 145408
    … (same address printed out to 18th call)
caught sig 11 18th call at addr 145408
caught sig 11 19th call at addr 145408
```

图7-27　使用系统调用brk及其输出

在用系统调用signal安排捕俘“段违例”(segmentation violation)软中断信号之后，该进程调用sbrk并打印出它初始的break值。然后，它进入循环，每次循环使字符指针加1并写其内容。直到它企图向一个超出其数据区的地址写，从而引起一个段违例软中断信号。捕俘到该信号之后，函数catcher调用sbrk为数据区再分配256个字节。进程从循环中的断点继续执行，

写入新得到的数据空间。当循环中再次超出数据区时，整个过程又重复一遍。在内存以页分配的机器上，如 3B20，会发生一个有趣的现象。页面是由硬件保护的最小内存单位，所以，当进程写一个超出它的 break 值，但仍在一个页面之内的“准合法”地址时，硬件是无法检测出来的。图 7-27 中的输出反映出这一现象：第一次 sbrk 调用返回 140 924。这说明在该页还剩下 388 字节，在 3B20 机器上，一个页面有 2K 字节。但进程只有当寻址到下一页，即地址 141 312 时，才会出现内存错。函数 catcher 将 break 值增加 256 个字节，新 break 值为 141 180，但仍低于一个页的起始地址。因此，进程又立刻出错并打印出同样的地址，141 312。在又一次调用 sbrk 之后，内核分配了一个新的内存页，因此进程可以寻址另外的 2K 字节，一直到 143 360，尽管这时的 break 值还没有这样高。这样，当进程再次出错时，进程需要调用 8 次 sbrk，才能继续运行。因此，一个进程可以偷越它的法定 break 值，虽然这是不好的程序设计风格。

当用户栈溢出时，内核执行一个类似于 brk 的算法，自动地扩充用户栈。一个进程开始时含有足以存放系统调用 exec 参数的（用户）栈空间。但随着在它执行期间不断将数据压入栈，它的初始栈空间就会溢出。当栈溢出时，机器产生一个存储器错，因为该进程企图存取一个其地址空间之外的存储单元。通过（出错）栈指针的值与栈区大小的比较，内核识别出该存储器错是由于栈溢出而引起的。内核为该栈区分配新的空间，就像上面为系统调用 brk 分配空间一样。当进程从该中断返回时，进程就获得了它运行所必需的栈空间。

7.8 shell 程序

本章已讨论了解释 shell 程序如何工作的足够的材料。虽然 shell 程序要比这里描述的复杂得多，但本节给出的进程间的关系却反映了真实的程序。图 7-28 给出了 shell 程序的主循环，并演示了异步执行、输出的重新定向和管道。

```
/* 读命令行直到“文件尾”  */
while (read(stdin,buffer,numcnars))
{
    /* 分析命令行  */
    if (/* 命令行含有 &  */)
        amper:=1;
    else
        amper=0;
    /* 对于非 shell 命令语言的命令部分  */
    if (fork() ==0)
    {
        /* 重新定向 I/O 否?  */
        if (/* 输出重定向  */)
        {
            fd=creat(newfile, fmask);
            close(stdout);
            dup(fd);
            close(fd);
            /* 标准输出被重定向  */
        }
        if (/* 建立管道  */)
        {
            pipe(flides);
```

图 7-28 shell 程序的主循环

```
        if (fork() ==0)
        {
            /* 命令行的第一个成分 */
            close(stdout);
            dup(fildes[1]);
            close(fildes[1]);
            close(fildes[0]);
            /* 标准输出到管道 */
            /* 子程序执行命令 */
            execlp(command1, command1, 0);
        }
        /* 命令行的第二个成分 */
        close(stdin);
        dup(fildes[0]);
        close(fildes[0]);
        close(fildes[1]);
        /* 标准输出从管道来 */
    }
    execve(command2,command2,0);
}
/* 父进程从此处继续…
 * 如果需要则等待子进程退出
 */
if (amper==0)
    retid=wait(&status);
```

图 7-28 （续）

shell 进程从它的标准输入读入命令行，并按照一组规则对命令进行解释。注册 shell 的标准输入和标准输出通常是用户注册所用的终端，大家在第 10 章中会看到这一点。如果 shell 识别出输入字符串是一个内部命令(build - in command)(例如，命令 cd，for，while 等)，那么它在内部执行该命令，不创建新的进程。否则，它假定该命令是一个可执行文件。

最简单的命令行含有一个程序和一些参数，如：

```
who
grep -n include*.c
ls -1
```

shell 进程调用 fork 创建一个子进程，子进程执行用户在命令行定义的程序。父进程，即用户正在使用的 shell 进程，等待子进程从命令中退出（exit），然后循环回来读下一个命令。

为了异步地运行一个进程（在后台），如

```
nroff -mm bigdocument &
```

当 shell 进程扫描到“&”符号时，它设置一个内部变量 amper。在循环末尾，如果 shell 发现这一内部变量被设置，它就不执行 wait，而是立即重新开始循环并读入下一个命令行。

图中给出了子进程在系统调用 fork 之后存取 shell 命令行的一个副本。为了重新定向标准输出到一个文件，如：

```
nroff－mm bigdocument＞output
```

子进程创建命令行所定义的输出文件，如果创建失败（例如，以错误的许可权在一个目录中创建一个文件），子进程将立即返回。如果创建成功，子进程关闭以前的标准输出文件并复制新的输出文件的文件描述符。标准输出文件描述符现在变为重新定向的输出文件。然后子进程关闭创建该输出文件时得到的那个文件描述符，为了使要执行的程序节省文件描述符。shell 进程以类似的方法重新定向标准输出文件和标准错误文件。

图 7-28 中的代码显示出 shell 处理带有单个管道的命令行的算法，如：

```
ls-1|wc
```

当父进程调用 fork 创建子进程之后，子进程创建一个管道。该子进程再调用 fork，它和它的子进程各处理命令行的一个成分。由第二个 fork 所创建的子子进程执行头一个命令成分（ls）：它写管道，因此它关闭它的标准输出文件描述符，复制管道写描述符。然后再关闭原来的管道写描述符，因为它已无用了。最后创建的子进程（ls）的父进程（wc）是原来的 shell 进程的子进程（见图 7-29）。这个进程（wc）关闭它的标准输入文件，并复制管道的读描述符，使其成为标准输入文件描述符。然后，进程关闭原来的管道读描述符，因为它已不再需要这一描述符了。该进程执行命令行的第二个成分。执行命令行的两个进程异步地执行，一个进程的输出送到另一个进程的输入上。同时，父进程 shell 等待它的子进程（wc）退出，然后再继续象通常一样地执行。当 wc 进程退出后，整个命令行处理结束。shell 进程继续循环，读下一条命令。

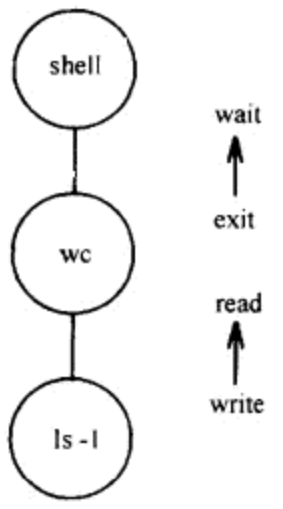


图 7-29 命令行 ls-1|wc 中进程间的关系

7.9 系统自举和进程 init

系统管理员要通过“自举”过程来初始化一个处于非活动状态的系统。自举过程因机器类型的不同而不同。但自举的目的对所有的机器都相同：将操作系统装入主存，并开始执行它。完成这一任务通常需要一系列步骤，因而称为自举。系统管理员可能要设置计算机控制台上的一些开关，来规定一个特殊的硬编码的程序地址，或只需按一下一个按钮，指示机器从它的微代码装入自举程序。这个程序可能仅由几条指令组成，它指示机器执行另一个程序。在 UNIX 系统中，自举过程最终要读一个磁盘的自举块（第 0 块），并将其装入内存。自举块中的程序将内核从文件系统（例如从文件“/unix”或从一个系统管理员定义的另一

个文件）中装入内存。在内核装入内存后，自举程序将控制转到内核的起始地址，内核便开始运行（算法 start，图 7-30）。

内核首先初始化它的内部数据结构。例如，构造空闲缓冲区和索引节点的链表，构造缓冲区和索引节点的散列队列，初始化区结构、页表项等。初始化工作完成之后，内核将根文件系统安装到根（“/”），并为进程 0 形成环境，建立 u 区，初始化进程表的第 0 项，并使根（/）成为进程 0 的当前目录，还有其他一些工作。

```
算法　start          /* 系统初启过程 */
输入：无
输出：无
{
    初始化全部内核数据结构；
    安装根；
    形成进程 0 的环境；
    产生(fork) 进程 1：
    {
        /* 此处为进程 1 */
        分配区；
        将区附接到 init 进程的地址空间；
        扩展区的大小来容纳要拷贝进来的代码；
        将用来执行 init 的代码从内核空间拷贝到用户空间；
        改变状态：从内核态返回用户态；
        /* 由于改变了状态，init 进程从不会达到此处，
         * init 调用 exec 执行 /etc/init，从系统调用
         * 的角度讲，它变成“正常的”用户进程
         */
    }
    /* 进程 0 从此处继续 */
    产生（fork）内核进程；
    /* 进程 0 调用对换程序，
     * 来管理将进程的地址空间分配到内存和对换设备上。
     * 这是一个无限循环；
     * 进程 0 通常在循环中睡眠，除非它有事可做。
     */
    执行对换程序的代码；
}
```

图 7-30　系统自举的算法

当设置好进程 0 的环境之后，系统就作为进程 0 运行。进程 0 调用 fork，因为进程 0 是在核心态运行，所以它从内核内部调用算法 fork。新的进程，进程 1，也运行在核心态。它创建一个数据区，并将其附接到它的地址空间，从而形成它的用户级上下文。进程 1 将该区扩展到合适的大小，然后，简略地说，从内核地址空间拷贝代码到新区：该代码现在形成了进程 1 的用户级上下文。进程 1 设置好保存的用户寄存器上下文，从核心态“返回”用户

态，然后执行其从内核拷贝的代码。与进程0不同，进程1是一个用户级进程。进程0是一个核心级进程，它总是运行在核心态。进程1的正文，即从内核拷贝的代码，由调用系统调用exec以执行程序“/etc/init”的代码组成。进程1按一般的方式调用exec和执行程序。进程1通常被称为初始化（init）进程，因为它负责新的进程的初始化。

为什么进程要将系统调用exec的代码拷贝到进程1的用户地址空间？它本来可以直接从内核调用exec的内部形式，但这要比上面所描述的实现更复杂。为了按后一种方法做，exec要在内核空间分析文件名，而不是像目前实现的那样，只在用户空间分析文件名。这种仅为init所需要的方法，如果作为通用性功能来实现，会使exec的代码复杂化，并且在更一般的情况下会降低它的性能。

进程init（图7-31）是一个进程派遣者。它除了产生其他进程外，还产生一些使用户在系统上注册的进程。进程init读文件“/etc/inittab”来得到关于要产生哪些进程的指示。文件“/etc/inttab”由一些行组成，每行有一个“标识”，即状态标识符（单用户，多用户等），一个“动作”（见习题7.43），和一个程序定义（见图7-32）。进程init读该文件，如果它被调用的状态与某行的状态标识符相匹配，则创建一个进程，执行给定的程序定义。例如，调用init进程的状态是多用户状态（状态2），init进程一般要产生getty子进程来监视配置到系统的终端线路。getty子进程要执行一个注册过程，当用户成功地注册后，再执行一个注册shell。第10章将描述注册shell。同时，init进程执行系统调用wait来监视子进程的死亡，以及由于父进程的退出而产生的“孤儿”进程的死亡。

```
算法 init    /* init进程，即系统的进程1 */
输入：无
输出：无
{
    fd=open("/etc/inittab", O_RDONLY);
    while（从文件中读一行到缓冲区）
    {
        /* 读文件的每一行 */
        if（调用状态！=缓冲区状态）
            continue;
        /* 状态匹配 */
        if (fork()==0)
        {
            /* 这里是子进程 */
            调用exec执行缓冲区规定的程序;
            exit();
        }
        /* init进程不等待，继续while循环 */
    }
    while ((id=wait((int*)0))!==-1)
    {
        /* 检查如果是本进程派生的子进程死，则考虑重新派生该子进程；
           否则，继续while循环； */
    }
}
```

图7-31 init进程的算法

```
Format: identifier, state, action, process specification
Fields separated by colons.
Comment at end of line preceded by '#'

co::respawn:/etc/getty console console          # 机房中的控制台
46:2:respawn:/etc/getty -t 60 tty46 4800H       # 此处为注释
```

图 7-32　文件 inittab 的样本例子

UNIX 系统中的进程有三种：用户进程、守护进程（称为 daemon 进程）及核心进程。在一个典型系统上的大多数进程是用户进程，它们与某个终端相联系。守护进程不与任何用户相关联，而是执行系统功能，如管理和控制网络，执行与时间相关的活动，行打印机假脱机输出，等等。进程 init 可以产生存在于整个系统生存期的守护进程。偶而用户也可以产生这样的进程。这些进程像用户进程一样运行于用户态，像用户进程一样调用系统调用来得到系统的服务。

核心进程只在内核态运行。进程 0 产生内核进程，如页面回收进程 vhand。进程 vhand 进一步变为对换进程。核心进程在提供系统范围服务方面类似于守护进程，但在执行优先权上具有更大的权力，因为它们是内核代码的一部分。它们可以不用通过系统调用而直接存取内核算法和数据结构，因而它们的权力极大。然而，核心进程不如守护进程灵活，因为要改变它们必须重新编译内核。

7.10　本章小结

本章讨论了管理进程上下文和控制进程执行的系统调用。系统调用 fork 通过复制所有与父进程相关联的区来创建一个新进程。实现系统调用 fork 的难点是初始化进程保存的寄存器上下文，这使子进程从系统调用 fork 的内部开始执行，并识别自己是子进程。所有进程都调用系统调用 exit 来终止。exit 断开区与进程的联系，并向它的父进程发一个“子进程死”软中断信号。一个父进程可以用系统调用 wait 使它的执行与子进程的终止相同步。系统调用 exec 允许用户调用其他进程，即用一个可执行文件的内容作为它的地址空间。内核断接老进程的区，并根据可执行文件分配新区。共享正文的文件和驻留位方式改善了内存的利用率和执行一个程序的启动时间。系统允许普通用户通过系统调用 setuid 或 setuid 程序，以其他用户、很可能是超级用户的特权来执行程序。系统调用 brk 允许一个进程改变其数据区的大小。进程通过系统调用 signal 来控制它们对软中断信号的处理。当进程捕俘某一软中断信号时，内核改变用户栈和用户保存的寄存器上下文，为调用软中断信号处理函数做准备。进程可以用系统调用 kill 发送软中断信号，并通过系统调用 setpgrp 来控制发给特殊进程组的信号的接收。shell 进程和 init 进程使用标准的系统调用来提供通常只能在其他操作系统的内核中才能找到的高级功能。shell 进程使用系统调用来解释命令，重新定向标准输入、标准输出和标准错误，产生子进程，在子进程之间设置管道，与子进程同步地执行以及记录命令的退出状态码。类似地，init 进程产生各种子进程，特别是控制终端运行的子进程。如果 init 进程创建的一个子进程退出了，init 进程为执行同样的功能，可以按需要再创建一个

新的子进程，这要在文件“/etc/inittab”中事先定义。

7.11 习题

1. 在一个终端运行图 7-33 中的程序，将它的标准输出重定向到一个文件并比较结果。

```
main()
{
        printf("hello \n");
        if (fork() ==0)
                printf("world \n");
}
```

图 7-33 系统调用 fork 和标准 I/O 程序包

2. 描述图 7-34 中的程序的结果，并与图 7-4 中的程序的结果作比较。

```
#include <fcntl.h>
int fdrd, fdwt;
char c;

main(argc, argv)
        int argc;
        char * argv[];
{
        if (argc ! =3)
                exit(1);
        fork();

        if((fdrd=open(argv[1],O_RDONLY))==-1)
                exit(1);
        if((fdwt=creat(argv[2],0666))==-1)&&
                ((fdwt=open(argv[2],O_WRONLY))==-1))
                exit(1);
        rdwrt();
}

rdwrt()
{
        for(;;)
        {
                if (read (fdrd, &c, 1)! =1)
                        return;
                write(fdwt, &c, 1);
        }
}
```

图 7-34 不共享文件存取的父进程与子进程

3. 重新考虑图 7-5 中的程序，其中两个进程通过一对管道互换消息。如果它们只通过一个管道互换消息会发生什么现象？

4. 一般来讲，一个进程在有机会处理之前，如果收到若干个同一种软中断信号，是否会丢失什么信息？（设想一个进程记录它收到的 interrupt 软中断信号的次数。）这个问题可以解决吗？

5. 描述系统调用 kill 的实现。

6. 图 7-35 中的程序捕俘“子进程死”软中断信号，并像许多软中断信号捕俘函数一样，重新设置软中断信号捕俘函数。程序的结果是什么？

```
# include <signal.h>
main()
{
        extern catcher();

        signal(SIGCLD, catcher);
        if (fork() ==0)
                exit();
        /* pause 挂起执行，直到收到一个信号 */
        pause();
}

catcher()
{
        printf("parent caught sig \n");
        signal(SIGCLD, catcher);
}
```

图 7-35　捕俘子进程死软中断信号

7. 如果一个进程收到了某些软中断信号但并不加以处理，内核要转储进程收到该信号时的内存映象。内核在进程的当前目录中创建一个名为“core”的文件，并拷贝 **u 区**、正文区、数据区和栈区到该文件中。用户以后可以用标准调试工具来检查转储的进程映象。描述内核创建一个“core”文件的算法。如果文件“core”已在当前目录中存在，算法应如何处理？如果多个进程在同一目录下转储，内核应如何处理？

8. 重新考虑图 7-12 中的程序，其中一个进程向另一个进程发送软中断信号，而第二个进程捕俘软中断信号。讨论如果软中断信号处理算法做如下两点变动会产生什么结果：

- 内核不改变软中断处理函数，直到用户显式地要求这样做；
- 内核使进程忽略软中断信号，直到用户重新调用 signal。

9. 重新设计处理软中断信号的算法，使内核自动地安排使一个进程忽略进一步到来的、与正在处理的软中断信号同类的信号，直到软中断信号处理函数返回。内核如何才能发现运行在用户态的软中断信号处理函数返回？这个定义接近 BSD 系统中对软中断信号的处理。

*10. 如果某进程在一个系统调用中睡眠于可被中断的优先级上，这时，收到了一个软中断信号，那么它要跳出（longjmp）该系统调用。如果定义了软中断信号处理函数，内核

要安排进程执行该函数。当进程从软中断信号处理程序返回时，在系统 V 上，就像以一个错误指示（中断的）从系统调用中返回一样。而 BSD 系统自动地为进程重新开始执行系统调用。如何实现这一特性？

11. 命令 mkdir 的传统实现是调用系统调用 mknod 创建一个目录结点，然后调用系统调用 link 两次，将目录项“.”和“..”联结到该目录结点上和该目录结点的父目录上。如果三个操作不完全，该目录则不会有正确的格式。如果正在执行时，mkdir 进程收到了一个软中断信号，将会发生什么情况？如果该软中断信号是不可捕俘的 SIGKILL 又会怎样？如果系统要实现系统调用 mkdir，考虑如何解决这个问题。

12. 当进程进入或离开睡眠状态时（假若它睡眠在可中断优先级上），或当它在完成了系统调用或处理完一个中断之后从内核态返回到用户态时，它要检查软中断信号。为什么进程不在执行系统调用进入内核时检查软中断信号？

13. 假设一个进程在系统调用之后即将返回用户态时，它发现没有收到软中断信号。但就在刚检查之后，内核处理了一个中断并向该进程发了一个软中断信号。（如：一个用户按下〈BREAK〉键。）当内核从中断返回时，进程要做什么？

*14. 如果有几个信号同时发送到一个进程，内核按它们列在手册中的次序处理这些信号。假定对收到的软中断信号有三种可能的响应：捕俘信号；在转储进程映象之后退出；不进行转储即退出。是否有更好的处理并发信号的次序？例如：假若一个进程收到了一个“quit”软中断信号（该信号引起一个内存转储）和一个“interrupt”软中断信号（该信号不会引起内存转储），先处理“quit”信号，还是先处理“interrupt”信号，哪种方法更有意义？

15. 实现一个新的系统调用：

```
newpgrp (pid, ngrp);
```

该系统调用把由进程标识号 pid 所标识的另一个进程的进程组置为新的进程组 ngrp。讨论它可能的用途和这一系统调用的危险。

16. 评论如下说法：在算法 wait 中进程可以睡眠在任何事件上，系统将会正确地工作。

17. 考虑一个新的系统调用的实现：

```
nowait (pid);
```

其中进程标识号 pid 标识了发出该系统调用进程的一个子进程。当调用这个系统调用时，进程通知内核它将不等待该子进程的退出，因此在子进程死亡后内核可以立即清除该子进程的进程表项。内核如何实现这一方案？讨论这样一个系统调用的优缺点并将它与使用“子进程死”软中断信号进行比较。

18. C 语言的连接装配程序自动地在可运行程序中嵌入一个“startup”子程序，这一子程序调用用户程序的主函数（main）。如果用户程序内部没有调用 exit，则 startup 子程序在用户从主函数返回时调用 exit。如果（因为连接装配程序有错）在进程从主函数返回时 startup 子程序没有调用 exit，则会发生什么情况？

19. 当子进程不带参数调用 exit 时，系统调用 wait 得知什么信息？即子进程调用 exit() 而不是 exit(n)。如果一个程序员在调用 exit 时总是不用参数，wait 检查的值如何决定？演示并证明你的结论。

20. 讨论图 7-36 中的进程调用 exec 执行自己时，会发生什么情形？内核如何避免在上锁的索引节点中发生死锁？

```
main(argc, argv)
        int argc;
        char *argv[];
{
        execl(argv[0],argv[0],0);
}
```

图 7-36　一个有趣的程序

21. 习惯上，系统调用 exec 的第一个参数是进程要执行的文件名的最后一个分量。当用户执行图 7-37 时会发生什么情况？如果“a.out”是图 7-36 中的程序经编译后产生的装入模块，则又会发生什么情况？

```
main()
{
        if (fork() == =0)
        {
                execl("a.out", 0);
                printf("exec failed \ n");
        }
}
```

图 7-37　一个非常规的程序

22. 假定 C 语言支持一个新的数据类型——“只读”数据。每当一个进程企图写“只读”数据时，便导致一个保护错。请描述一种实现方法（提示：与共享正文比较）。内核的哪些算法要改动？有哪些其他的实体在实现中要被当作区？

23. 描述对于驻留位文件，算法 open，chmod，unlink 和 umount 要作何变化。例如，当文件被拆除（unlink）时，内核对带驻留位的文件应做些什么工作？

24. 超级用户是唯一有权写口令字文件“/etc/passwd”的用户，这是为了防止恶意或无意的用户破坏文件的内容。程序 passwd 允许用户改变他自己的口令，但它必须保证用户不改变其他人的口令，这个程序如何才能保证这一点？

*25. 执行下列 shell 命令序列，其中“a.out”是一个可执行文件：

```
chmod 4777 a.out
chown root a.out
```

命令 chmod 设置 setuid 位（4777 中的 4），并且，习惯上所有者“root”是超级用户。执行这个命令系列能否造成安全性方面的缺口？

27. 运行图 7-38 中的程序会发生什么情形？为什么？

28. 库函数 malloc 通过调用系统调用 brk 给进程分配更多的数据空间；库函数 free 释放以前由 malloc 分配的存储空间。这两个库函数的格式为：

```
main()
{
        char *endpt;
        char *sbrk();
        int brk();

        endpt=sbrk(0);
        printf("endpt=%ud after sbrk\n",(int) endpt);

        while (endpt--)
        {
                if (brk(endpt)==-1)
                {
                        printf("brk of %ud failed\n", endpt);
                        exit();
                }
        }
}
```

图 7-38　一个紧缩进程空间的例

```
ptr=malloc (size);
free (ptr);
```

其中 size 是一个无符号整数，表示要分配的字节数；ptr 是一个指向新分配空间的字符指针。当 ptr 用作 free 的参数时，它必须是由以前的 malloc 返回的指针。请实现这个库函数。

29. 当运行图 7-39 中的程序时，会发生什么情况？请和系统手册中所说的结果作一比较。

```
main()
{
        int i;
        char *cp;
        extern char *sbrk();

        cp=sbrk(10);
        for (i=0;  i<10;  1++)
                *cp++='a'+i;
        sbrk(-10);
        cp=sbrk(10);
        for(i=0;  i<10;  i++)
                printf("char %d='%c'\n", i, *cp++);
}
```

图 7-39　一个简单的 sbrk 的例子

30. 当 shell 创建一个新进程去执行一个命令时，它如何知道该文件是可执行的？如果是可执行的，它如何区分是 shell 程序还是由编译产生的文件？检查上述情况的正确顺序是什么？

31. shell 的符号“≫”将输出追加到一个指定的文件。

例如：

```
run≫outfile
```

如果文件“outfile”不存在，该命令创建该文件并写（write）该文件，否则，它打开（open）该文件并在已经存在的数据后面追加写。请写出实现“≫”的代码。

32. shell 测试一个进程调用 exit 所返回的码，视零值为真，非零值为假（注意与 C 不一致）。假定图 7-40 所示的程序的可执行文件名是 truth。请描述 shell 执行下面的循环时所发生的情况。请修改该 shell 代码来处理这种情况。

```
while truth
do
truth &
done
```

```
main()
{
        exit(0);
}
```

图 7-40　truth 程序

33. 为什么 shell 必须按指定的顺序创建进程，来处理一个管道线中的两个命令成分（图 7-25）？

34. 使 shell 循环的样本程序在处理管道方面更一般化，也就是说，使它能处理命令行上任意数目的管道。

35. 环境变量 PATH 给出了 shell 查询的可执行文件所用的目录的有序集合。库函数 execlp 和 execvp 将 PATH 中所列出的目录加到不以斜杠“/”字符开始的文件名参数前面。请实现这两个函数。

＊36. 一个超级用户应设置好 PATH 环境变量，使 shell 不在当前目录中查找可执行文件。如果它在当前目录中查找可执行文件的话，存在什么安全性问题？

37. shell 如何处理命令 cd(改变目录)？对文件行：

```
cd pathname &
```

shell 做什么工作？

38. 当用户在终端按下“delete”或“break”键时，终端驱动程序发一个“interrupt”软中断信号给该注册 shell 组中所有的进程。用户很可能只想终止该 shell 进程所产生的那些子进程而不是退出注册。图 7-28 中的 shell 循环应做什么改动？

39. 用户可以键入命令

```
nohup command _ line
```

从而不允许“command _ line”产生的进程接收“hangup”软中断信号和“quit”软中断信号。图 7-28 中的 shell 循环应如何处理这一点？

40. 考虑下列 shell 命令序列：

```
nroff －mm bigfile1 > big1out &
nroff －mm bigfile2 > big2out
```

重新考虑图 7-28 所给出的 shell 循环。假如第 1 个 nroff 命令在第 2 个 nroff 命令之前执行完，会发生什么情况？应如何修改 shell 循环程序来正确地处理这一情况？

41. 用 shell 执行一个尚未调试的程序时，由 shell 打印出的一个常见错误信息是“Bus error－core dumped.”，即总线错——内存被转储。程序显然做了什么非法的事，shell 又如何知道它应打印一个出错信息呢？

42. 只有进程 init 可以做为进程 1 在系统上运行。然而，一个系统管理员可以调用命令 init 来改变系统的状态。例如，当系统自举后进入单用户状态，这意味着控制台终端是活动的，而用户终端却不活动。一个系统管理员在控制台终端打入：

```
init 2
```

将 init 进程的状态改为 2（多用户）。控制台 shell 调用 fork 并执行 init 算法。如果一个系统只有一个 init 进程可以活动，系统将会发生什么情况？

43. 文件“/etc/inittab”的格式允许定义一个与每一产生的进程相关联的动作。例如，与 getty 进程相关联的动作是“respawn”。这意味着在该进程死后，进程 init 应重新创建该进程。实际上这意味着用户在退出注册后，进程 init 要再创建另一个 getty 进程，以允许另一用户存取当前活动的终端线路。进程 init 如何才能实现“respawn”动作呢？

44. 某些内核算法需要查进程表，使用父指针、子指针和兄弟指针可以改善查找时间。父指针指向进程的父进程，子指针指向任意子进程，兄弟指针指向其他具有同一父进程的进程。一个进程根据子指针和兄弟指针可以找到它所有的子进程（循环是非法的）。哪些算法得益于这一实现？哪些算法并不得益于这一实现？

第 8 章　进程调度和时间

在分时系统中，内核给每个进程分一段 CPU 时间，这段时间称为时间片。当这段时间过去时，内核抢先该进程并调度另一个进程。过一会儿后，内核会重新调度该进程继续运行。UNIX 系统上的调度函数使用运行的相对时间作为下次调度哪个进程的参数。每个进程都有一个调度优先权。当内核做上下文切换时，它选取有最高优先权的进程，切换到该进程的上下文。当运行的进程从核心态转到用户态时，内核重新计算它的优先权，并定期地调整在用户态“就绪”的每个进程的优先权。

有些用户也有必要知道关于时间的信息。例如，命令 time 打印出执行另外一个命令所花费的时间，命令 date 给出日期和时间。有许多与时间有关的系统调用允许进程设置或查询内核的时间值或确定进程的 CPU 使用量。系统通过一个硬件时钟计时。这个硬件时钟以一个固定的、依赖于硬件的频率中断 CPU，一般这个频率是在每秒钟 50 到 100 次之间。每一次时钟中断称为一个时钟滴答（clock tick）。本章将讨论 UNIX 系统上与时间有关的活动，诸如进程调度、有关时间的系统调用以及时钟中断处理程序的功能等。

8.1　进程调度

UNIX 系统上的调度程序属于操作系统调度程序中常用的一种，叫作多级反馈循环调度（round robin with multiple feedback）。这种调度方法的意思是，内核给进程分一个 CPU 时间片，抢先一个超过其时间片的进程，并把它反馈到若干优先级队列中的某一个队列。一个进程在它结束之前，可能需要多次通过“反馈循环”。当内核做上下文切换和恢复一个进程的上下文时，该进程从它原来被挂起的地方继续执行。

8.1.1　算法

在上下文切换结束时，内核执行图 8-1 中所示的算法，来调度一个进程，即从处于“在内存中就绪”和“被抢先”状态的进程中，选取优先权最高的进程。选取一个尚未装入内存的进程是没有意义的，因为直到该进程被换入之前，它不能运行。如果若干进程都具有最高优先权，内核按循环调度策略选择在“就绪”状态时间最长的进程。如果没有合格的进程，内核则休闲，直到下次中断，下次中断最迟发生在下一个时钟滴答时。在处理完中断后，内核再次调度一个进程去运行。

8.1.2　调度参数

每个进程都有一个优先权域，用于进程的调度。在用户态的进程的优先权是它最近使用 CPU 时间的函数。最近使用过较多 CPU 时间的进程得到较低的优先权。进程优先权的范围分为两种（见图 8-2）——用户优先权和核心优先权。每种优先权有若干优先权值（或称为优先数），每个优先权都有一个逻辑上与它相关联的进程队列。具有用户级优先权的进程在它们从核心态返回到用户态时被抢先，而得到它们的用户级优先权；而具有核心级优先权的

```
算法　schedule _ process
输入：无
输出：无
{
    while（没有能被选取运行的进程）
     {
        for（就绪队列中的每个进程）
            取已装入内存的、优先权最高的进程；
        if（没有合格运行的进程）
            机器休闲；
            /*　下次中断使机器脱离休闲状态　*/
     }
    将选取的进程从就绪队列中移出；
    切换到被选取的进程的上下文，恢复其执行；
}
```

图 8-1　进程调度的算法

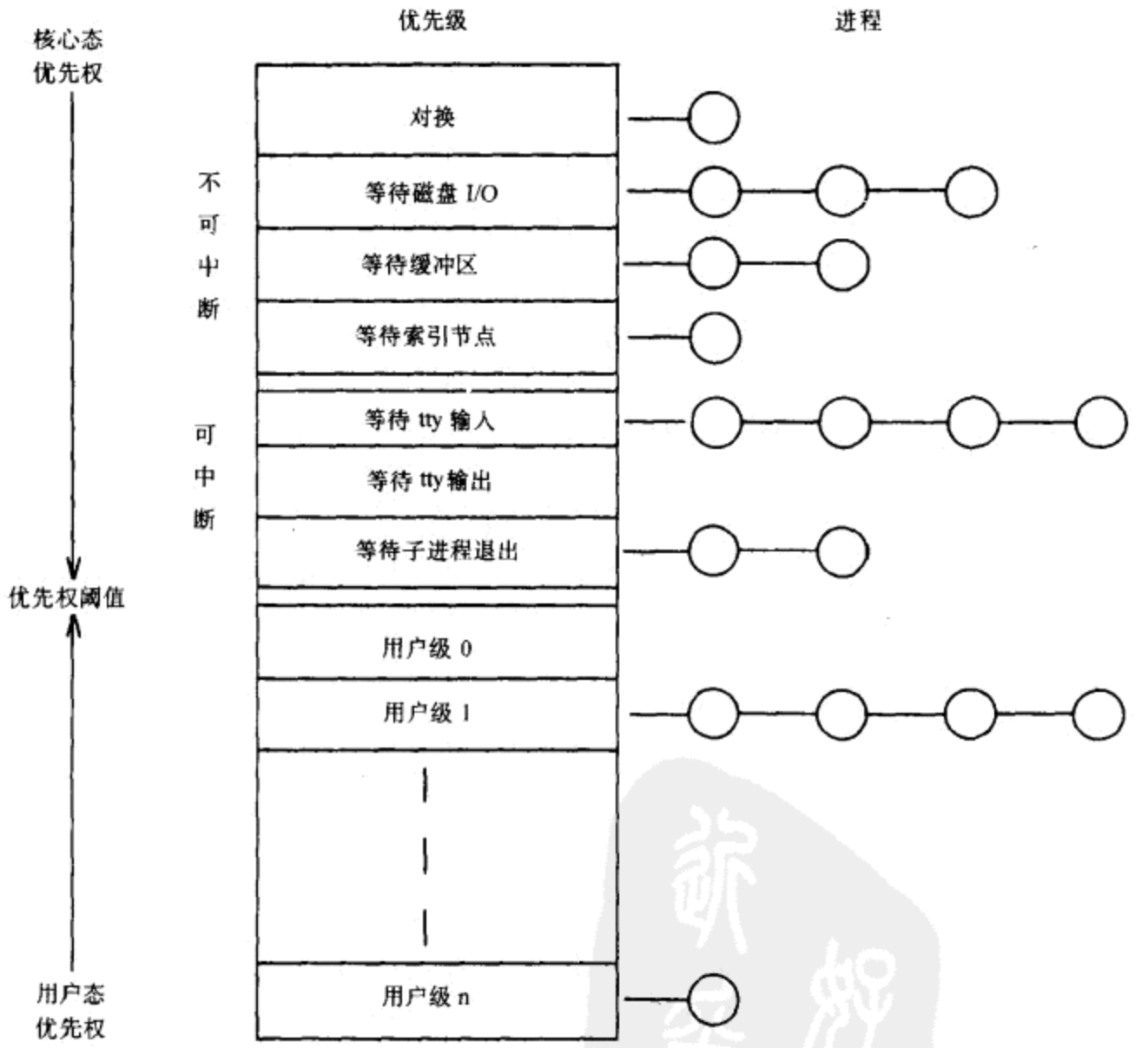


图 8-2　进程优先权的范围

进程是在算法 sleep 中得到核心级优先权的。用户级优先权低于一个阈值，核心级优先权高于该阈值。核心级优先权又可进一步划分为：在收到一个软中断信号时，具有低核心优先权的进程可被唤醒；有高核心优先权的进程却继续睡眠（见 7.2.1 节）。

图 8-2 给出了优先权阈值，它是在用户优先权和核心优先权之间，以“等待子进程退出”和“用户级 0”这两个优先级之间的双划线表示的。图中，被称为“对换”，“等待磁盘 I/O”，“等待缓冲区”和“等待索引节点”的优先级是不可中断的系统高优先级，分别有 1，3，2，和 1 个进程在排队。被称为“等待 tty 输入”，“等待 tty 输出”和“等待子进程退出”的优先级是可中断的系统低优先级，分别有 4，0，和 2 个进程在排队。该图将用户优先权分为“用户级 0”、“用户级 1”、直到“用户级 n”[⊖]，分别有 0，4，和 1 个进程在排队。

内核在特定的进程状态下计算一个进程的优先权。

• 内核将优先权赋给一个即将进入睡眠的进程，也就是说，将一个固定的优先权值和睡眠的原因联系起来。该优先权不依赖于进程的运行时特性（I/O 型的，或 CPU 型的），而是依照睡眠原因而定的常量。该常量是为每个 sleep 调用而硬编码的。在低层算法中睡眠的进程的不活动的时间越长，就越容易引起系统瓶颈问题，因此，它们比可能引起较少系统瓶颈的进程得到较高的优先权。举例说来，一个睡眠等待磁盘 I/O 的进程比等待一个自由缓冲区的进程具有较高的优先权。这是因为，首先，等待磁盘 I/O 完成的进程已经有了缓冲区，当它醒来时，它就有机会做足够的处理，从而释放该缓冲区以及可能的其他资源。它释放的资源越多，其他进程的运气就会越好，不用挤在那里等待资源。这样，系统将少做一些上下文切换，从而进程的响应时间和系统的吞吐量就会越好。其次，一个等待自由缓冲区的进程，可能正在等待由一个等待 I/O 完成的进程所占用的缓冲区。当这个 I/O 完成时，这两个进程都会醒来，因为它们是睡眠在同一个地址上。如果让等待该缓冲区的进程先运行，它会再次进入睡眠，直到另一个进程释放该缓冲区；因此，这个进程的优先权应该比等待 I/O 完成的进程的优先权低一些。

• 内核调整从核心态返回到用户态的进程的优先权。该进程以前可能已经进入了睡眠状态，其优先权已变到一个核心级优先权，该核心级优先权必须在返回用户态时被降低到用户级优先权，同时，为对其他进程公平起见，内核要处罚该进程，因为它刚刚占用过宝贵的内核资源。

• 时钟处理程序以 1 秒钟的间隔（在系统 V 上）调整用户态下的所有进程的优先权，并使内核运行调度算法，以防止某个进程垄断 CPU 的使用。

在一个进程的时间片中，时钟可能使它中断若干次，每次时钟中断时，时钟处理程序都要使该进程表中的一个字段增值。该字段记录了该进程的最近 CPU 使用时间。时钟处理程序还每秒一次地根据一个衰减函数：

```
decay(CPU)=CPU/2;
```

来调整每个进程的最近 CPU 使用时间（在系统 V 上）。当时钟处理程序重新计算最近 CPU 使用时间时，它还根据公式：

⊖ 在系统中，最高优先权的值是 0。因此，用户级 0 比用户级 1 具有更高的优先权。其他依此类推。

priority = ("recent CPU usage"/2) + (base level user priority)

重新计算在“被抢先”和“就绪”状态下的每个进程的优先权。式中，priority 为优先权值，recent CPU usage 为最近 CPU 使用时间，base level user priority 为基级用户优先权值，即上面曾说明过的，在核心态和用户态之间的优先权阈值。数字上的低值意味着高的调度优先权。考虑上面计算最近 CPU 使用时间的公式和计算进程优先权值的公式，我们可以看出，最近使用 CPU 的时间衰减得越慢，一个进程的优先权达到其基级优先权的时间就越长，因此，处于“就绪”状态的这些进程将经历更多的优先级。

每秒一次地重新计算优先权值的效果是，具有用户级优先权的进程在优先队列之间移动，如图 8-3 中所示。将图 8-3 和图 8-2 比较可知，有一个进程已经从用户级优先权为用户级 1 的队列转移到用户级 0 的队列。在一个实际系统中，图中所示的所有具有用户级优先权的进程都会改变优先级队列，但图中只画了一个。内核并不改变核心态进程的优先权，也不允许具有用户级优先权的进程跨越优先权阈值而获得核心级优先权，除非这些进程做系统调用并进入睡眠。

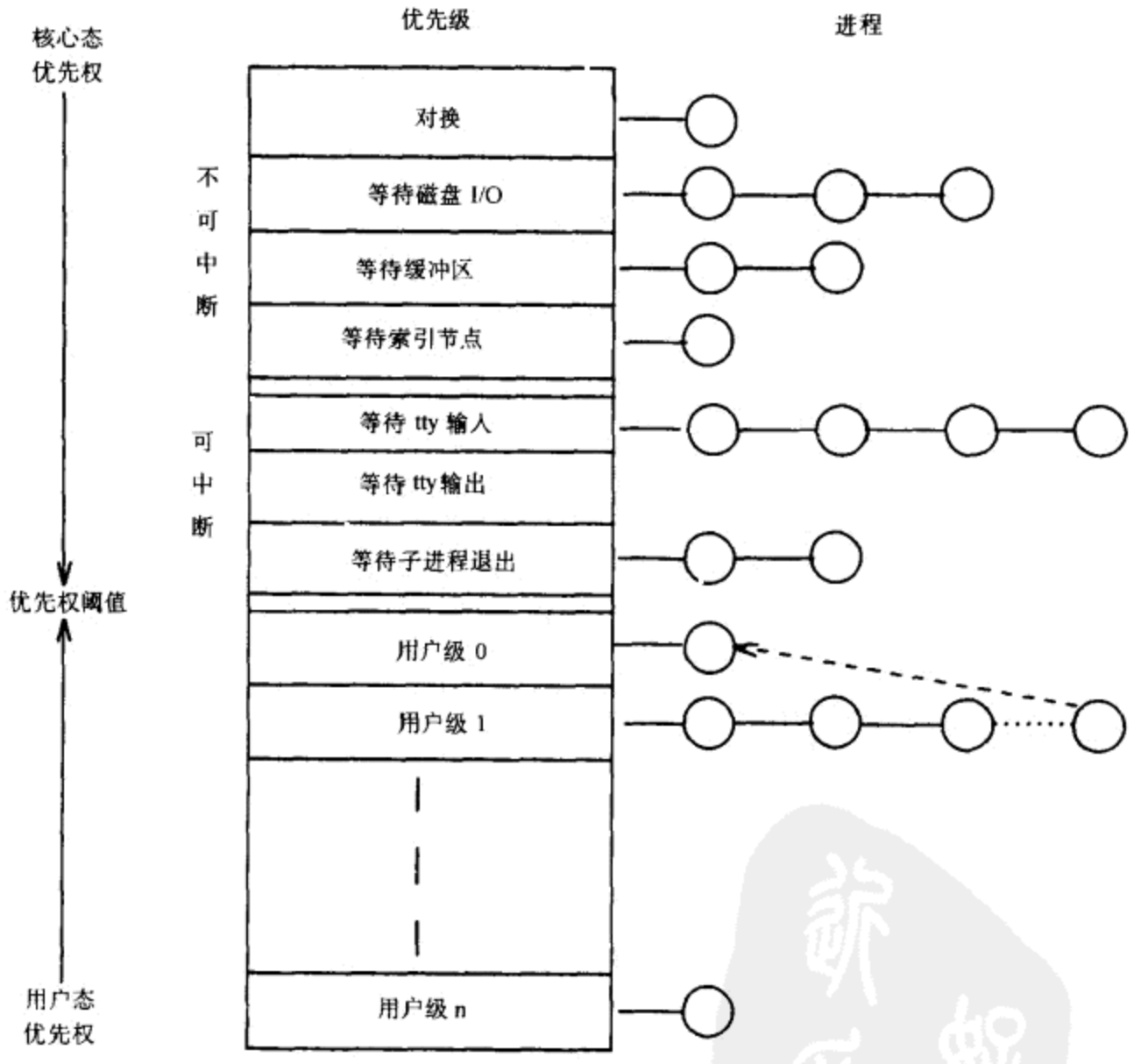


图 8-3　进程在优先级队列中的移动

内核试图每秒一次地重新计算所有活动进程的优先权。但这个时间间隔可以稍有变化。

当内核正在执行一段临界区（critical region）代码时，如果时钟中断已经到来（也就是说，当处理机优先级已被提高，但显然还没有高到足以屏蔽时钟中断时），内核并不重新计算优先权，因为那会使内核在临界区呆的时间过长。相反，内核只是记住它应该重新计算进程优先权，并当“以前”的处理机执行级降低之后的一次时钟中断时，做这次进程优先权的重新计算。周期地重新计算进程优先权确保了对在用户态执行的进程实行循环调度策略。内核对象文本编辑或表格输入这样的程序的交互式请求做出自然的响应：这样的进程具有较高的CPU休闲对CPU使用的比率，因而，当它们就绪运行时，它们的优先权值自然会提高（[Thompson 78]1973页）。调度机制的其他实现还使时间片在0和1之间动态地变化，这取决于系统的负载。因而，这样的实现对进程有较快的响应，因为它们不必一直等到1秒钟才运行；另一方面，由于额外的上下文切换，内核的开销较高。

8.1.3　进程调度的例子

在如下的假设下，图8-4给出了在系统Ⅴ上三个进程A、B、C的调度优先权：这三个

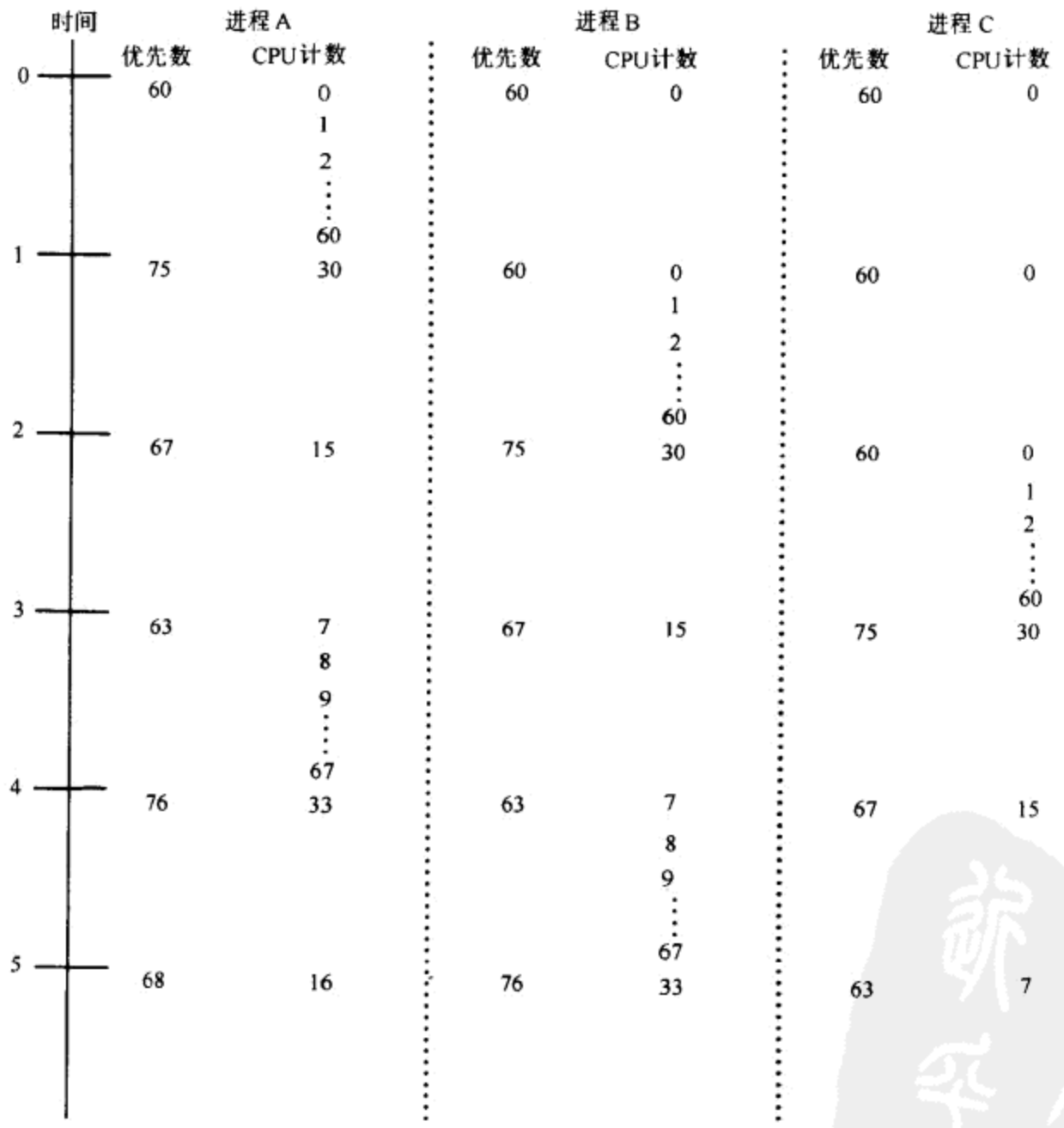


图8-4　进程调度的例子

进程是同时创建的，初始优先数是 60，最高的用户级优先数是 60（即最小优先权）。时钟每秒中断系统 60 次，这些进程没做任何系统调用，也没有其他进程就绪运行。内核由

```
CPU=decay(CPU)=CPU/2;
```

计算 CPU 使用时间的衰减值；由

```
priority=(CPU/2)+60;
```

计算进程的优先数。假定进程 A 先开始运行。它从一个时间片的开头开始，运行了 1 秒钟：在这段期间里，时钟使系统中断了 60 次，中断处理程序使进程 A 的 CPU 使用字段增值了 60 次（到 60）。内核在标志为 1 秒钟的地方强行做上下文切换，并且，在重新计算了所有进程的优先权值后，调度进程 B 去运行。在下 1 秒钟，时钟处理程序使进程 B 的 CPU 使用字段增值 60 次。然后，重新计算所有进程的优先权值并强行做上下文切换。按这种形式重复下去，内核轮换执行这三个进程。

现在，考虑具有图 8-5 所示的优先权的进程。假定系统中还有其他进程。在进程 A 已经连续地获得几个 CPU 时间片后，内核可能抢先进程 A，使它处于“就绪状态”，它的用户级

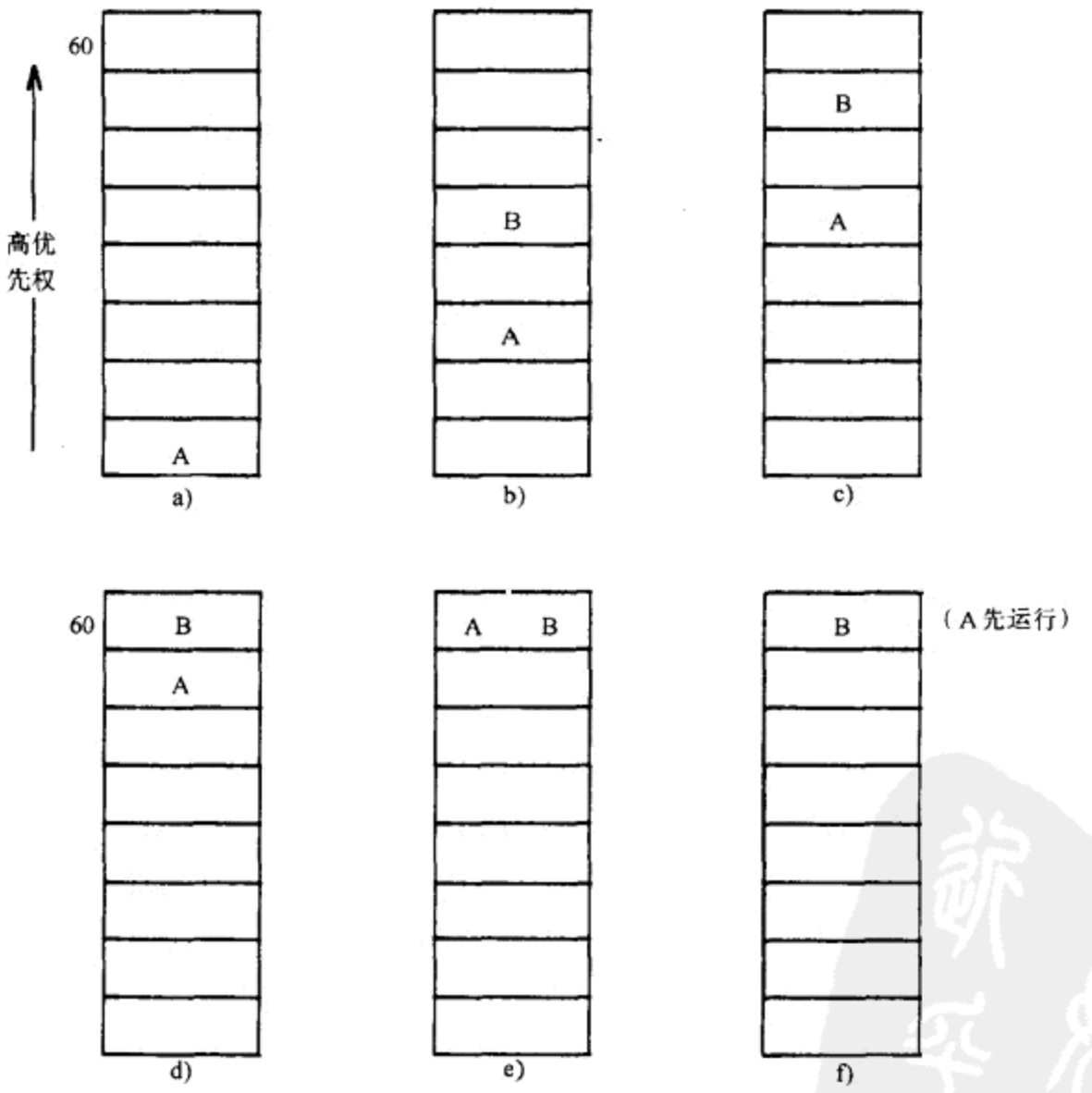


图 8-5　循环调度及进程的优先权

优先权可能很低（图 8-5a）。随着时间的继续，进程 B 可能进入就绪状态，此时，它的用户级优先权要比进程 A 的优先权高（图 8-5b）。如果内核在一段时间里没有调度这两个进程中的任何一个（它调度其他进程），那么这两个进程逐渐地都会达到同一用户优先级，尽管 B 可能会先进入那一级，因为它的起始级别一开始就离这个同一级别近一些（图 8-5c、图 8-5d 和图 8-5e）。然而，内核可能会先于进程 B 而调度进程 A，因为进程 A 在“就绪”状态的时间较长（图 8-5f）：这是对具有相同优先权的进程进行调度的原则。

请回顾 6.4.3 节，内核在上下文切换结束时调度一个进程：当进程进入睡眠或退出（exit）时，它必须做上下文切换；当它从核心态返回到用户态时，有机会做上下文切换。内核抢先一个正从核心态返回到用户态的进程，如果另一个具有较高优先权的进程就绪的话。当内核唤醒了一个比当前运行的进程具有更高优先权的进程时，或当时钟处理程序修改所有“就绪”进程的优先权时，也会存在这样的进程。在第一种情况下，既然可以得到一个具有较高优先级的核心态进程，那么，当前运行的进程就不应该继续在用户态运行。在第二种情况下，时钟中断处理程序确定正在运行的程序已经用尽了它的时间片，而且，因为许多进程的优先权已被修改，于是，内核做上下文切换，来重新调度一个进程。

8.1.4　进程优先权的控制

使用系统调用 nice：

```
nice(value);
```

进程可以自然地实行对它们的调度优先权的控制。这里，value 被加到进程优先权值的计算公式中：

```
priority = ("recent CPU usage"/constant) + (base priority) + (nice value)
```

系统调用 nice 按参数提供的值，增加或减少进程表中的 nice 字段，尽管只有超级用户才能提供提高进程优先权的 nice 值。同理，只有超级用户才能提供低于指定阈值的 nice 值。当执行“强计算”型作业时，通过调用 nice 来降低他们的进程优先权的用户，对系统中其他用户来说是友好的，系统调用 nice 由此得名。在系统调用 fork 期间，进程继承它们的父进程的 nice 值。系统调用 nice 仅作用于正在运行的进程；一个进程不能重新设置另一个进程的 nice 值。实际上，这意味着，如果由于一些进程消耗了太多的时间，一个系统管理员想降低它们的优先权值，那么，除了直接地将它们扼杀（kill）以外，没有其他方法。

8.1.5　公平共享调度

上述调度算法对不同类型的用户不做区分，也就是说，不可能将 CPU 时间分一半给一组特殊的进程，即使想要这样做。然而，在一个计算中心的环境中，考虑这一点是很重要的。在计算中心，一组用户可能想在有保证的前提下，租用一个机器的一半 CPU 时间，来确保一定级别的响应时间。本节描述一种叫做公平共享调度（fair share scheduler）的策略，是在 AT&T 贝尔实验室的 Indian Hill 计算中心实现的 [Henry 84]。

公平共享调度的原则是将用户团体分为一些公平共享组。这样，每组的成员，相对于组

中的其他进程而言，受到常规的进程调度的限制。然而，系统将 CPU 时间按比例分给每个组，并不考虑组中成员的多少。例如，假定在一个系统中有四个公平共享组，每组分 25% 的 CPU 时间，各组分别含有 1，2，3 和 4 个 CPU 型进程，这些进程从不自动地放弃处理器（例如，它们都处于无限循环中）。假定系统中没有其他进程，若按常规的调度算法，则 4 个组中的每个进程将得到 10% 的 CPU 时间（共有 10 个进程），因为没有其他方法能将它们区分开来。但是，用公平共享调度方法，在第 1 组中的进程所得到的 CPU 时间是第 2 组中的进程的两倍，是第 3 组中的进程的 3 倍，是第 4 组中的进程的 4 倍。在这个例子中，同组中的所有进程的 CPU 时间，在时间上讲，是相等的，因为它们都在一个无限循环中。

这种策略的实现方法是简单的，而下面这个特点使其更加吸引人：在计算进程优先权值的公式中，加进了一个称为“公平共享组优先权”的项。每个进程在其 u 区有个新字段，指向一个公平共享 CPU 使用字段，由该公平共享组中的所有进程共享。时钟中断处理程序就像它增加运行进程的 CPU 使用字段一样，增加运行进程的公平共享 CPU 使用字段，并每秒一次地衰减所有公平共享组 CPU 使用字段的值。在计算进程优先权值时，一个新的计算成分是共享组的 CPU 使用量，该值根据分配给该公平共享组的 CPU 时间量而被归一化(normalize)。一个组中的进程最近得到的 CPU 时间越多，该组的 CPU 使用字段的数字值就越高，因而，根据公式计算得到的优先权的值就越高，这意味着该公平共享组中的所有进程的优先权就越低。

以图 8-6 中描述的三个进程为例。假定进程 A 在一个组，进程 B 和 C 在另一个组。假

时间	进程 A			进程 B			进程 C		
	优先权值	CPU	组	优先权值	CPU	组	优先权值	CPU	组
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		⋮	⋮						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					⋮	⋮			⋮
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		⋮	⋮						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						⋮		⋮	⋮
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		⋮	⋮						
		78	78						
5	98	39	39	70	3	18	76	15	18

图 8-6　公平共享调度的例子——两组，三个进程

定内核先调度进程 A，它在下一秒钟将增加进程 A 的 CPU 使用字段和组的 CPU 使用字段。在 1 秒钟的标志处，中断处理程序重新计算进程的优先权，进程 B 和 C 得到最高优先权；假定内核调度进程 B。在下一秒钟期间，进程 B 的 CPU 使用字段达到 60，B 和 C 的组使用字段也达到 60。因此，在第 2 秒钟重新计算优先权时，进程 C 的优先数是 75（请比较图 8-4），内核将调度进程 A，其优先数为 74，因此，优先权比 B 和 C 都高。该图给出了这个模式是如何重复的：内核按 A，B，A，C，A，B 等等这样的次序进行调度。

8.1.6　实时处理

实时处理指的是对外部事件进行立即响应，从而，在该事件发生后的一个指定时间内，调度指定的进程去运行。例如，一个计算机用来作为监控医院病人的生命支持系统，在病人的状态发生某种变化时要采取立即行动。像文本编辑这样的进程不被考虑为实时进程：虽然对于用户来讲，响应时间越快越好，但不像等不得额外的几秒钟那样严重（尽管用户可能不这么想）。上面描述的调度算法是为一个分时环境设计的，它们在实时环境中并不合适，因为它们不能保证内核在一个固定的时间限度内，调度一个指定的进程。支持实时处理的另一个障碍是，内核是不可抢先的：核心态如果正在执行一个核心态进程，它就不能调度一个用户态的实时进程，除非做大的改动。目前，系统程序员必须将实时进程插到内核中，来获得实时响应。对该问题的真正解决方法是必须允许实时进程动态地存在（即，不是在内核中硬编码的），给它们提供一种机制来通知内核关于它们的实时要求。目前，还没有标准的 UNIX系统具有这种能力。

8.2　有关时间的系统调用

与时间有关的系统调用有：stime，time，times，alarm。头两个是关于全局的系统时间，后两个是关于单个进程的时间。

stime 允许超级用户将一个表示当前时间的值赋给一个内核全局变量：

```
stime(pvalue);
```

这里，pvalue 指向一个以秒为单位的长整数时间，是从格林威治时间 1970 年 1 月 1 日零点开始的，时钟中断程序每秒钟一次地使该变量加 1。time 查询由 stime 所设置的时间：

```
time(tloc);
```

这里 tloc 指向一个用户进程中用做返回值的单元，time 从系统调用中返回这个值。像 date 之类的命令则利用 time 来决定当前时间。

times 给出调用进程在用户态和核心态执行时所花费的累积时间以及它的所有僵死子进程在用户态和核心态执行时曾花费的累积时间。该调用的语法格式是：

```
times(tbuffer);
struct tms * tbuffer;
```

这里，tms 含有查询到的时间，它由下面的结构来定义：

```
struct tms {
        /*   time_t 是用于时间的数据结构   */
        time_t tms_utime;       /*   进程的用户时间   */
        time_t tms_stime;       /*   进程的内核时间   */
        time_t tms_cutime;      /*   子进程的用户时间   */
        time_t tms_cstime       /*   子进程的内核时间   */
};
```

times 返回“从过去的一个任意时刻”开始所经过的时间，通常是系统初启的时间。

在图 8-7 的程序中，一个进程创建了 10 个子进程，每个子进程循环 10 000 次。该父进程

```
#include   <sys/types.h>
#include   <sys/times.h>
extern long times ();

main ()
{
     int i;
     /* tms 是有 4 个时间元素的数据结构 */
     struct tms pb1, pb2;
     long pt1, pt2;
     pt1=times (&pb1);
     for (i=0; i<10; i++)
          if (fork () ==0)
               child (i);
     for (i=0; i<10; i++)
          wait ( (int *) 0);
     pt2=times (&pb2);
     printf ("parent real %u user %u sys %u cuser %u csys %u\n",
          pt2-pt1, pb2.tms_utime-pb1.tms_utime, pb2.tms_stime-pb1.tms_stime,
          pb2.tms_cutime-pb1.tms_cutime, pb2.tms_cstime-pb1.tms_cstime);
}

child (n)
     int n;
{
     int i;
     struct tms cb1, cb2;
     long t1, t2;
     t1=times (&cb1);
     for (i=0; i<10000; i++)
          ;
     t2=times (&cb2);
     printf ("child %d: real %u user %u sys %u\n", n, t2-t1,
          cb2.tms_utime-cb1.tms_utime, cb2.tms_stime-cb1.tms_stime);
     exit ();
}
```

图 8-7　使用系统调用 times 的程序

在创建子进程之前调用了 times，在所有子进程都退出（exit）后，又调用了 times。所有子进程在它们的循环之前和之后也调用了 times。有人可能天真地以为，父进程的子进程用户

时间等于它的各个子进程的用户时间之和；父进程的子进程系统时间等于其各个子进程的系统时间之和。然而，子进程的时间不含花费在系统调用 fork 和 exit 上的时间，并且，花费在中断处理和做上下文切换上的时间会给所有的时间统计造成偏差。

用户进程可利用系统调用 alarm 来设置闹钟软中断信号。例如，图 8-8 中的程序每分钟检查一次一个文件的存取时间，如果文件曾被访问过，则打印出一条信息。为做到这一点，该程

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>
main (argc, argv)
        int argc;
        char * argv [];
{
        extern unsigned alarm ();
        extern wakeup ();
        struct stat statbuf;
        time_t axtime;
        if (argc ! =2)
        {
             printf ("only 1 arg\n");
             exit ();
        }

        axtime= (time_t) 0;
        for (;;)
         {
             /* 查文件存取时间 */
             if (stat (argv [1], &statbuf) ==-1)
              {
                  printf ("file %s not there\n", argv [1]);
                  exit ();
             }
             if (axtime ! =statbuf.st_atime)
              {
                  printf ("file %s accessed\n", argv [1]);
                  axtime=statbuf.st_atime;
             }
             signal (SIGALRM, wakeup);        /* 重置闹钟 */
             alarm (60);
             pause ();          /* 睡眠等待软中断信号 */
        }
}
wakeup ()
{
}
```

图 8-8　使用系统调用 alarm 的程序

序进入一个无限循环：在每次循环期间，它调用 stat，报告该文件上次被访问的时间，如果在上一分钟内，该文件曾被访问过，则打印出一条信息。然后，该进程调用 signal 去捕俘闹钟信号，调用 alarm 来安排每 60 秒一次的闹钟信号，调用 pause 挂起它的活动，直到收到一个信号。60 秒之后，闹钟信号响起，内核设置该进程的用户栈来调用信号捕俘函数wakeup，该函数返回到 pause 之后的代码，从而，该进程再次循环。

在所有有关时间的系统调用中，共同的因素是它们对系统时钟的依赖：在处理时钟中断时，内核修改各种时间计数值，并发出相应的动作。

8.3　时钟

时钟中断处理程序的功能是：

- 重新启动时钟。
- 按内部定时器有计划地调用内部的内核函数。
- 对核心进程和用户进程提供运行直方图分析的能力。
- 收集系统和进程记帐及统计信息。
- 计时。
- 在有请求时，向进程发送闹钟软中断信号。
- 定期地唤醒对换进程（见下章）。
- 控制进程的调度。

有些操作是每个时钟中断时都做，有些操作则在若干个时钟滴答后才做。时钟中断处理程序在高处理机执行级上运行，以防止在中断处理程序工作期间发生其他事件（如来自外设的中断）。因此，时钟中断处理程序速度很快，这样，其他中断被屏蔽的临界时间会尽可能地小。图 8-9 给出处理时钟中断的算法。

8.3.1　重新启动时钟

当时钟中断系统时，多数机器要求由软件指令重新启动时钟，从而在适当的时间间隔后，它能再次中断处理机。这些指令是与机器有关的，这里不予讨论。

8.3.2　系统的内部定时

有些系统操作，特别是设备驱动和网络协议，要求在实时基础上调用内核的一些函数。例如，一个进程可以使一个终端处于原始方式，从而使内核以固定的时间间隔满足用户的 read 请求，而不是等待用户敲入回车键（见 10.3.3 节）。内核将必要的信息存放在 callout 表中（图 8-10），该表含有当定时时间到时所要调用的函数名、该函数的一个参数以及以时钟滴答为单位的定时时间。

用户不能直接控制 callout 表中的表项；这些表项是由内核在需要时，用各种内核算法创建的。对 callout 表的表项，内核不是按它们被放入表中的先后次序排序，而是按它们各自的“启动时间”进行排序。因为是按启动时间排序的，所以在 callout 表中，各表项的时间字段记录的是前一表项启动后，到该表项被启动时的时间量。对于表中的某一给定表项，其总的启动时间是从启动表中的第一项直到该表项（包括该表项）的时间的总和。

```
算法　clock
输入：无
输出：无
{
    重新启动时钟；　/*　为了使时钟再次中断　*/
    if　(callout 表非空)
    {
        修改 callout 时间；
        如果时间已消逝，安排调度 callout 函数；
    }
    if (内核直方图分析已打开)
        记下中断时刻的程序计数器；
    if (用户直方图已打开)
        记下中断时刻的程序计数器；
    收集系统统计信息；
    收集本进程统计信息；
    if　(自上次执行此语句以来已经过 1 秒钟或更多时间，而且中断不是在临界代码区)
    {
        for　(系统中的所有进程)
        {
            如果活动的话，调整闹钟时间；
            修改 CPU 的使用量；
            if　(进程在用户态执行)
                修改进程优先数；
        }
        必要的话，唤醒对换进程；
    }
}
```

图 8-9　时钟中断处理程序的算法

函数	启动时间
a ()	−2
b ()	3
c ()	10

之前

函数	启动时间
a ()	−2
b ()	3
f ()	2
c ()	8

之后

图 8-10　callout 表和函数 f 的新表项

图 8-10 给出在加入一个函数 f 的新 callout 表项以前和以后，callout 表的变化情况。（稍后将解释函数 a 的负时间字段的含义。）

当创建一个新表项时，内核找出新表项的位置，并适当地调整紧接新表项之后的那一项的时间字段。在图 8-10 中，内核计划在 5 个时钟滴答之后调用函数 f：它在 b 的表项之后为

f 建立了一个表项，其时间字段的值是 2（b 和 f 的时间字段之和为 5），并将 c 的表项的时间字段改为 8（c 将在 13 个时钟滴答时启动）。从实现角度讲，内核可以对 callout 表的每个表项使用链表，也可以在修改该表时，重新调整表项的位置。如果内核不是过多地使用 callout 表，那么后一种方法并不太费时。

时钟中断处理程序在每次时钟中断时，检查 callout 表中是否有表项。如果有，则使第一个表项的字段减 1。由于上述这种内核在 callout 表中的计时方法，因而，只要使第一个表项的时间字段减 1，便可以有效地使表中的所有其他表项的时间字段都减了 1。如果表中第一项的时间字段小于或等于 0，就应该调用对应的函数。时钟中断处理程序并不直接调用该函数，这样它就不会因疏忽而屏蔽后来的时钟中断：当前的处理机优先级是设在屏蔽时钟中断上，但内核并不知道该函数将花多少时间才能完成。如果该函数的运行时间超过一个时钟滴答，下次的时钟中断（以及所有其他发生的中断）就会被屏蔽掉。一般地，时钟中断处理程序通过引起一个“软件中断”来调度该函数。这个“软件中断”有时也称作“可编程中断”，因为它是由执行一条特殊指令引起的。因为软件中断的优先级比其他中断的优先级低，所以，直到内核处理完所有其他中断之前，软件中断一直是被屏蔽的。在内核准备好调 callout表中的某个函数的时刻和软件中断发生的时刻之间，许多中断，包括时钟中断都可能发生。因此，callout 表中第一项的时间字段可能被减为负值。当软件中断最终发生时，中断处理程序清除时间字段已过时的 callout 表项，并调用相应的函数。

在 callout 表中，由于头几项的时间字段可能为零或小于零，中断处理程序必须找出第一个时间字段为正值的表项，使该表项减 1。在图 8-10 中，函数 a 的表项的时间字段为 -2，这意味着在 a 有资格被调用之后，系统已过去了 2 个时钟中断的时间。假定 b 的表项在 2 次滴答之前就在表中，则内核跳过 a 的表项并使 b 的时间字段减 1。

8.3.3　直方图分析

内核直方图分析给出了一种确定系统在用户态的执行时间与系统在核心态的执行时间之比，以及内核执行内核中的各个子程序所花费的时间的度量方法。内核直方图驱动程序在时钟中断之时，对系统的活动进行采样，以便监视内核模块的相对性能。直方图分析程序有一个用于采样的内核地址表，表中通常含有内核函数的地址。进程事先通过写（write）直方图驱动程序来装入这些地址。在允许内核进行直方图分析时，时钟中断处理程序就调用直方图驱动程序的中断处理程序，它首先确定处理机在中断时是处于核心态还是用户态。如果是核心态，它使对应于程序计数器的一个内部计数器加 1；如果是用户态，它使用户执行计数器加 1。用户进程可以读直方图驱动程序，来获得内核计数并做统计。

以图 8-11 为例，图中给出的是几个假想的内核子程序的地址。如果在 10 个时钟中断期间，被采样的程序计数器值的顺序是 110、330、145，用户空间的地址是 125、440、130、320 及 104，那么图 8-11 给出内核应该保存的计数值。观察该图，你可以得出结论：内核在用户态花费了它的 20% 的时间，它的 50% 的时间花在执行内核算法 bread 上。如果做直方图分析的时间很长，程序计数器的采样图将趋于系统使用的真实比例。然而，这种方法并没计入花在执行时钟中断处理程序和屏蔽时钟级中断的代码上的时间，因为时钟不能中断这样的临界区代码段，从而不能在那里调用直方图中断处理程序。这是很遗憾的，因为对直方图

算法	地址	计数
bread	100	5
breada	150	0
bwrite	200	0
brelse	300	2
getblk	400	1
user	—	2

图 8-11　内核算法的采样地址

分析来讲，这样的内核临界区代码段通常是最重要的代码段。因此，必须有保留地对待内核直方图分析的结果。Weinberger［Weinberger 84］曾描述过一种产生代码的基本程序块计数的方法，用来精确地计算如“if-then”和“else”语句的语句体这样的基本程序块共执行了多少次。然而，这种方法使 CPU 时间增加了 50% 到 200%。因此，把它作为长久的内核直方图分析法是不实际的。

用户可以在用户级使用系统调用 profil 来对进程的执行做统计。该系统调用的语法格式是：

```
profil(buff,bufsize,offset,scale);
```

这里，buff 是用户空间中的一个数组的地址，bufsize 是该数组的大小，offset 是一个用户子程序（通常是第一个）的虚地址，scale 是将用户虚地址映射到数组中时使用的一个因子。内核将 scale 看作小数点在最“左”面的一个定点二进制小数．16 进制 0xffff 给出程序计数器与 buff 中的字的一对一映射，0x7fff 将一对程序地址映射到 buff 中的一个单个字，0x3fff 将 4 个程序地址构成的组映射到 buff 中的一个单个字，依次类推。内核将该系统调用的参数存在 u 区。当时钟中断一个在用户态的进程时，时钟中断处理程序就在中断时刻，检查该用户的程序计数器，将它和 offset 作比较，并使 buff 中的某个单元增值，该单元的地址是 bufsize 和 scale 的函数。

图 8-12 给出一个程序的例子。该程序对一个在无限循环中交替地调用函数 f 和 g 的程序的执行，进行直方图分析。该进程首先调用 signal，用来安排在一个 interrupt 软中断信号发生时，调用函数 theend。然后，它计算要进行统计的正文的地址范围，即从函数 main 的地址到函数 theend 的地址。最后，调用 profil 来通知内核，它想对它的运行做直方图分析。在一个负载不重的 AT&T 3B20 上，该程序运行大约 10 秒钟后，我们得到图 8-13 所示的输出。f 的地址比做统计的零地址大 204；f 的正文大小是 12 字节，在 AT&T 3B20 计算机上，一个整数是 4 个字节，因此 f 的地址被映射到 buf 的第 51、52 和 53 项。类似地，g 的地址映射到 buf 的第 54、55 和 56 项。buf 的第 46、48 和 49 项是函数 main 中的循环中的地址。在典型的使用中，要做直方图的地址范围是通过检查欲做直方图的程序的符号表中的正文地址来确定的。因为 profil 比较复杂，一般并不鼓励用户直接地使用它，而是在 C 编译程序中选取一个选择项，指示编译程序产生对进程做直方图的代码。

```
#include  <signal.h>
int buffer [4096];
main()
{
      int offset, endof, scale, eff, gee, text;
      extern theend (), f (), g ();
      signal (SIGINT, theend);
      endof= (int) theend;
      offset= (int) main;
      /* 计算程序正文的字数 */
      text= (endof-offset+sizeof (int) -1) /sizeof (int);
      scale=0xffff;
      printf ("offset %d endof %d text %d \n", offset, endof, text);
      eff= (int) f;
      gee= (int) g;
      printf ("f %d g %d fdiff %d gdiff %d \n", eff, gee, eff-offset, gee-offset);
      profil (buffer, sizeof (int) * text, offset, scale);
      for (;;)
       {
          f ();
          g ();
       }
}
f ()
{
}
g ()
{
}
theend ()
{
      int 1;
      for (i=0; i<4096; i++)
         if (buffer [i])
             printf ("buf [%d] =%d \n", i, buffer [;]);
      exit ();
}
```

图 8-12　调用 profil 的程序

```
offset 212 endof 440 text 57
f 416 g 428 fdiff 204 gdiff 216
buf [46] =50
buf [48] =8585216
buf [49] =151
buf [51] =12189799
buf [53] =65
buf [54] =10682455
buf [56] =67
```

图 8-13　profil 的采样输出

8.3.4 记帐和统计

当时钟中断系统时，系统可能在核心态执行，也可能在用户态执行。如果所有的进程都在睡眠，等待某个事件发生，那么内核就处于休闲状态。内核对每个处理机状态都保持有一个内部计数值，并在每个时钟中断期间，修改这些计数值，记下机器的当前状态。过后用户可以分析、收集在内核中的统计信息。

每个进程在其 u 区内都有一个字段，记录所消耗的内核时间和用户时间。当处理时钟中断时，内核根据进程是在核心态运行，还是在用户态运行，来修改该运行进程在 u 区中对应的字段。父进程在对其退出的子进程累加运行统计信息时，在系统调用 wait 中收集其子进程的统计信息。

每个进程在 u 区中还有一个让内核记录其内存使用情况的字段。当时钟中断一个运行的进程时，内核计算一个进程使用的总内存数，该数是该进程的私有存储区和它在共享存储区中所占的比例的函数。例如，如果一个进程和其他 4 个进程共享一个 50K 字节的正文区，并使用大小分别为 25K 和 40K 字节的数据区和栈区，内核则按该进程使用了75K(50K/5 + 25K + 40K)字节的内存空间收费。对于请求调页的系统，内核通过计算每个区中的有效页数来计算内存的使用情况。因此，如果某个被中断的进程使用了两个私有区，并和另一个进程共享一个区，内核按该进程使用的两个私有区中的有效页数，加上共享区中有效页数的二分之一收费。当该进程退出时，内核将这个信息写到记帐记录中，并用作向用户收费的依据。

8.3.5 计时

内核在每个时钟中断时，使一个时间变量加 1。系统的计时是以时钟滴答为单位，从系统自举之时开始。在使用系统调用 time 时，内核利用该时间变量返回一个时间值，并计算一个进程的总的运行时间（实际时间）。当一个进程在系统调用 fork 中被创建时，内核将该进程的起始时间记在 u 区中。当这个进程退出时（exit），内核将当前时间减去 u 区中的起始时间值，便得出该进程真正的运行时间。另外还有一个由系统调用 stime 来设置的时间变量，它保持着日历时间，每秒钟被修改一次。

8.4 本章小结

本章描述了 UNIX 系统上的进程的调度算法。内核将系统中的每个进程与一个调度优先权联系起来。进程优先权的赋值发生在进程进入睡眠时，或定期地发生在时钟中断处理程序中。进程在进入睡眠时所获得的优先权是一个固定的值，它取决于该进程当时正在执行的内核算法。在时钟中断处理程序中（或在进程从核心态返回到用户态时）所赋的优先权取决于该进程最近使用了多少 CPU 时间：如果它最近使用了 CPU，它将得到较低的优先权；否则，得到较高的优先权。系统调用 nice 允许用户调整计算进程优先权值的公式中的一个参数。

本章还介绍了与时间有关的系统调用：设置和查询内核时间，查询进程的运行时间，以及设置进程的闹钟软中断信号。最后，本章介绍了时钟中断处理程序的功能，其中包括保持系统时间、管理 callout 表、收集统计信息以及安排运行进程调度进程、进程对换进程和偷页进程。对换进程和偷页进程是下章中将要讨论的问题。

8.5　习题

1. 当进程进入睡眠时，内核给进程赋一优先权。对一个等待上了锁的索引节点的进程，内核赋给它的优先权高于一个等待上了锁的缓冲区的进程。同理，对等待读终端输入的进程所赋的优先权高于等待写终端输出的进程的优先权。请说明这样做的道理。

*2. 时钟中断处理程序以 1 秒钟为间隔，重新计算进程优先权并重新调度进程。请讨论一种能根据系统负载情况，动态地修改时间间隔的算法。所得到的优点相对于所增加的复杂性来说是否值得？

3. UNIX 系统的第 6 版使用下面的公式来调整进程的最近 CPU 使用时间：

```
decay(CPU)=max(优先权阈值,CPU-10);
```

第 7 版使用公式：

```
decay(CPU)=.8*CPU
```

这两个系统都用公式：

```
priority=CPU/16+(base level priority)
```

来计算进程的优先权。priority 为优先数。base level priority 为基级优先数。请将这些衰减函数用于图 8-4 的例子。

4. 请用 7 个（而不是 3 个）进程，重作图 8-4 所给的例子。再以每秒钟发生 100 次（而不是 60 次）时钟中断重做这个例子。请加以评论。

5. 设计一个算法，使系统对进程的执行时间加以限制。如果运行的进程超过该时间限制，则强迫它退出。用户如何将这样的进程和应该永远运行的进程区分开来？如果仅要求在 shell 中运行这种算法，应该做些什么？

6. 当一个进程执行系统调用 wait 并找到一个僵死的进程时，内核将该僵死子进程的 CPU 使用字段加到其父进程的 CPU 使用字段中。有什么理由要这样惩罚父进程呢？

7. 命令 nice 使紧跟在它后面的命令按给定的 nice 值运行，如：

```
nice 6 nroff-mm big_memo>output
```

请写出 nice 命令的 C 代码。

8. 跟踪图 8-4 中所示的进程调度情况，假定进程 A 的 nice 值是 5 或 -5。

9. 实现一个系统调用，renice x y，这里 x 是一个活动进程的进程标识号（ID），y 是其 nice 值应取的值。

10. 重新考虑图 8-6 所示的公平共享调度的例子。假定进程 A 所在的组买下了 33% 的 CPU 时间，进程 B 所在的组买下了 66% 的 CPU 时间。请问进程调度的顺序应当是什么样的？请使进程优先权的计算一般化，从而能使组的 CPU 使用字段归一化。

11. 请实现命令 date：没有参数时，该命令打印出系统的当前日期和时间；使用一个参数，如：

```
date mmddhhmmyy
```

一个（超级）用户能将系统的当前日期和时间设置为对应的月、日、小时、分和年。例如：

```
date 0911205084
```

将系统的日期和时间设置为 1984 年 9 月 11 日下午 8:50。

12. 程序可以用下面这个用户级函数使其运行被挂起指定的秒数：

```
sleep(seconds);
```

seconds 为挂起的秒数。请利用系统调用 alarm 和 pause 来实现该函数。如果一个进程在调用 sleep 之前曾调用过 alarm，将会发生什么情况？考虑两种可能性：以前的 alarm 调用会在该进程睡眠期间到期，或在 sleep 完成之后到期。

*13. 参见上个问题，内核可能在函数 sleep 中的 alarm 调用和 pause 调用之间做上下文切换，而进程可能在它调用 pause 之前就收到 alarm 软中断信号，这会发生什么情形？如何解决这种竞争条件？

第9章　存储管理策略

存储管理策略对上一章描述的进程调度算法有很深的影响。至少进程的一部分必须在内存中，它才能运行。或者说，CPU 不能执行一个全部内容都驻存在二级存储器中的进程。然而，内存是一种很有限的资源，它通常容纳不下系统中的全部活动进程。例如，若一个系统只有 8 兆字节的内存，那么 9 个 1 兆字节的进程就不能同时被放入内存。存储管理子系统决定哪一个进程应该驻留（至少是部分地）在内存中，并管理进程的虚地址空间中不在内存中的那些部分。它监视着可用的存储空间量，并定期地将进程写到一个称为对换设备的二级存储器上，以便提供更多的内存空间。过一会儿后，内核再将数据从对换设备读回内存。

早期的 UNIX 系统在内存和对换设备之间传送整个进程，而不是独立地传送一个进程的各个部分（共享正文除外）。这样的存储管理策略称为对换（swap）。在 PDP11 上实现这样的策略是有道理的。因为在 PDP11 上，进程的最大尺寸是 64K 字节，对这样的策略来说，进程的大小被限定在一个系统可用的物理存储量上。BSD（4.0 版本）是第一个主要的实现请求调页（demand paging）策略的系统。请求调页策略是在内存和二级存储器之间来回传送存储页，而不是整个进程。最近推出的 UNIX 系统 V 的版本也支持请求调页存储管理策略。整个进程并不需要全部驻留在内存就可运行。当进程访问页面时，内核按需要为进程装入该页。请求调页的优点是，它使进程的虚地址空间到机器的物理存储空间的映射具有更大的灵活性。它通常允许进程的大小比可用的物理存储空间大得多，还允许将更多的进程同时装入内存。对换策略的优点是，它的实现较为简单，因此系统开销较小。本章将讨论对换和调页这两种存储管理策略。

9.1　对换

对换算法的描述包括三个部分：对换设备上的空间管理，将进程换出内存和将进程换入内存。

9.1.1　对换空间的分配

对换设备是在一个磁盘的可配置段中的一个块设备。与对文件一次只分配一个磁盘块空间不同的是，内核在对换设备上是以一组连续的磁盘块为单位来分配空间的。为文件分配的空间静态地使用，这是因为该空间将存在很长时间，这样的分配机制适应于减少碎片量，从而减少文件系统中不可分配的空间。但是，在对换设备上，空间的分配是短暂的，它依赖于进程调度的模型。驻留在对换设备上的进程，最终会迁回主存，从而释放它在对换设备上所占据的空间。由于速度是至关重要的，而且系统以一个多块操作进行输入输出要比多个单块操作快得多，所以，内核在对换设备上分配连续的空间，而不考虑碎片问题。

因为用于对换设备的分配方法不同于用于文件系统的分配方法，所以，记录空闲区的数据结构也不一样。内核以一个空闲块链接表的方式来管理文件系统的空闲空间。在文件系统的超级块中可以存取到该链表。与此不同的是，内核通过存放在内存的、称为“映射图

(map)”的表，来管理对换设备的空间。映射图也用于管理除对换设备以外的其他一些资源(例如，一些设备驱动程序)。它允许采用最先适配（first fit）的算法，分配一个资源中的连续“块”。

映射图是一个数组。数组中的每一项都含有一个可分配资源的地址和在该地址上可用资源的单位数。内核根据映射图的类型来解释地址和单位的含义。开始时，映射图只含有一项。它指出资源的地址和全部可用资源的数目。例如，内核将对换映射图中的每一项看作一组磁盘块。图 9-1 给出了一个从地址 1 开始，具有 10000 块的初始对换映射图。随着分配和释放资源，内核不断修改映射图，使其总是保存着空闲资源的准确信息。

地址	单位数
1	10000

图 9-1　初始对换映射图

图 9-2 给出的是根据映射图分配空间的算法 malloc。内核从映射图中查找第一个含有足够大的空间来满足要求的表项。如果这一需求占用了该表项的全部资源，内核则从数组中删除这一表项并压缩映射图（也就是说，映射图少了一项)。否则，内核根据分配资源的数量，修改该表项中的地址和数量域。图 9-3 给出了分配 100 单位、50 单位以及又一个 100 单位之后的一系列对换映射图的内容。内核调整了对换映射图，以反映出从头开始的 250 个单位数已被分配。目前还有从地址 251 开始的 9750 个单位的空闲资源。当释放资源时，内核根据地址找出它们在映射图中的相应位置。这时有三种可能的情况：

```
算法　malloc　/*　分配映射空间的算法　*/
输入：(1) 映射地址　/*　指明使用哪个映射图　*/
      (2) 所请求的资源的单位数量
输出：如果成功的话，为地址
      否则为 0
{
    for（每个映射图表项）
    {
        if（当前映射图的表项能满足请求的单位数）
        {
            if（请求的数量 == 该表项中的单位数）
                从映射图中删除该表项；
            else
                修改该表项的起始地址及单位数量；
            return（该表项原来的地址）；
        }
    }
    return（0）
}
```

图 9-2　按映射图分配空间的算法

(1) 被释放的资源完全填满映射图中的一个空洞：它们的地址在映射图中刚好能连接紧接其前和其后的那些表项。在这种情况下，内核将所释放的资源与已存在的（两个）表项合并，形成映射图中的一项。

(2) 被释放的资源部分地填充映射图中的一个空洞：如果被释放的资源的地址紧接其前

或其后的表项，内核根据刚释放的资源修改相应表项的地址和单位数量字段，映射图的项数不变。

地址	单位数
1	10000

a)

地址	单位数
101	9900

b)

地址	单位数
151	9850

c)

地址	单位数
251	9750

d)

图 9-3 分配对换空间

(3) 被释放的资源部分地填充一个空洞但并不连接映射图中的任何表项。这时，内核在映射图中建立一个新的表项，并插入到相应的位置上去。

让我们再回到前一个例子。假设内核释放从地址 101 开始的 50 个单位的对换资源。因为放回的资源不与映射图中的已有表项相连，所以对换映射图中便含有一个对应于该释放资源的新项。此后，内核又释放从地址 1 开始的 100 单位的对换资源。因为这次释放的资源与对换映射图中的第一项相连，所以内核修改了第一项。图 9-4 给出了对应这些事件的对换映射图的内容。

地址	单位数
251	9750

a)

地址	单位数
101	50
251	9750

b)

地址	单位数
1	150
251	9750

c)

图 9-4 释放对换空间

假定内核现在需要 200 单位的对换空间。因为对换映射图的第一项仅含有 150 个单位，所以内核用第二项来满足这一请求（见图 9-5）。最后，假定内核释放从地址 151 开始的 300 个单位的对换空间。尽管这 300 个单位的空间是几次分配的结果，但内核没有理由不能一次全部释放。（实际上，内核对对换空间并不这样做，因为对对换空间的申请都是各自独立的。）内核分析出被释放的资源填满了介于对换映射图中的第一项和第二项之间的空区。它产生一个新表项来替代第一项和第二项（以及被释放的资源）。

传统的 UNIX 系统，其实现仅使用一个对换设备，但系统 V 的最新实现允许使用多个对换设备。内核采用循环算法来选用对换设备，以保证对换设备上有足够大的连续存储空间。管理人员可以动态地创建和取消对换设备。如果某个对换设备正在被取消，内核就不再

地址	单位数
1	150
251	9750

a)

地址	单位数
1	150
451	9550

b)

图 9-5　从映射图中的第二项分配对换空间

往该对换设备上对换数据。其上的数据被不断地取走，直到该对换设备被腾空而被取消。

9.1.2　进程的换出

如果内核需要内存空间，它就换出进程。以下事件可引起进程的换出：

(1) 系统调用 fork 必须为子进程分配空间；

(2) 系统调用 brk 扩大一个进程的大小；

(3) 一个进程由于其栈的自然增长而变大；

(4) 为了运行以前被换出、而现在又应被换入的进程。其中，由 fork 引起的情况最为突出，它是唯一不能放弃被进程先前占据的内存映象空间的情况。

当内核决定了某一进程应从主存中换出时，它使该进程中的每个区的引用数减 1，并把那些引用数减为 0 的区换出。内核分配对换设备空间，并将该进程锁在内存中（对应于上述事件 1、2 和 3），以防止在当前的对换操作过程中，对换进程将它对换出来（见习题 9.2）。内核将区的对换地址保存在区表表项中。

内核绕过高速缓冲，直接在对换设备和用户地址空间之间对换数据，并在一次 I/O 操作中对换尽可能多的数据。如果硬件不能一次传送多个页面的话，内核的软件必须循环地一次传送一页内存。因此，数据传送的精确速率及传送机制，除其他因素外，主要依赖于磁盘控制器的能力和存储管理的实现。例如，如果按页面来组织内存，那么，要被换出的数据在物理存储器上就可能是不连续的。内核必须收集被换出的数据的页面地址，磁盘驱动程序也可能要使用这些页面地址来启动 I/O。对换进程在换出剩下的数据之前要等待每一个 I/O 操作的完成。

内核没有必要将一进程的整个虚地址空间全部写到对换设备上去。内核仅将已分配给进程的物理存储拷贝到所分配的对换设备空间上去，而忽略那些尚未分配的虚地址。当内核将该进程换回到内存时，它知道该进程的虚地址映射，从而能将该进程重新分配到正确的虚地址上。内核通过把数据读到先前建立的与虚地址位置一致的物理存储器的位置去，消除了一次额外的从数据缓冲到物理存储器的拷贝。

图 9-6 给出了一个将一个进程的内存映象映射到对换设备上的例子[⊖]。该进程具有三个区，分别用于正文、数据和栈。正文区结束于虚地址 2K，数据区从虚地址 64K 开始，中间有 62K 字节的虚地址空间空区。当内核换出进程时，仅换出虚地址为 0、1K、64K、65K、66K 和 128K 的页面。内核不为介于正文区和数据区之间的 62K 字节的空区或介于数据和栈区之间的 61K 字节的空区分配对换空间，而是连续地填放对换空间。当内核将该进程换入

⊖ 为简单起见，在该图和以后的图中，进程的虚地址空间被表示为一个页表项的线性数组，而实际上，每个区通常有自己的页表。

时，通过查进程存储映射表得知该进程有一个62K字节的空区，并指定相应的物理存储。图9-7说明了这一情况。比较图9-6和图9-7可以看出，进程占据的物理地址单元在换出前

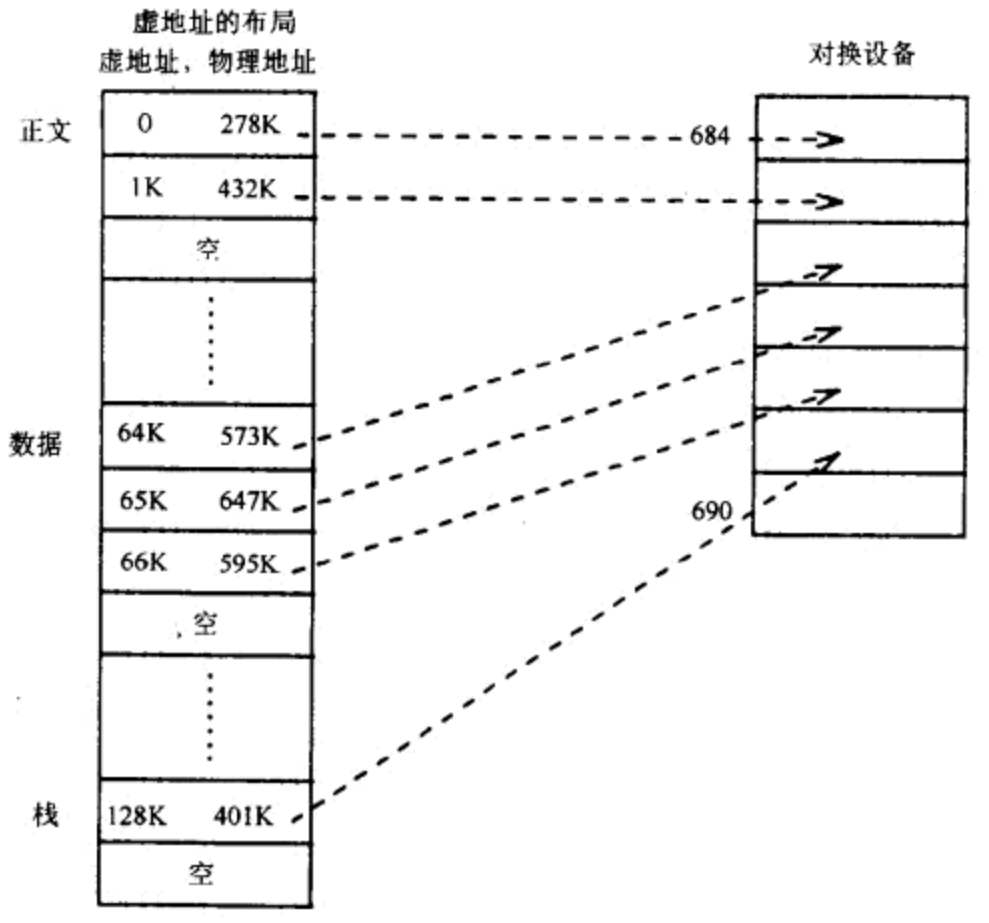


图9-6 进程空间到对换设备的映射

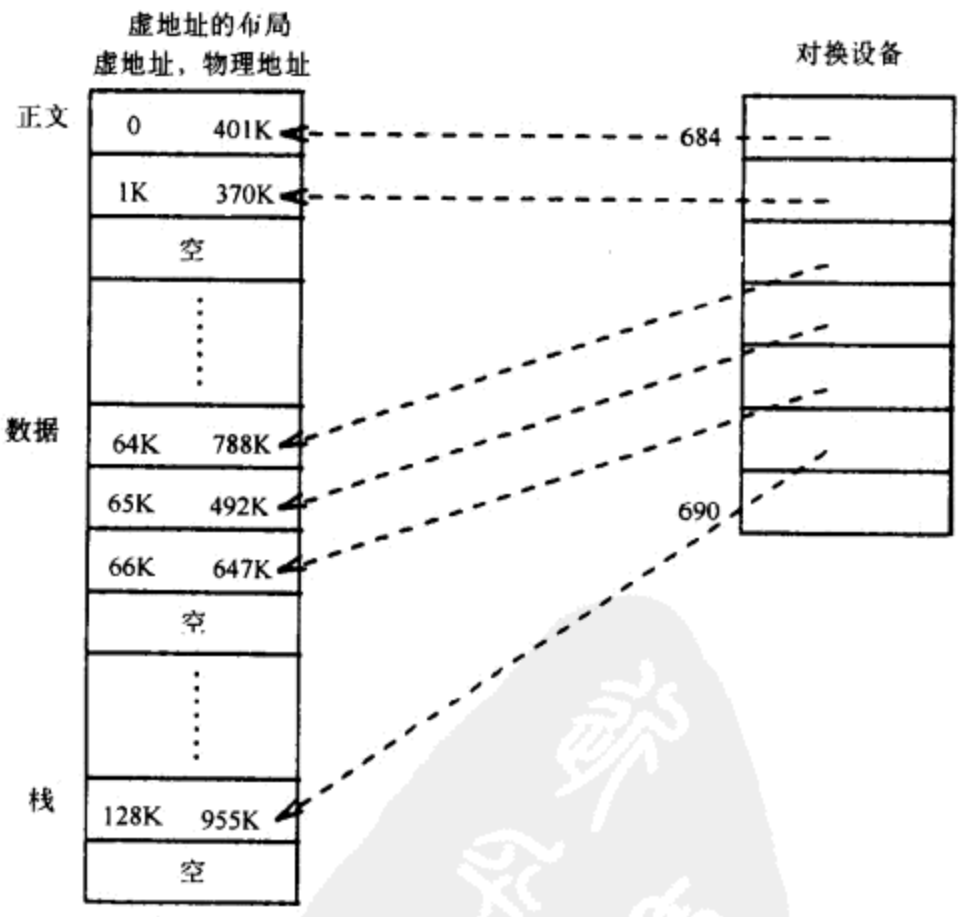


图9-7 将进程换入内存

和换入后是不同的，但在进程的用户级上是不会注意这一变化的，因为它的虚地址空间中的内容没有变。

从理论上说，进程的所有存储空间，包括它的 u 区和核心栈，都可以被换出，尽管在进行敏感操作时，内核可能暂时地将一个区锁在内存。然而，在实际上，如果 u 区含有进程地址转换表，则内核在实现上就不能换出 u 区。实现上还要决定是否一个进程能将其自身换出，还是必须由另一个进程来将其换出（见习题 9.4）。

1. 系统调用 fork 的对换

在系统调用 fork 的描述中（见 7.1 节），假定了父进程找到了足够的内存空间以创建子进程的上下文。否则，内核将换出该进程但并不释放被（父）进程的内存映象所占据的内存。当换出完成后，子进程在对换设备上，父进程将子进程置为就绪状态（见图 6-1），然后返回用户态。由于子进程处于就绪状态，所以对换进程总会将其换入内存。内核在内存中总会调度到它，这时子进程完成它的 fork 系统调用部分然后返回用户态。

2. 扩展对换

如果进程需要的内存比当前已分配给它的内存还多，不管这是由栈增长引起的还是由于调用系统调用 brk 所引起的，内核都要进行一次进程的扩展对换（expansion swap）。内核在对换设备上预定足够的空间以容纳进程的存储空间，其中包括新申请的空间。然后内核修改进程地址转换映射以适应新的虚存空间，但此时并不分配物理存储地址（由于没有可用的物理空间）。最后内核通过一次通常的对换操作将该进程换出，同时将对换设备上新分配的空间清零（见图 9-8）。当以后内核将该进程换入时，将按新的(增加了尺寸的)地址转换映射

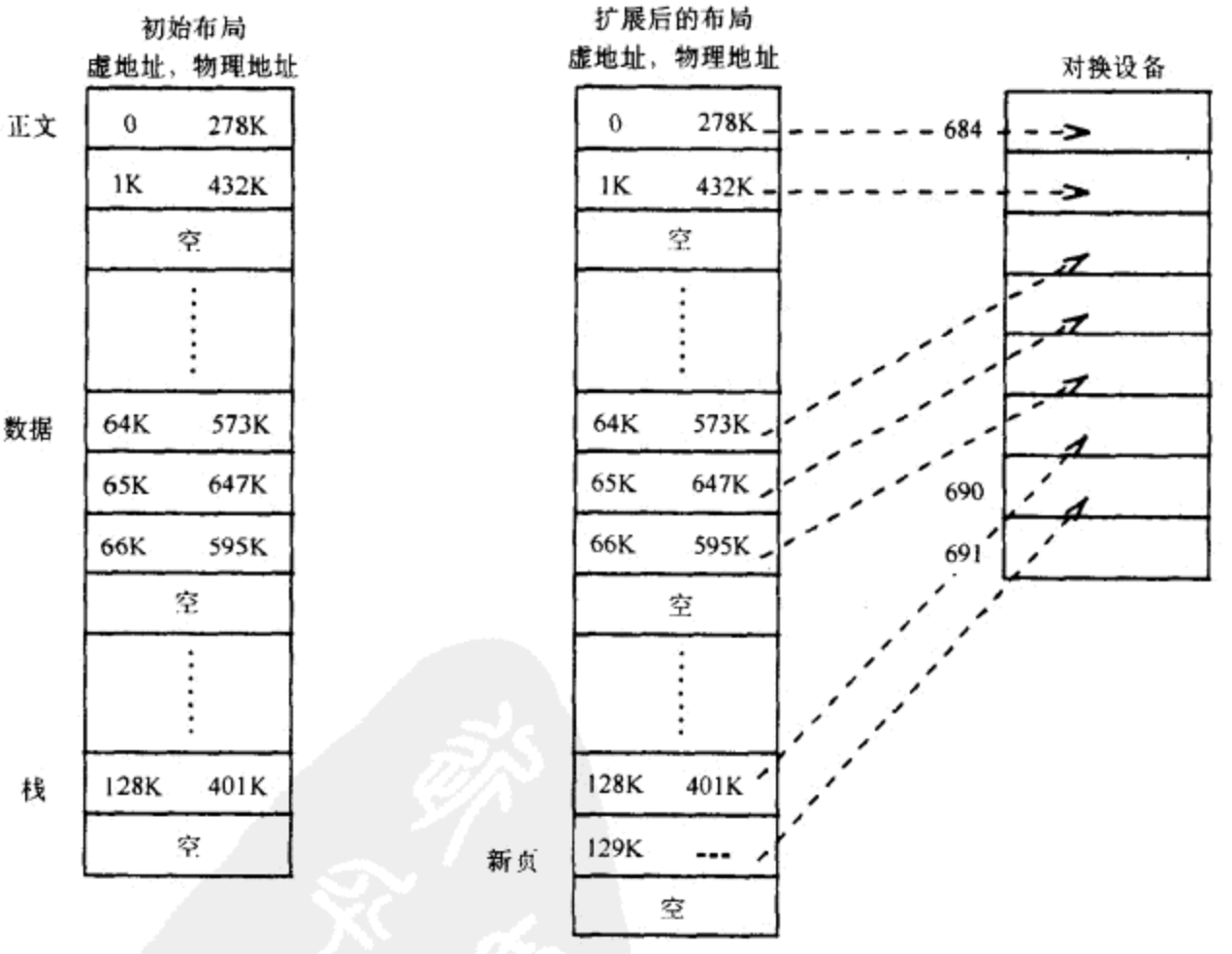


图 9-8　为扩展对换修改内存映象图

图来分配物理地址。这样，当该进程恢复执行时就有了足够的空间。

9.1.3 进程的换入

进程0，即对换进程，是唯一的将进程从对换设备上换入内存的进程。在系统初始化结束时，对换进程进入一个无限循环，就像在7.9节中提到的那样，它的任务就是将进程换入或换出。它总是试图将进程从对换设备上换入内存；如果需要主存空间，就将进程换出内存。如果对换进程没有事做（例如没有进程要换入）或不能做任何事情（例如没有进程可以被换出），它就睡眠。内核定期地唤醒对换进程。尽管对换进程具有高优先权，内核同调度其他进程一样调度对换进程运行，但对换进程仅在核心态下运行。对换进程不调用系统调用，但使用内核内部函数来执行对换，这是所有核心进程的主要工作方式。

第8章中曾简要提到，时钟处理程序度量每一个进程在内存中或被换出的时间。当对换进程被唤醒后，它进行换入进程的工作，此时，它查找所有处在“就绪且换出”状态的进程，从中选取换出时间最长者（图9-9），如果有足够的内存，就将进程换入。将进程换入

```
算法  swapper  /*  换入曾被换出的进程，
                  * 换出其他进程来腾出空间  */
输入：
输出：
{
    loop:
        for（所有已被换出的就绪进程）
            选取被换出时间最长的进程；
        if（没有这样的进程）
        {
            sleep（事件：必须换入进程）；
            goto loop；
        }
        if（主存中有足够的空间存放该进程）
        {
            换入该进程；
            goto loop；
        }
    /*  loop2：这里是在修改的算法中（见第219页）* /
        for（所有已装入内存、非僵死而且没被锁在内存的进程）
        {
            if（有一个睡眠进程）
                选取优先权值与驻留时间之和在数值上最高的进程；
            else  /* 没有睡眠的进程 * /
                选取驻留时间与nice值之和在数值上最高的进程；
        }
        if（所选择的进程不在睡眠或不满足驻留时间的要求）
            sleep（事件：必须换入进程）；
        else
            换出进程；
        goto loop；   /* 在修改算法中，goto loop2 * /
}
```

图9-9 对换进程的算法

的操作是换出操作的逆过程：分配物理存储，将进程从对换设备读入，然后释放对换空间。

如果对换进程成功地换入了一个进程，它仍继续查找状态为“就绪且换出”的进程来换入并重复上述过程。最终会发生下列情形之一：

• 对换设备上没有“就绪”的进程：这时，对换进程进入睡眠，直到一个对换设备上的进程被唤醒或内核换出一个“就绪”状态的进程（见状态图 6-1）。

• 对换进程找到了应被换入的进程，但系统没有足够的内存空间：此时，对换进程试图换出另一进程，如果成功则重新启动对换算法，查找需要换入的进程。

如果对换进程必须换出一个进程，那么对换进程检查在内存的每一进程：僵死进程不用换出，因为僵死进程不占据任何物理存储；锁在内存中的进程（例如正在进行区操作）也不能换出。内核换出正在睡眠的进程而不是换出“就绪”进程。因为“就绪”进程很可能很快就被调度上。选取哪一个睡眠进程来换出取决于进程的优先权和它在内存驻留的时间。如果内存中没有正在睡眠的进程，那么选择哪个“就绪”进程来换出就取决于进程的 nice 值和它在内存中驻留的时间。

一个就绪进程在被换出前必须至少在内存驻留了 2 秒，一个要被换入的进程必须至少已被换出了 2 秒。[⊖]如对换进程找不到可换出的进程，或者，要换入或要换出的进程在它们的环境里的驻留时间都不超过 2 秒，对换进程就睡眠于一个事件上。这一事件指出，对换进程要换入一个进程，但是找不到足够的内存来存放它。在这一状态下，时钟每秒唤醒一次对换进程。如果有任何进程要进入睡眠状态，内核也要唤醒对换进程，因为进入睡眠的进程可能比对换进程以前所查找的进程更应被换出。如果对换出了一个进程，或因不能换出一个进程而进入睡眠，它将从头执行对换算法，以换入可被换入的进程。

图 9-10 描绘了五个进程在经历一系列对换操作时，各自在内存或在对换设备上所用的时间。为简单起见，假定所有进程都是 CPU 密集型的，并且不进行任何系统调用。这样，上下文的切换仅仅由每秒一次的时钟中断所引起。对换进程总是以最高优先权运行，所以只要有工作可做，每隔一秒钟它总是要运行一会儿。我们进一步假定，所有的进程大小相同，并且系统同时在内存最多只放两个进程。在开始时，进程 A 和进程 B 在内存中，其他的进程都被换出。在开始的 2 秒内，对换进程不能对换任何进程，因为没有进程在内存或在对换设备上驻留了 2 秒（这是对换的驻留要求）。一旦到了 2 秒，对换进程换出进程 A 和进程 B，并换入进程 C 和进程 D。对换进程还试图换入进程 E，但却由于内存告罄而失败。在 3 秒钟时，进程 E 应被换入，因为它在对换设备上已呆了 3 秒钟。但是，由于呆在内存的进程都没有超过 2 秒钟，所以对换进程不能换出任何进程来为进程 E 腾出空间。到了 4 秒钟时，对换进程换出进程 C 和 D 并换入进程 E 和 A。

对换进程基于进程被换出的时间来选择要换入的进程。另一个换入标准是具有最高优先权的就绪进程，因为这样的进程应该得到较多的运行机会。在系统重负载的情况下，这种策略可获得“略微”好一点的系统吞吐量（见[Peachy 84]）。

但是，选择换出进程以得到内存空间的算法有更为严重的缺陷。第一，对换进程根据进程的优先权、内存驻留时间及它的 nice 值换出一个进程。尽管对换进程换出一个进程仅仅是为了腾出空间以换入另一个进程，但换出一个进程可能并不能为要换入的进程提供足够的

⊖ UNIX 系统的第 6 版在实现上并不为将要换入的进程腾出内存而换出一个进程，除非要换入的进程在磁盘上驻留了 3 秒。被换出的进程必须在内存至少驻留了 2 秒。这一时间间隔的选择改善了抖动并增加了系统的吞吐量。

时间	进程 A	B	C	D	E
0	0 运行	0	换出 0	换出 0	换出 0
1	1	1 运行	1	1	1
2	2 换出 0	2 换出 0	2 换入 0 运行	2 换入 0	2
3	1	1	1	1 运行	3
4	2 换入 0	2	2 换出 0	2 换出 0	4 换入 0 运行
5	1 运行	3	1	1	1
6	2 换出 0	4 换入 0 运行	2 换入 0	2	2 换出 0

图 9-10 对换操作序列

内存空间。举例来说，如果对换进程想要换入一个占 1 兆字节内存的进程而系统又没有空闲内存，这时换出一个仅占 2K 字节内存的进程是不解决问题的。一个替代的策略是，仅当一组进程能为换入进程提供足够的内存空间时，才换出这组进程。在一台 PDP 11/23 计算机上的使用经验说明，采用这一策略，在重负载的情况下系统的吞吐量将增长 10%（见[Peachey 84]）。

第二，如果对换进程因为不能为换入进程找到足够的内存空间而睡眠，它醒来时将重新寻找一个要换入的进程，尽管先前它已经有了一个要换入的进程。因为与此同时可能有其他的进程被唤醒，而它们又比先前选中的进程更应被换入。对先前已选中的进程的一点安慰是，对换进程仍然试图将其换入。在一些实现的版本中，对换进程在选择下一个换入进程之前总是试图换出多个较小的进程来为一个要换入的大进程提供空间。图 9-9 中的注释指出了对换进程算法的这一修改。

第三，假若对换进程选中了一个“就绪”进程换出，但可能该进程自从被换入之后一直没有运行。图 9-11 说明了这一情况。图中内核在 2 秒时将进程 D 换入，然后在 3 秒钟时为了进程 E 又将进程 D 换出（由于 nice 值的作用），尽管 D 还没有运行。这一抖动显然不是我们所希望的。

最后，值得一提的是另一个危险。如果对换进程要换出一个进程，但在对换设备上又找不到空区，这时若下列四个条件满足时，系统就会产生死锁；在主存中的所有进程都进入睡眠；所有“就绪”进程都已被换出；在对换设备上没有空间以容纳新的进程以及在内存中已无空间来存放要换入的进程。习题 9.5 探讨了这种情况。解决对换进程这一问题的希望在于请求调页算法。最近几年，请求调页算法已在 UNIX 系统上实现。

时间	进程 A	B	C	D	E
0	0 运行	0	换出 0	nice 25 换出 0	换出 0
1	1	1 运行	1	1	1
2	2 换出 0	2 换出 0	2 换入 0 运行	2 换入 0	2
3	1	1	1	1 换出 0	3 换入 0 运行
4	2 换入 0 运行	2	2 换出 0	1	1
5	1	3 换入 0 运行	1	2	2 换出 0
6	2 换出 0	1	2	3 换入 0 运行	1

图 9-11　由对换而引起的抖动

9.2 请求调页

存储器结构基于页面，并且CPU具有可再启动指令[⊖]的计算机，可以支持一个实现请求调页算法的内核，即内核在对换设备和主存之间对换存储页。请求调页系统将进程从大小限制中解放出来，否则进程大小将被限制在一个计算机的可用物理存储量上。例如，在请求调页系统中，仅有1兆或2兆字节物理存储的机器，可运行大小为4兆或5兆的进程。但是，内核仍要限制进程虚地址空间的大小，这取决于机器可寻址的虚拟存储量。由于进程不能全部装入内存，所以内核必须动态地将进程的适当部分装入内存并执行该进程，尽管该进程的其他部分不在内存。除了虚拟空间大小的限制之外，请求调页对用户程序是透明的。

进程常表现为仅执行进程正文空间的一小部分指令，如程序循环和频繁调用的子程序。进程的数据访问也倾向于聚集在全部进程数据空间的一些小子集上。这种现象被称为“局部化原理”。Denning(见[Denning 68])提出了一个进程的“工作集”表示法。工作集(working set)是最近n次内存访问所访问的页面集合。数字n称为工作集的窗口(window)。因为工作集只是整个进程的一小部分，所以比起在对换系统中的进程来，可以有更多的进程被同时放入内存。因为对换传输的减少，潜在地提高了系统的吞吐量。当一进程寻址一个不在工作集内的页面时，会导致一个页面错。在处理页面错时，内核更新工作集，并在需要时，从二级存储器中读入该页面。

图9-12画出了一个进程可能进行的一系列页面访问，还给出了遵循最近最少使用的替

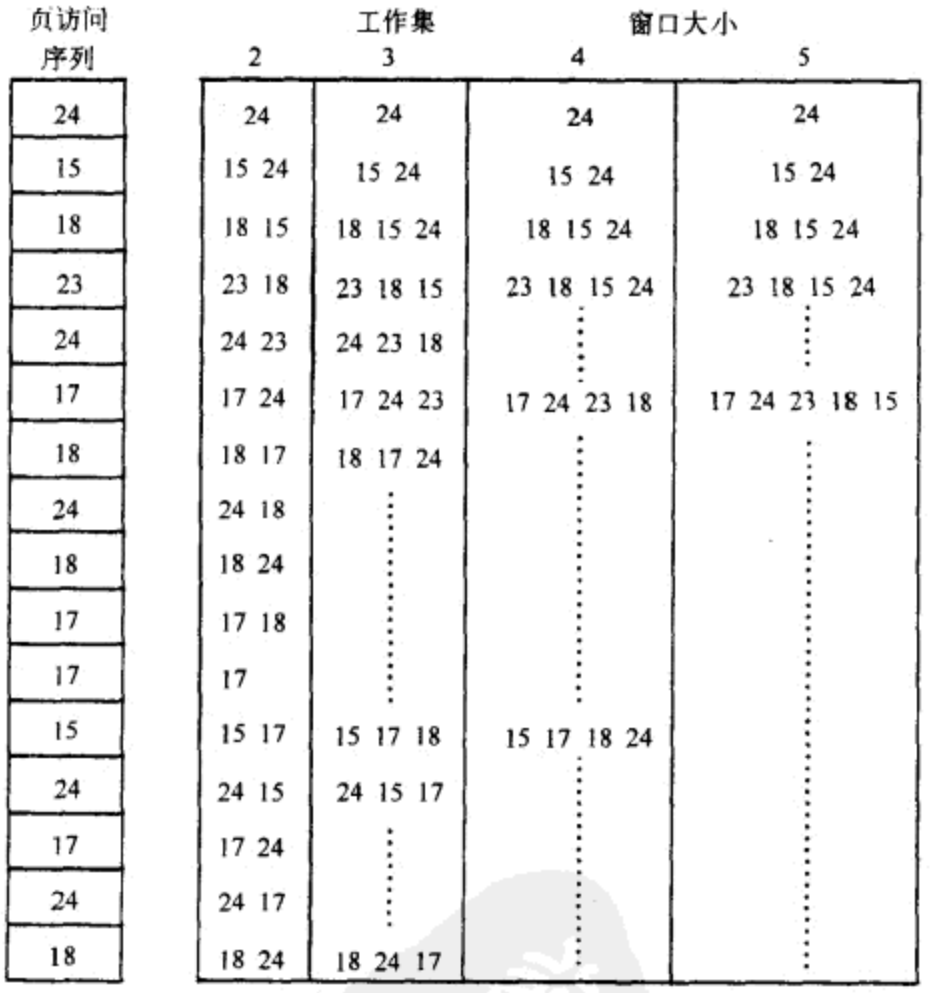


图9-12 一个进程的工作集

⊖ 假若一台计算机只部分地执行了一条指令并导致一个缺页错，CPU在出错处理之后必须重新执行该指令，因为在页面错之前所得到的中间结果可能已经丢失了。

换策略时，不同窗口大小的工作集。随着进程的执行，它的工作集也随着变化。这一变化取决于进程所做的存储器访问的情况，较大的窗口产生较大的工作集，这意味着进程不会经常出现页面错。实现一个纯工作集模型是不实际的，因为记住页访问次序的代价是昂贵的。系统采用另一种方法来粗略地模拟工作集模型：当进程访问一内存页时，则其访问位被置为1，内核定期地抽样检查内存的访问情况。如果某页最近被访问过，它就是工作集的一部分，否则，这一页在内存中的“年龄”不断增长，直到它可被换出。

当一个进程存取一个不属于工作集的页面时，将产生一个有效性页面错（validity page fault）简称为有效性错。内核挂起该进程，直到内核将该页读入内存并使其可被该进程存取。当页面装入内存后，进程重新执行导致有效性错的那条指令。由此可以看出，调页子系统的实现有两部分：将不常使用的页面换到对换设备上去以及处理有效性错。以上是对调页系统的一般性描述，它可扩大到非 UNIX 系统中。本章的其余部分将详细地讨论 UNIX 系统 V 的请求调页机制。

9.2.1 请求调页的数据结构

支持低层存储管理和请求调页的主要内核数据结构有 4 个：页表表项、磁盘块描述项（disk block descriptors）、页面数据表（page frame data table，简称 pfdata）和对换使用表（swap－use table）。在系统的生存期内，内核仅为 pfdata 分配一次空间，对其他数据结构则动态地分配内存页。

回忆第 6 章我们知道，一个区有一个页表，以存取物理存储器。页表的每一表项（见图 9-13）含有该页的物理地址，用以指示是否允许进程读、写或执行该页的保护位（protection bit），以及为支持请求调页而设的下列位字段：

- 有效位。
- 访问位。
- 修改位。
- 写时拷贝位。
- 年龄位。

内核置上有效位（valid bit）来指示该页的内容是有效的。下面将会看到，当有效位没有被置上时，对该页的访问不见得一定是非法的。访问位（reference bit）指示最近是否有进程访问了该页。修改位（modift bit）指示最近是否有进程修改了该页的内容。写时拷贝位（copy on write bit）用于系统调用 fork，指出当进程修改该页内容时，内核必须为该页建立一个新的拷贝。最后，内核通过管理年龄位（age bit）来记录该页作为一个进程的工作集中的一员有多长时间了。以下我们假定内核管理页表项的有效位、写时拷贝位及年龄位，而硬件设置页表项中的访问位和修改位。9.2.4 节将考虑硬件没有这些能力的情况。

每一个页表项都与一个磁盘块描述项相关联。该磁盘块描述项描述了该虚拟页的磁盘拷贝。因此，共享一个区的所有进程也共享页表项和磁盘块描述项。一个虚拟页的内容或在一个对换设备的特定块中，或在一个可执行文件中，即不在对换设备上。如果该虚拟页在对换设备上，则磁盘块描述项中含有存放该页的逻辑设备号和块号。如果该页在一个可执行文件中，则磁盘块描述项含有该文件中的逻辑块号，虚拟页就在这一逻辑块中。内核可以很快地将这个逻辑块号映射到它的磁盘地址上去。磁盘块描述项还含有在系统调用 exec 时设置的

两个特殊条件：一个是页“请求填入（demand fill）”，另一个是页“请求清零（demand zero）”。我们将在第2小节中解释这两个特殊条件。

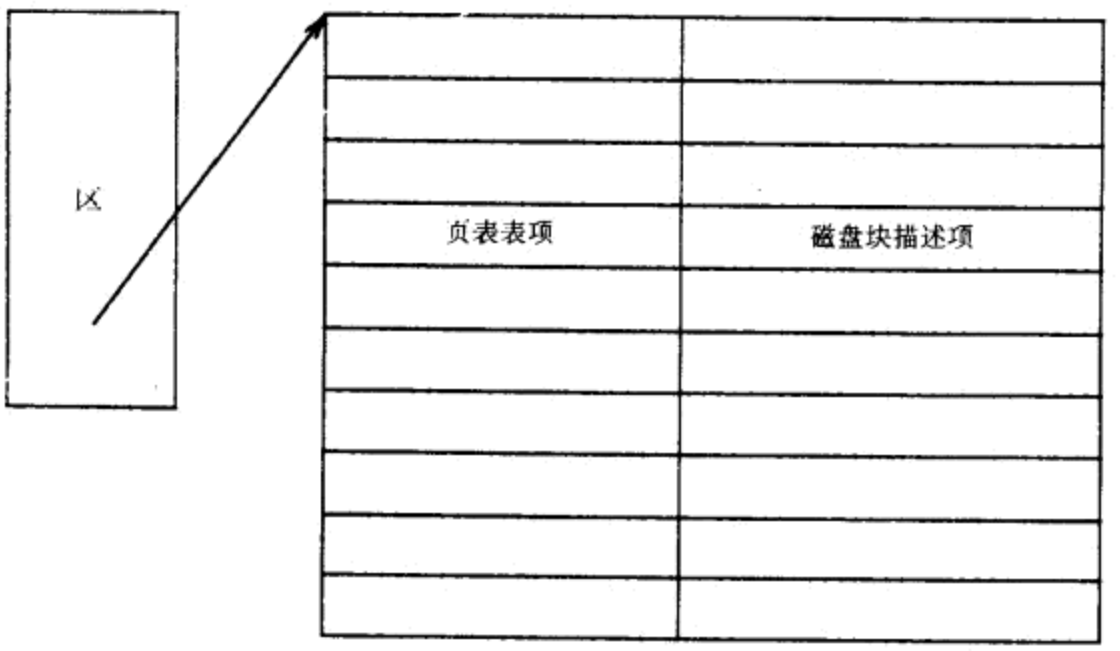


图 9-13 页表表项和磁盘块描述项

pfdata 表描述了每个物理页。它以页号来索引。每一表项的字段有：

• 页状态，指示该页是在对换设备上，或在可执行文件中；并指示 DMA 是否正在为该页服务（从对换设备上读数据）；还指示该页是否可被重新分配。

• 引用该页的进程个数。这个引用数等于引用该页的有效页表项的数目。它可能不同于共享含有该页的那些区的进程数目。在下面重新考虑系统调用 fork 算法时，我们再来讨论这一点。

• 逻辑设备（对换或文件系统）和含有该页一个拷贝的块号。

• 指向在空闲页链表中其他 pfdata 表项的指针和指向一个页散列队列中的其他 pfdata 表项的指针。

内核将 pfdata 表的所有表项链到一个空闲页链表和一个散列链表中，类似于高速缓冲的链接表。空闲链表是页的缓冲池，其中的页面可被重新分配。进程可能在一个地址上出页面错，但仍可能在空闲链表中发现相应的页是原封不动的。因此，空闲链表允许内核避免不必要的读对换设备的操作。内核按最近最少使用的次序从空闲链表上分配页面。内核还按页面的（对换）设备号和块号对 pfdata 表项进行杂凑。这样，只要给定一个设备号和块号，只要对应的页在内存，内核就可以迅速地定位。为了给一个区分配一个物理页，内核从空闲链表

头摘下一个空表项，修改其对换设备号和块号，并将其放入恰当的散列队列中去。

对换设备上的每一页都占有对换使用表的一个表项。该表项含有一个引用数，它表示了有多少页表项指向对换设备上的该页。

图 9-14 给出了页表项、磁盘块描述项、pfdata 表项以及对换使用表之间的关系。一个进程的虚地址 1493K 映射到一个指向物理页为 794 的页表表项；该页表表项的磁盘块描述项指出，该页的一个拷贝在对换设备 1 上的第 2743 号磁盘块上。物理页 794 的 pfdata 表项也指出该页的一个拷贝在对换设备 1 上的第 2743 块中，并且它的内存引用数为 1。在 9.2.4 节中将讨论为什么磁盘块号在 pfdata 表和磁盘块描述项中重复出现。对应这一虚拟页的对换使用数为 1，这意味着只有一个页表项指向该对换拷贝。

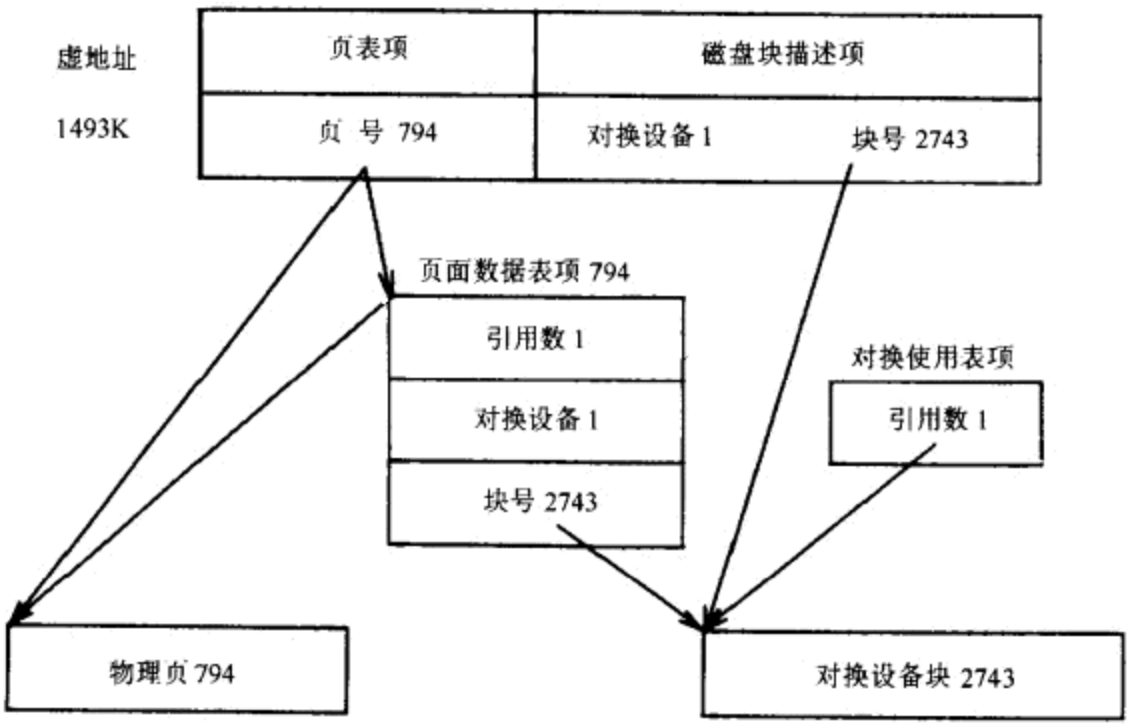


图 9-14　请求调页的各数据结构之间的关系

1. 调页系统中的系统调用 fork

像在 7.1 节中解释的那样，在调用系统调用 fork 时，内核拷贝父进程的每个区，并将其附接到子进程上。传统上，一个对换系统的核心要做一个父进程的地址空间的物理拷贝。这常常是浪费的操作，因为进程经常紧接着系统调用 fork 之后调用 exec，马上释放刚刚拷贝的内存。在系统 V 的调页系统中，内核通过管理区表，页表项和 pfdata 表项避免了页拷贝：它仅仅增加共享区的区引用数。对于像数据区和栈这样的私有区，内核分配一个新的区表表项和页表，然后检查父进程的页表项：如果一个页是有效的，则增加其 pfdata 表项中的引用数，以指出通过不同区而共享该页的进程数目（不同于通过共享区而共享该页的进程数）。如果该页在对换设备上，则增加该页的对换使用表中的引用数。

这时，该页可通过两个区来引用，这一共享持续到一个进程写该页。这时内核拷贝该页，从而每一个区有了自己的页。为了实现这一点，内核在执行 fork 期间对父进程和子进程私有区中的每一个页表项都置上“写时拷贝”位。不论哪个进程写这些页，都会导致一个保护错，在处理这一错误时，内核为出错的进程做一个该页的新拷贝。这样，物理页的拷贝延迟到进程真正需要时才进行。

图 9-15 给出了一个进程执行 fork 时的数据结构。因为两个进程共同引用共享正文区 T 的页表，所以 T 区的引用数为 2，该正文区内页的 pfdata 引用数为 1。内核分配了一个新的子进程数据区 C1，它是父进程中 P1 区的一个拷贝。这两个区的页表项是完全一样的（图中以虚地址为 97K 的那个表项说明这一点）。有两个页表项指向 pfdata 表项 613。它的引用数为 2，这说明有两个区引用了该页。

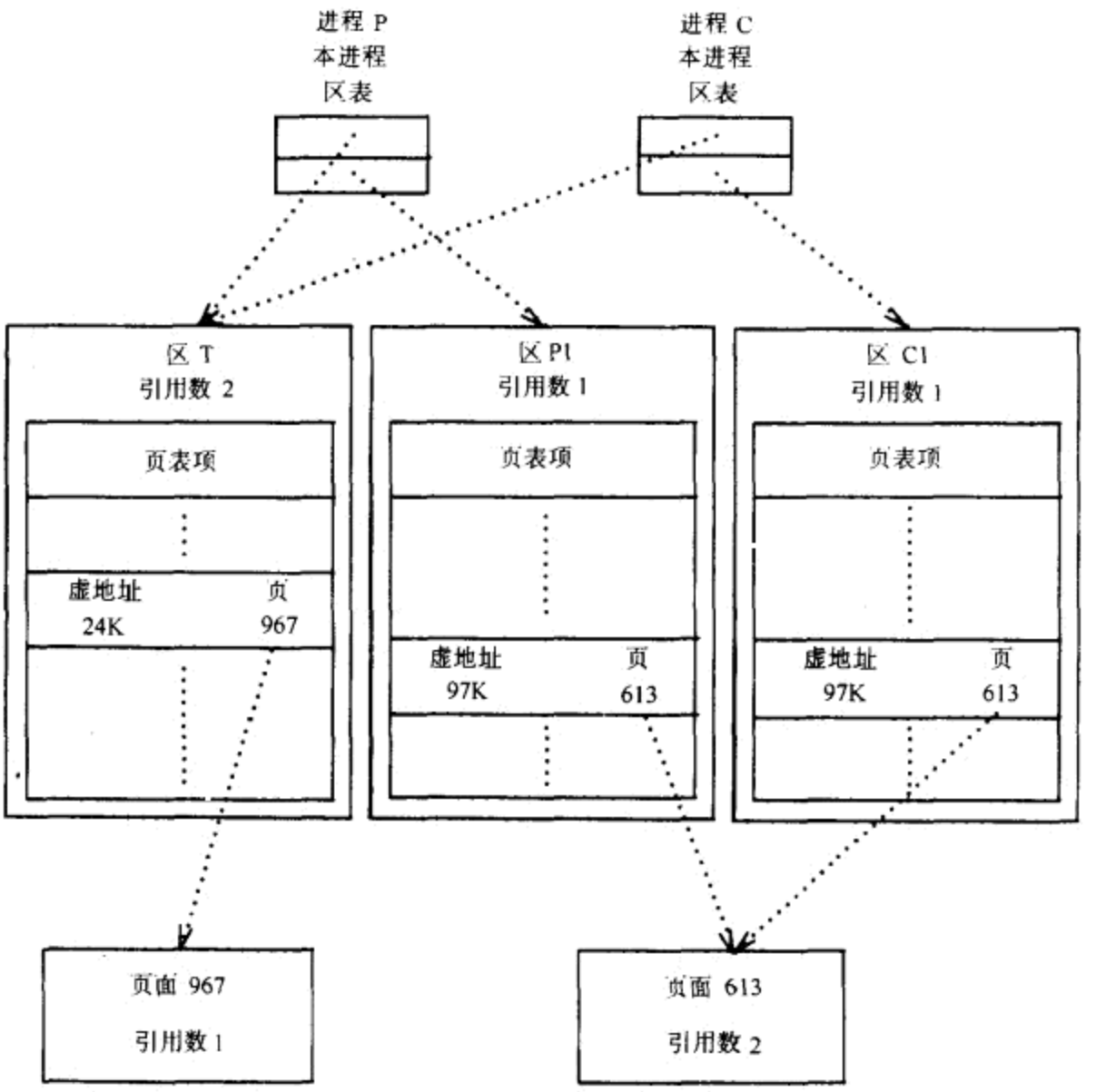


图 9-15　进程执行 fork 时的某一页面

在 BSD 系统中，系统调用 fork 的实现做了父进程所有页的物理拷贝。然而，由于认识到不进行这样的拷贝所带来的性能上的改善，BSD 系统又提供了一个系统调用 vfork。系统调用 vfork 假定子进程从系统调用 vfork 返回时，将立刻调用 exec。因为 vfork 不拷贝页表，所以它比系统 V 的 fork 快。但子进程与父进程在同一个物理地址空间上执行（直到调用exec 或 exit），因此可能重写父进程的数据和栈。如果程序员不正确地使用 vfork，将会造成危险的情况。所以，正确使用 vfork 的责任在于程序员。系统 V 和 BSD 系统在实现上的分歧具有哲理性：内核是应将其实现上的特点对用户隐藏起来，还是让高级用户有机会利用实现上的特点来更有效地完成逻辑功能呢？

以图 9-16 中的程序为例。在系统调用 vfork 之后，子进程没有执行 exec，而是重置了变

量 global 和 local，然后退出[⊖]。系统保证了父进程挂起，直到子进程执行 exec 或 exit。当父进程最后恢复运行时，却发现它的两个变量的值与调用 vfork 前不同了！如果子进程从调用 vfork 的函数返回的话，将会产生更惊人的结果（见习题 9.8）。

```
int global;
main()
{
    int local;
    local=1;
    if (vfork () ==0)
      {
        /* 子进程 */
        global=2;       /* 写父进程的数据空间 */
        local=3;        /* 写父进程的栈 */
        _exit ();
    }
    printf ("global %d local %d \n", global, local);
}
```

图 9-16　vfork 与进程空间的破坏

2. 调页系统中的系统调用 exec

第 7 章描述了当一个进程调用系统调用 exec 时，内核从文件系统中将可执行文件读入内存。但在请求调页系统中，可执行文件可能因太大而不能被全部装入可用的内存。所以，内核并不为可执行文件预先分配内存，而是“假装地”认为文件已在内存，并随着需要而逐步分配内存。内核首先为可执行文件分配页表和磁盘块描述项，并将页表表项设置为“请求填入”（对非 bss 数据）或“请求清零”（对 bss 数据）。在用算法 read 读入文件时，进程每读一页都产生一个有效性错。出错处理程序查看该页是请求填入还是请求清零。“请求填入”表示该页的内容将立即被可执行文件的内容所覆盖，所以不用清除。“请求清零”表示该页的内容应被清掉。9.2.3 节中的有效性错处理程序描述了如何做这些工作。假如进程不能装入内存，“偷页”进程（page-stealer）定期地将一些页换出内存，为要进入内存的文件腾出空间。

很明显，这一方案效率不高。首先，在读可执行文件的每一页时，进程都要产生一个页面错，尽管它可能永远不会存取该页。第二，偷页进程可能在 exec 完成之前就将某些页换出。如果进程一开始就需要这些页的话，这就造成了这些页的两次额外的页对换操作。为了使 exec 效率高一些，如果数据适当地对准，并由一个特殊的魔数所指示，内核就可以直接从可执行文件中调页。然而，使用标准算法（如第 4 章中的算法 bmap）来存取一个文件，将会使从间接块调页开销很大，因为读一个块需要多次存取高速缓冲。而且由于算法 bmap 不是可再入的，所以可能产生一致性的问题。在系统调用 read 期间，内核要设置在 u 区中的各种 I/O 参数。如果一个进程在系统调用 read 期间企图往用户空间拷贝数据时，产生了一个页面错，则为了从文件系统读入该页，内核就会重置 u 区中的这些字段。因此，内核不

⊖ 退出使用了_exit。因为 exit“清除”子进程和父进程的标准 I/O（用户级）的数据结构，这样会使父进程中的 printf 语句不能正确工作。这是 vfork 的另一个不幸的副作用。

能使用一般的算法来从文件系统中读入所缺的页。当然，在一般情况下，这些算法是可再入的，因为每一进程有独立的u区，并且一个进程不能同时执行多个系统调用。

为了从一个可执行文件中直接调页，内核在做exec时，找出该可执行文件的所有磁盘块号，并将这一串块号与文件的索引节点联系起来。在为这样一个可执行文件设置页表时，内核将含有该页的逻辑块号（从文件中的第0块开始）填入磁盘块描述项。以后，有效性错处理程序将使用这一信息将该页从文件装入内存。图9-17给出了一个典型的情况，其中，磁盘块描述项指出了该页位于文件中第84个逻辑块。内核则根据区到索引节点的指针，找到该索引节点并查出相应的磁盘块号（279）。

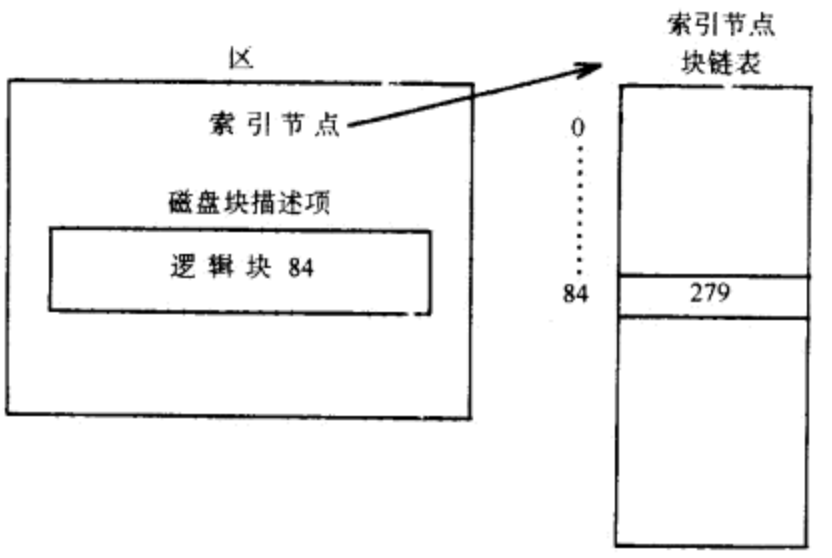


图9-17 将文件映射到一个区

9.2.2 偷页进程

偷页进程是一个核心进程，它将不再是进程工作集的页偷偷地换出内存。内核在系统初启时创建偷页进程，并在系统的生命期内，每当空闲页很少时就调用它。偷页进程检查每一个活动的、非上锁的区，跳过上锁的区，期待下遍扫描区链表时再检查它们，并增加所有有效页的年龄字段。当一个进程在一个区中出现有效性错，内核就锁住该区，这样偷页进程就不能将出错页换出了。

在内存中的页有两个调页状态：该页的年龄在增大，但还不能对换；或该页可被对换并可被重新分配给其他虚拟页。第一种状态表示进程最近存取过该页，所以该页是进程的工作集中的一员。一些机器硬件在它们访问一个页面时要置访问位，但如果硬件没有这一功能的话，也可以用软件的方法来替代（9.2.4节）。偷页进程复位这些页面的访问位，但记住自从上次该页被访问以来已检验了多少遍。这样，对应于在该页可被换出之前偷页进程对它的检查次数，第一种状态有若干子状态（见图9-18）。当遍数超过了某个阈值，内核就将该页置为第二种状态，即该页可被换出了。在可被换出之前，一个页面可被计龄的最大限度取决于实现。它受限于页表表项中可用位的位数。

图9-19给出了进程访问一个页面和偷页进程检查该页之间的交互作用。开始时，该页在内存中，图中数字显示了在二次内存访问之间，偷页进程检查的遍数。在第二遍访问之后，一个进程访问了该页，该页的年龄便降为0。类似地，在另一遍检查之后，一个进程又访问了该页。最后，偷页进程查出该页已连续被检查了三遍而中间没有被访问，它便将该页

面换出。

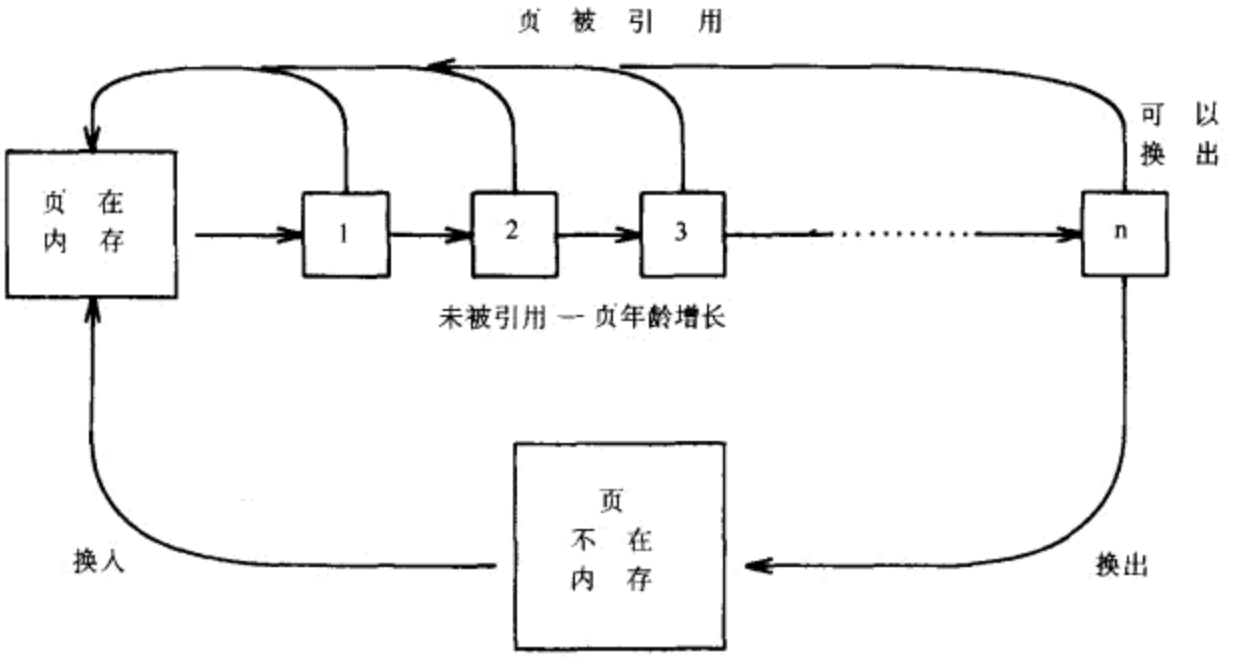


图 9-18　页面计龄的状态图

页状态	时间（上次访问以来）	
在 内 存	0	
	1	
	2	页被访问
	0	
	1	
	0	页被访问
	1	
	2	
	3	页被换出
不 在 内 存		

图 9-19　某一页面计龄的例子

如果有两个或更多的进程共享一个区，这些进程更新同样一组页表项的访问位。因此，页面可能是多个进程工作集的一部分，但这对偷页进程没有影响。若某一页面是任一进程工作集的一部分，它就驻留在内存；若它不在任何进程的工作集内，则它可被换出。一个区如果比其他区具有更多的在内存的页也没有关系：偷页进程并不试图从所有活动的区中换出相同数目的页。

当系统中可用的空闲存储低于某一低水准标志时，内核就唤醒偷页进程，偷页进程不断地将一些页换出，直到系统中可用的空闲存储超过了某一高水准标志。高水准标志和低水准标志的使用减少了抖动：如果内核只用一个阈值，内核可以换出足够的页以得到高出阈值的空闲页，但为了把缺页换回内存，空闲页的数目又将迅速降到阈值以下，结果是偷页进程在阈值附近抖动。通过将页面换出，直到空闲页的数目超过了一个高水准标志，使得空闲页数

目要经过较长的时间才能降到低水准标志。因此，偷页进程的运行就不那么频繁了。系统管理员可以配置高水准和低水准标志的值，以获得最好的系统性能。

当偷页进程决定要换出某页时，它考虑在对换设备上是否已有该页的拷贝。这时有三种可能性：

(1) 如果对换设备上没有该页的拷贝，内核要“安排”换出该页：偷页进程将该页放到一个要换出的页面链上，然后继续运行，此次对换便逻辑地完成了。当要换出的页面链达到一定的限度时（取决于磁盘控制器的能力），内核便将这些页写到对换设备上去。

(2) 如果该页的拷贝已在对换设备上，而且没有进程修改过它在内存中的内容（其页表项的修改位为0），内核将该页表项的有效位清零，pfdata 表项的引用数减 1，并将该表项放入空闲链表以供进一步的分配。

(3) 如果页面的拷贝已在对换设备上，并且进程已修改了该页面在内存的内容，像上面一样，内核要安排换出该页，并释放它在对换设备上目前占据的空间。

在第一种和第三种情况下，偷页进程要将该页拷贝到对换设备上。为了说明后两种情况之间的不同，假设对换设备上的某一页在进程发生有效性错后被换入内存，假设内核不自动清除该磁盘拷贝，还假设最终偷页进程又决定换出该页。如果自从该页面被换入后，没有进程修改过该页，则内存的拷贝与磁盘上的拷贝完全一致，因此没有必要将该页写到对换设备上去。反之，如果某个进程写了该页面，则内存拷贝就不同于磁盘拷贝。这样，在释放了该页目前在对换设备上占据的空间后，内核必须将该页拷贝到对换设备上。内核并不立即重新使用刚释放的对换设备上的空间，这样就可以保持对换设备空间的连续性以获得更好的性能。

偷页进程填写一个要被换出的页面的链表，其中的页可能来自不同的区。当链表满时，偷页进程将它们换到对换设备上去。并不是进程的每一页都要换出：例如，某些页的年龄可能还不够大。这点不同于对换进程的策略，对换进程从内存中换出进程的所有页面，但往对换设备上写数据的方法却与 9.1.2 节中描述的对换系统中的算法完全一样。如果对换设备上没有足够的连续空间，那么内核只得一次换出一页，显然这样开销要大一些。调页机制要比对换机制具有更多的存储碎片，因为内核一次成功地换出多个页面，但一次只换入一个页面。

当内核向对换设备写一页面时，它要清除该页表项的有效位，并将 pfdata 表项的引用数减 1。如果引用数减为 0，内核就将 pfdata 表项放在空闲链表的尾部，它一直被存放在这里，直到被重新分配。如果引用计数不为 0，说明由于先前调用了 fork，造成有几个进程正在共享该页，但内核仍然将该页换出。最后，内核分配对换空间，将对换地址保存在磁盘块描述项内，并将该页的对换使用表中的计数加 1。如果一个进程发生了一个有效性错，但该页还在空闲链中，内核仍然可以从内存中恢复它，而不一定要从对换设备上找回它。然而，如果它在对换链中的话，该页还是要对换出去的。

举例来说，假如偷页进程分别从进程 A，B，C 和 D 换出 30、40、50 和 20 页，并且偷页进程在每一次磁盘写操作中，可向对换设备写 64 个页。如果偷页进程以 A，B，C，D 这样的顺序来检查进程的页面，图 9-20 给出了将会发生的一系列页对换操作。偷页进程在对换设备上分配 64 个页面的空间，换出进程 A 的 30 个页面和进程 B 的 34 个页面。然后，它在对换设备上分配更多的空间以容纳另外 64 个页面，并且换出进程 B 剩下的 6 个页面，进

程 C 的 50 个页面和进程 D 的 8 个页面。对这两次写操作，在对换设备上的这两块区域不必是连续的。偷页进程将进程 D 剩下的 12 个页面留在准备换出的页面链表里，在这一链表被填满之前，并不将它们换出。随着进程因缺页而从对换设备上换入页面，或因进程退出而不再使用页面，对换设备上的空闲空间会不断地扩大。

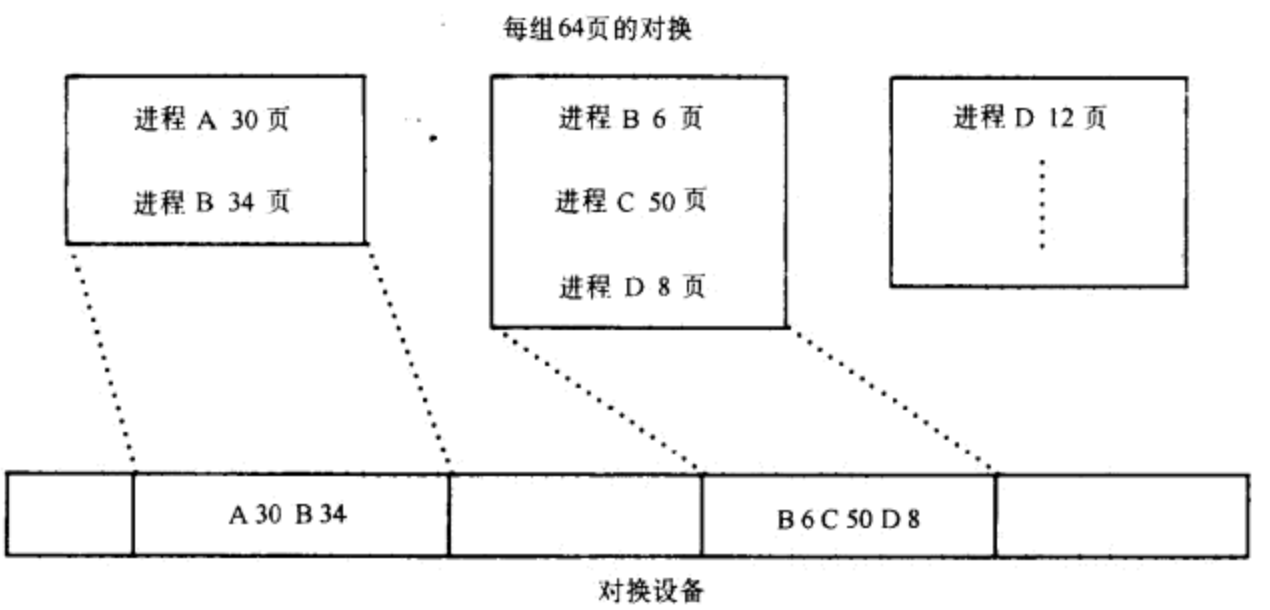


图 9-20　调页机制中对换空间的分配

总而言之，将页换出内存有两个阶段。第一个阶段，偷页进程找出应换出的页面，并将页号放入要被换出的页面链表中；第二个阶段，内核在适当的时候将页面拷贝到对换设备上去，清除页表表项中的有效位，将 pfdata 表项的引用数减 1。如果引用数为 0 的话，则将 pfdata 表项放在空闲链表的尾部。直到该页面被重新分配以前，该物理页的内容还是有效的。

9.2.3　页面错

系统产生两种页面错：有效性错及保护错。因为页面错处理程序可能要将磁盘上的某页读入内存，并在 I/O 操作期间进入睡眠，所以，对于中断处理程序不能睡眠这个一般规则而言，页面错处理程序是个例外。尽管如此，因为页面错处理程序睡眠在发生存储错的进程上下文中，也就是说，页面错总是与正在运行的进程相关联，因此，不会使任意的其他进程进入睡眠。

1. 有效性错处理程序

如果一个进程企图存取一个有效位为零的页，它将产生一个有效性错，内核要调用有效性错处理程序（图 9-21）。对于在一个进程虚空间范围以外的页面，及虽然在虚空间以内但当前没有分配物理页面的页，其有效位为零。硬件向内核提供存取虚空间时产生的内存错的地址，由内核找出相应页面的页表项及磁盘块描述项。内核锁住含有该页表项的区，以防止资源竞争条件。如果偷页进程企图换出该页，就会产生这种竞争条件。如果磁盘块描述项没有出错页面的记录，那么试图进行的内存访问即为非法的，内核将向违例进程发送一个“段违例”软中断信号（参见图 7-25）。这类似于在对换系统中，当一个进程存取一个非法地址时的处理过程，所不同的是在对换系统中，可以立即识别出错误访问，因为一个进程的所有有效页全都驻留在内存中。如果内存访问是合法的，则内核分配一个页面的内存，来读入对

换设备上或可执行文件中该页的内容。

```
算法 vfault /* 有效性错处理程序 */
输入：进程发生页面错的地址
输出：无
{
    找出对应于出错地址的区、页表项、磁盘块描述项，锁该区；
    if (出错地址在进程虚空间之外)
    {
        向进程发软中断信号（SIGSEGV：段违例）；
        goto out;
    }
    if (出错地址现在已有效)
        /* 进程在此之前可能睡眠 */
        goto out;
    if (页面在高速缓冲中)
    {
        从高速缓冲中移出该页；
        修改页表项；
        while（页的内容无效） /* 另有进程先缺此页 */
            sleep（事件：页的内容有效）；
    }
    else /* 页不在高速缓存中 */
    {
        给该区分配新页表；
        将新页放入高速缓冲，更新 pfdata 表项；
        if（该页以前未被装入而且该页“请求清零”）
            将分配的页清零；
        else
        {
            从对换设备或可执行文件中读虚页；
            sleep（事件：I/O 完成）；
        }
        唤醒进程（事件：页内容有效）；
    }
    设置页有效位；
    清修改位，年龄位；
    重新计算进程优先权；
out：解锁区；
}
```

图 9-21 有效性错处理程序的算法

引起出错的页面的状态是下列五种状态之一：

(1) 在对换设备上，且不在内存中。

(2) 在内存中的空闲页链表上。

(3) 在一个可执行文件中。

(4) 标志为“请求清零”。

(5) 标志为“请求填入”。

下面让我们来详细考查每一种状态的情况。

如果一个页面在对换设备上，但不在内存中（状态 1），则说明该页曾一度在内存中，但已被偷页进程换出。内核从磁盘块描述项中找到存放该页面的对换设备和块号，并证实该页不在页缓冲区内。内核修改页表项，使其指向要读入的页面，将 pfdata 表项放入一个散列链表，以便加速有效性错处理程序以后的操作。然后，内核从对换设备上读入该页面。出错进程进入睡眠，直到 I/O 操作完成，这时内核唤醒那些等待该页的内容被读入的进程。

作为一个例子，考虑图 9-22 中对应虚地址 66K 的页表表项。如果进程在存取该页时产

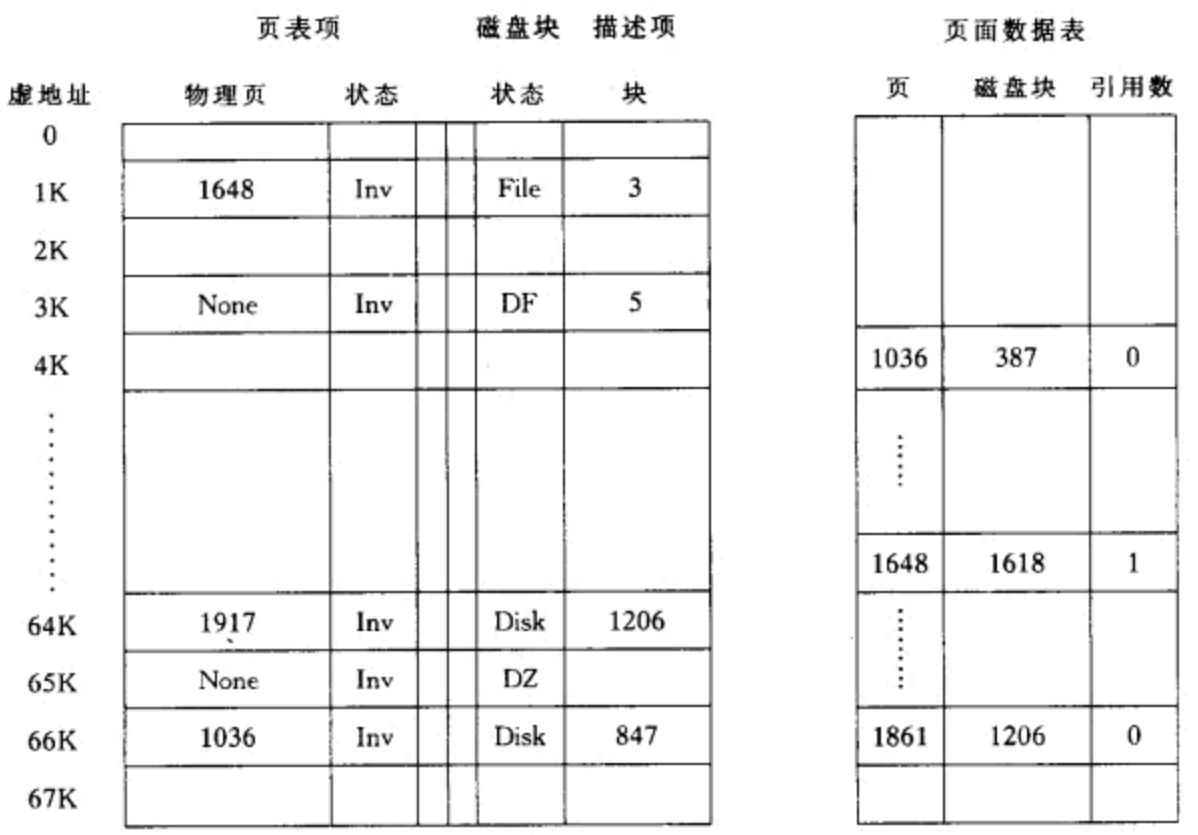


图 9-22　有效性错的一个实例

生一个有效性错，有效性错处理程序查找磁盘块描述项，发现该页在对换设备（假定只有一个对换设备）的第 847 块上。因此，这一虚地址是合法的。然后，有效性错处理程序查找页面缓冲区，但没有找到对应第 847 块的项，所以在内存中没有该虚拟页的拷贝。出错处理程序必须从对换设备上读入该页。内核分配第 1776 页（见图 9-23），并从对换设备上读入该页的内容，然后修改该页的页表项，使它指向第 1776 页。最后，内核修改磁盘块描述项，以指明该页面正在换入，内核还修改第 1776 页的 pfdata 表项，用以说明在对换设备的第 847 块上有该虚拟页面的拷贝。

当发生有效性错时，内核并不总是要进行 I/O 操作，甚至磁盘块描述项指示该页面被换出时也是如此（状态 2）。可能有这样的情况：页面被换出后，内核还没有重新分配这一物理页或另一进程缺页时又将该虚页读入到另一物理页上。在以上两种情况下，有效性错处理程序在页缓冲区中找出该页；删除磁盘块描述项中的块号；重置页表表项以指向刚找到的

图 9-23 页面被换入内存之后

页表；将它的引用数加 1；并在需要的情况下，将该页从空闲链表中取出。例如，假定一个进程存取图 9-22 中所示的虚地址 64K 时发生有效性错。内核在查页面缓冲池时找到了与磁盘块 1206 相关联的页面 1861。磁盘块 1206 与磁盘块描述项中的块号一致。内核为虚地址 64K 重新设置页表表项，指向第 1861 页面；设置有效位为有效，然后返回。这样，通过磁盘块号将页表表项与 pfdata 表项联系起来。这说明了为什么要在这两个表中都保存磁盘块号。

与上述情况类似，如果另一个进程在同一页上已发生有效性错，但还没有完全地将该页读入内存，那么有效性错处理程序也不必再将其读入内存。有效性错处理程序找到由另外一次有效性错处理程序锁住的该页表项所在的区。它进入睡眠，直到那个有效性错处理程序完成。该进程被唤醒后，发现该页已成为有效页，然后返回。图 9-24 给出了这一情形。

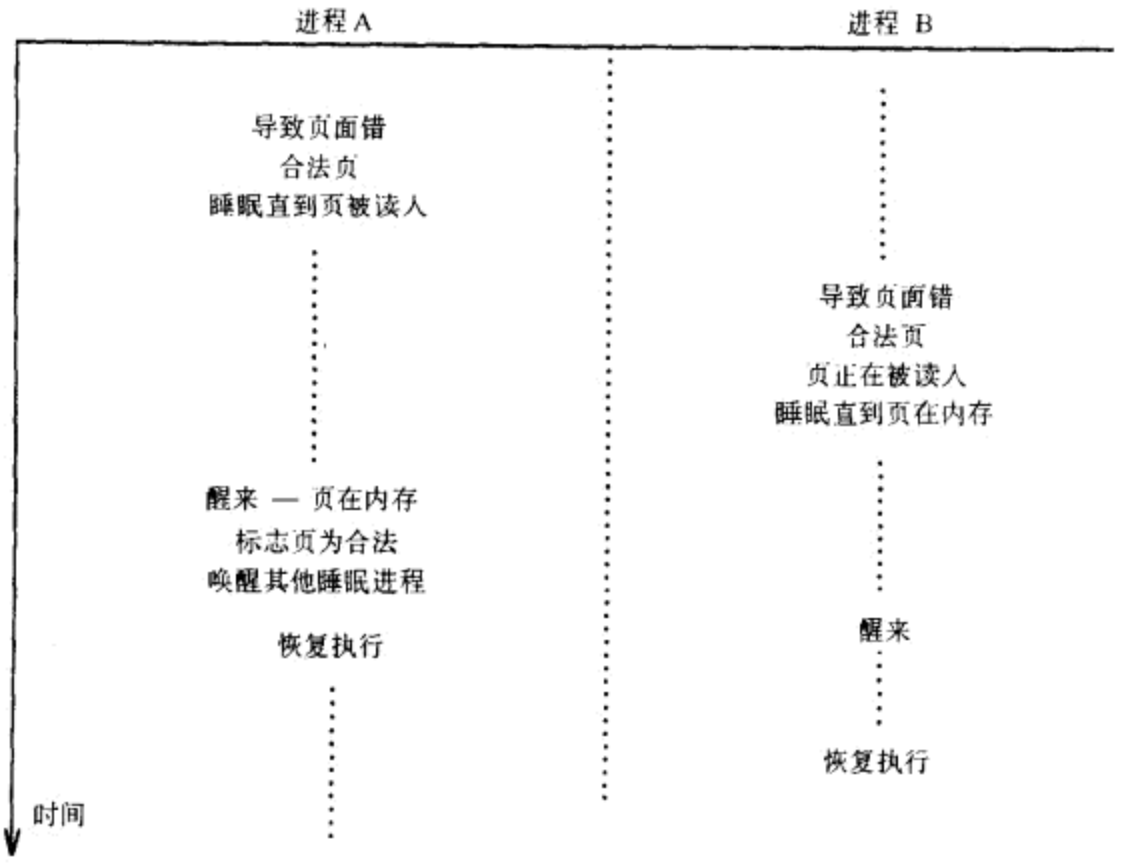


图 9-24 同一页面上的二重错

如果一个页面的拷贝不在对换设备上，而是在原始的可执行文件中（状态 3），那么内核就要从原始文件中读入该页。有效性错处理程序检查磁盘块描述项，找出该页在其所在的文件中的逻辑块号以及与该区表项相关联的索引节点。它以逻辑块号为偏移量，查找在系统调用 exec 时挂在该索引节点上的磁盘块号数组，找出相应的磁盘块号，然后将该页读入内存。例如，在图 9-22 中，虚地址 1K 的磁盘块描述项显示出该页的内容在可执行文件中的第

3 个逻辑块上。

如果进程在一个标有“请求清零（DZ）”或“请求填入（DF）”的页面上发生缺页（状态 4 和状态 5），则内核在内存中分配空闲页并修改相应的页表项。对于“请求清零”的页，内核还要将该页清零。最后，内核清除“请求清零”或“请求填入”标志。该页现在为内存中的有效页面，而且，它的内容没有被复制到对换设备上或一个文件系统中。在存取图9-22中所示的虚地址 3K 和 65K 时，就会发生上述情况：自从文件被系统调用 exec 执行以来还没有进程存取过这几页。

有效性页面错处理程序以清除修改位和设置有效位为结束动作。它要重新计算进程的优先权，因为进程可能在有效性错处理程序中以核心级优先权睡眠，这使它返回用户态时具有不公平的高调度优先权。最后，当它返回用户态时，要检查在处理有效性错期间，是否收到任何软中断信号。

2. 保护错处理程序

第二类内存错，即进程引起的保护错，是进程试图存取一个许可位关闭的有效页面所产生的。（请回忆图 7-22 中的例子，一个进程企图写它的正文空间。）在一个进程试图写一个页面，而该页面在系统调用 fork 时被置上了“写时拷贝”位时，也会导致一个保护错。内核必须要判断不许存取是由于页面“写时拷贝”的要求而引起的，还是真的发生了某些非法操作。

硬件为保护错处理程序提供错误发生时的虚地址，错误处理程序负责找到相应的区和页表表项（见图 9-25）。它要锁住该区以防当保护错处理程序在其上操作时，该页面被偷页进程换出。如果保护错处理程序判断错误是由于页面设置了“写时拷贝”标志而引起的，并且如果该页被其它进程所共享的话，内核则分配一个新页并将老页的内容拷贝到新页上。其他的进程仍然保持引用老的页面。在页面拷贝和用新的页号修改页表表项后，内核要将老页面的 pfdata 表项的引用数减 1。图 9-26 给出了这种情形：这里，有三个进程共享物理页 828。进程 B 写该页，但发生了一个保护错，因为该页设置了“写时拷贝”位。保护错处理程序分配一个新页，786，并将 828 页的内容拷贝到新页上，然后，将 828 页的引用数减 1，修改进程 B 所存取的页表表项，使其指向 786 页。

如果设置了“写时拷贝”位，但没有其他进程共享该页，则内核允许进程重新使用该物理页。内核清除“写时拷贝”位并解除该页与其磁盘拷贝的联系，因为其他进程可能共享该磁盘拷贝。然后，内核将该页的 pfdata 表项从页面队列中移出，这是因为虚页的新拷贝并不在对换设备上。内核将该页的对换使用数减 1。如果减为 0，则释放对换空间（见习题 9.11）。

考虑某页表表项无效而且设置了“写时拷贝”位，此时又产生了保护错的情况。假定系统在一个进程存取该页时，首先处理有效性页面错（习题 9.17 给出了相反的情况）。不管怎样，保护错处理程序总要检查该页是否有效，因为在锁一个区时它可能进入睡眠，而在此期间，偷页进程可能会将该页换出内存。如果该页非有效（即有效位被清除），则出错处理程序立即返回，进程将产生一个有效性错。内核处理有效性错，但进程还将产生保护错。更可能的是，内核最终处理保护错而不再有进一步的干扰，因为页面被计龄到足以被换出需要很长时间。图 9-27 给出了这种情况下发生的一系列事件。

```
算法 pfault /* 保护错处理程序 */
输入：进程发生页面错的地址
输出：无
{
    找出相对于出错地址的区、页表项、磁盘块描述项、页面数据表，并锁区；
    if (页面在内存中无效)
        goto out;
    if (未设置写对拷贝位)
        goto out;   /* 真发生程序错——发信号 */
    if (页面引用数>1)
    {
        分配新的物理页；
        将老页的内容拷贝到新页；
        老页面的引用数减 1；
        修改页表项使其指向新的物理页；
    }
    else    /* "偷页"，因为没有其他用户在使用它 */
    {
        if (在对换设备上有该页拷贝)
            释放对换设备上的空间，解除与页面的联系；
        if (页面在页面散列队列中)
            从页面队列中取出；
    }
    设置修改位，清除页表项中的写时拷贝位；
    重新计算进程优先权；
    检查软中断信号；
out: 解锁区；
}
```

图 9-25 保护错处理程序的算法

当保护错处理程序结束执行时，它设置修改位和保护位，清除写时拷贝位。它重新计算进程的优先权，并检查是否有软中断信号。这和有效性错处理程序结尾时所做的一样。

9.2.4 在简单硬件支持下的请求调页系统

如果硬件能设置访问位及修改位，并且，当一个进程写一个设置了"写时拷贝"位的页面时，硬件能产生一个保护错，则请求调页的算法是最高效的。但是，如果硬件仅能识别有效位和保护位，仍然可能实现我们在这里所描述的算法。如果有效位被复制到一个软件有效位，它指示该页是真有效还是无效，则内核可以关闭硬件有效位，并用软件模拟来设置其他的指示位。例如，VAX-11 的硬件没有访问位（见[Levy 82]）。内核可关闭某页的硬件有效位，并按下述方案行事：如果一个进程访问该页，将导致一个页面错，因为该页的硬件有效位被关闭。这之后，页面错中断处理程序检查这一页面。因为软件有效位被设置，所以内核知道该页确实有效并在内存中；内核设置上软件访问位并打开硬件有效位，由此内核得到了该页已被访问过的信息。接下来的对该页的访问将不再引起页面错，因为硬件有效位已被打

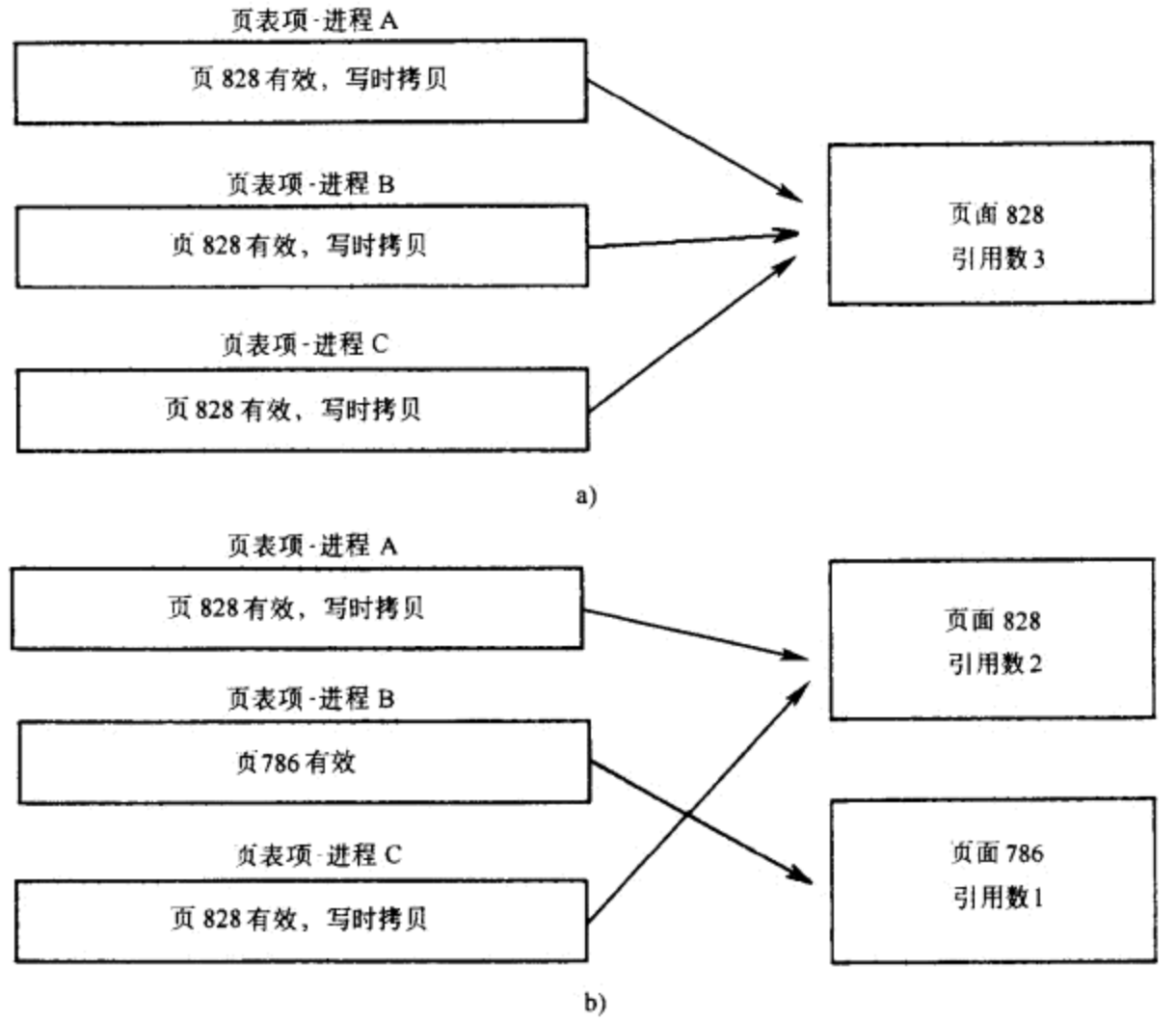


图 9-26 写时拷贝位所引起的保护错

a）进程 B 导致页保护错之前 b）为进程 B 运行了保护错处理程序之后

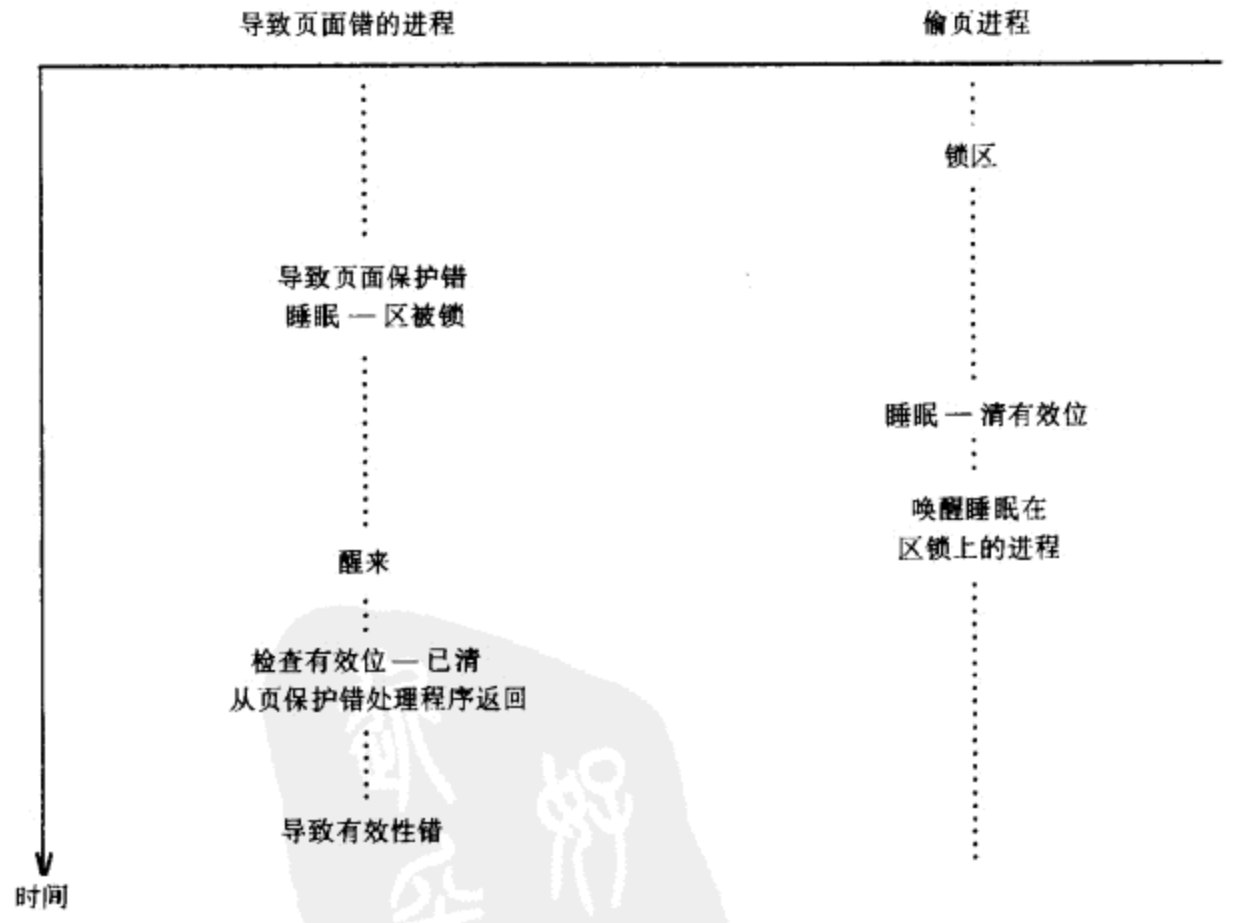


图 9-27 保护错和有效性错的交互作用

开。当偷页进程检查该页时，它重新关闭硬件有效位，从而在进程访问该页时又引起页面错，重复上面所描述的过程。图 9-28 给出了这一方案的例子。

图 9-28 用软件模仿硬件的修改位

a）修改页之前 b）修改页之后

9.3 对换和请求调页的混合系统

尽管请求调页系统比对换系统更灵活地使用内存，但由于缺少内存，也可能产生偷页进程和有效性错处理程序抖动的现象。如果所有进程的工作集的总和大于一个机器上的物理存储空间，那么，因为有效性错处理程序不能为进程分配页面，所以它经常睡眠。偷页进程也不能足够快地进行换页，因为所有的页面都在工作集内。系统此时的吞吐量很差，因为内核消耗了过多的时间开销，以疯狂的步伐来反复安排内存空间。

系统 V 的内核同时运行对换和请求调页算法，以避免抖动问题。当内核不能为一个进程分配页面时，内核就唤醒对换进程，并将调用进程置为与“就绪且换出”等价的一个状态。可能有若干进程同时处于这一状态。对换进程换出整个进程，直到可用内存达到高水准标志。对每个换出的进程，内核使一个“就绪且换出”的进程变为运行就绪的进程。它并不通过通常的对换算法换入这些进程，而是让它按需要来请求调页。如果系统中有足够的内存空间，对换进程将允许其他进程进行请求调页。这一方法降低了系统的缺页速度，减少了抖动。从原理上，它类似于 VAX/VMS 操作系统所采用的方法（[Levy 82]）。

9.4 本章小结

本章讨论了 UNIX 系统 V 的进程对换和请求调页算法。对换算法在主存和对换设备之间对换整个进程。如果因进程大小的增加使内存没有空间了（由于系统调用 fork，exec 或 sbrk 的作用，或正常的栈增长的结果），或必须为要换入的进程腾出空间时，内核就要换出进程。内核用对换进程，即进程 0 来换入进程。每当对换设备上存在“就绪”进程时，内核就执行对换进程。对换进程换入所有“就绪”进程，直到对换设备上没有“就绪”进程了，或内存没有空间了。在后一种情况下，它试图从内存中换出进程，但为了减少抖动，它禁止换出不满足对换驻留条件的进程。因此，对换进程每次运行并不总是成功地换入所有进程。如果对换进程有事可做，时钟处理程序就每秒一次地唤醒对换进程。

请求调页系统的实现使得进程甚至在其整个虚地址空间都不在内存时也能运行。因此，进程的虚地址空间的大小可以超过在一个系统中可用的物理存储的大小。当内核只剩很少空闲页时，偷页进程遍历所有区的每个活动页面，标记出那些已计龄到可被换出的页面，并最终将这些页面拷贝到对换设备上去。当进程寻址到某个已被换出的页面时，它将导致一个有效性错。内核调用有效性错处理程序为该区分配一个新的物理页，并拷贝该虚页的内容到主存中。

请求调用算法的实现改善了系统的性能。首先，系统利用写时拷贝位改进了系统调用

fork，从而在大多数场合下避免了页面的物理拷贝。第二，内核可以从文件系统的可执行文件中调出页面的内容，这避免了系统调用 exec 要将文件立即读入内存的要求。这一点提高了系统的性能，因为有些页面在一个进程的生存期内可能永远不会被用到，同时，它还减少了额外的抖动。这些抖动可能是由于在页面被使用前就被偷页进程换出所引起的。

9.5　习题

1. 粗略地设计一个算法 mfree，它释放空间，将所释放的空间还给一个映射图。

2. 9.1.2 节中指出，系统要锁住正在被换出的进程，因而保证在开始操作的同时，没有其他进程能将其换出。如果系统不锁该进程，会发生什么情况？

3. 假定 u 区中含有一个进程的段表和页表，内核如何才能将 u 区换出？

4. 若核心栈是在 u 区的话，为什么进程不能将其自身换出？怎样编写一个核心进程的程序来换出其他进程？它应如何被调用？

* 5. 假定内核试图换出一个进程来为对换设备上的一个进程腾出空间，如果在任何对换设备上都没有足够的空间，则对换进程进入睡眠，直到有更多的存储空间变成可用的。是否有可能出现这样的情况：所有在内存的进程都在睡眠，同时在对换设备上的进程都处在就绪状态？试讨论一下这种情况是如何发生的；内核应当如何纠正这种情况？

6. 重新考虑图 9-10 中的例子，如果在内存中只有可容纳 1 个进程的空间，情况又将如何？

7. 重新考虑图 9-11 中的例子，构造一个例子，使其中有一个进程总是用不上 CPU。有什么方法可以防止这一情况的发生？

8. 在 4.2BSD 系统中执行图 9-29 中的程序，将会发生什么情况？父进程的栈会发生什么问题？

```
main ()
{
    f ();
    g ();
}

f ()
{
    vfork ();
}

g ()
{
    int blast [100], i;
    for (i=0; i<100; i++)
        blast [i] =i;
}
```

图 9-29　系统调用 vfork 及更严重的破坏

9. 在一个系统调用 fork 之后，如果共享页被设置了写时拷贝位，则为什么在父进程之前调度子进程更具有优越性？内核如何才能强制地使子进程先运行？

*10. 有效性错算法表现为一次只换入一个正文页。如果将出错页的前后页面也预先换入的话，将改进算法的效率。试改进有效性错算法，使它可以预先换入页面。

11. 偷页进程和有效性错处理程序的算法都假定页面大小等于磁盘块大小。如果它们的大小不等，应怎样修改算法来适应这种情况？

*12. 当一个进程执行 fork 时，所有共享页的 pfdata 表项的页使用数都要增加。假定偷页进程换出一个（共享）页面到对换设备上，一个进程（假定是父进程）一会儿后又在该页上发生有效性错。该虚页现在还驻留在物理页上。解释子进程为什么总是可以找到该页合法的拷贝，甚至在父进程已写过该页面之后也是如此。如果父进程写该页面，为什么必须立即解除它自己与磁盘拷贝的联系？

13. 如果系统调用用完所有页面，则有效性错处理程序应做些什么事情？

*14. 设计一个算法来换出内核中不经常使用部分的页面。内核的哪些部分是不能被换出的？应如何标识这些部分？

```
struct fourmeg
{
    int page [512];      /* 假定 int 为 4 个字节长 */
} fourmeg [2048];

main ()
{
    for (;;)
    {
        switch (fork ())
        {
        case -1:   /* 父进程不能派生子进程---子进程太多 */
        case (): /* 子进程 */
            func ();
        default:
            continue;
        }
    }
}

func ()
{
    int i;

    for (;;)
    {
        printf ("proc %d loops again \n", getpid ());
        for (i=0; i<2048; i++)
            fourmeg [i] .page [0] =i;
    }
}
```

图 9-30 一个捣乱的程序

15. 设计一个算法，它用位图的方法代替本章中描述的映射图方法来管理对换设备上的空间。比较这两种方法的效率。

16. 假定机器硬件没有有效位但是有许可读、写和执行某页面的保护位。试模拟软件有效位的管理。

17. VAX-11 机器的硬件在检查有效性错之前检查保护错。这对缺页错处理程序的算法会产生什么影响？

18. 系统调用 plock 允许超级用户将调用该系统调用的进程的正文区和数据区锁在内存或解锁。对换进程和偷页进程不能将上锁的页面移出内存。使用这一系统调用的进程永远不用等待被换入，这保证了它们的响应速度要快于其他进程。这一系统调用应如何实现？是否应有一个选择项用以将栈区也锁在内存？如果被 plock 锁住的区的总存储空间大于可用的机器内存空间，则会发生什么情况？

19. 图 9-30 中的程序在做什么？试设计另一种调页策略。这一策略使每一进程的工作集有一个最大允许页面数。

第 10 章　输入/输出子系统

输入/输出子系统（简称 I/O 子系统）使进程能与外围设备如磁盘、磁带、终端、打印机及网络进行通信，而控制设备的内核模块常称为设备驱动程序（device driver）。通常设备驱动程序与设备类型是一一对应的：系统可以只含有一个磁盘驱动程序以控制所有的磁盘；一个终端驱动程序控制所有的终端；一个磁带驱动程序控制所有的磁带。对于配置有多于一个制造商的设备的系统——如两种商标的磁带，可以把它们视为两种不同的设备类型，配有两个分离的驱动程序，因为它们可能要求不同的命令序列以便正常操作。一个设备驱动程序（device driver）可以控制一种给定类型的许多物理设备，比如一个终端驱动程序可以控制连接到系统的所有的终端。驱动程序要对它所控制的这许多设备加以区分，也就是说，欲送往某一终端的输出决不能送往另一个终端。

系统支持“软设备”（software device），软设备没有与之相关联的物理设备。举例来说，它把物理存储器视为一种设备，使一个进程能存取它的地址空间范围以外的物理存储器，虽然存储器不是一种外围设备。例如 ps 命令从物理存储器读内核数据结构，以此报告进程统计量。类似地，驱动程序可以往物理存储器写对调试有用的跟踪记录，而跟踪驱动程序允许用户读这些记录。最后，第 8 章所描述的内核直方图（kernel profiler）是作为一个驱动程序实现的：一个进程写入在核心符号表中可找到的内核子程序的地址，并读直方图结果。

本章考察进程与 I/O 子系统之间的接口，以及机器和设备驱动程序之间的接口。它研究设备驱动程序的一般结构和功能，然后把磁盘驱动程序和终端驱动程序作为一般接口的特例来详细地讨论。本章以描述一种实现设备驱动程序的新方法结束，这种方法被称为流（stream）。

10.1　驱动程序接口

UNIX 系统包含两类设备：块设备（block device）和原始（或字符）设备（rew device）。正如第 2 章中所定义的，块设备如磁盘和磁带，对系统的其余部分来说好象是随机存取的存储设备；字符设备包括所有的其他设备，如终端和网络介质。块设备也可以有一个字符设备接口。

用户通过文件系统与设备接口（回忆图 2-1）：每个设备有一个像文件名那样的名字，并对它像一个文件那样存取。设备特殊文件有一个索引节点，在文件系统目录树中占据一个节点。设备文件以存储在它的索引节点中的文件类型与其他文件相区别，其类型或是“块”的，或是“字符”的特殊文件，相应于它所代表的设备。如果一个设备既有一个块接口，又有一个字符接口，它由两个设备文件所代表：它的块设备特殊文件以及它的字符设备特殊文件。正规文件的系统调用如 open，close，read 和 write，对设备有一个适当的含义，这点将在下面解释。系统调用 ioctl 提供给进程一个控制字符设备的接口，但它不可用于正规文件[○]。然而，并不是每一个设备驱动程序需要支持每个系统调用接口。举例来说，前面提到

○　相反地，系统调用 fcntl 提供在文件描述符级而非设备级操作的控制。若干其他的实现为所有文件类型使用 ioctl。

的跟踪驱动程序允许用户读由其他驱动程序写的记录，但它不允许用户写。

10.1.1　系统配置

系统配置是系统管理员规定与装备有关的参数的过程。某些参数规定内核表的大小，如进程表、索引节点表、文件表，以及分配给缓冲池的缓冲数目。其他参数规定设备配置：告诉内核这一装置包含哪些设备和它们的“地址”。比如，一个配置可以规定一块终端控制板是插到硬件底板上的哪一个特殊插槽上的。

设备配置的规定可以在三个阶段上进行。第一阶段，系统管理员可以把配置数据写入文件，在生成内核代码时这些文件被编译和连接。配置数据通常是以简单的形式规定的，然后由一个配置程序将之转换成适合于编译的文件。第二阶段，管理员在系统已经运行以后提供一些配置信息，内核动态地修改内部配置表。最后，自标识的设备允许内核辨认哪些设备是被安装上的，内核读入硬件开关以进行自身的配置。有关系统配置的细节已超出了本书的范围，然而在所有情况下，配置过程是产生或填入作为内核代码一部分的表格的过程。

内核与驱动程序的接口是由块设备开关表（block device switch table）及字符设备开关表（character device switch table）描述的（图 10-1）。每一种设备类型在表中有若干表项，这些表项在系统调用时引导内核转向适当的驱动程序接口。设备文件的系统调用 open 和 close

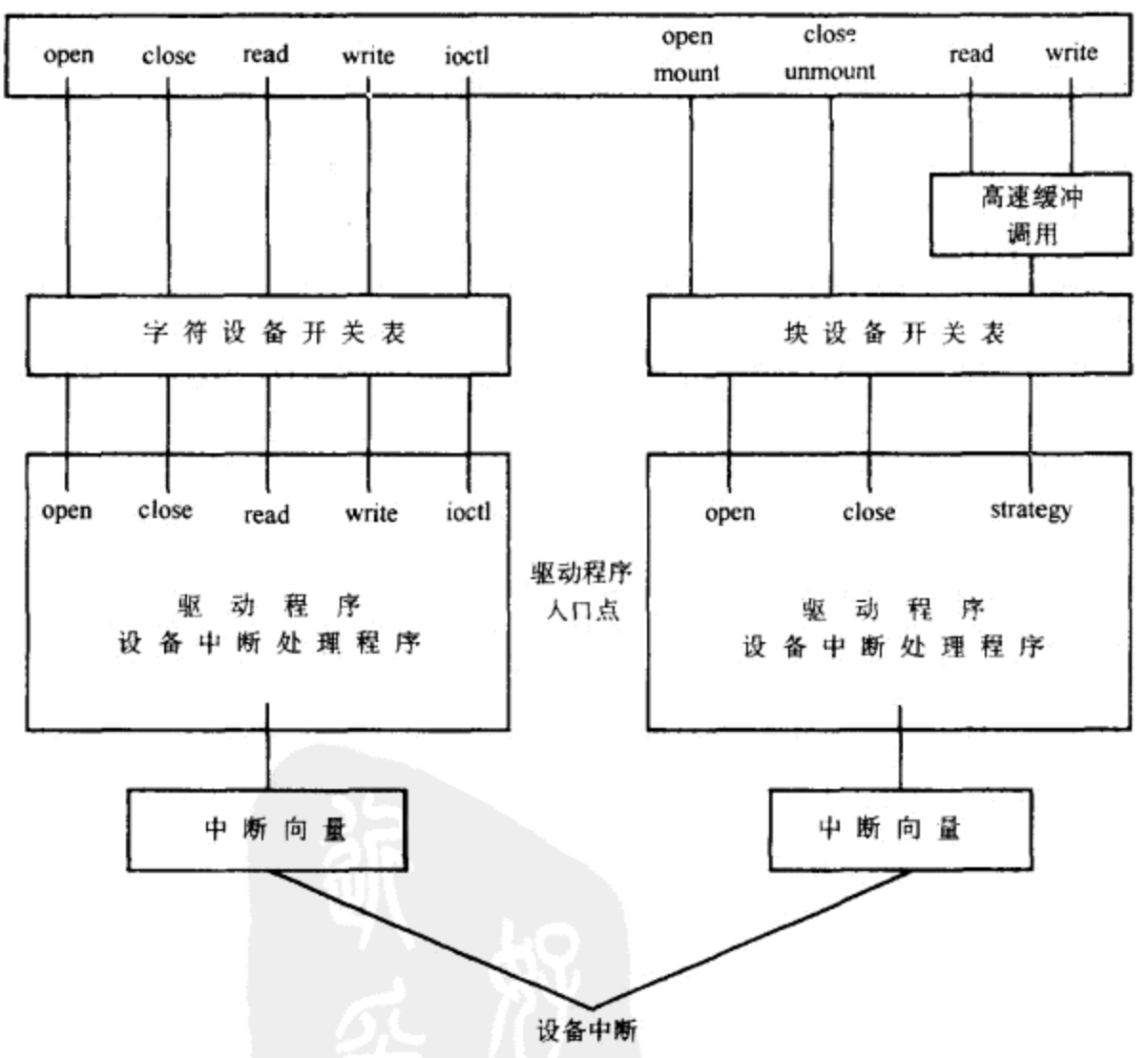


图 10-1　驱动程序入口点

根据文件的类型通过这两个设备开关表汇集，对块设备，系统调用 mount 和 unmount 也调用设备打开和关闭的过程。字符设备特殊文件的系统调用 read，write 和 ioctl 转向字符设备开关表中相应的过程，块设备以及安装在文件系统上的文件的系统调用 read 和 write 调用高速缓冲算法，后者又调用相应设备的策略（strategy）过程。某些驱动程序从它们的读和写过程内部地调用策略过程，这一点将在下面的论述中看到。下一节将详细地研究每一个驱动程序接口。

硬件与驱动程序的接口是由与机器有关的控制寄存器或操纵设备的 I/O 指令，以及中断向量组成的：当一设备中断出现时，系统识别发出中断的设备，并调用适当的中断处理程序。显然，软设备，如内核直方图驱动程序（见第 8 章），并没有硬件接口，但其他中断处理程序可以直接地调用一个“软件中断处理程序”，比如，时钟中断处理程序调用内核直方图中断处理程序。

系统管理员用 mknod 命令建立设备特殊文件。命令中提供文件类型（块或字符设备）和主、次设备号，mknod 命令调用系统调用 mknod 来产生设备文件。例如，在以下命令行

```
mknod /dev/tty13 c 2 13
```

中“/dev/tty13”是设备的文件名，c 规定这是一个字符设备特殊文件（b 规定一个块设备特殊文件），2 是主设备号（major number），而 13 是次设备号（minor number）。主设备号指示一种设备类型，它相应于块或字符设备开关表中适当的表项。次设备号指示该类设备的一个单元。如果进程打开块设备特殊文件“/dev/dsk1”，且其主设备号是 0，内核调用块设备开关表中表项为 0 的子程序 gdopen（图 10-2）；如果一个进程读字符设备特殊文件“/dev/mem”且其主设备号是 3，内核调用字符设备开关表中表项为 3 的子程序 mmread。子程序 nulldev 是一“空的”子程序，当不需要一个特殊的驱动程序功能时使用它。许多外围设备可以与一个主设备号相联系；次设备号用来把它们彼此区分开。设备特殊文件并不必在每次系统初启时都产生一次，只当系统配置改变时，如将某些设备添加到系统装备中去时才需改变。

块设备开关表			
表项	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

字符设备开关表					
表项	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodey

图 10-2　块和字符设备开关表实例

10.1.2　系统调用与驱动程序接口

本节叙述内核与设备驱动程序间的接口。对于那些使用文件描述符的系统调用，内核从用户文件描述符的指针找到内核文件表以及索引节点，并检查文件类型，根据需要存取块设

备或字符设备开关表。它从索引节点中抽取主设备号和次设备号，使用主设备号作为索引值进入适当的开关表，根据用户所发的系统调用来调用驱动程序中的函数。设备文件与正规文件的系统调用间的一个重要区别是：当内核执行驱动程序时，特殊文件的索引节点是不上锁的。这是因为驱动程序频繁地睡眠，等待着硬连接或数据的到来，因此内核不能确定一个进程要睡眠多长时间。如果对索引节点上锁，则其他存取此索引节点的进程（如通过系统调用stat）会无限期地睡眠下去，因为另一进程正在驱动程序中睡眠。

设备驱动程序把系统调用的参数解释为对该设备的适当的参数。一个驱动程序维护着描述它所控制的每一设备单元状态的数据结构，根据该状态以及要做的动作（如输入或输出数据）执行驱动程序中的子程序和中断处理程序。现在，我们将详细的叙述每一个接口。

1．系统调用 open

为了以系统调用 open 打开一个设备，内核使用与它打开正规文件同样的过程，即：分配一个内存索引节点，增加其引用数，赋予一个文件表表项和用户文件描述符，内核最后将用户文件描述符返回给调用的进程。所以打开一个设备看上去就像打开一个正规文件。然而，它在返回到用户态以前，调用设备专用的打开过程（图 10-3）：对块设备，它调用块设

```
算法　open　　/*　专为设备驱动程序　*/
输入：路径名
　　打开方式
输出：文件描述符
{
　　将路径名变换成索引结点，增加索引结点引用数，
　　与正规文件打开一样，分配文件表中表项、用户文件描述符；
　　从索引结点取主、次设备序号；
　　保存上下文（算法 setjmp）以防驱动程序需要执行 longjmp；
　　if （块设备）
　　{
　　　　使用主设备号作为块设备开关表的索引；
　　　　调用相应该索引的驱动程序打开过程：
　　　　　　传递参数为次设各号、打开方式；
　　}
　　else
　　{
　　　　使用主设备号作为字符设备开关表索引；
　　　　调用相应该索引的驱动程序打开过程：
　　　　　　传递参数为次设备号、打开方式；
　　}

　　if （open 在驱动程序中失败）
　　　　减少文件表、索引结点引用数；
}
```

图 10-3　打开设备的算法

备开关表中的打开过程；对字符设备，它调用字符设备开关表中的打开过程。如果一个设备既是一个块设备又是一个字符设备，内核将根据用户所打开的特定的设备文件，调用适当的打开过程。依赖于所用的驱动程序，这两个打开过程甚至可以是相同的。

设备专用的打开过程在调用的进程与被打开的设备之间建立起一个连接，并初始化私有的驱动程序数据结构。举例来说，对一个终端，打开过程可以让进程进入睡眠，直到机器检测到一个（硬件）载波信号，该信号指示有个用户正试图注册到机器。然后它根据适当的终端的预置参数（如终端波特率）来初始化驱动程序数据结构。对于软设备如系统存储器，打开过程可能不需要做初始化工作。

当打开一个设备时，如果由于某种外部的原因进程必须睡眠，可能会发生这样的情况：从睡眠中唤醒该进程的事件可能永远不出现。例如，如果老也没有用户注册到某特定的终端时，打开终端的 getty 进程（见 7.9 节）会睡眠很长时间，直到一个用户试图注册。内核必须能够从睡眠中唤醒该进程，并在收到一个软中断信号时取消 open 调用，即：由于未能打开设备，它必须重置索引节点、文件表表项、用户文件描述符这些它在进入驱动程序以前已分配的资源。因此，内核在进入设备专用的 open 子程序以前，应使用算法 setjmp（见 6.4.4 节）保存进程上下文，如果进程由于软中断信号被从睡眠中唤醒，那么内核使用算法 longjmp（见 6.4.4 节）将进程上下文恢复到它进入驱动程序之前的状态，并且释放它为了打开设备所分配的全部数据结构。类似地，驱动程序能捕俘这一软中断信号，并当需要时将私用的数据结构清除掉。当驱动程序遇到错误情况时，比如当一用户试图存取一个未被配置的设备时，内核也要再调整文件系统的数据结构。在这些情况下，系统调用 open 失败。

进程可以规定各种各样的选择项以限定对设备的打开。最普通的选择项是“不延迟”，它的意义是：在打开过程中如果设备没有准备好，进程将不进入睡眠，系统调用立即返回，而用户进程不知道是否建立了硬连接。下面我们将看到，以“不延迟”打开一个设备也会影响系统调用 read 的语义。

如果一个设备被打开多次，如第 5 章所述，则内核巧妙地处理用户文件描述符、索引节点以及文件表表项，为每一个系统调用 open 请求设备专用的打开过程。这样，设备驱动程序能对一个设备被打开了多少次进行计数，并且当此计数值不恰当时，让 open 调用失败。举例来说，允许多个进程打开一个终端进行写操作以便用户间能交换消息是有意义的。但是，允许多个进程同时打开一个打印机进行写操作就讲不通了，因为这会造成它们彼此的数据被改写。上述区别与其说是实际的区别倒不如说是实现上的区别：允许同时对终端写促进了用户间的通信；阻止同时对打印机写增加获得可读的打印输出的机会[⊖]。

2．系统调用 close

一个进程通过系统调用 close 断开它与一打开设备的连接。然而，仅当该设备的最后一个关闭操作到来时，也就是说，仅当没有其他进程仍然打开着该设备时，内核才调用那个设备专用的关闭过程。这是因为关闭过程要终止与硬件的连接，很明显，这必须等到没有进程再要存取该设备的时候。由于在每一次系统调用 open 时内核调用设备的打开过程，而仅调用设备的关闭过程一次，因此该设备驱动程序永远不能确切地知道有多少进程仍在使用那个

⊖ 实际上，通常打印机是由特殊的假脱机进程控制的，而且建立只有假脱机进程才能存取打印机的许可权。然而这样的比喻仍是合适的。

设备。如果不仔细地进行编程，驱动程序会很容易使它们自己陷入混乱状态。例如如果它们在关闭过程中睡眠，而在系统调用 close 完成前另一个进程打开这个设备，如果打开和关闭的组合导致一个无法辨认的状态，则这个设备会变成无用的设备。

关闭一个设备蹬算法类似于关闭一个正规文件的算法（图 10-4），然而，在内核释放索引结点以前，它进行设备文件专用的操作：

（1）它搜索文件表以确定没有其他的进程仍然打开着该设备。根据文件表中的引用数来指示这是一个设备的最后一次关闭操作是不够的，因为几个进程可以通过不同的文件表表项存取该设备；同样，依靠索引节点中引用数也是不够的，因为几个设备文件可以表示同一设备。比如，下面的 ls -l 命令的结果表明两个字符设备文件（每一行中的第一个字母“c”）指的是一个设备，因为它们的主设备号和次设备号是相等的。每一个文件的联结数为 1 意味着有两个索引节点：

```
crw--w--w-    1 root   vis   9, 1 Aug 6 1984   /dev/tty01
crw--w--w-    1 root unix   9, 1 May 3 15:02  /dev/tty01
```

如果多个进程独立地打开这两个文件，它们存取不同的索引节点，然而是同一设备。

```
算法 close    /* 设备专用 */
输入：文件描述符
输出：无
{
    执行正规的算法 close（见第 5 章）；
    if （文件表引用数非 0）
        goto finish;
    if （存在另一个打开文件其主、次设备号与欲关闭文件的相同）
        goto finish;      /* 毕竟不是最后的关闭 */
    if （字符设备）
    {
        以主设备序号作为字符设备开关表的索引；
        调用驱动程序关闭子程序；参数为次设备号；
    }
    if （块设备）
    {
        if （设备已安装）
            goto finish;
        将高速缓冲中该设备的数据块写回设备；
        用主设备号作为块设备开关表的索引；
        调用驱动程序关闭子程序；参数为次设备号；
        使高速缓冲中该设备的数据块无效；
    }
finish:
    释放索引节点；
}
```

图 10-4　关闭设备的算法

(2) 对于一个字符设备，内核调用设备的 close 过程并返回用户态。对于一个块设备，内核搜索安装表以确认该设备不包含一个已安装的文件系统。如果该块设备上含有一个已安装的文件系统，内核不能调用设备的关闭过程，因为这不是该设备的最后一次关闭操作。甚至如果该设备虽已不包含一个已安装的文件系统，但高速缓冲中仍然可能包含有先前安装的文件系统遗留下来的数据块，而且因为它们是以“延迟写”标记的，还从未写回设备。因此，内核在调用设备关闭过程以前，先要搜索高速缓冲中这样的数据块，把它们写回该设备。在关闭设备之后，内核再一次扫描高速缓冲，使所有装有现在已关闭的设备数据块的缓冲无效，因而允许含有有用数据的缓冲在高速缓冲中停留更长些。

(3) 内核释放设备文件的索引节点。

总而言之，设备关闭过程断开与设备的连接，并重新初始化驱动程序数据结构和设备硬件，使内核以后能重新打开该设备。

3. 系统调用 read 和 write

对一个设备读和写操作的算法类似于对正规文件的算法。如果进程要读或写一个字符设备，内核调用设备驱动程序的读或写过程。虽然也有内核直接在用户地址空间和设备之间传送数据的重要情形，但设备驱动程序通常在其内部缓冲数据。比如，终端驱动程序使用 clist (见 10.3.1 节)来缓冲数据，在这种情况下，在写操作期间，设备驱动程序分配一个“缓冲”，先从用户空间拷贝数据到“缓冲”，再从“缓冲”输出数据到设备。驱动程序的写过程还要调节输出的数据量(称为流量控制(flow control))：如果进程产生数据的速度比设备能输出它们快，那么写过程应使进程进入睡眠，直到设备可以接受更多的数据。对于读操作，设备驱动程序先从设备接收数据到一个缓冲，再将数据从缓冲拷贝到系统调用中规定的用户地址去。

驱动程序与设备通信的确切的方法依赖于硬件。某些机器提供映射到存储的 I/O (memory mapped I/O)，它的含义是在内核地址空间中的某些地址不是物理存储器中的存储单元，而是控制特定的设备的特殊寄存器。驱动程序通过往所规定的寄存器写入控制参数来控制该设备。举例来说，VAX-11 计算机的 I/O 控制器含有用来记录设备状态（控制和状态寄存器）和数据发送（数据缓冲寄存器）的特殊寄存器，这些寄存器被配置在物理存储器的特殊地址上。具体来讲，VAX DZ11 终端控制器控制 8 个用于与终端通信的异步线路（有关 VAX 体系结构的更详细的材料见[Levy 80]）。假设一个特定的 DZ11 的控制和状态寄存器是配置在地址 160120，发送数据缓冲寄存器在地址 160126，接收数据缓冲寄存器在地址 160122（图 10-5）。为了写一个字符到终端“/dev/tty09”，终端驱动程序把数字 1（1 = 9 mod8)写入控制与状态寄存器的一个规定的位中去，然后把该字符写入发送数据缓冲寄存器，写入发送数据缓冲寄存器的操作就发送了该数据。当 DZ11 控制器准备好接受更多的数据时，它就将控制与状态寄存器中的“完成”位置位。驱动程序还能有选择地置位控制和状态寄存器中的“发送中断允许”位，这使 DZ11 在准备好接受更多的数据时中断系统。从 DZ11 读数据是类似的。

其他机器使用可编程 I/O，其含义是机器包含用于控制设备的指令。驱动程序靠执行适当的指令控制设备。例如，IBM370 计算机有一个用来启动设备做一次 I/O 操作的指令——start I/O。驱动程序与外围设备进行通信的方法对用户是透明的。

由于设备驱动程序与它下面的硬件之间的接口是依赖于机器的，在这一级不存在标准的接口。无论对映射到存储的 I/O 或可编程 I/O，一个驱动程序都能对一个设备发出控制序列，在设备和存储器间建立直接存储器存取（DMA)。系统允许在设备和存储器间成块地以

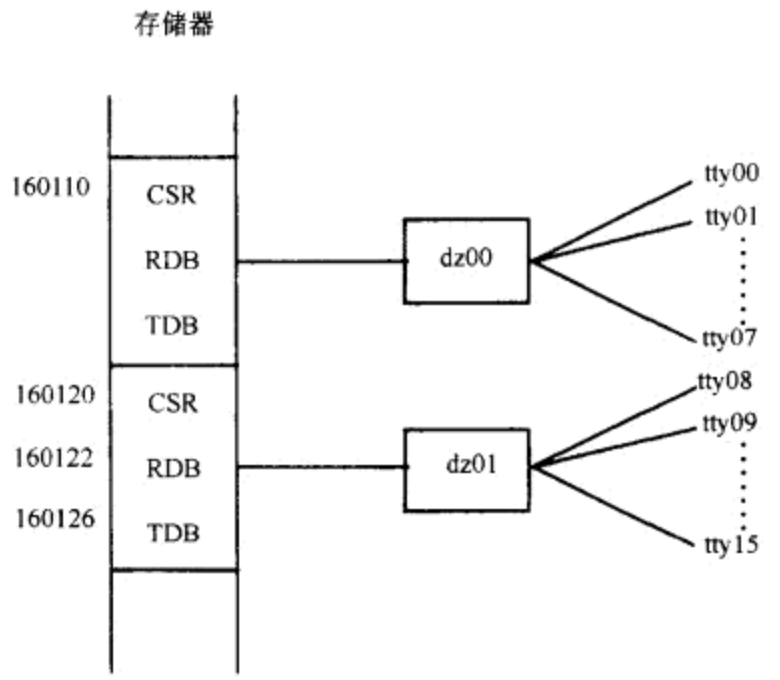


图 10-5　VAX 机 DZ11 控制器的映射到存储的 I/O

DMA 方式传送数据，它和 CPU 的操作并行，当这样的传送完成时设备会中断系统。驱动程序建立虚拟存储器映射，使 DMA 使用存储器中正确的位置。

高速设备有时可以在设备和用户地址空间之间直接地传送数据，无需内核缓冲的介入。这一点造成了高速传送速率，因为减少了一次往内核拷贝的操作，而且每次传送操作中传送的数据量也不再受内核缓冲大小的限制。使用这种“原始”的 I/O 传送的驱动程序通常从它们的字符读和写过程调用块设备的策略接口（见下节），如果它们有一个对应的块设备驱动程序的话。

4. 策略接口

内核使用策略接口（strategy interface）在高速缓冲和设备之间传送数据。虽然，如上面提到过的那样，字符设备的读和写过程有时使用它们的（对应的块设备的）策略过程来在设备与用户地址空间之间直接传送数据。策略过程可以把对一个设备的 I/O 作业送入一个工作表队列，或者做更复杂的调度 I/O 作业的处理工作。驱动程序可以建立与一个物理地址之间的数据传送，或者如果需要的话，建立与很多物理地址的数据传送。内核把一个缓冲头部地址传递给驱动程序策略过程，该头部包含一个为从设备传送数据或传送数据到设备的（页面）地址及大小的表，这也是第 9 章中讨论的对换操作的工作原理。就高速缓冲而言，内核从一个数据地址传送数据，而当对换操作时，内核从许多数据地址（页面）传送数据。如果数据是从用户地址空间拷贝或拷贝到用户地址空间去的，则驱动程序必须将进程（或者至少有关的页面）锁在存储器中，直到 I/O 传送完成。

举例来说，在安装一个文件系统以后，内核用设备号和索引节点号识别该文件系统中每一个文件，设备号是由主设备号和次设备号编码得到的。正如第 3 章所描述的，当内核从一个文件存取一块时，它把设备号和块号拷贝到缓冲头部中去。当高速缓冲算法（如 bread 或 bwrite）存取该磁盘时，它们调用由主设备号所指定的策略过程，策略过程则使用在缓冲头部的次设备号和块号来确定到设备上的什么地方去寻找数据，并用缓冲地址来确定数据应当传送到什么地方去。类似地，如果一个进程直接地存取一个块设备（即该进程打开那个块设

备并读或写它），它使用高速缓冲算法，而接口按照刚刚描述的方法工作。

5. 系统调用 ioctl

系统调用 ioctl 是对 UNIX 系统较早的版本中提供的专门用于终端的系统调用：stty（预置终端参数）和 gtty（取终端预置参数）的一种普遍化，它为所有的设备专用命令提供一个一般的、万能的入口点。它允许一个进程去预置与一个设备相联系的硬件选择项和与一个驱动程序相联系的软件选择项。由系统调用 ioctl 规定的专门的动作对每个设备是不同的，并由设备驱动程序定义。使用系统调用 ioctl 的程序必须知道它们正在与什么类型的文件打交道，因为这些都是设备专用的。这对于系统不应在不同文件类型间加以区别的一般规则是一个例外。10.3.3 节将对终端如何使用系统调用 ioctl 提供更多的细节。

该系统调用的语法是：

```
ioctl (fd, command, arg);
```

其中 fd 是先前的一个系统调用返回的文件描述符；command 是使驱动程序完成一个特定动作的请求命令，而 arg 是对该命令的一个参数（可能是指向一个结构的指针）。命令是驱动程序专用的，因此，每个驱动程序根据其内部的说明解释命令，而数据结构 arg 的形式依赖于该命令。驱动程序可以根据预先定义的形式从用户空间读入数据结构 arg，或者它们可以用 arg 将设备的预置参数写回到用户空间去。例如，ioctl 接口允许用户置终端波特率；允许用户在磁带驱动器上反绕磁带；最后它允许一些网络操作如规定虚电路数目和网络地址。

6. 与文件系统有关的其他系统调用

文件系统调用如 stat 和 chmod 对设备文件也和对正视文件一样地动作，它们只处理索引节点而不访问驱动程序。甚至系统调用 lseek 也为设备文件工作。举例来说，如果一进程要在磁带上移动一特定字节的偏移量，内核修改文件表中的偏移量，但是不做驱动程序专门的操作。以后，当该进程进行读或写时，和对正规文件所做的一样，内核将文件表中的偏移量移到 u 区，而由该设备物理地找到由 u 区指示的偏移量的正确位置。10.3 节将以一个例子叙述这一情形。

10.1.3　中断处理程序

如前面所解释的（6.4.1 节），一个中断的出现将引起内核执行一个中断处理程序，而执行哪一个中断处理程序是根据发出中断的设备和中断向量表中的偏移量的相互关系决定的。内核调用该设备专用的中断处理程序，将设备号或其他参数传递给它，以便识别引起中断的特定的设备单元。举例来说，图 10-6 给出了中断向量表中为处理终端中断（“ttyintr”）的两个入口点，每个入口点处理 8 个终端的中断。如果设备 tty09 中断了系统，系统将调用与发出中断设备的硬件位置相联系的中断处理程序。由于一个中断向量入口点可以与许多物理设备相联系，驱动程序必须能够判定是哪一个设备引起的中断。在本图中，“ttyintr”的两个中断向量入口点被标记为 0 和 1，这意味着当调用该中断处理程序时，系统要以某种方式区分这两个向量入口点，例如使用数字 0 或 1 作为调用参数。这样，中断处理程序会使用这个数字以及由中断机构传递的其他参数来弄清是设备 tty09 中断了系统，而不是其他设备，例如 tty12，中断了系统。这个例子是实际系统的一种简化，在实际系统中会有几级控制器和它们的中断处理程序进入本图，但本图说明了一般的原则。

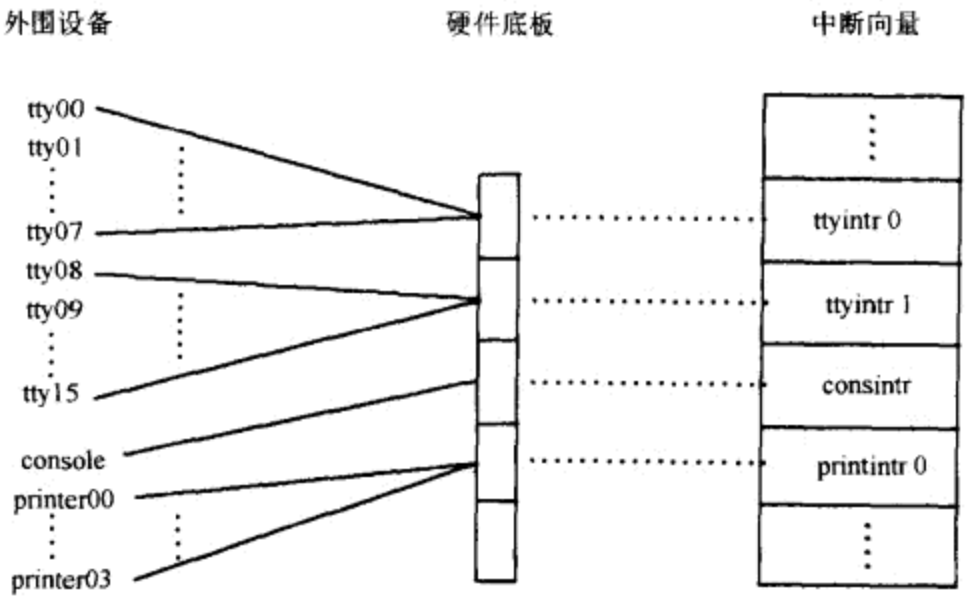


图 10-6　设备中断

总之，中断处理程序使用设备号识别一个硬件单元，设备文件中的次设备号为内核识别一个设备，而设备驱动程序使次设备号与硬件单元发生联系。

10.2　磁盘驱动程序

回顾历史，UNIX 的磁盘曾被配置为多个磁盘段，每段包含一个文件系统，为的是使“磁盘组被分割成更多的易于管理的块”（见[System V 84b]）。举例来说，如果一个磁盘含有四个文件系统，系统管理员可以留下一个未安装的文件系统，将另一个安装成“只读”的，最后两个安装为“可读可写”的。虽然所有的文件系统同时存在于一个物理设备上，但用户既不能用第 4 章和第 5 章中所叙述的存取方法来存取未安装的文件系统中的文件，也没有任何用户可以往“只读”的文件系统中写文件。此外，由于每一磁盘段（因此每一文件系统）在磁盘上覆盖邻接的磁道和柱面，因此拷贝整个文件系统变得比如果它们分散在整个磁盘卷上要容易得多。

磁盘驱动程序把由一个逻辑设备号和块号组成的文件系统地址翻译成磁盘上特定的扇区号。驱动程序使用以下两种方法之一来得到该地址：或者策略程序使用缓冲池中的一个缓冲，而该缓冲头部含有设备号和块号；或者将逻辑设备号（次设备号）作为一个参数传递给读过程或写过程，它们再把保留在 u 区的字节偏移量变换为适当的块地址。磁盘驱动程序用设备号来识别物理盘以及所使用的特定的磁盘段，维护内部表格以便找到一个磁盘段开始的扇区。最后，它把文件系统的块号加到起始扇区号上去，以便识别 I/O 传送所用的扇区号。

历史上磁盘段的大小和长度曾是按磁盘类型而固定不变的。例如，DEC RP07 磁盘被划分成如图 10-7 所示的若干段。假设设备特殊文件“/dev/dsk0”，“/dev/dsk1”，“/dev/dsk2”和“/dev/dsk3”相应于一个 RP07 磁盘的磁盘段 0 到磁盘段 3，并相应于次设备号 0 到 3。假定一个逻辑文件系统块的大小与一个磁盘块的大小是一样的，如果内核试图存取“/dev/dsk3”文件系统中的第 940 块，磁盘驱动程序将这一请求变换为存取磁盘上的第 336940 块（因磁盘段 3 从第 336000 块开始；336000 + 940 = 336940）。

图 10-7 中磁盘段的大小是不等的，系统管理员把各文件系统配置到适当的段中去：大

的文件系统进入大的段，等等。各磁盘段在磁盘上可以重叠。比如在 RP07 磁盘上的第 0 段和第 1 段是邻接着的，但它们加在一起覆盖了整个磁盘，即从第 0 块到第 1008000 块。图中的第 7 段也跨越了整个磁盘。只要包含在磁盘段中的文件系统被配置成互不重叠的，磁盘段的重叠并不要紧。而且让一个磁盘段包含整个磁盘是有利的，因为这样整个磁盘卷能被快速地拷贝。

段号	起始块号	块数
	块大小＝512 字节	
0	0	64000
1	64000	944000
2	168000	840000
3	336000	672000
4	504000	504000
5	672000	336000
6	840000	168000
7	0	1008000

图 10-7　RP07 磁盘的磁盘段

固定段的使用限制了磁盘配置的灵活性。有关磁盘段的划分也不应当以硬编码形式放入磁盘驱动程序，而应当放入磁盘上的一个可配置的卷内容表中去。然而，在所有的磁盘上找到一个统一的位置存放卷内容表，又要保持与先前版本的系统兼容是困难的。当前系统 V 的实现是让一个磁盘上第一个文件系统的引导块占据该卷的第一个扇区，虽然那是卷内容表逻辑上最合理的位置。然而，磁盘驱动程序可以将有关某特定磁盘卷内容表的存放地点的信息硬编码到它里面，以便允许磁盘段大小的变化。

由于磁盘的交通量高是 UNIX 系统的特征，磁盘驱动程序必须使数据吞吐量达到最大，以获得最好的系统性能。多数现代磁盘控制器能管理磁盘任务调度、定位磁盘臂和在磁盘及 CPU 之间传送数据，若磁盘控制器不做这些事情，磁盘驱动程序必须完成这些任务。

实用程序可以使用原始的或块接口来直接地存取磁盘数据，绕过第 4 章和第 5 章中所讨论的正规文件系统存取方法。直接与磁盘打交道的两个重要的程序是 mkfs 和 fsck，mkfs 为形成一个 UNIX 文件系统格式化一个磁盘段，在新文件系统上产生一个超级块、索引节点表、磁盘块自由链表和一个根目录。fsck，如第 5 章中所述，检查一个已存在的文件系统的一致性，并纠正错误。

下面让我们考查一下图 10-8 中的程序和两个设备特殊文件“/dev/dsk15”和“/dev/rdsk15”，假设 ls 命令打印出以下信息：

```
ls -1/dev/dsk15   /dev/rdsk15
br----------   2 root   root   0, 21 Feb 12 15:40   /dev/dsk15
crw-rw----   2 root   root   7, 21 Mar   7 09:29   /dev/rdsk15
```

这表明，文件“/dev/dsk15”是一个由“根”拥有的块设备，只有“根”能读它，它的主设备号是 0，次设备号是 21。文件“/dev/rdsk15”是一个由“根”拥有的字符设备，但允许

所有者和同组用户（这里都是“根”）有读和写许可权，它的主设备号是7，次设备号是21。一个打开这些文件的进程分别通过块设备开关表和字符设备开关表来存取这个设备，而次设备号通知驱动程序要存取哪一个磁盘段——例如，物理盘2，第1段。由于每一个文件的次设备号是相同的，假定这是一个设备[⊖]，两者都涉及相同的磁盘段。这样，一个进程运行图10-8的程序，用系统调用open两次打开同一驱动程序（通过不同的接口），用系统调用lseek找到该设备上的偏移量8192字节处，并用系统调用read从该位置读数据。假设没有其他的文件系统活动，两个read的结果应当是相同的。

```
#include "fcntl.h"
main()
{
        char buf1[4096],buf2[4096];
        int fd1,fd2,i;

        if(((fd1=open("/dev/dsk5",O_RDONLY))==-1)||
                ((fd2=open("/dev/rdsk5";O_RDONLY))==-1))
        {
             printf("failure on open\n");
             exit();
        }
        lseek(fd1,8192L,0);
        lseek(fd2,8192L,0);
        if((read(fd1,buf1,sizeof(buf1))==-1)||(read(fd2,buf2,sizeof(buf2)==-1))
        {
             printf("failure on read\n");
             exit();
        }

        for(i=0;i<sizeof(buf1);i++)
             if(buf1[i]!=buf2[i])
             {
                  printf("different at offset %d\n",i);
                  exit();
             }
        printf("reads match\n");
}
```

图10-8　使用块和原始接口读磁盘数据

直接地读和写磁盘的程序是危险的，因为它们可能读或写敏感的数据，危害系统的安全性。因此系统管理员必须通过对磁盘文件设置适当的许可权来保护块和原始的接口。例如，磁盘文件“/dev/dsk15”和“/dev/rdsk15”的所有者应当是“根”，而它们的许可权应是只允许“根”去读该文件，不允许任何其他的用户读或写。

⊖ 除了检查系统配置表和驱动程序的编码以外，没有什么方法能证实一个字符设备驱动程序与块设备驱动程序指的是同一设备。

直接地读和写磁盘的程序也会破坏文件系统数据的一致性。在第 3、4、5 章中解释的文件系统的算法并配合以磁盘的 I/O 操作，为的是保持对磁盘数据结构有一致的看法，包括磁盘块自由链表和从索引节点到直接和间接的数据块的指针。直接存取磁盘的进程绕过这些算法，即使它们是经过仔细地编程的，当其他文件系统的活动正在进行时，如果它们运行，也仍然存在着一致性问题。正因为如此，fsck 不应当在一个活动着的文件系统上运行。

两种磁盘接口之间的区别是它们是否处理高速缓冲。当访问块设备接口时，内核使用与处理正规文件同样的算法，所不同的是在把逻辑字节偏移量变换成一个逻辑块偏移量之后，它把逻辑块偏移量当作文件系统中的一个物理块号。然后，它通过高速缓冲存取数据，并最终地调用驱动程序的策略接口。然而，当通过原始接口存取磁盘时，内核并不将字节偏移量变换成文件系统的块号，而是通过 u 区立即将该偏移量传递给驱动程序，驱动程序中读和写子程序把字节偏移量变换成一个块偏移量，并绕过高速缓冲直接地把数据拷贝到用户地址空间去。

这样，如果一个进程向一个块设备写，另一个进程在同一地址从一个原始设备读，第二个进程读的数据可能不是第一个进程写入的数据，因为第一个进程写入的数据可能仍在高速缓冲中，而不在磁盘上。然而，如果第二个进程也是从块设备读，只要新数据在高速缓冲中存在，它就会自动地拿到该数据。

使用原始接口也可能带来些奇怪的行为。例如，当一个进程以小于块大小的单位对一个原始设备进行读或写时，其结果是与驱动程序有关的。比如，对一个磁带多次写，每次写一个字节，结果每个字节可能出现在不同的磁带块上。

假定不考虑数据被高速缓存以备后用所带来的好处的话，那么使用原始接口的优点是提高速度。以块设备进行存取的进程所传送的数据块的大小是由文件系统的逻辑块大小限制的。举例来说，如果一个文件系统的逻辑块大小是 1K 字节，则每次 I/O 操作最多传送 1K 字节。然而，以原始设备存取磁盘的进程在一次磁盘操作过程中可以传送许多磁盘块，仅受磁盘控制器能力的限制。从功能上看，进程得到的是相同的结果，但是原始接口可能快的多。例如在图 10-8 的程序中，对于一个块大小是 1K 字节的文件系统，如果一个进程要使用块设备接口读 4096 字节，在从系统调用 read 返回之前，内核要内部地循环四次，每次循环都要访问磁盘。而如果它使用原始接口读，驱动程序可能以一次磁盘操作就读完。更有甚者，使用块设备接口必然伴有额外的在用户地址空间和内核缓冲之间拷贝数据，而这一点在原始接口中是避免了的。

10.3　终端驱动程序

终端驱动程序具有与其他驱动程序相同的功能，即控制从终端来和到终端去的数据。然而，由于终端是用户与系统的接口，所以终端又是特殊的。为使用户交互式的使用 UNIX 系统，终端驱动程序包含一个与行规则（line discipline）模块的内部的接口，行规则程序的作用是对输入和输出的数据进行解释。在标准方式（canonical）下，行规则程序把在键盘上敲入的原始数据序列变换成一种标准的形式（即用户真正的意思是什么），然后再把数据送给一个接收进程；同样地，行规则程序把一个进程输出的原始输出数据变换成用户所期望的形式。在原始方式（raw）下，行规则程序仅在进程和终端之间传送数据，不做上述变换。

比如，程序员是一些出了名的快的但易敲错的打字员，终端提供了一个“擦除键”（或者可能是这样来指定的一个键），以便用户能够从逻辑上部分地擦除敲入的序列，然后敲入

校正的序列。终端把整个序列包括擦除字符[⊖]送给主机。在标准方式下，行规则程序把数据按行（直到一个回车[⊜]的字符序列）缓冲起来，并且在把修改过的序列送给正在读的进程之前，内部地对擦除字符进行处理。

一个行规则程序的功能是：

- 通过分析将输入字符串变成行。
- 处理擦除键。
- 处理“抹行”字符，它使在当前行上到目前为止键入的所有字符无效。
- 把接收的字符回显（写）到终端。
- 扩展输出，如把制表符变成一系列空格字符。
- 为终端挂起（hangup）、断线或响应用户键入的 delete 键，向进程产生软中断信号。
- 允许不对特殊字符如擦除、抹行或回车进行解释的原始方式。

支持原始方式意味着可以使用一个异步的终端，这样进程可以在字符被键入时就读它们，而不是等待用户键入一个回车或“enter”键时才读。

Ritchie 特别提到在 70 年代初系统开发期间所使用的最早的终端行规则程序是在 shell 和编辑程序中的，而不是在内核中的(见[Ritchie 84]第 1580 页)。然而，由于许多程序都需要它们的功能，所以，它们的合理位置应当是放在内核中。虽然行规则程序完成的功能从逻辑上来说是在终端驱动程序与内核的其余部分之间，但内核并不直接调用行规则程序，而是通过终端驱动程序来调用。图 10-9 给出了通过终端驱动程序和行规则程序的逻辑数据流，以及

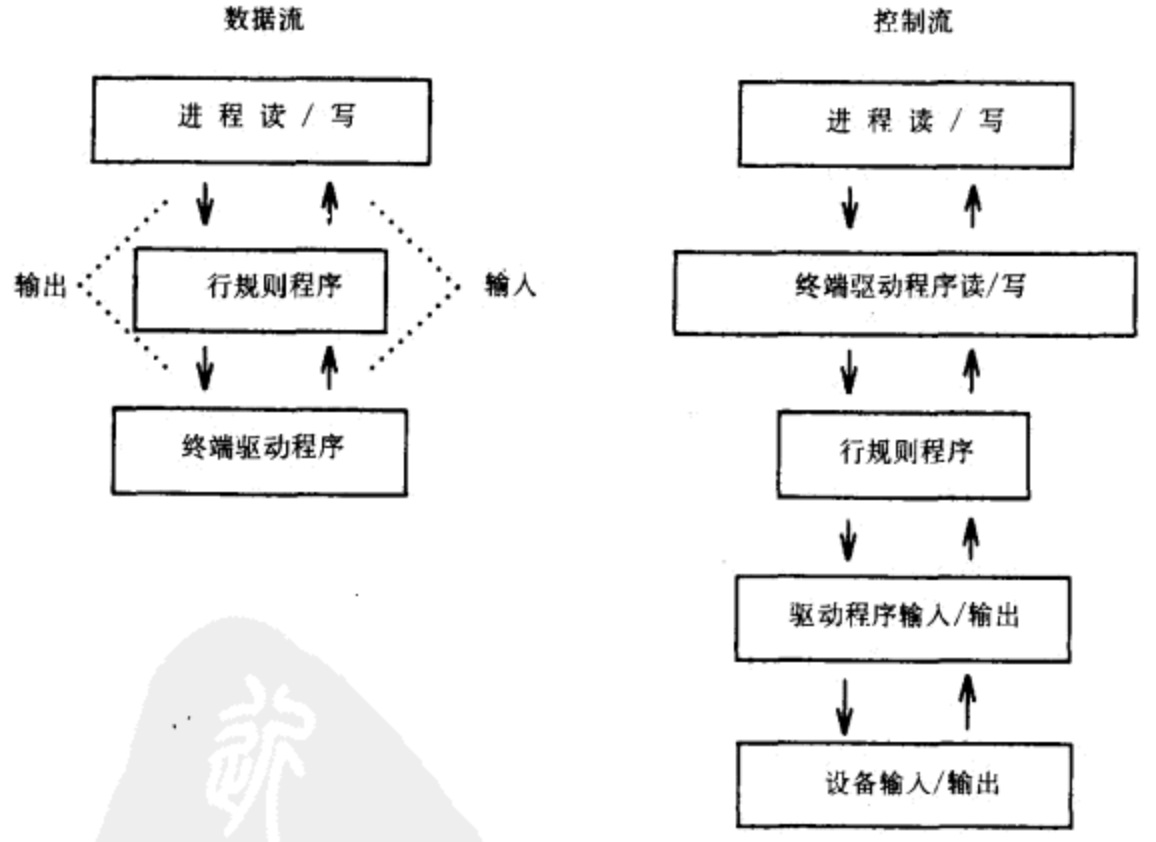


图 10-9　有关行规则程序的调用顺序及数据流

⊖ 本节假设的是使用笨终端，它将用户敲入的全部字符不加处理的进行发送。

⊜ 本章中使用“回车”这个通用术语表示“回车”和“换行”字符。

通过终端驱动程序的相应的控制流。用户可以通过系统调用 ioctl 来规定应当使用哪一个行规则程序。然而要想实现一种设计，在这种设计中一个设备同时使用几个行规则程序，而且为了依次地处理数据，每个行规则模块相继的调用下一个模块是困难的。

10.3.1　字符表 clist

行规则程序在字符表上操纵数据。字符表，或称为 clist，是字符缓冲 cblock 的变长度的链表，并携带有表中的字符计数。一个 cblock 含有一个指向该链表中下一个 cblock 的指针，一个用来存放数据的小的字符数组，一组用来指示 cblock 中有效数据位置的偏移量（图 10-10）。起始偏移量指示数组中第一个有效数据的位置，终止偏移量指示第一个无效数据的位置。

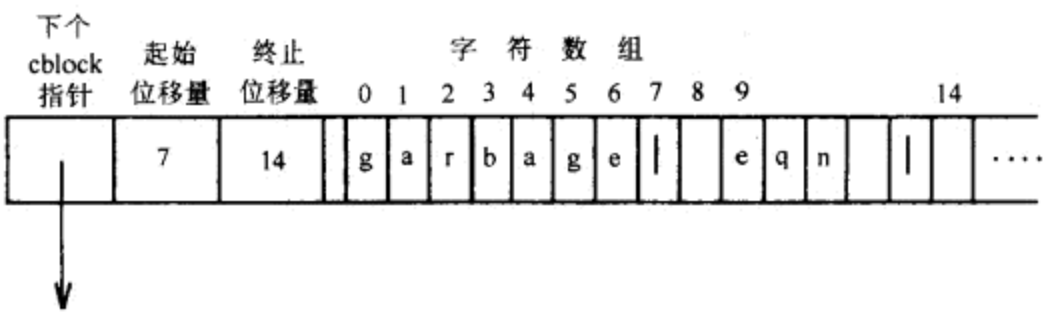


图 10-10　cblock 结构

内核维护一个空闲 cblock 的链表，并规定在 clist 和 cblock 上有六种操作：

(1) 它有一个从空闲表中分配一个 cblock 给驱动程序的操作。

(2) 它有一个把一个 cblock 归还给空闲表的操作。

(3) 内核可以从一个 clist 中提取一个字符：它从 clist 表中第一个 cblock 中移出一个字符，并调整 clist 的字符计数和到该 cblock 的索引值，使以后的操作不再提取同一字符。如果提取操作移出的是某 cblock 中最后一个字符，则内核把腾空的 cblock 放回空闲表，并调整 clist 的指针。如果提取操作完成后一个 clist 中已不包含字符，则内核返回空白字符。

(4) 内核能把一个字符放入 clist 表的末尾，它找到该 clist 表的最后一个 cblock，将该字符放进去，并调整偏移量。如果该 cblock 已满，则内核分配一个新的 cblock，将之链入该 clist 的末尾，并把字符放入新 cblock 中去。

(5) 内核可以从一个 clist 表的开头移去一组字符，每次一个 cblock。这个操作等价于每一次移去一个 cblock 的所有字符。

(6) 内核可以把一个 cblock 的字符放到一个 clist 表的末尾。

clist 表提供了一种简单的缓冲机构，这种机构对于少量数据的传送——典型地，如终端这样的慢速设备——是很有用的。它们允许每次对一个字符的数据或一组 cblock 的数据进行操纵。举例来说，图 10-11 给出了从一个 clist 移出字符的情形；内核从该 clist 的第一个 cblock中每次移去一个字符（图 10-11a～c），直到该 cblock 中不再有字符时为止（图 10-11d），然后它调整该 clist 的指针，使它指向下一个 cblock，而这个 cblock 就变成该链表的第一个 cblock 了。类似地，图 10-12 给出了内核是怎样把字符放入 clist 表的，这里我们假设一个 cblock 中最多存放 8 个字符。内核将一个新的 cblock 链接到链表的末尾的情形示于图

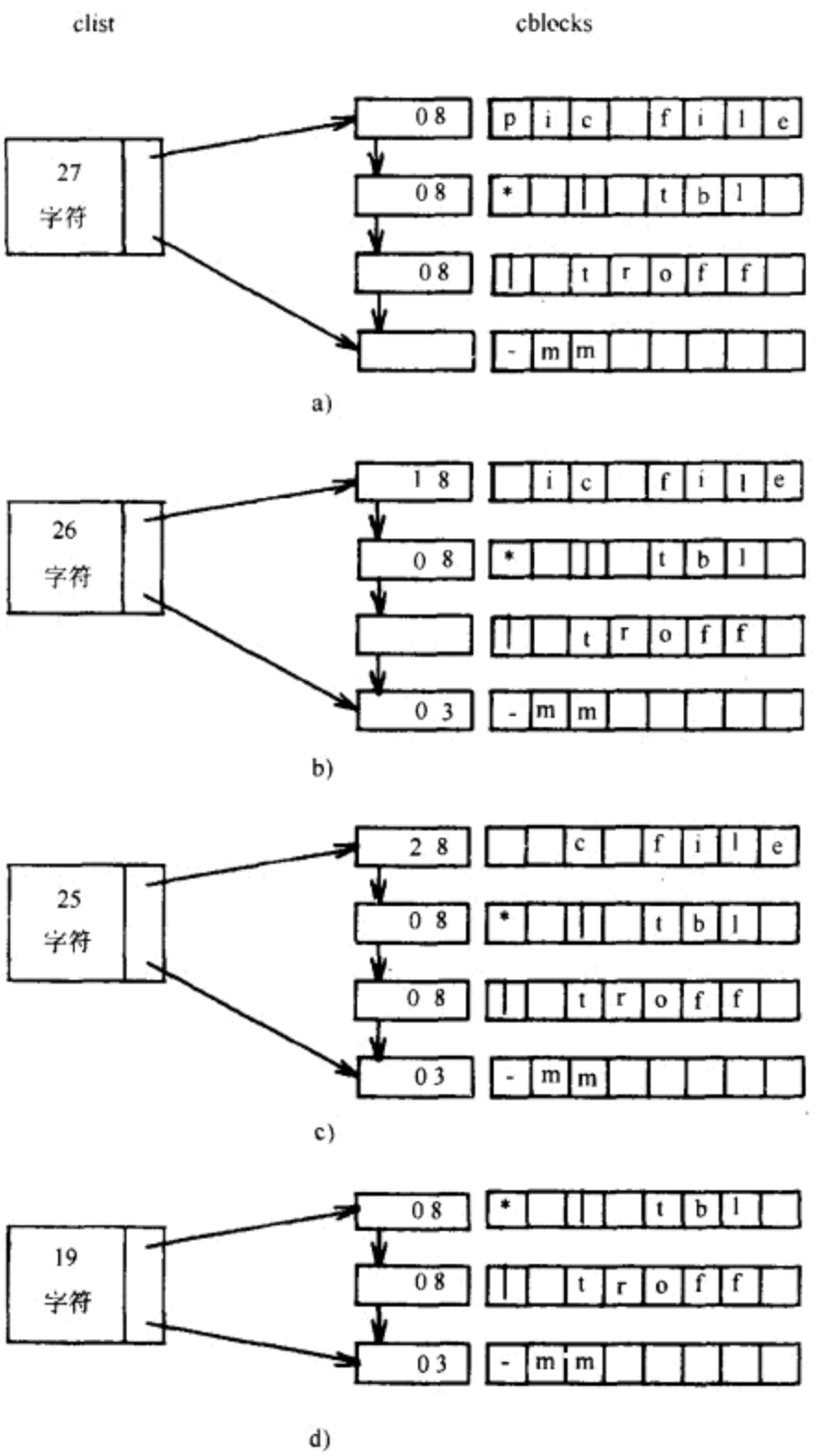


图 10-11 从 clist 中移去字符

10-12d。

10.3.2 标准方式下的终端驱动程序

终满驱动程序的数据结构中含有三个与之相关联的 clist；一个 clist 用来存放输出到终端去的数据；一个 clist 用来存放“原始”输入数据，这是当用户敲入字符时，终端中断处理程序将之填入的数据；一个 clist 用来存放“已加工”的输入数据，常称它为标准 clist，这是在行规则程序将原始 clist 中的特殊字符如擦除（erase）、抹行（kill）字符变换后的输入数据。

当一个进程要向一个终端写时（图 10-13），终端驱动程序调用行规则程序，行规则程序进

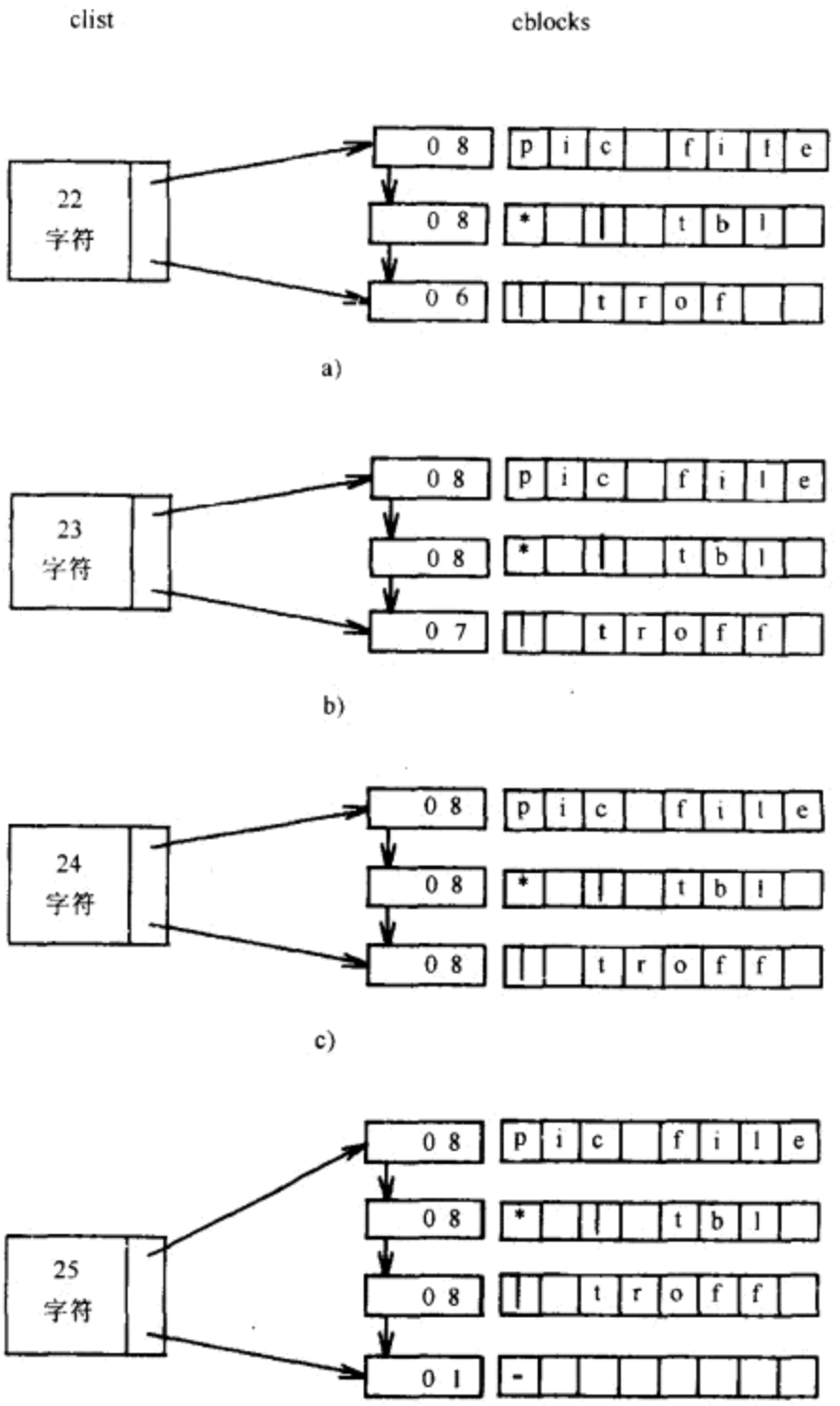


图 10-12　将字符放入 clist

入一个循环——从用户地址空间读欲输出字符，将之放入输出 clist 中去，直到它取尽数据。在此过程中，行规则程序对输出字符加以处理，例如将制表符扩展为一组空格符。如果输出 clist 上的字符数开始大于最高水准时，行规则程序调用驱动程序将输出 clist 中的数据发送到终端，并使写进程进入睡眠。当输出 clist 中的数据降低到低于最低水准时，终端驱动程序唤醒所有的睡眠在等待终端能接受更多数据的事件上的进程。正如以前所说的，当行规则程序将所有输出数据都从用户空间拷贝到输出 clist 时，终止循环，并调用驱动程序将数据发送到终端。

如果多个进程向一个终端写，它们独立地使用上面给出的过程，输出的数据可能会是混乱的，也就是说，各进程所写的数据在终端上可能是交错着的。当一个进程用几个系统调用

write 对终端写时，也会发生这种情况。这是因为在相继的系统调用 write 间，该进程处在用户态，内核可能切换上下文，当原来的进程睡眠时，新被调度的进程可能对该终端写。另外，由于一个写进程可能在系统调用 write 中为等待以前的输出数据从系统中泄出而睡眠，在一个终端上输出的数据也会混乱，内核可能在原来的进程被再次调度之前调度其他也要对该终端进行写的进程。由于这种情况，内核不能保证由一个系统调用 write 输出的数据缓冲中的内容在终端上连续地出现。

```
算法  terminal _ write
{
    while  (尚有数据待从用户空间拷贝)
    {
        if  (tty 结构填满待输出数据)
        {
            启动硬件写操作以便送出输出 clist 中数据;
            sleep(tty 能接受更多数据的事件);
            continue;     /*  回到 while 循环开始  */
        }
        从用户空间拷贝 cblock 大小的数据到输出 clist;
                行规则程序变换制表符等;
    }
    启动硬件写操作以便送出输出 clist 中的数据;
}
```

图 10-13　写数据到终端的算法

让我们来考察图 10-14 的程序。父进程创建了多达 18 个的子进程，每个子进程在数组 output 中形成一定格式的字符串（通过库函数 sprintf），该字符串包含一条消息和在系统调用 fork 时 i 的值，然后进入一个循环，每次循环过程中将该字符串写到它的标准输出文件去。如果标准输出是终端，则终端驱动程序控制数据流输出到终端，由于输出的字符串大于 64 个字符，在系统 V 实现中因其太大不能被放入一个 cblock（64 字节长），因此对每一个系统调用 write，终端驱动程序需要使用多于一个 cblock，而且输出可能变得混乱。例如，下面的两行就是在 AT&T 公司 3B20 计算机上运行该程序时所产生的输出的一部分：

```
this is a sample output string from child 1
this is a sample outthis is a sample output string from child 0
```

在标准方式下从终端读数据是一个更复杂的操作。系统调用 read 规定该进程想要读的字节数，然而行规则程序在接收到一个回车符后就完成该 read，即使尚未达到所规定的字符数。这一点是很实际的，因为要一个进程预先知道用户在键盘上将键入多少个字符是不可能的，而且等待用户键入大量的字符也是无意义的。举例来说，用户对 shell 键入命令行，并且期待 shell 在收到一个回车符时对命令给出响应。无论命令是简单的如“date”或“who”，或者是较复杂的命令序列如：

```
pic file* | tbl | eqn | troff-mm-Taps | apsend
```

都应是一样的。终端驱动程序和行规则程序对 shell 的语法一无所知，而且这是理所当然的，因为从终端读数据的其他程序（如编辑程序）有不同的命令语法。因此，行规则程序在收到一个回车符时就完成系统调用 read。

```
char form[ ] = "this is a sample output string from child";
main()
{
    char output[128];
    int i;

    for(i = 0;  i < 18;  i++)
    {
        switch(fork())
        {
        case -1:        /*  错误---已达进程数量大值 */
            exit();

        default:        /*  父进程 */
            break;

        case 0:             /* 子进程 */
            /* 在变量 output 中形成输出字符串 */
            sprintf(output, "%s%d\n%s%d\n", form, i, form, i);
            for(;;)
                write(1, output, sizeof(output));
        }
    }
}
```

图 10-14　在标准输出上输出数据

图 10-15 给出了读终端的算法。假定终端是在标准方式下，10.3.3 节将讨论原始方式的情形。如果当前在任意一个输入 clist 中没有数据，则读进程睡眠着直到一行数据到来。当数据被输入时，终端中断处理程序把数据放入原始 clist，以便输入给读进程，并且放入输出 clist 以便回显到终端。如果输入字符串中含有一个回车符，则中断处理程序唤醒全部读进程。当一个读进程运行时，驱动程序从原始 clist 中移出字符，对擦除或抹行符进行处理，并把字符放入标准 clist 中去。然后它把字符拷贝到用户地址空间中去，直到接收到回车符或它达到了系统调用 read 中的 count 值时为止，就看这两个值中哪一个数字小一些。然而，一个进程可能会发现它被唤醒的那些数据已不复存在了。在第一个进程被重新调度之前，其他进程可能从终端读，并从原始 clist 中移走了数据。这与当多个进程从一个管道中读数据时发生的情况相类似。

由于存在两个输入 clist，一个输出 clist，因而输入和输出方向的字符处理是不对称的。行规则程序从用户空间输出数据、对它们进行处理、然后将它们放入输出 clist 中去。若为了保

```
算法 terminal_read
{
    if (标准 clist 中无数据)
    {
        while (原始 clist 中无数据)
        {
            if (tty 是以"不延迟"选择打开的)
                return;
            if (tty 是基于定时的原始方式且定时器未激活)
                设置定时器唤醒(callout 表);
            sleep(事件:数据从终端到来);
        }
        /* 原始 clist 上有数据 */
        if (tty 是原始方式)
            将全部数据从原始 clist 拷贝到标准 clist;
        else /* tty 是标准方式 */
        {
            while (原始 clist 中有字符)
            {
                每次从原始 clist 拷贝一个字符到标准 clist:
                        进行擦除、抹行处理;
                if (字符是回车符或文件尾)
                    break; /* 退出循环 */
            }
        }
    }
    while (标准 clist 中有字符且欲读字节数未满足)
        从标准 clist 的 cblock 拷贝到用户地址空间;
}
```

图 10-15 读终端的算法

持对称性,应只设一个输入 clist,然而,这就要求中断处理程序去处理擦除和抹行符,这不仅使中断处理程序变得更加复杂和费时,而且在一段临界时间内会将其他中断封锁住。使用两个输入 clist 意味着中断处理程序能只将输入字符转储到原始 clist 中;在唤醒读进程之后,由读进程适当地承担处理输入数据的开销。然而,中断处理程序应立即把输入字符放入输出 clist,以便用户经历最小的延迟就看到所敲入的字符出现在终端上。

图 10-16 示出了由一个进程产生很多子进程的程序,这些子进程都在读它们的标准输入文件,彼此对终端数据进行竞争。通常终端输入太慢以致不能满足所有的正在做系统调用 read 的进程,因此这些进程将它们的主要时间消耗在 terminal_read 算法中睡眠,等待输入数据。当用户键入一行数据时,终端中断处理程序唤醒所有的读进程,由于它们睡眠在同一优先级,它们都是在同一优先级上符合运行条件的。这样,用户不能预知哪一个进程将运行和读该

行数据，成功的进程会打印在它被创建时的 i 的值。所有其他的进程最终将被调度成运行状态，但是它们很可能会发现在输入 clist 中已没有输入数据了，因而再回去睡眠。对每一个输入行整个过程重复一遍，而且不能保证一个进程能扫到所有的输入数据。

```
char input[256];

main()
{
    register int i;

    for(i =0;  i<18;  i++)
    {
        switch(fork())
        {
        case-1:        /* 错误 */
            printf("error cannot fork \n");
            exit();

        default:       /* 父进程 */
            break;

        case0:              /* 子进程 */
            for(;;)
            {
                read(0,input,256);       /*  读一行  */
                printf("%d read %s \n",i,input);
            }
        }
    }
}
```

图 10-16　对终端输入数据的竞争

对一个终端允许多个读者本身是含糊的，但是内核已尽可能好的处理这种情形了。从另一方面说，内核必须允许多个进程读一个终端，否则由 shell 创建的那些读标准输入文件的进程就会永远不能工作，因为 shell 也对标准输入进行存取。简言之，进程必须在用户级同步对终端的访问。

当用户敲入一个“文件尾”符（ASCII 码 Ctrl-d）时，行规则程序以直到文件尾符（但不包括文件尾符）的输入字符串完成对终端的系统调用 read。如果它在 clist 中只遇到文件尾符时，read 返回的是无数据（返回值为 0），这时调用进程要负责判别它已读到文件尾，因而它不应再从终端读了。参考第 7 章中 shell 程序的编码，我们看到当用户键入 Ctrl-d 时，shell 终止循环，系统调用 read 返回 0，因而 shell 退出。

本节所考虑的是笨终端硬件的情形，这种终端每一次准确地在用户键入一个字符的时刻，送一个字符的数据到机器。智能终端在其设备中能对输入数据进行加工，因而将 CPU 释放出来，以便从事其他的工作，它们的终端驱动程序和笨终端的驱动程序很相像，然而根据设备的能力要改变行规则程序的功能。

10.3.3 原始方式下的终端驱动程序

用户可以使用系统调用 ioctl 预置终端参数如擦除符、抹行符，或读取当前预置的值。类似地，他们可以控制终端是否要对它的输入进行回显、置终端波特率（位传输速率）、倾泻输入和输出 clist 或手动地开始或停止字符的输出。终端驱动程序保存这些预置的控制参数（见[SVID 85] 第 281 页），而行规则程序在收到系统调用 ioctl 的参数时预置终端数据结构中有关的字段，或读取有关的字段。当一个进程预置终端参数时，它为所有的使用该终端的进程预置这些参数，但当改变预置的进程退出时，并不自动地复位对终端的预置。

进程也可以使终端工作于原始方式下，这时行规则程序所传送的字符与用户敲入的完全一样，完全不做输入处理。然而，由于这时回车符已被视为普通的输入字符了，内核必须知道它什么时候应该完成用户的系统调用 read。它可以在终端输入了最少数目的字符之后让系统调用 read 返回，或者从终端接收任何字符以来已等待了固定的时间。在后者的情况下，内核以向 callout 表（见第 8 章）中插入表项的方法来对从终端输入字符进行定时。这两种判据（最少数目的字符和固定的时间）也是用系统调用 ioctl 预置的。当该特定的判据成立时，行规则中断处理程序唤醒所有睡眠的进程。这时，终端驱动程序将原始 clist 中的全部字符移到加工 clist 中，使用与标准方式下相同的算法，完成用户的读请求。原始方式对面向屏幕的应用，例如屏幕编辑 vi 特别重要，因为屏幕编辑 vi 有许多命令是不以回车符终结的，如命令 dw 用来删除在当前光标位置的某个字。

图 10-17 示出了通过执行一个系统调用 ioctl 来保存文件描述符 0 ——标准输入文件描述符的当前终端预置参数的程序，ioctl 的命令 TCGETA 指示驱动程序读取预置值，并将它们存放在用户地址空间的 savetty 结构中。这个命令通常用来确定一个文件是否是终端设备文件，因为它对系统中的任何东西都不做修改，即：如果它失败返回，进程就认为该文件不是终端设备文件。在本程序中，进程执行第二个系统调用 ioctl，使终端进入原始方式，它停止字符的回显，并安排在下列条件满足时，完成从终端读的操作：当从终端至少收到 5 个字符，或当收到任意数目的字符但从收到第一个字符以后经历了大约 10 秒钟。当它收到一个 SIGINT 软中断信号时，进程重置原来的终端选择并终止。

10.3.4 终端探询

有时探询一个设备是很方便的，也就是说，如果有数据出现就读数据，否则继续正常的处理。图 10-18 的程序描述了这一情形：由于以“不延迟”的选择打开终端，其后的读操作如果不存在数据，则进程不睡眠，而是立即返回（参考图 10-15 的算法 terminal _ read）。如果一个进程监视很多设备时，这个方法也是可行的：它以“不延迟”打开每一个设备，并且探询所有的设备，等待从它们之中任意一个输入。然而这个方法浪费处理能力。

BSD 系统有一个允许探询设备的系统调用 select，这个系统调用的语法是：

```
select (nfds, rfds, wfds, efds, timeout)
```

其中 nfds 给出了欲选择的文件描述符的数目；rfds、wfds 和 efds 是指向“所选择的”打开文件描述符的位图的指针，也就是说，如果用户要选择文件描述符 fd，则 1≪fd（1 左移文件描述符的值那么多位）位被置位；timeout 指出为等待数据的到来 select 要睡眠多长时间。

```
#include        <signal.h>
#include        <termio.h>
struct termio savetty;
main()
{
        extern sigcatch();
        struct termio newtty;
        int nrd;
        char buf[32];
        signal(SIGINT,sigcatch);
        if  (ioctl(0,TCGETA, &savetty)==-1)
        {
                printf("ioctl failed: not a tty\n");
                exit();
        }
        newtty=savetty;
        newtty.c_lflag &=~ICANON;          /*停止标准方式*/
        newtty.c_lflag &=~ECHO;            /*停止字符回显*/
        newtty.c_cc[VMIN]=5;               /*最小值为5个字符 */
        newtty.c_cc[VTIME]=100;            /*10秒间隔*/
      if(ioctl(0,TCSETAF, &newtty)==-1)
        {
            printf("cannot put tty into raw mode\n");
            exit();
        }
        for(;;)
        {
                nrd=read(0,buf,sizeof(buf));
                buf[nrd]=0;
                printf("read %d chars '%s'\n",nrd,buf);
        }
}
sigcatch()
{
        ioctl(0,TCSETAF, &savetty);
        exit();
}
```

图 10-17　以原始方式读入五字符组

举例来说，在 timeout 值截止以前在某些文件描述符上到来了数据，select 返回，并在位图中指示是哪一些文件描述符。又比如，某用户想在收到文件描述符 0、1、2 的输入以前一直睡眠，则 rfds 应指向值为 7 的位图，当 select 返回时，该位图就会被一个指示哪一个文件描述符已有数据准备好的位图所改写。位图 wfds 对写文件描述符完成类似的功能，位图 efds 对特定文件描述符指示什么时候有例外情况存在，这点对网络是很有用的。

```
#include<fcntl.h>

main()
{
        register int i,n;
        int fd;
        char buf[256];

        /*以只读方式及"不延迟"选择打开终端*/
        if((fd =open("/dev/tty",O_RDONLY|O_NDELAY))==-1)
                exit();

        n=1;
        for(;;)         /*无限循环*/
        {
                for(i =0;  i<n;  i++)
                        ;
                if(read(fd,buf,sizeof(buf))>0)
                {
                        printf("read at n %d\n",n);
                        n--;
                }
                else        /* no data read;returns due to no-delay */
                        n++;
        }
}
```

图 10-18　终端探询

10.3.5　建立控制终端

控制终端是用户借以注册到系统中去的终端，它控制用户在该终端创建的那些进程。当一个进程打开一个终端时，终端驱动程序就会打开行规则程序，如果该进程是一个进程组组长——这是先前的一次系统调用 setpgrp 的结果，而且该进程还没有一个与之相关联的控制终端时，行规则程序就使上述打开的终端成为它的控制终端。它把该终端设备的主设备和次设备号存储在该进程的 u 区，并把打开终端的进程的进程组号存储在终端驱动程序数据结构中。下面我们将会看到，这个打开终端的进程是个控制进程，通常是该用户的注册 shell 进程。

控制终端在处理软中断信号时起着重要的作用。当用户按下 delete，break，rubout 或 quit 键时，中断处理程序调用行规则程序，行规则程序向该控制进程组中所有的进程发适当的软中断信号。类似的，如果用户挂起，终端中断处理程序会从硬件接收到一个挂起指示，而行规则程序向该控制进程组中所有的进程发出一个 SIGHUP（hangup）软中断信号，用这种方法使一个特定的终端上创建的所有进程都收到 SIGHUP 信号。收到这一信号的多数进程的缺省反应是退出，这就是当一个用户突然关掉终端时这些杂散的进程被杀掉的方法。在发出 SIGHUP 信号后，终端中断处理程序断开该终端与该进程组的联系，使该进程组中的

进程再也不能接收从该终端发出的软中断信号了。

10.3.6　间接终端驱动程序

进程常常需要直接从控制终端上读或写数据，甚至当标准输入和标准输出文件可能已被重定向到其他文件上了。例如，一个 shell 命令文件能直接送紧急的消息到终端，虽然它的标准输出和标准错误文件可能被重定向到其他什么地方去了。UNIX 系统提供通过设备文件“/dev/tty”进行的“间接的”终端访问，而该设备文件标志每一个有控制终端进程的控制终端，通过不同的终端注册的用户都能访问“/dev/tty”，只是他们访问的终端不同。

对于内核，有两种从文件名“/dev/tty”找到控制终端的一般的方法。第一种方法，内核可以为间接终端文件定义一个特殊的设备号，并在字符设备开关表中设一个特殊的入口，当调用间接终端时，间接终端的驱动程序从 u 区中取控制终端的主设备和次设备号，并通过字符设备开关表调用实终端驱动程序。从文件名“/dev/tty”找到控制终端的第二种通常使用的方法是在调用驱动程序的 open 子程序以前，测试一下该主设备号是否是间接终端的主设备号，如果是的话，它释放“/dev/tty”的索引结点，分配索引结点给控制终端，重置文件表项使它指向控制终端索引结点，然后调用该终端驱动程序的 open 子程序，这时打开“/dev/tty”返回的文件描述符直接引用的就是控制终端以及它的通常的驱动程序了。

10.3.7　注册到系统

如第 7 章所述，1 进程——init 进程，执行一个无限循环，该循环从文件“/etc/inittab”中读入有关进入系统状态如“单用户”或“多用户”时如何做的指示。进入多用户状态后，init 的一个主要责任是允许用户通过终端注册（图 10-19），它创建若干个被称为 getty 的进程

```
算法　login　　/*　注册的过程　*/
{
    getty 进程执行：
    置进程组(系统调用 setpgrp)；
    打开 tty 线路；　　/*　睡眠直到被打开　*/
    if （打开成功）
    {
        执行 login 程序；
        显示输入用户名要求；
        停止回显字符,显示输入口令字要求；
        if （成功）　　/* 与　/etc/passwd 中口令字相符　*/
        {
            将 tty 置为标准方式(用 ioctl)；
            执行 shell 程序；
        }
        else
            计算企图注册的次数,小于某值时再试；
    }
}
```

图 10-19　系统注册算法

(该进程是为接通终端或接通“tty”的)，并记录哪一个 getty 进程打开哪一个终端；每个 getty 进程使用系统调用 setpgrp 重置它的进程组标识号，打开一个特定的终端线路，并且通常睡眠在系统调用 open 中，直到机器检测到与终端的硬件连接建立为止。当 open 返回时，getty 通过系统调用 exec 执行注册程序 login，该程序要求用户使用注册名和口令字标识他们自己。如果用户成功地注册了，则 login 最后通过系统调用 exec 执行 shell，而该用户就可以开始工作，这次对 shell 的调用称为注册 shell，该 shell 进程与原来的 getty 进程有相同的进程标识号，因此注册 shell 就是一个进程组组长。如果一用户未能成功地注册，则在适当的时间限制之后 login 退出，关闭已打开的终端线路，而 init 进程为该线路创建另一个 getty 进程。init 进入暂停状态，直到它接收到子进程死的软中断信号为止。当它被唤醒时，它查明该僵死的进程是否是一个注册 shell 进程，若是，它创建出另一个 getty 进程以便打开该终端，以此来代替已死亡的那个进程。

10.4　流

虽然实现设备驱动程序的机构是满足要求的，但存在某些缺点，随着时间的推移这些缺点变得愈加明显了。不同的驱动程序往往会有功能性的重复，特别是对实现网络协议的驱动程序，因为这类驱动程序通常包含一个设备控制部分和一个协议部分。虽然协议部分对所有网络设备应当是共同的，但实际实现时并非如此，因为内核并不提供适当的公共使用的机制。例如，字符表 clist 对它们的缓冲能力是有用的，然而由于逐个字符地进行操作使其开销过大，为了提高性能，企图绕过这一机制又会导致 I/O 子系统模块性的破坏。在驱动程序级缺乏公用性也会渗透到用户命令级，其中若干命令可能在不同的介质上完成公共的逻辑功能。驱动程序机制的另一个缺点是网络协议也要求一种类似于行规则程序的能力，每个规则程序实现协议的一部分，又要能以一种灵活的方式将各组成部分结合在一起。然而，要把传统的行规则程序叠加到一起是困难的。

Ritchie 最近实现了一个被称为流（stream）的机制，以提高 I/O 子系统的模块性和灵活性，此处的叙述是基于他的工作[Ritchie 84b]，虽然在系统 V 中的实现与此稍有区别。流，是一进程与一设备驱动程序间的一个全双工的连接，它由 一组线性地链接着的队列对所组成，每个队列对中的一个成员用于输入，另一个成员用于输出。当进程向流写数据时，内核把数据送到输出队列；当一个设备驱动程序接收了输入数据，它将输入队列的数据送给一个正在读的进程。队列还根据明确定义的接口将消息传递给相邻的队列。每个队列对与某内核模块的一个实例相联系，例如驱动程序、行规则程序或协议模块，这些模块操纵通过它的队列的数据。

每个队列是一个包含下列元素的数据结构：

- 一个在系统调用 open 执行时调用的打开过程。
- 一个在系统调用 close 执行时调用的关闭过程。
- 一个将一消息送入队列去时调用的“放入”过程。
- 一个当队列被调度执行时调用的“服务”过程。
- 一个指向该流中下一个队列的指针。
- 一个指向等待服务的消息表的指针。
- 一个指向维护该队列的状态的私用数据结构的指针。

• 用来进行流量控制、调度和维护队列状态的标志位以及高、低水准线。

核心分配存储器中相邻的位置给队列对，因此，一个队列可以容易地找到该队列对中另一个成员。

一个具有流驱动程序的设备是一字符设备，在字符设备开关表中它有一特殊字段，该字段指向流初始化结构，这个结构含有上面提到过的若干子程序的地址以及高、低水准线。当内核执行系统调用 open 时，发现该设备文件是一个字符特殊文件，它检查字符设备开关表中这一新字段，如果在那里没有入口，则该驱动程序不是一个流驱动程序，因而内核执行通常的字符设备过程。然而，当第一次打开一个流驱动程序时，内核分配两个队列对，一个队列对作为流头标，另一个是为驱动程序使用的。流头标模块对所有已打开流的实例是相同的，它含有通用的放入和服务过程，也是与实现系统调用 read，write 和 ioctl 的高层内核模块的接口。内核还对驱动程序队列结构进行初始化工作，赋值队列指针，并从每个驱动程序初始化结构中拷贝驱动程序的子程序地址，然后调用驱动程序打开过程。驱动程序打开过程完成通常的初始化工作，但将信息保存到与之相联系的队列中去，以便再次调用。最后，内核在内存索引节点中赋值一特殊指针以指向流头标(图 10-20)。这样，当其他进程打开此设备时，内核就可以通过索引节点指针找到先前所分配的流，并调用该流上所有模块的打开过程了。

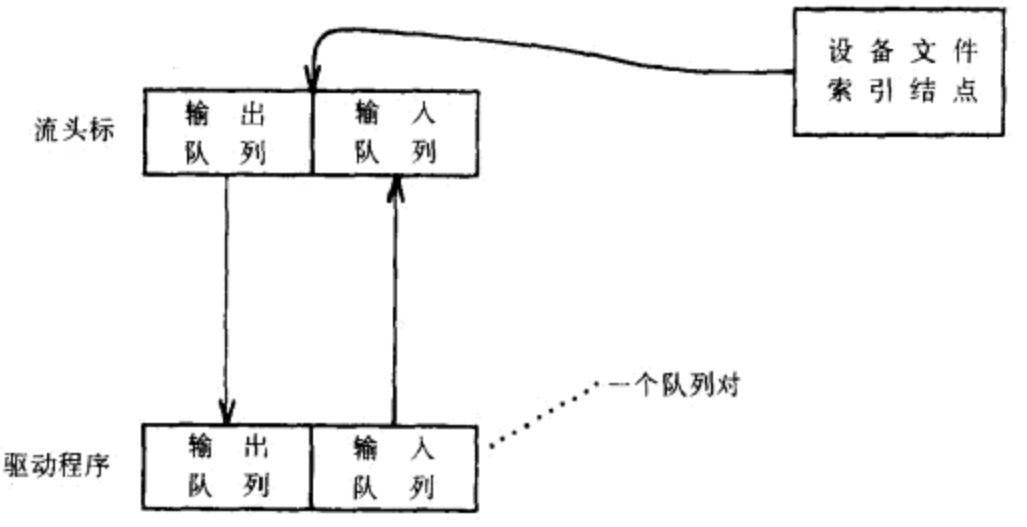


图 10-20　打开后的流

模块间通信靠在一个流上向相邻模块传递消息来实现。一个消息是由一个消息块头部的链接表组成的，每个块头部指向该块数据的起始和终止位置。有两类消息——控制消息和数据消息，由消息头部中的类型指示进行标识。控制消息可以由系统调用 ioctl 或如终端挂起这样的特殊条件引起，而数据消息可能是由系统调用 write 或数据从一个设备中到来引起。

当某进程向一个流写时，内核将数据从用户空间先拷贝到由该流头标分配的消息块中去，流头标模块调用下一个队列模块的放入过程。这个过程可以处理该消息；立即传递给下一个队列；或者将之送入队列以便以后处理。对于最后这一种情形，该模块将那个消息块头部链入一个链接表中，形成一个双向链接表（图 10-21），然后它在它的队列数据结构中置位一个标志位，指示它有数据要处理，并调度自己进行服务：这个模块将该队列放入请求服务的队列链接表中，并向一个调度机构发出请求，由调度程序调用这个表中每个队列的服务过程。内核也可以用软件中断来调度模块，这一点与它调用 callout 表中的函数（如第 8 章中所述）相类似，软件中断处理程序再调用某个服务过程。

进程可以通过发出系统调用 ioctl 将模块压入一个已打开的流中去，此时内核将被压入的模块嵌到紧挨着流头标下面，并连接队列指针使该结构保持为一双向链接表。流中低层的模块并不关心它们是与流头标进行通信的，还是与一个被压入的模块进行通信的，其接口是该流上的下一个队列的放入过程，而下一个队列是属于刚被压入的模块的。举例来说，一个进程可以将一个行规则模块压到一终端驱动流上去以便进行对擦除和抹行符的处理（图 10-22），虽然这里的行规则模块与 10.3 节中所描述的行规则程序的接口不一样，但其功能是相同的。如果没有行规则模块，则终端驱动程序不对输入字符进行处理，因而这些字符不加改变地到达流头标。一个调用系统调用 open 打开一个终端、以及压入一个行规则模块的程序段可以像以下方式书写：

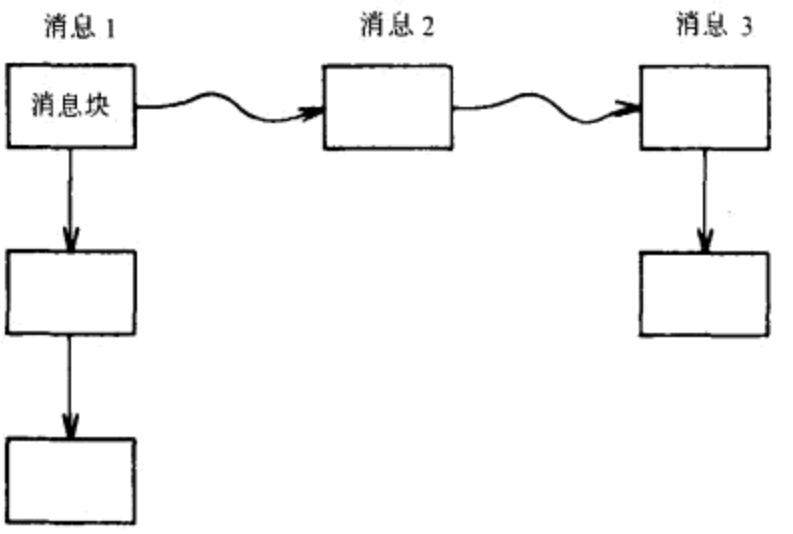


图 10-21　流中的消息

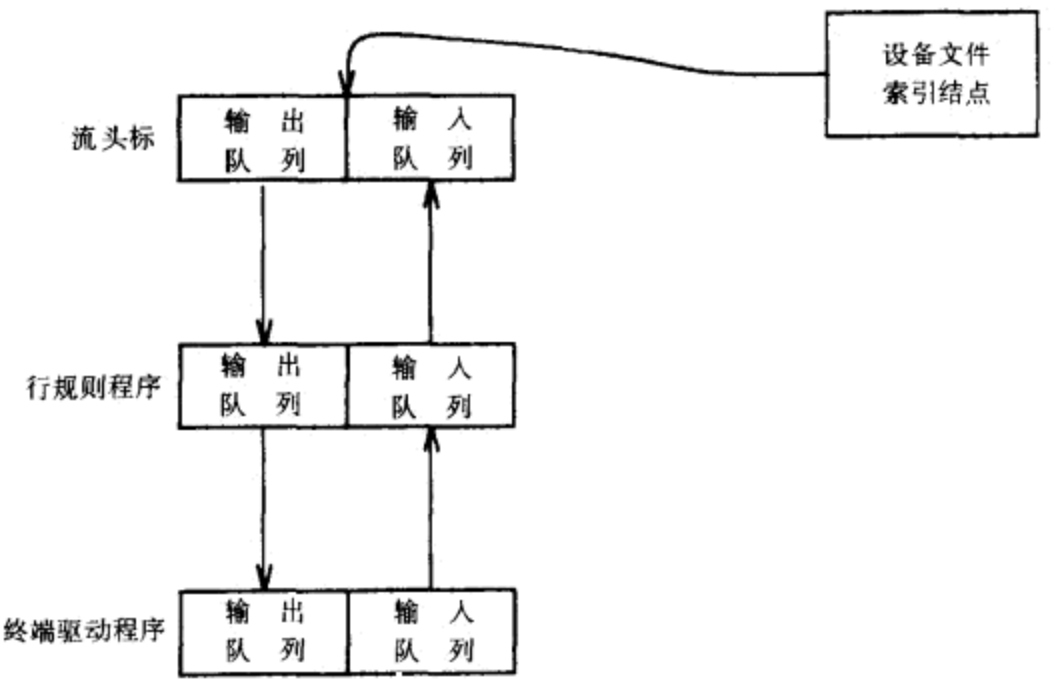


图 10-22　将一模块压入流

```
fd=open("/dev/ttyxy",O_RDWR);
ioctl(fd,PUSH,TTYLD);
```

其中 PUSH 是命令名，而 TTYLD 是标识行规则模块的一个数字。压多少个模块到一个流是不受限制的，而且一个进程可以按照后进先出的次序从流中“上托”掉模块，这是靠使用另一个系统调用 ioctl 实现的：

```
ioctl(fd,POP,0);
```

假定一个终端行规则模块实现通常的终端处理功能，它下面的设备可以是一个网络连接，而不是与一个单个终端设备的连接，行规则程序以同样的方式工作，不管它下面的模块

是什么。本例表明了通过内核模块的组合所得到的较大灵活性。

10.4.1　流的详细的示例

Pike 叙述了利用流来实现多路复用虚终端（virtual terminal，见[Pike 84]），用户可见到几个虚终端，每个虚终端在物理终端上占有一个窗口。虽然 Pike 的文章所描述的是对一个智能终端的方案，但它对笨终端也是适用的。每个窗口可以占据整个屏幕，用户通过敲入一个控制序列完成虚窗口间的切换。

图 10-23 示出了进程和内核模块的安排。用户请求一个被称为 mpx 的进程来控制物理终端，mpx 读物理终端线路，并等待有关控制事件的通知，例如创建一个新窗口、切换控制到另一个窗口、删除一个窗口等等。当它接收到一个用户想要创建一个新窗口的通知时，mpx 产生一个进程去控制新窗口，并通过一个虚终端（简称 pty）来与之通信。pty 是一个以成对方式操作的软设备：送往某对的一个成员的输出被送到该对的另一个成员的输入，而输入被送到其上的模块。为了建立一个窗口（图 10-24），mpx 分配一个 pty 对，并打开其中一个成员，建立一个到它去的流（驱动程序 open 应保证该 pty 以前未被分配过）。mpx 创建一个新进程，新进程打开该 pty 对的另一个成员。mpx 将一个消息模块压入它的 pty 流中去，以便将控制消息变换为数据消息（下一段将解释原因），而子进程在执行 shell 以前将一个行规则模块压入它的 pty 流。这时，shell 运行在一个虚终端上，对用户来讲，这与一个物理终端没有区别。

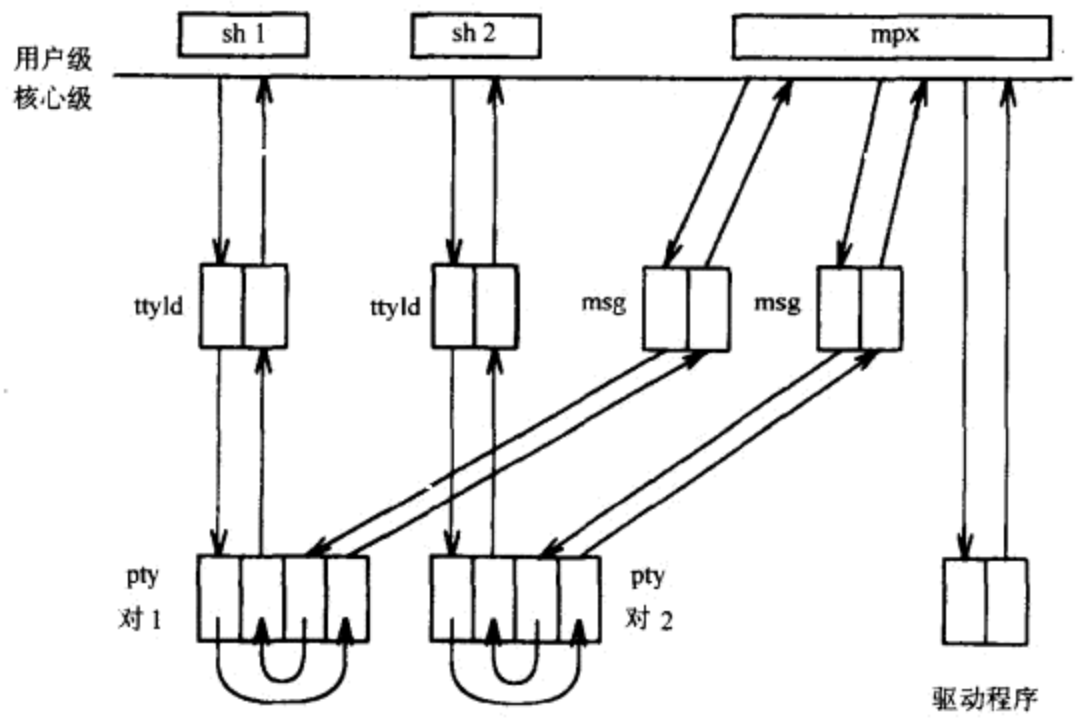


图 10-23　在物理终端上的虚终端窗口

mpx 进程是一个多路转接器，它把虚终端的输出发送到物理终端上，并将物理终端的输入多路分配给正确的虚终端。mpx 使用系统调用 select 等待在任意线路上数据的到来，当数据从物理终端到来时，mpx 要确定这是否是通知它产生一个新窗口或删除一个老窗口的控制消息，或者这是否是欲送往正在读虚终端的进程去的数据消息。若是后一种情形，该数据应当含有一个用来识别目的虚终端的头部，mpx 从该消息中剥去头部，并将数据写到适当的 pty 流中去，pty 驱动程序又通过终端行规则程序将数据送给读进程。当一个进程对虚终端进行写操作时发生相反的过程：mpx 添加头部到数据上，通知物理终端该数据应当显

示在哪一个窗口上。

```
/*  假设文件描述符 0 和 1 已经指的是物理终端  */
for(;;)    /*  无限循环  */
{
    select(输入);    /*  等待某线路有输入  */
    读输入线路;
    switch(有输入数据的线路)
    {
    case 物理终端:    /*  在物理终端线路上输入  */
        if (控制命令)    /*  例如产生新窗口  */
        {
            打开一个空闲虚终端;
            派生一新进程;
            if (父进程)
            {
                在 mpx 侧压入一消息行规则模块;
                continue;    /* 回到循环开始 */
            }
            /*  子进程从此处执行  */
            关闭不必要的文件描述符;
            打开虚终端对的其他成员,使之成为子进程的 stdin、stdout、stderr;
            压入 tty 行规则模块;
            执行 shell 程序;    /*  如同虚终端  */
        }
        /*  从终端送上来的到某虚终端的"正规"数据  */
        对从物理终端读的数据解复用,剥去头部并写到适当的虚终端去;
            continue;    /*  回到 for 循环开始  */
    case 逻辑终端:    /*  一虚终端正向某窗口写  */
        形成指示数据欲送到哪个窗口的头部;
        将头部和数据写到物理终端;
        continue;    /*  回到 for 循环开始  */
    }
}
```

图 10-24　多路复用窗口的伪码

如果进程在一个虚终端上发系统调用 ioctl，则终端行规则程序完成对它的虚线路的必要的预置，但对每个虚终端的预置可能是不同的。然而，依赖于具体设备，某些信息可能需要送到物理终端去，消息模块把由 ioctl 产生的控制消息转换成适合于 mpx 读和写的数据消息，这些数据消息被送往物理设备。

10.4.2　对流的分析

Ritchie 提到他曾试图仅用放入过程来实现流，或仅用服务过程来实现。然而，服务过程对流量的控制是必需的，因为如果某模块的相邻模块暂时不能接收更多的数据时，则该模块必须将数据送入队列。放入过程也是必需的，因为常常必须直接地把数据递交给相邻的模

块。例如，终端行规则程序必须尽可能快的将输入数据回显到终端，这只有使系统调用 write 能直接地调用下一个队列的放入过程，该过程接着又调用再下一个队列的放入过程等等，而不需要调度机构才有可能。另外，一个进程当输出队列拥挤时会睡眠，然而，在输入这一侧的模块不能睡眠，因为它们是由中断处理程序调用的，这样，就会使一个无辜的进程进入睡眠。在输入和输出方向上，模块间通信是不对称的，降低了这个方案的完美性。

将每个模块设计为分离的进程本来也是更为可取的，但是使用大量的模块会造成进程表的溢出。所以它们是以特殊的调度机制——软件中断实现的，这个机制与正常的进程调度无关。因此，模块不可能进入睡眠，因为这会使一个任意的进程（被中断的进程）进入睡眠。模块也必须在其内部保存它们的状态信息，这使它们的编码比起若允许进入睡眠时的情形更加臃肿些。

流的实现中有几个反常情况：

• 以流实现时，进程记账变得困难，因为模块并不必须在使用流的进程的上下文中运行。另外假设所有的进程均匀地分担流模块的执行开销也是错误的，因为某些进程可能要求使用复杂的网络协议，而另一些进程可能使用简单的终端行规则。

• 原来用户能够这样来使一个终端驱动程序进入原始方式：如果没有数据时，过了一短暂的时间后，系统调用 read 返回（例如，在图 10-17 中使 newtty.c _ cc [VMIN] = 0）。用流来实现这一点是困难的，除非在流头标一级引入特殊情况的编码。

• 流是线性地连接着的，不容易在内核中多路复用。例如，前一节中的窗口的例子是在用户级进程完成多路复用的。

虽然存在这些反常情况，流对改进驱动程序模块的设计仍是大有希望的。

10.5　本章小结

本章给出了 UNIX 系统上设备驱动程序的概貌。设备或者是块设备，或者是字符设备，设备与内核的其余部分的接口依赖于设备类型。块设备的接口是块设备开关表，它由设备打开、关闭和策略过程的入口点组成，策略过程控制数据传送到块设备和从块设备传送数据。字符设备接口是字符设备开关表，它由设备打开、关闭、读、写和 ioctl 过程的入口点组成。系统调用 ioctl 使用字符设备的 ioctl 接口，它允许在进程和设备间发送控制信息。当接收到一个设备中断时，内核根据中断向量表中存储的信息，以及由发出中断的硬件所提供的参数，调用设备中断处理程序。

磁盘驱动程序将文件系统使用的逻辑块号变换成物理磁盘上的位置。块设备接口允许内核缓冲数据，原始设备接口允许较快地从磁盘输入或输出到磁盘，但绕过高速缓冲，造成较多的文件系统被破坏的机会。

终端驱动程序支持与用户的主要接口。内核以三个字符表 clist 与每个终端相联系：一个用于从键盘敲入的原始输入；一个用于考虑了擦除、抹行和回车符的作用的、经处理的输入；还有一个用于输出。系统调用 ioctl 允许进程控制内核如何看待输入的数据，因而能将终端置为标准方式或设置原始方式下的各种参数。getty 进程打开终端线路，等待连接建立。然后它设置它的进程组标识号，使注册 shell 最终成为一个进程组组长；通过 ioctl 初始化终端参数；并在通过注册序列后给用户送提示符。这样建立的控制终端就会对进程组中的进程发软中断信号，作为对诸如用户挂起或按下 break 键发出的事件的响应。

流是一种改善设备驱动程序和协议的模块性的方案。流是进程和设备驱动程序间的一个全双工连接，它可以包含行规则和协议模块，以便在途中处理数据。流模块的特点是有明确定义的接口，以及具有与其他模块结合起来使用的灵活性。它们提供的灵活性为网络协议和驱动程序的开发带来极大的好处。

10.6　习题

*1. 假设一个系统含有两个具有相同主设备号和次设备号的设备文件，而且都是字符设备。如果两个进程想要同时打开这个物理设备，试说明它们打开相同的设备文件或不同的设备文件是没有区别的。它们关闭这个设备时会发生什么情况？

2. 回忆第 5 章中系统调用 mknod 为创建一个设备特殊文件要求超级用户的权限。已知设备的存取是由文件的许可权方式决定的，为什么 mknod 要求超级用户的许可权？

3. 编写一个程序，它能证实磁盘上的文件系统不相重叠。这个程序取两个参数：一个代表磁盘卷的设备文件；一个给出该类型磁盘的扇区数目和扇区长度的描述符文件。该程序应读入超级块以确信文件系统是互不重叠的。这样的程序所给出的结果是不是永远是正确的？

4. mkfs 程序通过创建一个超级块、为索引节点保留空间、将所有的数据块放入一个链接表以及形成一个根索引节点目录来初始化一个文件系统。你想怎样来编写 mkfs 的程序？如果要有一个卷内容表时，这个程序应怎样改变？它应怎样初始化这个卷内容表？

5. 试对 mkfs 和 fsck 是用户级的程序，而不是内核的一部分进行评述。

6. 设想一个程序员编写一个运行在 UNIX 系统上的数据库系统。数据库程序在用户级运行，而不是作为内核的一部分运行。这个数据库系统应如何与磁盘交互？并考虑以下问题：

- 比较使用正规文件系统接口和原始磁盘接口，
- 对速度的要求，
- 需要知道什么时候数据真正驻留在盘上，
- 数据库的大小：它能装入一个文件系统、一个整个的磁盘卷或几个磁盘卷吗？

7. UNIX 内核从策略上假设文件系统是装在完美的磁盘上的。然而，磁盘可能含有些毛病，使某些扇区不能使用，虽然磁盘的其余部分仍是“好的”。试说明磁盘驱动程序（或智能磁盘控制器）怎样才能允许少量的坏区。这会对性能有什么样的影响？

8. 当安装一个文件系统时，内核调用驱动程序的打开过程，但在系统调用结束时释放设备特殊文件的索引节点。当拆卸一个文件系统时，内核存取设备特殊文件的索引节点，调用驱动程序关闭过程，然后释放索引节点。试对索引节点操作及驱动程序打开和关闭的顺序与打开和关闭一个块设备的顺序进行比较和评论。

9. 运行图 10-14 的程序，但将输出重定向到一个文件。试对该文件内容与当输出到终端时的输出结果进行比较。为停止运行你必须中断进程，故让它运行足够长时间以得到足够的输出量。如果程序中的系统调用 write 被置换为

```
printf(output);
```

会发生什么情况？

10. 当用户试图在后台执行正文编辑时：

```
ed file &
```

会发生什么情况？为什么？

11. 当用户已注册时，终端文件通常有如下的存取许可权

```
crw--w--w-   2 mjb    lus    33, 11 Oct 25 20:27   tty61
```

也就是说，对用户“mjb”允许读或写，对其他用户只允许写，为什么？

12. 假设你知道一个朋友的终端设备文件名，试编写一个程序能让你把消息写到你的朋友的终端上去。你还需要什么其他信息以便编写一个与通常的 write 命令类似的合理的屏幕图象？

13. 实现 stty 命令。当不带参数时，它读取终端的预置值，并将它们报告给用户；否则用户可以交互式的设置各种预置参数。

14. 编写一个行规则程序，它在每行输出开始时写上机器名。

15. 在标准方式下，用户通过在键盘上键入“Ctrl-s”来暂时停止向终端的输出，键“Ctrl-q”来继续输出。标准行规则程序应如何实现这个特性？

*16. 进程 init 为系统中每一个终端创建一个 getty 进程。假设两个 getty 进程同时对一个终端存在着并等待用户注册，会发生什么情况？内核能防止这种情况发生吗？

17. 假设 shell 被编码成“忽略”文件尾符，并继续读标准输入。当用户（在注册 shell 中）键入文件尾符并继续键入时会发生什么情况？

*18. 假设一进程读它的控制终端但忽略或捕俘 hangup 软中断信号。当该进程继续读控制终端时会发生什么情况？

19. getty 程序负责打开一条终端线路，而 login 程序负责检查注册和口令字信息。两部分功能由分开的程序完成的优点是什么？

20. 考虑 10.3.6 节说明的实现间接终端（“/dev/tty”）的两种方法。用户会观察到什么区别？（提示：考虑系统调用 stat 和 fstat）

21. 试设计一种调度流模块的方法，其中内核含有一个特殊的进程，当模块的服务过程被调度执行时，它执行这些过程。

*22. 试设计一种使用普通的（非流的）驱动程序的虚拟终端（窗口）方案。

*23. 试设计一种使用流实现虚终端的方案，其中由一个内核模块，而不是用户进程，在虚终端和物理终端间完成多路复用 I/O。说明一种允许扇入和扇出的连接流的机制。将多路复用模块放入内核是否比以用户进程来建立它好一些？

24. ps 命令报告一个正在运行的系统中进程活动的有趣的信息。传统实现时，ps 从内核存储区直接读进程表中的信息。在进程表项的大小会改变的开发环境中，ps 不容易找到进程表的正确的字段，这种方法是不稳定的。编写一个迫切地需要这种改变的环境的驱动程序。

第 11 章　进程间通信

进程间通信机制允许任意进程间交换数据和同步地执行。我们已经考察过几种形式的进程间通信了，如管道、有名管道和软中断信号。管道（无名的）存在着这样的缺点：只有成为调用系统调用 pipe 的进程的子孙后代的那些进程才知道这些管道，因而相互无关的进程不能通过管道通信。虽然有名管道能使相互无关的进程通信，但它们一般来讲既不能用于跨越一个网络的进程（见第 13 章），也不容易用它们来为不同的通信进程集合建立多条通信通路，也就是说，不可能多路复用一个有名管道以便为多对通信的进程提供私用的通道。任意进程间也可以通过系统调用 kill 发送软中断信号来相互通信，然而所传递的“消息”只有软中断信号的序号。

本章讨论进程间通信的其他几种形式。讨论是从进程跟踪开始的，进而讨论一个进程跟踪和控制另一个进程的执行。然后解释系统 V 进程间通信（Interprocess Communication，以下简称 IPC）程序包：消息（message）、共享存储区（shared memory）和信号量（semaphore），接着评述进程通过网络与其他机器上的进程通信的传统方法，最后给出 BSD 套接字（socket）在用户级的概貌。本章并不讨论网络中的一些专门的问题，诸如协议、寻址和名字服务等，因为这些已超出了本书的范围。

11.1　进程跟踪

UNIX 系统为跟踪进程的执行提供了一个原语形式的进程通信法，这对调试是很有用的。一个调试进程，例如 sdb，派生一个被跟踪的进程，并以系统调用 ptrace 来控制它的执行、设置断点和清除断点、以及从它的虚地址空间读和写数据。因此，进程跟踪是由同步调试进程与被跟踪进程以及控制被跟踪进程的执行组成的。

图 11-1 的伪码示出了调试程序的典型结构。调试程序派生一个子进程，子进程调用系统调用 ptrace，其结果是：内核在子进程的进程表项中设置一跟踪位。然后，子进程执行被跟踪的程序，例如，如果用户正在调试 a.out 程序，则子进程应当执行 a.out。和通常一样，内核执行系统调用 exec，然而，当执行到最后时注意到跟踪位被置位，因而给子进程发送一个“SIGTRAP”软中断信号。当从系统调用 exec 返回时，与任何其他系统调用返回时内核检查软中断信号一样，它也检查软中断信号，并发现了它刚刚给自己发的“SIGTRAP”软中断信号，因而执行作为处理软中断信号的一种特殊情形的进程跟踪代码。注意到它的进程表项中的跟踪位是被置位的，子进程唤醒正睡眠在系统调用 wait 中的父进程（这一点下面我们会看到），进入一个类似于睡眠状态的跟踪状态（图 6-1 的进程状态图中未画出），并完成上下文的切换。

一般地说，父进程（调试程序）在此时应当进入一个用户级的循环，执行系统调用 wait，等待由被跟踪的进程唤醒。当被跟踪进程唤醒了调试进程时，调试进程从 wait 中返回，读用户输入的命令，并将这些命令变换成一系列的系统调用 ptrace 以便控制子进程（被跟踪的进程）。系统调用 ptrace 的语法是：

```
ptrace (cmd, pid, addr, data);
```

其中 cmd 规定各种命令如读数据、写数据、继续执行等等；pid 是被跟踪进程的进程标识号；addr 是欲读或写的数据在子进程中的虚地址；而 data 是欲写的整数值。当执行系统调用 ptrace 时，内核先证实该调试程序有一个进程标识号为 pid 的子进程，而且该子进程正处于被跟踪的状态，然后它使用一个全局的跟踪数据结构，用来在两个进程间传送数据。它将

```
if ((pid=fork())==0)
{
    /* 子进程—被跟踪进程 */
    ptrace (0, 0, 0, 0);
    exec ("此处给出被跟踪程序名")
}
/* 调试进程从此处继续执行 */
for (;;)
{
    wait ( (int *  ) 0);
    read (输入跟踪指令)
    ptrace (cmd, pid, …);
    if (停止跟踪)
        break;
}
```

图 11-1　调试进程的结构

跟踪数据结构上锁，防止其他跟踪进程对它改写，将 cmd，addr 和 data 拷贝到该数据结构，唤醒子进程并使其进入“就绪”状态，然后进入睡眠直到子进程响应。当子进程继续执行（在核心态）时，它执行适当的跟踪命令，把它的回答写到跟踪数据结构中去，然后唤醒调试程序。根据跟踪命令的类型，子进程可能再次进入跟踪状态，并等待一个新的命令，或者从软中断信号处理中返回，继续执行。当调试程序继续执行时，内核将被跟踪进程提供的“返回值”保存起来，对跟踪数据结构解锁，并返回到用户态。

如果当子进程进入跟踪状态时调试程序并未睡眠在系统调用 wait 中，那么，直到调试程序调用 wait 时它才能发觉它的被跟踪的子进程，而在此时它立即返回，并像刚刚描述的那样进行处理。

让我们来考察图 11-2 和图 11-3 中的两个程序，它们分别被称为 trace 和 debug。如果在终端运行 trace 时，数组 data 的值是 0，该进程在屏幕上打印出 data 的地址后退出。但是现在我们用一个等于 trace 打印出的值作为参数运行 debug，debug 将该参数保留在变量 addr 中，然后派生一个子进程，该子进程调用 ptrace 来使自己符合跟踪条件，接着执行 trace 程序。在系统调用 exec 结束时内核送给子进程（称它为 trace 进程）一个 SIGTRAP 软中断信号，因而 trace 进程进入了跟踪状态，等待着从 debug 发来的命令。如果 debug 一直在 wait 中睡眠，那么它被唤醒，找到被跟踪的进程，并从 wait 中返回。然后 debug 调用 ptrace，将循环变量 i 的值写入 trace 的数据空间 addr 处，并增大 addr。在 trace 程序中，addr 是数组 data 中的一个元素的地址。debug 进程的最后一次调用 ptrace 导致了 trace 进程运行，而此时

数组 data 中已含有数值 0 到 31 了。像 sdb 这样的调试程序可以存取被跟踪进程的符号表，从符号表它可以确定用来作为系统调用 ptrace 参数的地址值。

```
int data [32];
main ()
{
    int i;
    for (i=0; i<32; i++)
        printf ("data [%d] = %d \n", i, data [i]);
    printf ("ptrace data addr 0x%x \n", data);
}
```

图 11-2　trace ——一个被跟踪进程

```
#define TR_SETUP 0
#define TR_WRITE 5
#define TR_RESUME 7
int addr;

main (argc, argv)
    int argc;
    char * argv [];
{
    int i, pid;

    sscanf (argv [1],"%x", &addr);

    if ( (pid=fork ()) ==0)
    {
        ptrace (TR_SETUP, 0, 0, 0);
        execl ("trace","trace", 0);
        exit ();
    }
    for (i=0; i<32; i++)
    {
        wait ( (int * ) 0);
        /* 将 i 的值写入进程 pid 在地址 addr 中去 */
        if (ptrace (TR_WRITE, pid, addr, i) ==-1)
            exit ();
        addr+=sizeof (int);
    }
    /* 被跟踪进程应继续执行 */
    ptrace (TR_RESUME, pid, 1, 0);
}
```

图 11-3　debug ——一个跟踪进程

使用 ptrace 进行进程跟踪是原语性的，并存在以下若干缺点：

- 为在调试进程与被跟踪进程间传送一个字的数据，内核必须完成四次上下文的切换：

在调试进程中内核在系统调用 ptrace 中切换上下文，直到被跟踪的进程对询问进行回答时，内核切换到被跟踪进程的上下文，再从被跟踪的进程上下文切换出来，当接收到 ptrace 的回答时，又要把上下文切换回调试进程去。这一开销是必需的，因为调试进程没有其他方法来存取被跟踪进程的虚地址空间，因此进程跟踪是很慢的。

• 一个调试进程能同时跟踪若干子进程，虽然这一特点在实际中是几乎不使用的。更重要的是，一个调试程序只能跟踪它的子进程，倘若被跟踪的子进程又派生子进程，调试进程对其孙子进程是无法控制的，这在调试复杂程序时是一个严重的缺陷。如果被跟踪进程调用了系统调用 exec，由于原来有 ptrace，后来被系统调用 exec 执行的映象仍然被跟踪，然而调试程序可能不知道被执行的映象的名字，因而难以进行符号调试。

• 一个调试程序不能跟踪一个已在执行的进程，如果这个被调试的进程未曾调用过 ptrace 使内核知道它同意被跟踪，这是很不方便的，因为一个进程若需要调试必须先被杀死，再以跟踪方式重新启动。

• 跟踪 setuid 程序是不可能的，因为用户能通过 ptrace 写入它们的地址空间去，及做些非法的操作，这可能破坏安全性。例如，假设 setuid 程序以文件名“privatefile”调用系统调用 exec，一个聪明的用户可能会以 ptrace 将文件名改写为“/bin/sh”，以未被批准的许可权执行 shell（和由 shell 执行的所有程序）。因此，当进程被跟踪时，exec 忽略 setuid 位，以防止用户改写 setuid 程序的地址空间。

Killian［Killian 84］描述了跟踪进程的一种不同的方案，这一方案是基于第 5 章中所说的文件系统的。系统管理员安装一文件系统“proc”，用户用他们的 PID 来标识进程，并把它们视为“proc”中的文件，内核根据进程的用户标识号和进程组标识号给予打开这些文件的许可权。用户能通过读该文件检查进程地址空间，他们也可以通过写该文件设置断点，系统调用 stat 可以返回关于该进程的各种统计结果。这个方法消除了 ptrace 的三个缺点：第一，由于调试进程在每次系统调用时能够比使用 ptrace 传送更多的数据，因而这个方法较快；第二，调试程序可以跟踪任意的进程，不必须是其子进程；最后，被跟踪的进程并不必须做允许跟踪的事先安排，即调试程序能够跟踪已存在的进程。然而，作为正规文件保护机构的一部分，只有超级用户才能对设置用户标识号为 root 的进程进行调试。

11.2　系统Ⅴ IPC

UNIX 系统Ⅴ IPC 程序包由三种机制组成：消息允许进程发送格式化的数据流到任意的进程；共享存储区允许进程间共享它们虚地址空间中的部分区域；信号量允许进程间同步地执行。由于三者是作为一个整体实现的，它们有以下共同的性质：

• 每种机制都包含一个表，其中的表项描述该机制的所有实例。

• 每个表项都含有一个数值关键字，它是它的用户选择的名字。

• 每种机制都含有一个“获取”系统调用，以创建一个新表项或检索一个已存在的表项。这个系统调用的参数包括关键字和若干标志，内核搜索适当的表以便找到一个由该关键字命令的表项。进程可以用关键字 IPC _ PRIVATE 调用“获取”系统调用，以保证返回一个未用的表项；如果给定关键字的表项尚不存在，它们还可以在标志字段里置位 IPC _ CREAT 位，以便产生一个新表项；如果对应于该关键字已有一表项存在，它们还可以用置位 IPC _ EXCL 和 IPC _ CREAT 标志来迫使通知错误。“获取”系统调用返回一个内核选择

的可用于IPC其他系统调用的描述符，因此它类似于文件系统调用creat和open。

• 对每种IPC机制，内核使用下列公式从描述符找到数据结构表的索引值：

索引值 = 描述符　mod（表中表项数目）

例如，如果消息结构表含有100个表项，表项1的描述符可以是1、101、201等等。当一个进程删除一个表项时，内核将与该表项相关联的描述符增大，增加的值为该表中表项的数目。这个增加后的值就成为该表项再次被“获取”系统调用分配使用的新描述符。这时，试图以老描述符来存取这个表项的进程就会失败。用前面的例子来说，如果与消息表中表项1相关联的描述符是201，当它被删除时内核赋给该表项一个新描述符301，试图存取描述符201的进程会收到错误指示，因为201已不再有效。当然内核最终会再循环使用原来的描述符，但这大概要经过很长时间。

• 每个IPC表项有一个许可权结构，该结构含有创建该表项的进程的用户ID和用户组ID，由“控制”系统调用（见下面）设置的用户ID和用户组ID，以及与文件存取权限相类似的为用户、同组用户及其他用户的读-写-执行的许可权。

• 每个表项含有若干其他状态信息，如最后修改该表项（发送消息、接收消息、附接共享存储区等）的进程的进程ID，以及最后存取或修改的时间。

• 每种机制含有一个“控制”系统调用，用来查询某表项的状态、设置状态信息或从系统中删除某表项。当进程查询某表项的状态时，内核在证实该进程有读许可权之后，将该表项的数据拷贝到用户地址中去。类似地，若进程欲设置某表项的参数，内核先要证实该进程的用户ID，它或者与该表项的用户ID或创建用户ID相符；或者该进程是由超级用户运行的。这里，仅有写许可权是不足以设置参数的。然后，内核将用户数据拷贝到该表项中去——设置用户ID、用户组ID、许可权方式以及与该机制的类型有关的其他字段。内核并不修改创建用户ID和用户组ID，因此创建某表项的用户仍保留对它的控制权。最后，用户能够删除一表项，如果他是超级用户，或者他的进程ID与该表项结构中的任意一个标识字段相符合。这时，内核增大描述符的编号，使得为分配该表项的下一个实例返回的是一个不同的描述符。这样，如前所述，倘若一进程试图以一个老的描述符来存取该表项，则系统调用会以失败返回。

11.2.1　消息

有四个系统调用用于消息：系统调用msgget返回（很可能是创建）一个消息描述符，该描述符指定一个消息队列以便用于其他系统调用；系统调用msgctl可设置和返回与一个消息描述符相关联的参数的选择项，以及用于删除消息描述符的选择项；系统调用msgsnd发送一消息；系统调用msgrcv接收一消息。

系统调用msgget的语法是：

```
msgqid = msgget (key, flag);
```

其中msgid是该系统调用返回的描述符；key和flag具有前面描述的一般的“获取”系统调用的语义。内核将消息存储在每个描述符的链接表（队列）中，并使用msgid作为进入消息队列头标数组的索引。除了前面提到的一般的IPC许可权以外，队列结构包含以下字段：

- 指向链接表中第一个及最后一个消息的指针。
- 链接表中消息的数目及数据字节总数。
- 链接表中所允许的最大数据字节数。
- 最后一次发送和接收消息的进程 ID。
- 最后一次 msgsnd，msgrcv 和 msgctl 操作的时间（以下称为时间戳）。

当用户调用 msgget 来创建一个新的描述符时，内核搜索消息队列头标数组以确定是否已有一个队列以给定的关键字存在了。倘若对所规定的关键字尚无表项存在，内核分配一个新的队列结构，将它初始化并返回给用户一描述符，否则的话，它检查许可权并返回。

进程使用系统调用 msgsnd 发送一消息：

```
msgsnd (msgqid, msg, count, flag);
```

其中 msgid 通常是由 msgget 返回的一个消息队列的描述符；msg 指向由消息类型及一字符数组组成的结构的指针，其中消息类型是由用户选择的一个整数值；count 给出数据数组的大小；flag 规定当内核用尽内部缓冲空间时应执行的动作。

内核首先检查（图 11-4）发送进程是否对该消息描述符有写许可权；消息长度不超过系统规定的限制；该消息队列尚未包含太多的字节以及消息的类型为一正整数。倘若所有的

```
算法 msgsnd  /* 发送消息 */
输入  (1) 消息队列描述符
      (2) 消息结构地址
      (3) 消息的大小
      (4) 标志位
输出：所发送的字节数
{
    检验描述符、许可权的合法性；
    while（缺少足够大的空间来存贮消息）
    {
      if（标志位规定不要等待）
          return;
      sleep（有足够大的空间的事件）；
    }
    取消息头部；
    从用户空间将消息正文读到内核；
    调整数据结构：消息头部送入队列，
                  消息头部指向数据，
                  修改计数值、时间戳、进程标识号；
    从队列中唤醒正在等待读消息的全部进程；
}
```

图 11-4　系统调用 msgsnd 的算法

检测都成功了，内核从一个消息映射图（回忆 9.1 节）为该消息分配空间，并从用户空间拷贝数据；内核分配一消息头部，将它链入该消息队列的消息头部链接表的末尾；它在消息头部中记录消息的类型和大小；设置消息头部以使它指向消息数据区，并修改消息队列头标中

的各种统计字段（队列中消息数和字节数、时间戳以及发送者的进程 ID）。然后，内核唤醒正在睡眠等待着在该队列上到来消息的进程。如果队列中的字节数超过了该队列的上限，则进程进入睡眠直到其他消息从该队列中移去。然而，如果进程被规定为不等待方式（用标志 IPC _ NOWAIT），则它立即以错误指示返回。图 11-5 示出了一个队列上的消息、队列头标、消息头部的链接表以及从消息头部到数据区的指针。

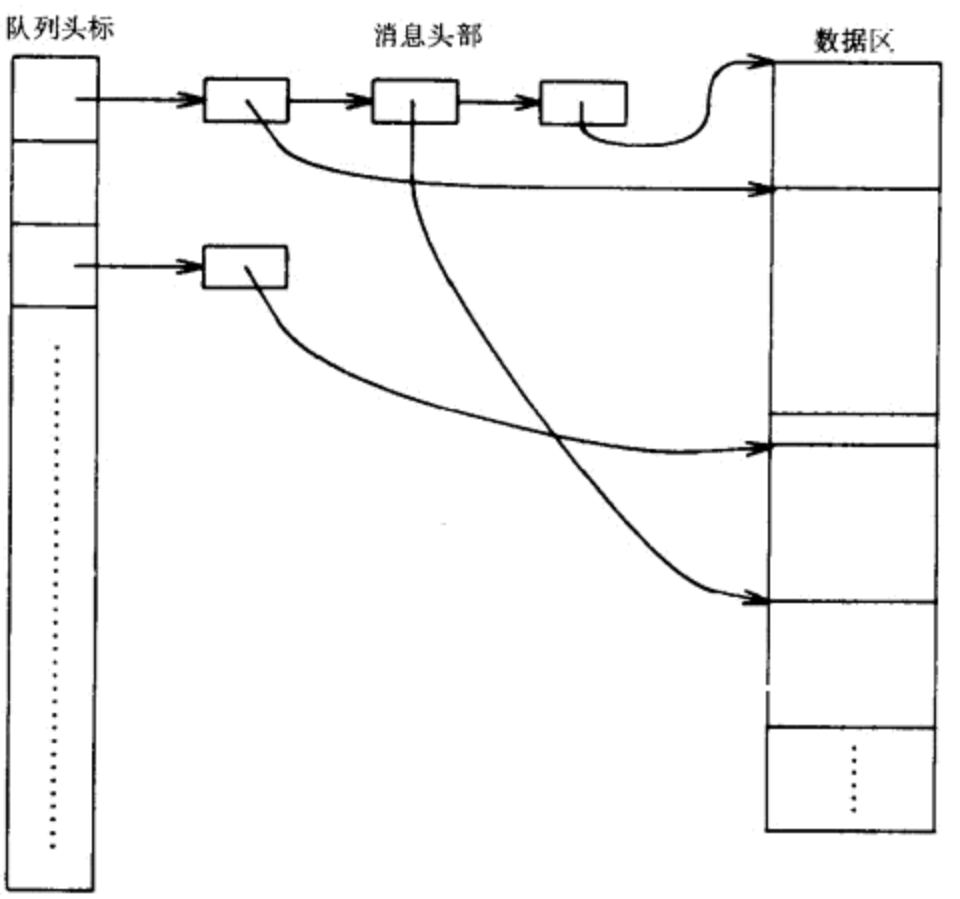


图 11-5　消息的数据结构

下面让我们来考察一下图 11-6 的程序。一个进程以 msgget 为关键字为 MSGKEY 的消息获取一描述符；虽然它只使用消息中的第一个整数，它仍建立一个长度为 256 字节的消息；将它的进程 ID 拷贝到消息正文中去；又将消息类型值赋为 1；然后调用 msgsnd 发送这一消息。以后我们还要回过来讨论这个例子。

进程接收消息时用

```
count = msgrcv (id, msg, maxcount, type, flag);
```

其中 id 是消息描述符；msg 是用来存放欲接收消息的用户数据结构的地址；maxcount 是 msg 中数据数组的大小；type 规定用户想要读的消息的类型，而 flag 规定倘若该队列上无消息内核应当做什么事。返回值 count 是返回给用户的字节数。

像前面一样，内核检查（图11-7）用户对该消息队列有必要的存取权限。如果所请求的消息类型为 0，内核寻找链接表上第一个消息，若它的大小小于或等于用户所请求的大小，则内核将消息数据拷贝到用户的数据结构中去，并适当地调整它的内部结构：减少队列中消息数和队列中数据字节数；设置接收时间和接收进程 ID；调整链接表以及释放原存储消息数据的内核空间。这时，如果有进程由于该表中无空闲空间而正等待发送消息，内核要唤醒

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

#define MSGKEY    75

struct msgform {
    long    mtype;
    char    mtext[256];
};

main()
{
    struct msgform msg;
    int msgid, pid, *pint;

    msgid=msgget(MSGKEY, 0777);

    pid=getpid();
    pint=(int *) msg.mtext;
    *pint=pid;    /* 将 pid 拷贝到消息正文中去 */
    msg.mtype=1;

    msgsnd(msgid, &msg, sizeof(int), 0);
    msgrcv(msgid, &msg, 256, pid, 0);    /* pid 用作消息的类型 */
    printf("client: receive from pid %d\n", *pint);
}
```

图 11-6　顾客进程

它们。如果消息比用户规定的 maxcount 大，则内核让消息继续留在队列上，对系统调用返回错误。然而如果用户忽略大小的限制（在 flag 中置位 MSG_NOERROR），内核截断该消息，返回所请求的字节数，并从该表中删除整个消息。

一个进程能够适当地设置 type 参数来接收特定类型的消息。如果该参数为正整数，内核返回的是所给定类型的第一个消息。如果是负整数，内核找到队列中小于或等于 type 的绝对值的所有消息中最低的类型，并返回该类型的第一个消息。例如，倘若一队列包含三个类型分别为 3、1 和 2 的消息，而用户请求一个类型为 -2 的消息，则内核返回的是类型 1 的消息。在所有的各种情况下，当队列中没有消息能满足接收请求时，内核让进程进入睡眠，除非该进程曾经置位 flag 中的 IPC_NOWAIT 位，它规定立即返回。

下面来考察图 11-6 和图 11-8 的程序，图 11-8 的程序示出了一个为顾客进程提供一般服务的服务者结构。举例来说，服务者可以从顾客进程接收从数据库提供信息的请求，这样，服务者进程就成为该数据库的单一存取点，便于进行一致性和安全性控制。服务者在系统调用 msgget 中用置位 IPC_CREAT 标志来创建一个消息结构，并接收所有类型 1 ——从顾客进程来的请求——的消息，然后它读消息正文，找到顾客进程的进程 ID，并将返回的消息类型置为顾客的进程 ID。在本例中，它在消息正文中把它自己的进程 ID 送回给顾客进程，而顾客进程只接收消息类型等于它的进程 ID 的消息。这样，服务者进程只接收顾客进程送给它的消息，而顾客进程只接收服务者进程送给它们的消息。上述进程通过协作在一个消息

队列上建立了多个通道。

```
算法　msgrcv　/* 接收消息 */
输入：(1) 消息描述符
      (2) 为存放到来的消息的数据数组地址
      (3) 数据数组大小
      (4) 所请求的消息类型
      (5) 标志位
输出：返回的消息中的字节数
{
    检验许可权；
  loop:
    检验消息描述符的合法性；
    /* 寻找返回给用户的消息 */
    if (所请求的消息类型 = = 0)
        考虑队列中的第一个消息；
    else if (所请求的消息类型>0)
        考虑队列中给定类型的第一个消息；
    else  /* 所请求的消息类型<0 */
        考虑队列中小于或等于所请求类型的绝对值的所有消息中最低类型
            的第一个消息；
    if (存在一个这样的消息)
    {
      调整消息大小或若用户给出的大小太小返回错误；
      从内核拷贝消息类型、消息正文到用户空间；
      从队列中删除该消息；
      return;
    }
    /* 不存在这样的消息 */
    if (标志位规定不要睡眠)
        以错误返回；
    sleep (消息到达队列的事件)；
    goto loop;
}
```

图 11-7　接收消息的算法

消息被格式化为类型-数据对，而文件中数据是字节流。这个类型前辍允许进程，如果需要的话，选择一种特定类型的消息，这是文件系统中所不提供的特性。这样，进程能够从消息队列中提取特定类型的消息，其次序是这些消息到达的次序，而内核负责维护这一适当的次序。虽然以文件系统在用户级实现消息传递也是可能的，但系统Ⅴ IPC 的消息以更有效的方式为应用程序提供进程间的数据传送能力。

进程能够用系统调用 msgctl 查询一个消息描述符的状态、设置它的状态以及删除一个消息描述符，该系统调用的语法是：

```
msgctl (id, cmd, mstatbuf);
```

其中 id 用来识别该消息描述符；cmd 规定命令的类型；mstatbut 是含有控制参数或查询结果的用户数据结构的地址。这个系统调用的实现是简单的．附录中详细地规定了它的参数。

让我们再回到图 11-8 中的服务者的例子。如果进程捕俘到软中断信号时，则调用函数 cleanup 来从系统中删除该消息队列，如果它捕俘不到软中断信号或者它接收了一个 SIGKILL 软中断信号（这个信号是不能捕俘的），则该消息队列即使没有进程引用它也会继续保留在系统中，而且后来试图以该关键字创建（互斥地）一个新消息队列的尝试都会失败，直到它被删除为止。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGKEY     75
struct msgform
{
     long mtype;
     char mtext [256];
} msg;
int msgid;

main ()
{
     int i , pid, *pint;
     extern cleanup ();

     for (i=0; i<20; i++)
          signal (i, cleanup);
     msgid=msgget (MSGKEY, 0777|IPC_CREAT);

     for (;;)
     {
          msgrcv (msgid, &msg, 256, 1, 0);
          pint= (int *) msg.mtext;
          pid= *pint;
          printf ("server: receive from pid %d \n", pid);
          msg.mtype=pid;
          *pint=getpid ();
          msgsnd (msgid, &msg, sizeof (int), 0);
     }
}

cleanup ()
{
     msgctl (msgid, IPC_RMID, 0);
     exit ();
}
```

图 11-8　服务者进程

11.2.2　共享存储区

进程能够通过共享它们虚地址空间的若干部分，然后对存储在共享存储区中的数据进行读和写来直接地彼此通信。操纵共享存储区的系统调用类似于对消息的系统调用：系统调用shmget建立一个新的共享存储区或返回已存在的一个共享存储区；系统调用shmat从逻辑上将一个共享存储区附接到一进程的虚地址空间上；系统调用shmdt从一进程的虚地址空间断接一共享存储区；系统调用shmctl对与共享存储区相关联的各种参数进行操纵。进程使用与它们读写普通存储器一样的机器指令来读写共享存储区，所以，在附接了共享存储区以后，该共享存储区就变成了进程虚地址空间的一部分，进程就能以存取其他虚地址一样的方法存取它，这里不需要通过系统调用来存取共享存储区中的数据。

系统调用shmget的语法是：

```
shmid = shmget (key, size, flag);
```

其中size是存储区中的字节数。内核搜索共享存储区表中具有给定关键字的存储区，倘若它发现了一个表项，且许可权方式是可接受的，它返回该表项的描述符；倘若它未发现一个表项，而且用户设置了IPC_CREAT标志以创建一个新存储区时，内核在证实了参数size是在系统范围内的最大和最小值之间以后，用算法allocreg（6.5.2节）分配一个区数据结构，内核将许可权方式、大小和指向区表项的指针保存在共享存储区表中（图11-9），并在其中设置一个标志以指示尚无存储空间与该区相关联。只当一个进程将该区附接到自己的地址空间时，内核才为该区分配存储空间（页表等）。内核还要在该存储区表项中设置一个标志，

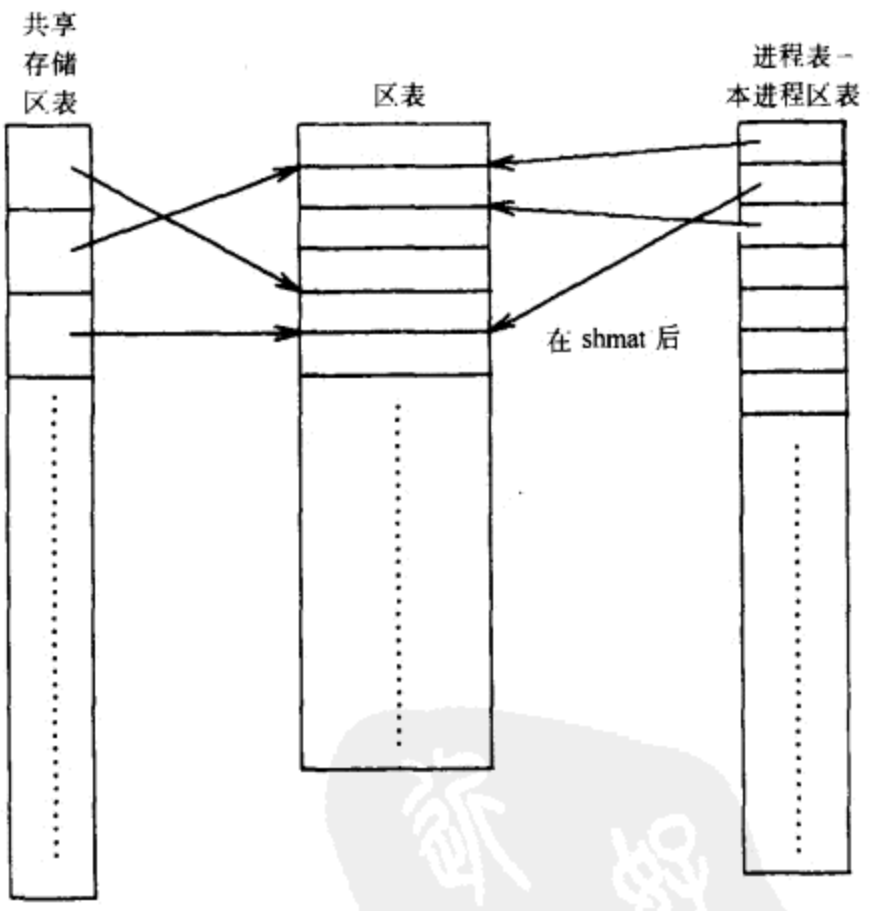


图11-9　共享存储区的数据结构

指示当最后一个附接到它的进程退出时该区不应被释放。这样，即使已无进程将某共享存储区包括在它们的地址空间了，该区中的数据仍原封不动地保存着。

进程使用系统调用 shmat 将一共享存储区附接到它的虚地址空间去：

```
virtaddr = shmat (id, addr, flags);
```

其中 id 是由先前的系统调用 shmget 返回的、用来识别某共享存储区；addr 是用户想要使共享存储区附接的虚地址；flags 规定对此区是否是只读的，以及内核是否应对用户规定的地址做舍入操作。返回值 viraddr 是内核将该共享存储区附接的虚地址，这里，它不必须是进程请求的虚地址值。

当执行系统调用 shmat 时，内核首先证实该进程有存取该区的许可权（图 11-10），然后它检查用户规定的地址：如果是 0，则内核选择一个合适的虚地址，由于共享存储区决不能与进程虚地址空间中的其他区相重叠，所以该地址必须明智地选择，使进程的其他区不至于增长到这个共享存储区中来。举例来说，进程可以通过系统调用 brk 增加它的数据区的大小，而新数据区是虚拟地与先前的数据区相邻接的，所以，内核不应将一共享存储区附接到邻近数据区的地方去。类似地，它也不应当将共享存储区放到邻近栈顶的地方，这样栈就不会增长到它中间来。如果栈是向高地址方向增长的，则附接共享存储区最好的地方，比如说，就是紧挨着栈区开始位置前的地方。

```
算法 shmat    /* 附接共享存储区 */
输入：(1) 共享存储区描述符
      (2) 欲附接的虚地址
      (3) 标志位
输出：存储区所附接的虚地址
{
    检验描述符、许可权的有效性；
    if  (用户规定了虚地址)
    {
        根据标志位所规定的对虚地址进行舍入；
        检验虚地址、区大小的合法性；
    }
    else  /* 用户要求内核找到合适的虚地址 */
        内核寻找虚地址：若找不到则记录错误；
    将存储区附接到进程地址空间 (算法 attachreg)；
    if  (该区第一次被附接)
        为该区分配页表、存储空间 (算法 growreg)；
    return (所附接的虚地址)；
}
```

图 11-10　附接共享存储区的算法

内核在检查了该共享存储区对进程的地址空间是恰当的以后，使用算法 attachreg 附接该区，如果调用进程是第一个附接该区的进程，则内核使用算法 growreg 分配必要的表，调整共享存储区表中“最后附接时间”字段，并将所附接的共享存储区的虚地址返回。

进程从其虚地址空间断接一个共享存储区使用系统调用：

```
shmdt (addr);
```

其中 addr 是先前的系统调用 shmat 所返回的虚地址。虽然以一个标识符作为参数传递似乎更符合逻辑，但这里使用的是共享存储区的虚地址，为的是一个进程可以对附接到它的地址空间的一个共享存储区的若干个实例加以区分，另外也因为这个标识符可能已经被删除了。内核搜索附接在所指示的虚地址的进程的区，并用算法 detachreg（6.5.7 节）将它从进程地

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/shm.h>
#define SHMKEY    75
#define    K    1024
int shmid;

main ()
{
    int i, * pint;
    char * addr1, * addr2;
    extern char * shmat ();
    extern cleanup ();

    for (i=0; i<20; i++)
        signal (i, cleanup);
    shmid=shmget (SHMKEY, 128 * K, 0777|IPC_CREAT);
    addr1=shmat (shmid, 0, 0);
    addr2=shmat (shmid, 0, 0);
    printf ("addr1 0x%x addr2 0x%x\n", addr1, addr2);
    pint= (int * ) addr1;

    for (i=0; i<256; i++)
        * pint++=i;
    pint= (int * ) addr1;
    * pint=256;

    pint= (int * ) addr2;
    for (i=0; i<256; i++)
        printf ("index %d\tvalue %d\n", i, * pint++);

pause ();

}

cleanup ()
{
    shmctl (shmid, IPC_RMID, 0);
    exit ();
}
```

图 11-11　共享存储区两次附接到进程

址空间中断接出去。由于区表没有指向共享存储区表的反向指针，因而内核搜索共享存储区表，以便找到指向该区的表项，然后调整该区最后被断接的时间字段。

现在让我们来考察一下图 11-11 中的程序：一进程建立一个 128K 字节的共享存储区，并在它的地址空间中不同的虚地址两次附接该区，它将数据写入“第一个”共享存储区，又从“第二个”共享存储区中读数据。图 11-12 示出了附接了同一共享存储区的另一进程（为了表明每个进程可以附接一个共享存储区中不同的数量，它只取其中的 64K 字节），它等待着直到第一个进程在共享存储区中第一个字中写入一个非零值后读出该存储区；这时第一个进程暂停着以使第二个进程有机会去执行上述操作；当第一个进程捕俘到一个软中断信号时，它将该共享存储区删除。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY 75
#define K 1024
int shmid;

main ()
{
    int i, * pint;
    char * addr;
    extern char * shmat ();

    shmid=shmget (SHMKEY, 64 * K, 0777);

    addr=shmat (shmid, 0, 0);
    pint= (int * ) addr;

    while ( * pint= =0)
        ;
    for (i=0; i<256; i++)
        printf ("%d\n", * pint++);
}
```

图 11-12　进程间共享存储区

进程使用系统调用 shmctl 来查询某共享存储区的状态和设置有关的参数：

```
shmctl (id, cmd, shmstatbuf);
```

其中 id 标识共享存储区表项；cmd 规定操作的类型，而 shmstatbuf 是一个用户级数据结构的地址，当查询或设置共享存储区状态时，这个数据结构中含有该表项的状态信息。类似于对消息的实现，内核把 cmd 作为查询状态、改变所有者和许可权的命令。当删除一共享存储区时，内核释放该表项，并观察区表项；如果此时已没有进程将它附接在自己的虚地址空间，那么，内核用算法 freereg（6.5.6 节）释放该区表项和它的所有资源；如果该区仍然附接在某些进程上（即它的引用数大于零），那么，内核仅仅清除指示当最后一个进程断接该区时不释放该区的标志位，这时，正在使用该共享存储区的进程可以继续使用它，但没有新

的进程能够附接它了。当所有的进程都断接了该区时，内核释放它。这与文件系统中一个进程打开了一个文件，但是在拆除与该文件的联结（unlink）之后仍可继续存取它是十分相似的。

11.2.3 信号量

信号量系统调用允许若干进程在一组信号量上原子地做一系列操作以便同步地执行。在实现信号量以前，如果一个进程要封锁一个资源，它用系统调用 creat 来创建一个锁文件。这样，倘若该锁文件已经存在，creat 会失败，因而该进程会认为有另一个进程已将该资源封锁了。这个方法的主要缺点是进程不知道什么时候再试，并且当系统崩溃或再自举时，由于疏忽锁文件可能仍被遗留着。

Dijkstra 发表过的 Dekker 算法描述了信号量的一种实现方法——整数值对象和对它们定义的两种原子操作（atomic operation）：P 和 V（见［Dijkstra 68］）。P 操作减少一个信号量的值，如果它的值大于零；而 V 操作增加它的值。由于这些操作都是原子的，因而任何时候在一个信号量上最多有一个 P 或 V 操作能成功地执行。系统 V 的信号量系统调用是 Dijkstra P V 操作的普遍化，其中若干操作可以同时完成，而且增或减操作的值都可以大于 1。内核原子地执行这些操作，这就是说，在所有这些操作完成之前，没有其他进程能调整这些信号量的值。如果内核不能完成所有的操作，则它不做其中任何一个操作，进程睡眠直到它能进行所有的操作，这一点我们下面将要解释。

系统 V 中一个信号量由以下几部分组成：

- 信号量的值。
- 最后的一个操纵信号量的进程的进程 ID。
- 等待着信号量的值增加的进程数。
- 等待着信号量的值等于零的进程数。

信号量系统调用有：用于产生一组信号量以及得以存取它们的系统调用 semget；用于在一组信号量上进行各种控制操作的 semctl；以及对信号量的值进行操纵的 semop。

系统调用 semget 产生一个信号量数组：

```
id = semget (key, count, flag);
```

其中 key、count 和 flag 类似于用于建立消息和共享存储区的这些参数。内核分配一个表项，该表项指向有 count 个元素的信号量结构的数组（图 11-13）。该表项还规定数组中信号量的个数、最后一次系统调用 semop 的时间以及最后一次 semctl 的时间。例如，图 11-14 中的系统调用 semget 产生一个含有两个元素的信号量。

进程使用系统调用 semop 对信号量进行操纵：

```
oldval = semop (id, oplist, count);
```

其中 id 是由系统调用 semget 返回的描述符；oplist 是指向信号量操作数组的指针；count 是该数组的大小。返回值 oldval 是在那一组操作中最后被操作的信号量在操作完成前的值。oplist 中每个元素的形式是：

- 信号量序号，它标识要操作的信号量数组元素。

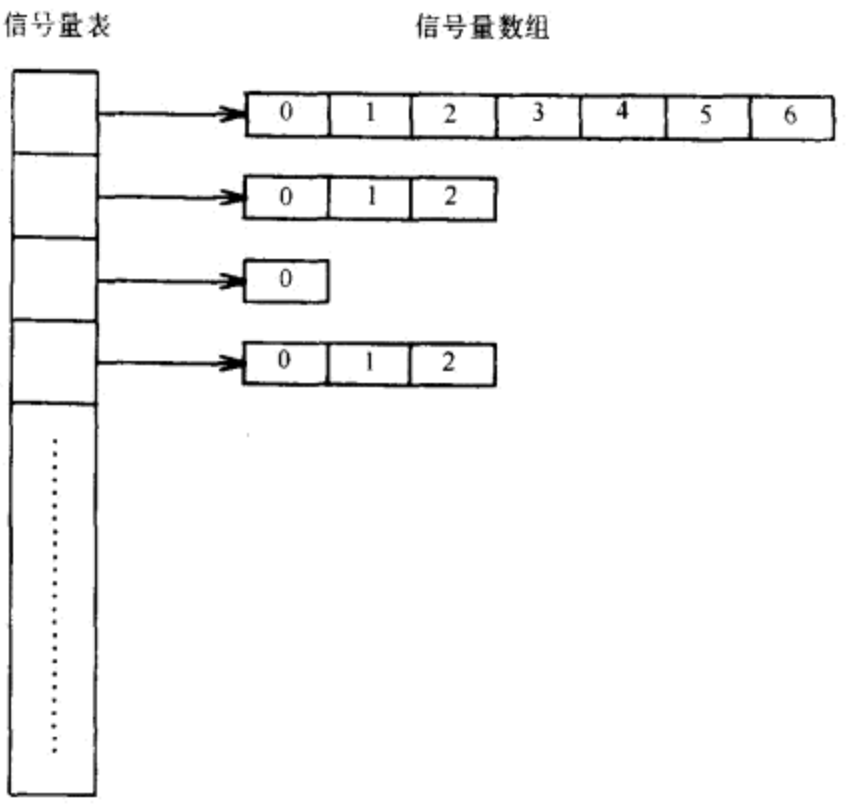


图 11-13　信号量数据结构

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

#define SEMKEY    75
int semid;
unsigned int count;
/* 在文件 sys/sem.h 中定义了 sembuf
 * struct sembuf {
 *      unsigned shortsem_num;
 *      short sem_op;
 *      short sem_flg;
}; */
struct sembuf psembuf, vsembuf;          /* P 和 V 操作 */

main (argc, argv)
    int argc;
    char * argv [];
{
    int i, first, second;
    short initarray [2], outarray [2];
    extern cleanup ();

    if (argc==1)
    {
        for (i=0; i<20; i++)
```

图 11-14　加锁和解锁操作

```
        signal (i, cleanup);
        semid=semget (SEMKEY, 2, 0777|IPC_CREAT);
        initarray [0] =initarray [1] =1;
        semctl (semid, 2, SETALL, initarray);
        semctl (semid, 2, GETALL, outarray);
        printf ("sem init vals %d %d\n", outarray [0], outarray [1]);
        pause ();       /* 睡眠直到被一软中断信号唤醒 */
    }
    else if (argv [1] [0] == 'a')
    {
        first=0;
        second=1;
    }
    else
    {
        first=1;
        second=0;
    }
    semid=semget (SEMKEY; 2, 0777);
    psembuf.sem_op=-1;
    psembuf.sem_flg=SEM_UNDO;
    vsembuf.sem_op=1;
    vsembuf.sem_flg=SEM_UNDO;

    for (count=0;; count++)
    {
        psembuf.sem_num=first;
        semop (semid, &psembuf, 1);
        psembuf.sem_num=second;
        semop (semid, &psembuf, 1);
        printf ("proc %d count %d\n", getpid (), count);
        vsembuf.sem_num=second;
        semop (semid, &vsembuf, 1);
        vsembuf.sem_num=first;
        semop (semid, &vsembuf, 1);
    }
}

cleanup ()
{
    semctl (semid, 2, IPC_RMID, 0);
    exit ();
}
```

图 11-14 （续）

- 欲进行的操作。
- 标志。

内核从用户地址空间读入信号量操作数组 oplist，证实信号量序号是合法的以及进程有读或改变这些信号量的必要的许可权（图 11-15）。倘若不具备这个许可权，系统调用失败返回。如果当内核执行这个操作表时必须睡眠，那么它将它已经操作了的信号量恢复到系统调用开始时的值，然后睡眠，直到它等待的事件出现时它才重新开始执行该系统调用。由于内核将信号量操作保存在一个全局数组变量中，因此如果它必须重新开始执行这个系统调用，那么它必须再次从用户空间读入这个数组。这样，操作是原子地完成的——或者一次全部完成或者完全不执行。

内核根据欲进行的操作值改变信号量的值。如果操作值为正整数，则它将该信号量增加这个值，并唤醒所有的等待此信号量值增加的进程。如果信号量操作值为零，则内核检查信号量的值：若为零，它继续执行操作数组中的其他操作；否则，它增加等待信号量的值等于零的睡眠进程数，并进入睡眠。如果信号量操作是负整数且其绝对值小于或等于信号量值，则内核将操作值（一个负整数）加到信号量值之上；如果信号量值小于信号量操作值的绝对值，则内核让进程睡眠在等待信号量值增加的事件上。无论在信号量操作中间的什么时候进程睡眠，它都睡眠在一个可中断的优先级上，因此，当它接收到一个软中断信号时，它都被唤醒。

```
算法 semop  /* 信号量操作 */
输入：(1) 信号量描述符
      (2) 信号量操作数组
      (3) 数组中元素数
输出：最后被操作的信号量的初始值
{
    检验信号量描述符的合法性；
start：从用户空间将信号量操作数组读入内核空间；
      检验所有信号量操作的许可权；
    for（数组中每个信号量操作）
    {
        if（信号量操作值为正）
        {
            将“操作值”加到信号量值上；
            if  （该信号量操作 UNDO 标志置位）
                修改进程的 undo 结构；
            唤醒所有睡眠着等待信号量增加的进程；
        }
        else if  （信号量操作值为负）
        {
            if（“操作”+信号量值>=0）
            {
                将“操作值”加到信号量值上；
                if  （UNDO 标志是置位的）
                    修改进程 undo 结构；
```

图 11-15　信号量操作算法

```
            if (信号量值为 0)
                唤醒所有等待信号量值变为 0 的事件的进程;
            continue;
            }
            废弃这次系统调用已完成的所有信号量操作 (以前的迭代);
            if (标志规定不得睡眠)
                错误返回;
            sleep (信号量值增加的事件);
            goto start;      /* 从头开始循环 */
        }
        else  /* 信号量操作值为 0 */
        {
            if (信号量值非 0)
            {
                废弃这次系统调用已完成的所有信号量操作;
                if (标志规定不得睡眠)
                    错误返回;
                sleep (信号量值为 0 的事件);
                goto start;   /* 再开始循环 */
            }
        }
    }   /* for 循环在此处结束 */
    /* 信号量操作全部成功 */
    修改时间戳、进程 ID;
    返回最后被操作的信号量在系统调用成功前的值;
}
```

图 11-15 （续）

下面让我们考虑图 11-14 中的程序，并假设用户按以下顺序对它（a.out）执行三遍：

```
a.out &
a.out a &
a.out b &
```

当不带任何参数运行时，进程产生一个有两个元素的信号量集，并将它们的值初始化为 1。然后它暂停并睡眠直到被一软中断信号唤醒，这时它在子程序 cleanup 中删除该信号量。当以参数“a”执行这个程序时，进程（称为 A）在循环中执行四次分离的信号量操作：它将信号量 0 的值减 1；信号量 1 的值减 1；执行打印语句；然后增加信号量 1 和 0 的值。一个进程如果试图减少一个值已经是 0 的信号量值时，它要进入睡眠，因此，这个信号量被认为是上了锁的。由于信号量被初始化为 1，而且没有其他进程正在使用这些信号量，所以，进程 A 永远不会睡眠，信号量将在 1 和 0 之间振荡。当以参数“b”执行这个程序时，进程（称为 B）以与进程 A 相反的次序对信号量 0 和 1 减 1。当进程 A 和 B 同时运行时，以下的情形可能出现：进程 A 已对信号量 0 上了锁且要对信号量 1 加锁，然而，进程 B 已对信号量 1 上了锁又要对信号量 0 加锁。两个进程都必须睡眠，都不能继续执行。它们陷入了死锁，仅当接收到软中断信号时才能退出。

为了避免这样的问题出现，进程可以同时执行多个信号量操作。在前述的例子中使用下面的结构就会得到所期望的效果。

```
struct sembuf psembuf [2];

psembuf [0] .sem _ num=0;
psembuf [1] .sem _ num=1;
psembuf [0] .sem _ op=-1;
psembuf [1] .sem _ op=-1;
semop (semid, psembuf, 2);
```

psembuf 是同时减小信号量 0 和 1 的信号量操作数组。如果任一操作不能成功地执行，进程进入睡眠直到两个操作都能成功。比如，如果信号量 0 的值为 1 而信号量 1 的值为 0，内核将原封不动地保持这些值，直到它能够减少两个值。

进程能在系统调用 semop 中置位 IPC _ NOWAIT 标志，这时，如果内核遇到一种必须使进程进入睡眠的情况——它必须等待信号量值超过一特定的值或等于零——则内核以错误指示从系统调用返回。这样就有可能实现一种条件信号量，使得在进程不能执行原子操作时并不进入睡眠。

如果一进程执行了信号量操作——可能对某个资源上了锁，然后，在尚未复位信号量的值时就退出了，那么，危险的情况在这种条件下就出现了。这种情况的出现或者是由于程序员的错误所引起的，或者是由于接收到一个软中断信号造成进程的突然终结。让我们还是返回到图 11-14 的例，如果在将信号量的值减少以后进程接收到 SIGKILL 软中断信号，由于 SIGKILL 信号是不能捕俘的，它就没有机会去重新增加该信号量的值。因此，即便是对它加锁的进程已不复存在，其他进程会发现该信号量总是上了锁的。为了避免这样的问题出现，进程可以在系统调用 semop 中置位 SEM _ UNDO 标志，这样当它退出时，内核会废弃该进程已做的每一个信号量操作的效果。为了实现这一特性，内核保持一个表，系统中每一个进程在该表中有一表项，每个表项指向一组 undo 结构，而该进程使用的每一个信号量有一个这种结构（图 11-16）。每个 undo 结构又是一个三元组数组，每个数组元素由信号量 ID、该 ID 识别的信号量集合中的信号量序号以及一个调整值组成。

当进程第一次执行它的设置了 SEM _ UNDO 标志的系统调用 semop 时，内核动态地分配 undo 结构。当它以后执行设置了 SEM _ UNDO 标志的系统调用 semop 时，内核搜索该进程的 undo 结构以便找到一个有着与 semop 操作所规定的相同的信号量 ID 和序号的三元组：倘若它找到了一个，它从调整值中减去信号量操作值，这样，该 undo 结构中就含有该进程在这个信号量上所有的设置了 SEM _ UNDO 标志的信号量操作累计的逆效果；倘若这样的 undo 结构不存在，内核建立一个，并按信号量 ID 和序号对该结构的表排序。当调整值降为零时，内核删除该 undo 结构，当进程退出时，内核调用一个特殊子程序，该子程序搜索与该进程相关联的 undo 结构，并对所指示的信号量执行规定的操作。

让我们再次回到图 11-14，进程每一次减少信号量值时内核建立一个 undo 结构；进程每一次增加信号量值时内核删除一个undo结构，因为这时undo结构的调整值是零了。图 11－17 示出了当以参数“a”运行该程序时的 undo 结构序列。在第一次操作以后，该进程对 semid 有了一个信号量序号为 0、调整值为 1 的三元组；第二次操作以后，它有了第二个

信号量序号为 1、调整值为 1 的三元组。如果此时进程突然退出，内核会搜索这些三元组，并将每个信号量值加 1，把它们的值恢复为 1。而在正常情况下，在第三次操作时内核从信号量 1 的调整值中减少相当于该操作欲增加的信号量 1 的值，由于这时的调整值降为 0，它删除该三元组。在第四次操作以后，由于所有的调整值均为 0，进程不再有三元组了。

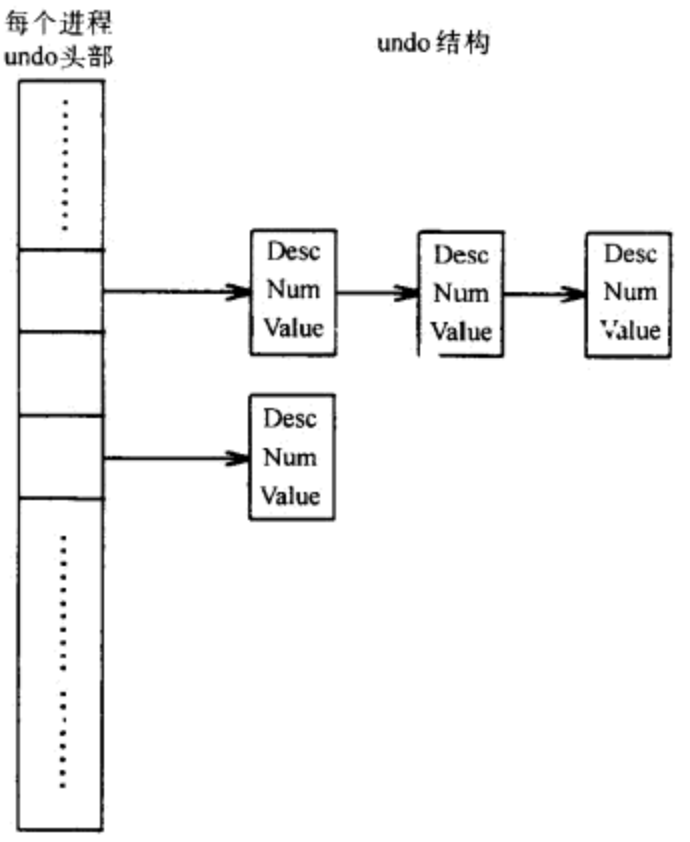


图 11-16　信号量的 undo 结构

信号量标识	semid
信号量号	0
调整值	1

a)

信号量标识	semid	semid
信号量号	0	1
调整值	1	1

b)

信号量标识	semid
信号量号	0
调整值	1

c)

空

d)

图 11-17　undo 结构序列

a）第一个操作后　b）第二个操作后　c）第三个操作后　d）第四个操作后

综上所述，对信号量的数组操作使进程避免死锁问题，然而这些操作复杂，而多数的应用并不完全需要这样的能力。所以，应当让需要使用多个信号量的应用在用户级处理死锁条件，而不应让内核包含这样复杂的系统调用。

系统调用 semctl 包含对信号量的许多控制操作：

```
semctl (id, number, cmd, arg);
```

其中 arg 是一联合结构：

```
union semunion {
    int val;
    struct semid_ds * semstat;      /* 其定义见附录 */
    unsigned short * array;
} arg;
```

内核根据命令 cmd 的值解释 arg，这一点与它解释 ioctl 命令的方法类似。对这些命令——查询或设置控制参数（许可权或其他）、设置一信号量集合中一个或所有的信号量值、读信号量值——执行所要求的动作，附录中给出每一个命令的详细说明。对于删除命令，IPC_RMID，内核找到该信号量有 undo 结构的进程并删去相应的三元组。然后重新初始化该信号量数据结构，唤醒所有正在睡眠等待某种信号量事件的进程，当这些进程继续执行时，它们会发现该信号量 ID 已不再有效，因而给调用者返回一个错误。

11.2.4　总的评价

文件系统与 IPC 机构间有某些共性："获取"系统调用类似于系统调用 creat 和 open；"控制"系统调用中的从系统中删除描述符的操作类似于系统调用 unlink，但是没有一个操作类似于文件系统的系统调用 close。内核没有记录哪个进程可以存取哪一个 IPC 机构，甚至某些进程从来未执行过"获取"系统调用，如果它们猜到正确的 ID 和取得适当的许可权，那么它们确实可以存取一个 IPC 机构；内核也不能自动地清除那些无用的 IPC 机制，因为它从来不知道什么时候这些 IPC 机制已不再需要了。因此流动不定的进程会留下些不需要且无用的结构，使系统混乱。虽然在进程死亡以后内核可以在 IPC 机构中保存状态信息和数据，但针对这些目的最好还是使用文件。

IPC 机制引入了一个新的名字空间——关键字——代替传统的、普遍使用的文件名。然而，很难将关键字的语义扩展到一个网络上，因为在不同的机器上它们可能描述不同的对象。简言之，IPC 机制是为单一机器环境而设计的。文件名则更适合于一个分布式的环境，这一点我们将在第 13 章中看到。使用关键字而不用文件名也意味着 IPC 是一个对于特殊目的应用较为有用的自封闭实体，但是缺乏诸如管道和文件那样内在的建造工具的能力。它们的多数功能能用其他系统设施来一模一样地实现，因此，从美学角度讲，它们不应当放在内核中。然而，它们为紧密合作的应用软件包提供了比标准文件系统更好的性能。

11.3　网络通信

历史上，想要和其他机器通信的程序，如电子邮件、远程文件传输和远程机器注册曾使

用特别的方法以建立连接和交换数据。例如，标准的电子邮件程序把用户邮件信息的正文保存在一特定的文件中，如用户“mjb”的保存在文件“/usr/mail/mjb”中。当某人送邮件到同一机器上的另一用户时，邮件管理程序 mail 将该邮件附加到收信人的文件上去，在此过程中使用锁文件和临时文件以保持一致性。当某人读邮件时，mail 程序打开这人的邮件文件，并读出消息。为了发送邮件给另一台机器上的用户，mail 程序必须最终地找到那台机器上相应的邮件文件。但是，由于它不可能直接地在那台机器上操纵文件，所以在那台机器上必须有一个进程充当本地邮件管理进程的代理，而本地进程需要跨越机器的边界与它远程的代理通信。本地进程常被称为远程服务者进程的顾客。

由于 UNIX 系统是通过系统调用 fork 来创建新进程的，因此服务者（server）进程必须在顾客（client）进程试图建立一个连接之前就已经存在了。如果仅当一个建立连接的请求从网络中送来时，远程的内核才创建一个新进程，就会造成与整个系统的设计不一致，因而，代之以某进程——通常是 int 进程——创建一个服务者进程，该进程始终读一个通信信道，直到它接收到一个服务请求，然后按某协议来完成连接的建立。顾客程序和服务者程序通常根据应用数据库中的信息来选择网络介质和协议，或者用硬编入程序的数据来做这些工作。

例如，uucp 程序允许跨越一个网络传输文件和在远程执行命令（见 [Nowitz 80]）。用户进程从一个数据库中查询地址和路由选择的信息（如电话号码），打开一自动拨号设备，将信息写或以系统调用 ioctl 送到打开的文件描述符上，并呼叫远程的机器。远程的机器可能有专门为 uucp 使用的特殊线路，该机的 init 进程派生 getty 进程——服务进程——来监视这些线路，并等待连接的通知。在硬件的连接建立以后，顾客进程以通常的注册规则进行注册，而 getty 执行在/etc/passwd 文件中规定的一个特殊的命令解释程序 uucico，这时顾客进程将命令序列写到远程机器，造成远程机器代表本地机器执行。

由于传递的消息中常常必须包含数据和控制两部分，网络通信向 UNIX 系统提出了一个问题。消息中的控制部分可能包含地址信息以规定一个消息的目的地，而地址信息又要根据网络的类型和所使用的协议来结构化。因此，进程需要知道正与之通信的网络的类型，这一点恰好违反 UNIX 上用户不必知道一个文件的类型——因为所有的设备看上去都象文件一样——的原则。结果是，传统的实现通信的方法都极大地依赖于系统调用 ioctl 来规定控制信息，但是对不同的网络类型所使用的是不统一的。这一点造成了令人遗憾的副作用：为一种网络设计的程序可能不能在其他网络上运行。

为了改进 UNIX 系统的网络接口，人们曾经做过相当多的努力。在系统 V 的最新版本中流的实现为支持网络提供了一种完美的机制，因为通过将协议模块压到一个打开的流上去能够把它们灵活地结合在一起，并且它们的使用在用户级是一致的。下一节中我们将简单地描述 BSD 对这个问题的解决方法——套接字（socket）。

11.4 套接字

前一节说明了不同机器上的进程是怎样通信的。然而，它们建立通信的方法可能是不同的，这依赖于协议和信道。此外，这些方法可能不允许一进程与同一机器上的其他进程通信，原因是它们采取的是存在一个服务进程，这个服务进程在系统调用 open 或 read 时睡眠

在驱动程序中。为了提供进程间通信的一般方法和允许使用复杂的协议，BSD 系统提供了一种称为套接字的机制（见［Berkeley 83］）。本节简要地描述套接字的某些用户级方面的问题。

内核结构由三个部分组成：套接字层、协议层和设备层（图11-18）。套接字层提供系统

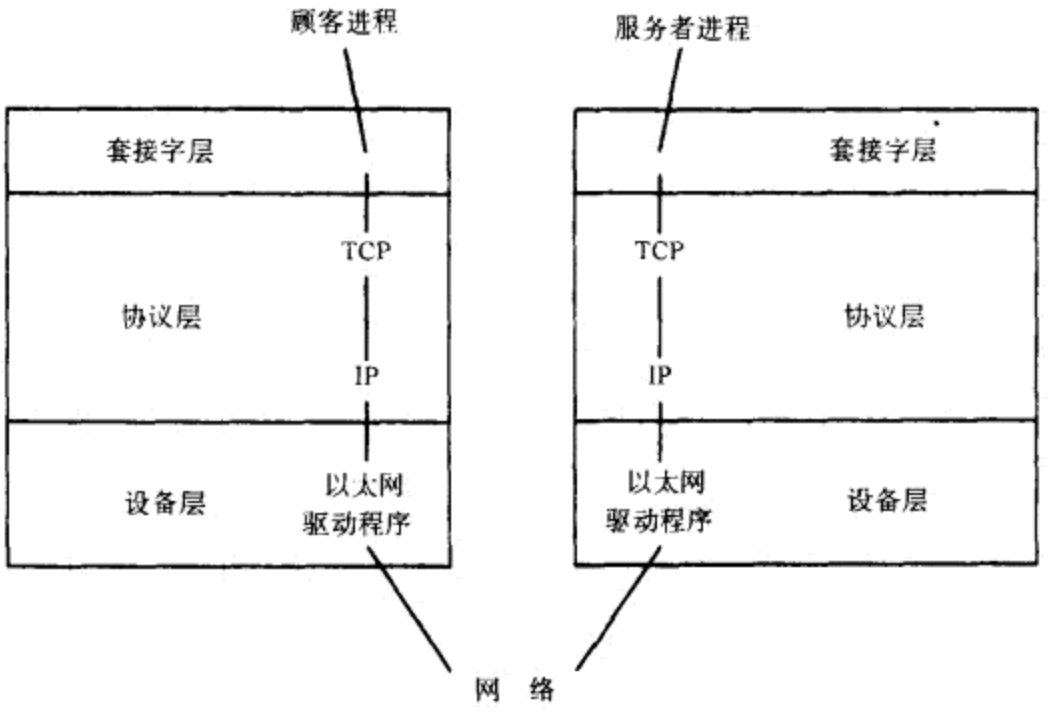


图 11-18　套接字模型

调用和低层次之间的接口；协议层包含通信使用的协议模块（图中的 TCP 和 IP）；设备层包含控制网络设备的设备驱动程序。当配置系统时，要规定协议和驱动程序的合法的组合，因此，这个方法不如压入流模块那样灵活。进程通信使用顾客-服务器模型：一个服务进程在作为一个双向通信通路一侧的端点的套接字上倾听；而顾客进程通过另一个套接字与服务进程通信，这个套接字是该通信路径的另一侧的端点，而且可能在另一台机器上。内核保存着内部的连接并为从顾客到服务器的数据确定路径。

具有共同通信性质——如命名法和协议地址格式——的套接字被划入域（domain）。BSD4.2 支持“UNIX 系统域”——它是为一台机器上的进程通信的，以及“互连网域”（internet domain）——它是为跨越一个使用 DARPA（美国国防部高级研究计划局）通信协议（见［Postel 80］和［Postel 81］）的网络上的进程通信的。每个套接字有其类型——虚电路（virtual circuit）类型（用伯克利术语为流）或数据报（datagram）类型。虚电路允许数据顺序地、可靠地传递。数据报不保证顺序地、可靠地或无重复地传递，但由于它不需要开销大的连接建立操作，它比虚电路开销小，因此，它对某些类型的通信是有用的。对每一种域和套接字类型的合法的组合，系统设置一种缺省协议，比如，在互连网域中传输控制协议（TCP）提供虚电路服务，用户数据报协议（UDP）提供数据报服务。

套接字机制包含几个系统调用。系统调用 socket 建立一个通信链路的端点：

```
sd = socket (format, type, protocol);
```

其中 format 规定通信域（UNIX 域或互连网域）；type 指示在套接字上通信的类型（虚电路或数据报）；protocol 指示控制通信的一个特定的协议；sd 是在其他系统调用中进程使用的

套接字描述符。系统调用 close 关闭套接字。

系统调用 bind 把一个名字与套接字描述符联系起来：

```
bind (sd, address, length);
```

sd 是套接字描述符；address 指向在系统调用 socket 中规定的通信域与协议专用的一个地址标识结构；length 是结构 address 的长度。由于在不同的域与协议中这个结构是变化的，所以如果没有 length 这个参数，内核会不知道该地址多长。比如，在 UNIX 系统域中地址是文件名。服务进程以系统调用 bind 把地址绑定到套接字，并“通告”它们的名字，使顾客进程能识别它们。

系统调用 connect 请求内核与一个已存在的套接字建立一个连接：

```
connect (sd, address, length);
```

这些参数的语义与 bind 的参数是相同的，只是这里的 address 是成为通信信道另一个端点的目的套接字地址。两个套接字必须使用相同的通信域和协议，这样内核才能正确地安排其间的通信链路。如果套接字的类型是数据报，系统调用 connect 只是通知内核在该套接字上随后的系统调用 send 使用的地址，而在此调用时刻并不建立连接。

当服务进程准备在一条虚电路上接受连接时，内核必须将到来的请求送入队列，直到服务进程能够为它们服务。系统调用 listen 规定最大队列长度：

```
listen (sd, qlength);
```

其中 sd 是套接字描述符；qlength 是待处理请求的最大数目。

系统调用 accept 接受外来的一次与服务进程的连接请求：

```
nsd=accept (sd, address, addrlen);
```

其中 sd 是套接字描述符；address 指向一用户数据结构，内核可将所连接的顾客的地址填入以返回给用户；addrlen 指示用户数据结构的大小。当从 accept 返回时，内核将指示顾客地址 address 所占的空间的数量重写到 addrlen 的内容中。accept 返回一个与套接字描述符 sd 不同的新的套接字描述符 nsd，服务进程可以在所通告的地址上继续倾听，同时在另一分开的通信信道上与顾客进程进行通信（图 11-19）。

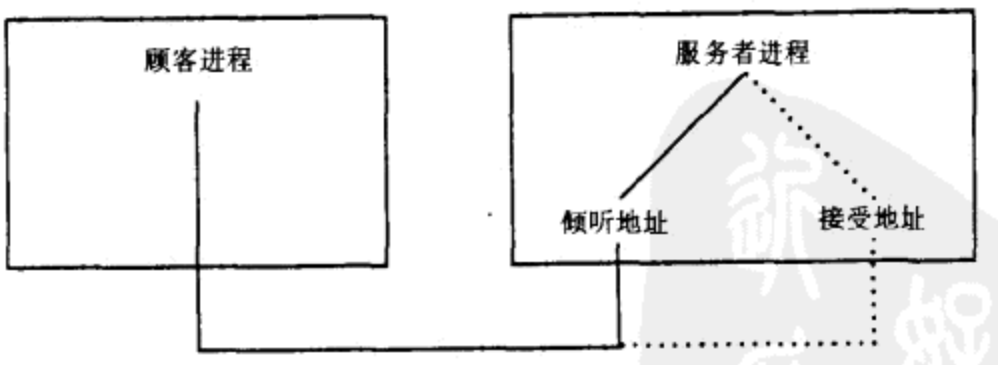


图 11-19　服务员接受一次调用

系统调用 send 和 recv 在一个已连接的套接字上发送和接收数据：

```
count = send (sd, msg, length, flags);
```

其中 sd 是套接字描述符；msg 是指向欲发送数据的指针；length 是它的长度；count 是实际发送的字节数；参数 flags 可置为值 SOF _ OOB 以发送“加速”数据。“加速”的含意是所发送的数据不属于通信进程间正常顺序的数据交换部分。例如，“远程注册”程序就可以发送一“加速”报文来模拟用户在终端上敲 delete 键。系统调用 recv 的语法是：

```
count = recv (sd, buf, length, flags);
```

其中 buf 是用来存放到来的数据的数据数组；length 是所期待的长度；count 是拷贝到用户程序的字节数；通过置位 flags 还能“偷看”一到来的报文、检查它的内容但不把它从队列中移去，或接收“加速”数据。这些系统调用的数据报形式——sendto 和 recvfrom 还有额外的地址参数。在流套接字上进程在连接建立后还能以系统调用 read 和 write 来代替 send 和 recv，这样，服务者只需要关心具体网络协议的协商和派生进程，这些进程就像使用正规文件一样地使用系统调用 read 和 write。

系统调用 shutdown 关闭一套接字连接：

```
shutdown (sd, mode)
```

其中 mode 指示是否发送方、接收方或双方都不再允许发送数据了，它通知它下面的协议关闭网络通信，但是套接字描述符仍原封不动。系统调用 close 释放套接字描述符。

系统调用 getsockname 用来获取由先前的一次系统调用 bind 绑定到一个套接字的名字：

```
getsockname (sd, name, length);
```

系统调用 getsockopt 和 setsockopt 根据套接字的通信域和协议获得或设置与套接字相关联的各种选择。

下面我们来考察图 11-20 的服务器程序。进程在“UNIX 系统域”中产生一个套接字，并把名字 sockname 绑定到它，然后它调用系统调用 listen 规定外来的报文的内部队列长度，并进入一个循环，等待外来的请求。系统调用 accept 进入睡眠直到它下面的协议层通知它有一个欲与此绑定名的套接字建立连接的请求来到，这时 accept 为到来的请求返回一个新的描述符。服务进程派生出一个进程以便与顾客进程通信，父进程和子进程各关闭相应的描述符，使它们相互不干扰彼此的通信业务。子进程继续与顾客进程对话，在本例中，它在从系统调用 read 返回后终止。而服务进程继续循环，在系统调用 accept 中等待着另一个连接请求。

图 11-21 中示出了与此服务进程相对应的顾客进程。顾客在与服务器同样的域中产生一个套接字，并发出一个与名字为 sockname 的套接字的连接请求，而该名字是由服务者绑定到那个套接字上去的。当 connect 返回时，顾客进程与一个服务进程间建立了一条虚电路。在本例中，该顾客进程写了一条消息后退出。

如果服务进程要为网络上的进程服务，它可以用以下方法规定套接字是在“互连网域”中：

```
#include <sys/types.h>
#include <sys/socket.h>

main ()
{
    int sd, ns;
    char buf [256];
    struct sockaddr sockaddr;
    int fromlen;

    sd=socket (AF_UNIX, SOCK_STREAM, 0);

    /* 绑定名字——不包括名字中的 NULL 字符 */
    bind (sd,"sockname", sizeof ("sockname") -1);
    listen (sd, 1);

    for (;;)
    {
        ns=accept (sd, &sockaddr, &fromlen);
        if (fork () ==0)
        {
            /* child */
            close (sd);
            read (ns, buf, sizeof (buf));
            printf ("server read'%s'\n", buf);
            exit ();
        }
        close (ns);
    }
}
```

图 11-20 UNIX 系统域中的服务进程

```
#include<sys/types.h>
#include<sys/socket.h>

main ()
{
    int sd, ns;
    char buf [256];
    struct sockaddr sockaddr;
    int fromlen;

    sd=socket (AF_UNIX, SOCK_STREAM, 0);

    /* 连接到服务者的名字——NULL 字符不是名字的一部分 */
    if (connect (sd,"sockname", sizeof ("sockname") -1==-1)
        exit ();

    write (sd,"hi guy", 6);
}
```

图 11-21 UNIX 系统域中的顾客进程

```
socket (AF_INET, SOCK_STREAM, 0);
```

并且绑定到一个从名址服务器那里得到的网络地址，BSD 系统利用库程序完成这些功能。与此相类似，顾客程序中 connect 的第二个参数应包含用来识别该网络上某主机的寻址信息（或通过中间的机器将消息发送到目的主机的路由地址）以及用来识别该目的主机上特定的套接字的额外信息。如果该服务器既要倾听网络上进程又要倾听本地进程的请求，它应当使用两个套接字，再用系统调用 select 来确定是哪儿的顾客正在请求连接。

11.5　本章小结

本章介绍了进程间通信的几种形式。首先考虑的是进程跟踪，这里两个进程相互合作为程序调试提供一种有用的设施。然而，通过 ptrace 进行跟踪是原始的和高开销的：因为每次调用 ptrace 只能传送有限量的数据；其间出现多次上下文切换；通信仅限于父子进程以及在执行前进程就必须同意被跟踪。UNIX 系统 V 提供了一个包含消息、信号量及共享存储区的 IPC 软件包，遗憾的是它们是为特殊目的设计的，与其他操作系统原语也未能紧密配合，而且不能扩展到网络上去。然而，它们对很多应用是有用的，与其他方案相比提供了较好的性能。

UNIX 系统支持各种各样的网络。传统的实现协议的协商的方法极大地依赖于系统调用 ioctl，然而，对不同网络类型它们的使用是不统一的。BSD 系统引入了套接字系统调用，以便为网络通信提供更通用的结构。将来，系统 V 将使用第 10 章中叙述的流机制来统一地处理网络配置问题。

11.6　习题

1. 如果 debug 程序（图 11-3）中遗漏了系统调用 wait 会发生什么情况？（提示：两种可能性。）

2. 某调试程序每次调用 ptrace 时从一个被跟踪程序读一个字的数据。为使每次调用读许多字的数据，内核应做哪些修改？ptrace 必须做些什么修改？

3. 试将系统调用 ptrace 扩展到下述情形：其中的 pid 不必是调用者的子进程。并考虑以下安全性问题：在什么情况下，应能允许一进程读另一任意进程的地址空间？在什么情况下它应能写另一进程的地址空间？

4. 利用正规文件、有名管道和 locking 原语在用户级程序库实现消息系统调用集。当建立一个消息队列时，创建一个记录该队列状态的控制文件，而且这个文件应以文件上锁或其他习惯的机制被保护起来。当发送一给定类型的消息时，创建一个为该类型的所有消息使用的有名管道文件（若这个文件尚不存在），然后将数据（数据前添加字节数）写入这个有名管道。控制文件应将类型号与有名管道的名字相互关联起来。当读消息时，控制文件引导进程到正确的有名管道。试从性能、编码复杂性及功能三方面对这种实现方案与本章所描述的机制进行比较。

5. 图 11-22 中的程序想要做什么？

*6. 试编写一程序，其共享存储区被附接到十分邻近其栈顶的位置，并使栈顶增长到这个共享存储区中去。什么时候它会引起一个存储错？

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>
#define ALLTYPES 0

main ()
{
    struct msgform
    {
        long mtype;
        char mtext [1024];
    } msg,
    register unsigned int id;

    for (id=0;; id++)
        while (msgrcv (id, &msg, 1024, ALLTYPES, IPC_NOWAIT) >0)
            ;
}
```

图 11-22　窃听程序

7. 使用 IPC_NOWAIT 标志重写图 11-14 的程序，以使信号量操作是有条件的。说明这是怎样避免死锁的？

8. 说明如何以有名管道实现 Dijkstra 的 P 和 V 信号量操作。你想怎样实现条件 P 操作？

9. 试使用（a）有名管道；（b）系统调用 creat 和 unlink；（c）消息系统调用来编写封锁资源的程序，并比较它们的性能。

10. 试编写程序来比较消息系统调用和在有名管道上的 read 和 write 的性能。

11. 试编写程序来比较以共享存储区和消息进行数据传输的速度，对共享存储区的程序应包含信号量以便同步读和写的完成。

第 12 章　多处理机系统

UNIX 系统的传统设计是以使用单一处理机（uniprocessor）体系结构为前提的，所说的单一处理机由一个 CPU、主存及外围设备组成。一个多处理机（multiprocessor）体系结构由共享公共主存及外围设备的两个或多个 CPU 组成（图 12-1），由于进程能在不同的处理机上并发地运行，所以它潜在地提供了更大的系统吞吐量。每个 CPU 独立地执行，但它们都执行内核的一个拷贝。各进程的行为严格地与它们在单一处理机系统上的行为相同——每个系统调用的语义都保持相同，但它们能透明地在处理机之间迁移。遗憾的是，进程一点儿也没有少耗费 CPU 时间。有些多处理机系统由于外围设备不能被所有的处理机所存取，而被称为附属处理机系统（attached processor system）。本章将不在附属处理机系统与一般多处理机系统之间做什么区别，除非另有明确的说明。

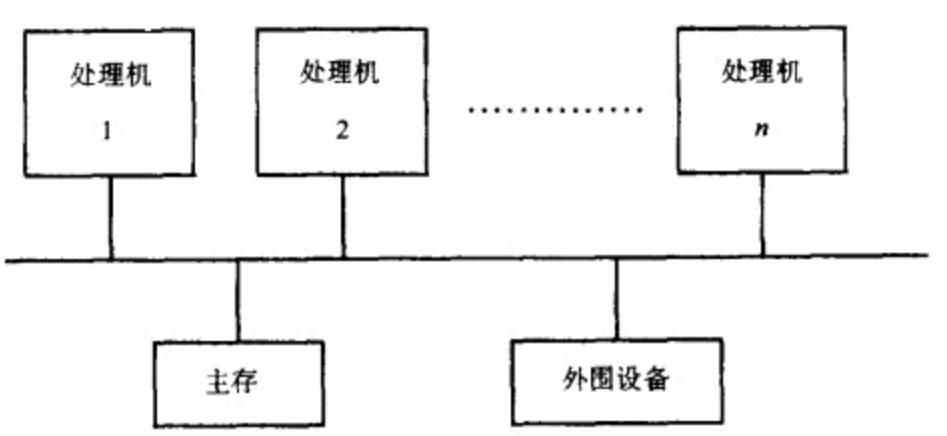


图 12-1　多处理机配置

在不使用保护机制的情况下，如果允许若干处理机代表不同进程同时地在核心态下执行，会引起完整性问题。本章解释了为什么 UNIX 系统的原始设计不能不加改变地运行在多处理机系统上，并且考虑了运行在多处理机上的两个设计。

12.1　多处理机系统的问题

回顾一下第 2 章，UNIX 系统设计采用两个策略，来保证内核数据结构的完整性：当一个进程在核心态下执行时，内核不能抢占它以及把上下文切换到其他进程上去；如果一个中断处理程序可能讹用内核数据结构的话，则当执行代码的临界区时，内核屏蔽中断。然而，在一个多处理机上，如果两个或多个进程同时在分离的处理机的内核中执行的话，则即使采用了对于单处理机系统来说是足够了的保护措施，内核仍会发生讹误。

举例来说，让我们重新考虑第 2 章的那个代码段（图 12-2），该代码段把一个数据结构（指针 bpl）放到一个已存在的结构（指针 bp）之后。假设在不同的处理机上的两个进程同时执行这段代码：处理机 A 想把结构 bpA 放到 bp 之后，而处理机 B 想把结构 bpB 放到 bp 之后。由于对处理机的相对执行速度不能做什么假设，所以可能会发生如下所述的最坏情况：处理机 B 在处理机 A 能够执行另一个语句之前执行四个 C 语句——比如，处理一个中

断能够延迟处理机A上的代码序列的执行。即使封锁了中断，讹误仍可能象第二章所说明的那样发生。

```
struct queue {

} * bp, * bpl;
bpl->forp = bp->forp;
bpl->backp = bp;
bp->forp = bpl;
/* 此处考虑可能的上下文切换 */
bpl->forp->backp = bpl;
```

图 12-2　把一个缓冲区放到双链表中

内核必须确保像这样的讹误永远不会发生。如果让可能出现讹误情况的窗口开放着的话，则不管讹误是多么罕见，内核总是不安全的，并且它的行为是不可预测的。有三个方法可防止这样的讹误（见［Holley 79］）：

（1）让所有的关键活动都在一个处理机上执行，依靠标准的单一处理机方法防止讹误；

（2）使用加锁原语把对代码临界区的存取顺序化；

（3）重新设计算法以回避对数据结构的竞争。

本章描述了使内核不发生讹误的前两种方法。有一个习题考察了第三种方法。

12.2　主从处理机解决方法

Goble 在一对改进型 VAX 11/780 机上实现了一个系统，其中有一台称为主导（master）的处理机，能在核心态下执行；还有一台称为从属（slave）的处理机，仅能在用户态下执行（见［Goble 82］）。虽然 Goble 的实现只包含两台机器，但是其技术可扩展到一个主导与多个从属的情况。主导处理机负责处理所有的系统调用和中断，从属处理机执行用户态下的进程，并在一个进程发出一个系统调用时向主导处理机报告。

调度算法决定一个进程应该在哪个处理机上执行（图 12-3）。进程表中的一个新字段标出了处理机 ID，指出进程必须在这个处理机上运行。为简单起见，假设它既可指出主导处理机，也可指出从属处理机。当从属处理机上的一个进程执行系统调用时，从属处理机的内核填写进程表中的处理机 ID 字段，指出进程只应在主导处理机上运行，并且进行上下文切换以调度其他进程（图 12-4）。主导处理机内核调度到必须在主导处理机上运行的优先权最高的进程，并且执行该进程。当它结束该系统调用时，它把进程表中的处理机 ID 字段置成从属，让进程重新在从属处理机上运行。

如果进程必须在主导处理机上运行，那么，最好是主导处理机立即运行它们，而不使它们等待。这类似于在单处理机上当从一个系统调用返回时允许进程被抢先，以使更紧迫的处理尽快地进行。如果从属处理机请求系统调用服务时，主导处理机正在用户态下执行一个进程，根据本方案，主导进程会继续执行下去，直到下一次上下文切换为止。如果从属处理机设置一个全局标志，主导处理机在时钟中断处理程序中检查这一全局标志，则主导处理机会响应得更快些，至多等一个时钟滴答，主导处理机就会做上下文切换。另一种方案是，从属处理机能中断主导处理机并强迫它立即进行上下文切换，但这需要特殊的硬件能力。

```
算法　schedule _ process
输入：无
输出：无
{
    while（没有被选中执行的进程）
    {
        if（在主导处理机上运行）
            for（就绪队列中的每个进程）
                选出已装入存储器中的优先权最高的进程；
        else  /* 在从属处理机上运行 */
            for（就绪队列中不需要在主导处理机上运行的每个进程）
                选出已装入存储器中的优先权最高的进程；
        if（没有符合执行条件的进程）
          使机器进入空转状态；
            /* 中断会让机器脱离空转状态 */
    }
    把被选中的进程从就绪队列中移出；
    把上下文切换到被选中的进程，恢复它的执行；
}
```

图 12-3　调度算法

```
算法　syscall  /* 修改了的引用系统调用的算法 */
输入：系统调用号
输出：系统调用结果
{
    if（在从属处理机上执行）
    {
        在进程表表项中置处理机 ID 字段；
        进行上下文切换；
    }
    执行实现系统调用的正规算法；
    将处理机 ID 字段重置为“任意”（从属）；
    if（其他进程必须在主导处理机上运行）
      进行上下文切换；
}
```

图 12-4　系统调用处理程序的算法

从属处理机上的时钟中断处理程序使诸进程确实能周期性地被调度到，从而没有哪个进程独占处理机。除了这点以外，时钟处理程序每秒一次把从属处理机从空转状态“唤醒”。从属处理机总是调度不需要在主导处理机上运行的优先权最高的进程。

内核数据结构的讹用的唯一可能来自调度算法，因为调度算法没能防止一个进程会被选在两个处理机上执行。例如，若某配置是由一个主导处理机和两个从属处理机组成的，则有

可能这两个从属处理机发现处于用户态下的同一个进程处于“就绪”状态。假若两个处理机同时调度到该进程，则它们将会读、写及破坏其地址空间。

系统能以两种方式避免这一问题。第一种，主导处理机能够指明进程应该在哪个从属处理机上执行，允许多个进程被分配到一个处理机上。但这样又出现了负载均衡的问题：可能有大量进程分配给一个处理机，而另一些处理机是空闲的。主导处理机内核必须把进程负载分布在各个处理机上。第二种，主导的内核能使用下一节所描述的信号量机制，允许在一个时刻只有一个处理机执行调度循环。

12.3　信号量解决方法

支持多处理机配置上的 UNIX 系统的另一个方法是把内核分成多个临界区，在任一时刻最多有一个处理机能执行临界区代码。像这样的多处理机系统设计已经用到 AT&T3B20A 计算机和 IBM370 计算机上，是使用信号量把内核分成多个临界区（见［Bach 84］）。此处的描述将遵循这两个系统上的实现。有两个问题：如何实现信号量以及在哪儿定义临界区。

正如在第 2 章指出的那样，单一处理机 UNIX 系统上的各种算法使用睡眠锁将其他进程置于临界区之外，以防第一个进程随后到临界区内去睡眠。置锁的机制是：

```
while (锁为忙状态)    /* 测试操作 */
        sleep (直至锁变为空闲);
置锁为忙状态;
```

而解锁机制是：

```
解锁;
把在条件锁集合上睡眠的所有进程唤醒;
```

睡眠锁描绘了一些临界区，但正如图 12-5 所指出的那样，睡眠锁在多处理机系统上不起作用。假设一个锁是空闲的，而且位于两个处理机上的两个进程同时试图测试该锁，并置锁。

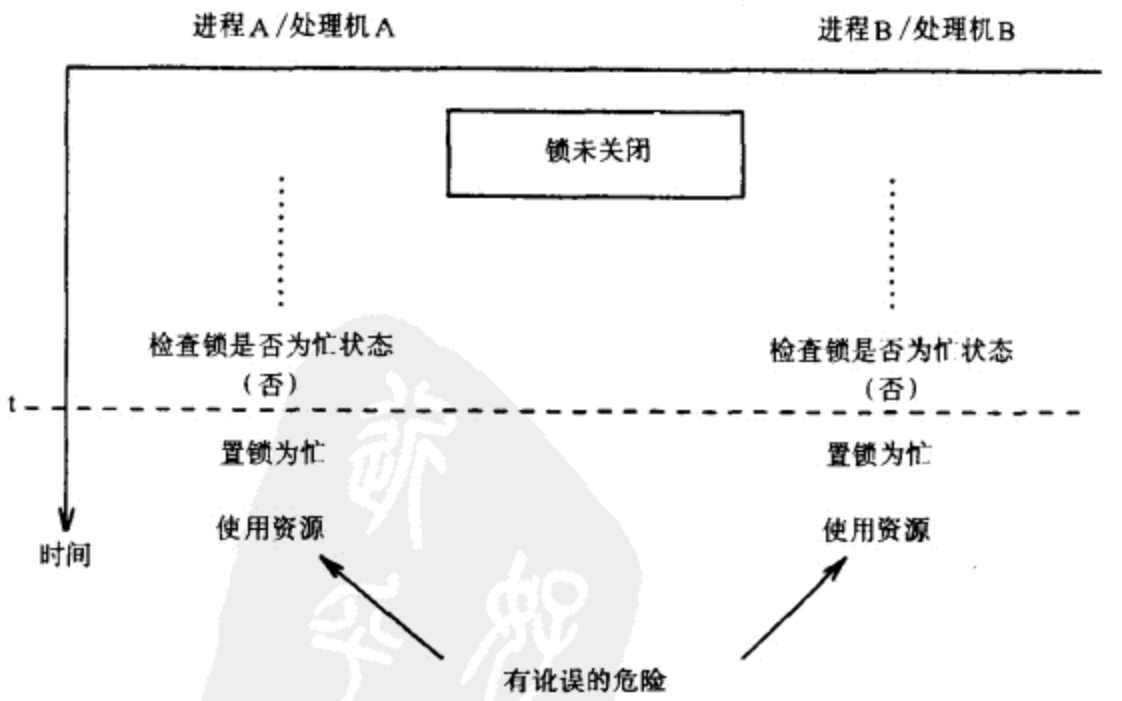


图 12-5　多处理机上的睡眠锁的竞争条件

它们发现在 t 时刻锁是空闲的，于是上锁，进入临界区，从而会讹用内核数据结构。在请求的同时性上有可允许的误差：无论哪个进程在另一进程执行测试操作之前执行了上锁操作，睡眠锁都不会失效。反之，则睡眠锁会失效。例如，若处理机 A 在发现锁是空闲的之后处理一个中断，并且在它正在处理中断时，处理机 B 检查锁，并且上了锁，则处理机 A 从中断返回后会再次上锁。为防止这一情况发生，加锁机制应当是原子的：对锁状态的测试动作及上锁动作必须在单一的、不可分的操作中做完，即在任一时刻仅有一个进程能对锁进行操纵。

12.3.1　信号量定义

一个信号量是一个由内核操纵的取整数值的对象，对它定义了如下的原子操作：

- 置初值操作：将信量初值置为非负值。
- P 操作：将信号量值减 1。如果在减 1 之后信号量的值小于 0，则做 P 操作的进程去睡眠。
- V 操作：将信号量值加 1。如果在加 1 之后信号量的值小于或等于 0，则唤醒因 P 操作而睡眠的一个进程。
- 条件 P 操作（简记为 CP)；如果信号量的值大于 0，则将信号量值减 1，且返回一个取值为真的指示；如果信号量的值小于或等于 0，则信号量的值不改变，且返回值为假。

这就是所定义的信号量。当然，这与第 11 章所描述的用户级信号量是无关的。

12.3.2　信号量实现

Dijkstra［Dijkstra 65］指出可以不使用专门的机器指令来实现信号量。图 12-6 介绍了实现信号量的 C 函数。函数 Pprim 通过检查数组 val 的值而把信号量锁住；系统中的每个处

```
struct semaphore
{
    int val [NUMPROCS];      /* 锁---为每个处理机设置一个登记项 */
    int lastid;              /* 最近一次获得信号量的那个处理机的 ID */
};
int procid;                  /* 处理机 ID，每个处理机对应着唯一的一个 */
int lastid;                  /* 最近一次获得信号量的那个处理机的 ID */

INIT (semaphore)
    struct semaphore semaphore;
{
    int i;
    for (i=0; i<NUMPROCS; i++)
        semaphore.val [i] =0;
}
Pprim (semaphore)
    struct semaphore semaphore;
```

图 12-6　用 C 语言实现加锁信号量

```
{
    int i, first;

loop:
    first = lastid;
    semaphore.val [procid] = 1;
forloop:
    for (i = first; i < NUMPROCS; i++)
    {
        if (i == procid)
        {
            semaphore.val [i] = 2;
            for (i = 1; i < NUMPROCS; i++)
                if (i! = procid && semaphore.val [i] == 2)
                    goto loop;
            lastid = procid;
            return;          /* 成功! 现在可使用资源 */
        }
        else if (semaphore.val [i])
            goto loop;
    }
    first = 1;
    goto forloop;
}
Vprim (semaphore)
    struct semaphore semaphore;
{
    lastid = (procid + 1) % NUMPROCS;      /* 重置为下一个处理机 */
    semaphore.val [procid] = 0;
}
```

图 12-6 （续）

理机都控制该数组中的一个登记项。当一个处理机想锁住信号量时，它去检查是否其他处理机已经把该信号量上锁了（此时它们在 val 中的登记项的值为 2），或者是否 ID 比它低的处理机当前正在企图锁住信号量（此时它们在 val 中的登记项的值为 1）。无论这两个条件中的哪一个为真，本处理机都把它自己在 val 中的登记项重置为 1，并且重来一次。Pprim 开始外层循环时，循环变量的值恰比最近使用资源的那个处理机 ID 的值大 1，以保证没有一个处理机能垄断资源（参见［Dijkstra 65］或［Coffman 73］的证明）。Vprim 通过清除正在执行着的处理机在 val 中的登记项及重置 lastid 来释放信号量，并允许其它处理机获得对资源的互斥存取。如下的代码序列能保护一个资源：

```
Pprim（信号量）;
此处使用资源：
Vprim（信号量）;
```

由于 Pprim 中的循环慢且性能低下，所以大多数机器上都有一个不可分的指令集合来做与上述等价的而且更经济的加锁操作。比如，IBM 370 系列支持一个原子的“比较与对换”（compare and swap）指令，而 AT&T 3B20 计算机支持一个原子的“读后清”（read and clear）指令。举例来说，当执行读后清指令时，机器把存储器某地点的值读出，清除它的值（将其置为 0），并且根据原始值是否为 0 来置条件代码。如果另一个处理机对同一存储地点同时使用读后清指令，则能保证一个处理机读出原始值而其它处理机读出 0 值：硬件保证了原子性。因此，使用读后清指令能更简单地实现函数 Pprim（图 12-7）。一个进程使用读后清指令循环，直至它读到一个非零值时为止。信号量锁分量必须被初始化为 1。

```
struct semaphore {/* 简化了的信号量结构 */
    int lock; /* 信号量结构中的锁字段 */
};

Init (semaphore)
    struct semaphore semaphore;
{
    semaphore.lock = 1;
}

Pprim (semaphore)
    struct semaphore semaphore;
{
    while (read_and_clear (semaphore.lock))
;
}

Vprim (semaphore)
    struct semaphore semaphore;
{
    semaphore.lock = 1;
}
```

图 12-7　使用读后清指令的信号量操作

这一信号量原语不能用在内核中，因为执行该原语的进程不断地循环直至它成功时为止：如果信号量是用来锁住一个数据结构的，则当进程发现信号量为上锁状态时它应该去睡眠，以便内核能把上下文切换到另一个进程上，从而做更有用的工作。给出 Pprim 和 Vprim 后，有可能构造出更复杂的、符合 12.3.1 节中定义的内核信号量操作 P 和 V 的集合。

首先，让我们把信号量定义为由对信号量存取进行控制的锁字段、信号量值字段及进程在信号量上睡眠用的队列所组成的结构。锁字段控制对信号量的存取，在 P，V 操作期间仅允许一个进程操纵该结构的其他字段。当 P、V 操作完成时它被重置。值字段决定一个进程是否应该具有对由信号量保护着的临界区的存取。在 P 算法的开始处（图 12-8），内核做一个 Pprim 操作，以保证对信号量的互斥存取，接着将信号量减值。如果信号量值非负，则正在执行的进程能存取临界区：它用 Vprim 操作重置信号量锁，以便其他进程能存取信号量，并且返回一个成功指示；如果作为减值的结果，信号量的值为负，则内核遵循类似于正规睡眠算法（第 6 章）的语义，把进程投入睡眠态：它按照优先权值来检查软中断信号，把正在

执行的进程送入睡眠进程的先进先出表中，并且做上下文切换。V算法（图12-9）通过Pprim原语来获得对信号量的互斥存取，并且为信号量增值。如果任何进程正处在信号量睡眠队列中，则内核把队列上的第一个进程摘下，把它的状态改为“就绪”。

P，V函数类似于sleep和wakeup函数，在实现上的主要区别在于：信号量是一个数据结构，而用于sleep和wakeup的地址仅为一个合适的数字。如果信号量的初始值为0，则对

```
算法 P  /* P信号量操作 */
输入：(1) 信号量
      (2) 优先权
输出：0——正常返回；
      -1——由于在内核中捕俘了软中断信号而异常唤醒；
      longjmp——由于不在内核中捕俘了软中断信号；
{
    Pprim（信号量的锁字段）；
    信号量的值字段减1；
    if  （信号量的值字段>=0）
    {
        Vprim（信号量的锁字段）；
        return（0）；
    }
    /* 必须去睡眠 */
    if  （检查软中断信号）
    {
        if  （有中断睡眠的软中断信号）
        {
            信号量的值字段加1；
            if  （在内核中捕俘了软中断信号）
            {
                Vprim（信号量的锁字段）；
                return（-1）；
            }
            else
            {
                Vprim（信号量的锁字段）；
                longjmp；
            }
        }
    }
    将进程送入信号量睡眠表的尾部；
    Vprim（信号量的锁字段）；
    做上下文切换；
    如上，检查软中断信号；
    return（0）；
}
```

图12-8　实现P操作的算法

```
算法　V　/* V信号量操作 */
输入：信号量地址
输出：无
{
    Pprim（信号量的锁字段）;
    信号量的值字段加 1;
    if　（信号量的值字段<=0）
    {
        移去信号量睡眠表上的第一个进程;
        将该进程置为就绪态（唤醒它）;
    }
    Vprim（信号量的锁字段）;
}
```

图 12-9　实现 V 操作的算法

信号量进行 P 操作时，进程总是去睡眠，因此 P 能代替 sleep 函数。然而，V 操作仅能唤醒一个进程，而单一处理机的 wakeup 函数唤醒在事件地址上睡眠的所有进程。

从语义上说，wakeup 函数的使用指示一个给定的系统条件再也不为真了，从而在该条件上睡眠的所有进程必须全部唤醒。比如说，当一个缓冲区不再被使用的时候，则进程在缓冲区忙这一事件上睡眠就是不正确的了，所以内核把在该事件上睡眠的所有进程唤醒。第二个例子是，如果多个进程往一个终端上写数据，由于终端驱动程序不能处理那么大量的数据，所以终端驱动程序可以把它们投入睡眠状态。以后，当该驱动程序决定它能够接受更多的数据并把它们输出时，它唤醒因等待输出数据而睡眠的所有进程。在诸进程一个挨一个地获得对资源的存取的情况下，P，V 操作是更实用的加锁操作，并且其他进程按它们对资源请求的次序被准予存取资源。由于当一个事件发生时，如果唤醒所有进程，大多数进程会发现锁仍是闭着的，从而立刻转去睡眠，所以现在的这一处理方法通常比单处理机睡眠锁更有效。但是，另一方面，在所有进程应该被同时唤醒的情况下，使用 P，V 操作就较为困难了。

若给出返回信号量值的原语，那么，下面的操作会与 wakeup 函数等价吗？

```
while （value（信号量）<0）
    V（信号量）;
```

假设没有来自其他处理机的干扰，则内核不停地执行该循环，直至信号量值大于或等于 0。这意味着，没有进程在该信号量上睡眠了。然而，有可能处理机 A 上的进程 A 测试该信号量，发现它的值等于 0，而就在 A 上的测试之后，处理机 B 上的进程 B 做了个 P 操作，把信号量值减至 −1（图 12-10）。进程 A 会自认为已经唤醒了在该信号量上睡眠的所有进程，从而继续执行。因此，该循环不能保证每个睡眠进程都被唤醒，因为它不是原子的。

考虑单处理机系统上使用信号量的另一种现象。假设两个进程 A 和 B，为一个信号量而竞争：进程 A 发现该信号量空闲而进程 B 在睡眠；此时信号量的值为 −1。当进程 A 用一个 V 操作来释放该信号量时，它唤醒进程 B，并且把信号量值增至 0。现在，假设进程 A 仍然在核心态执行，它试图再次锁住该信号量，因为此时该信号量值为 0，所以即使该资源仍为

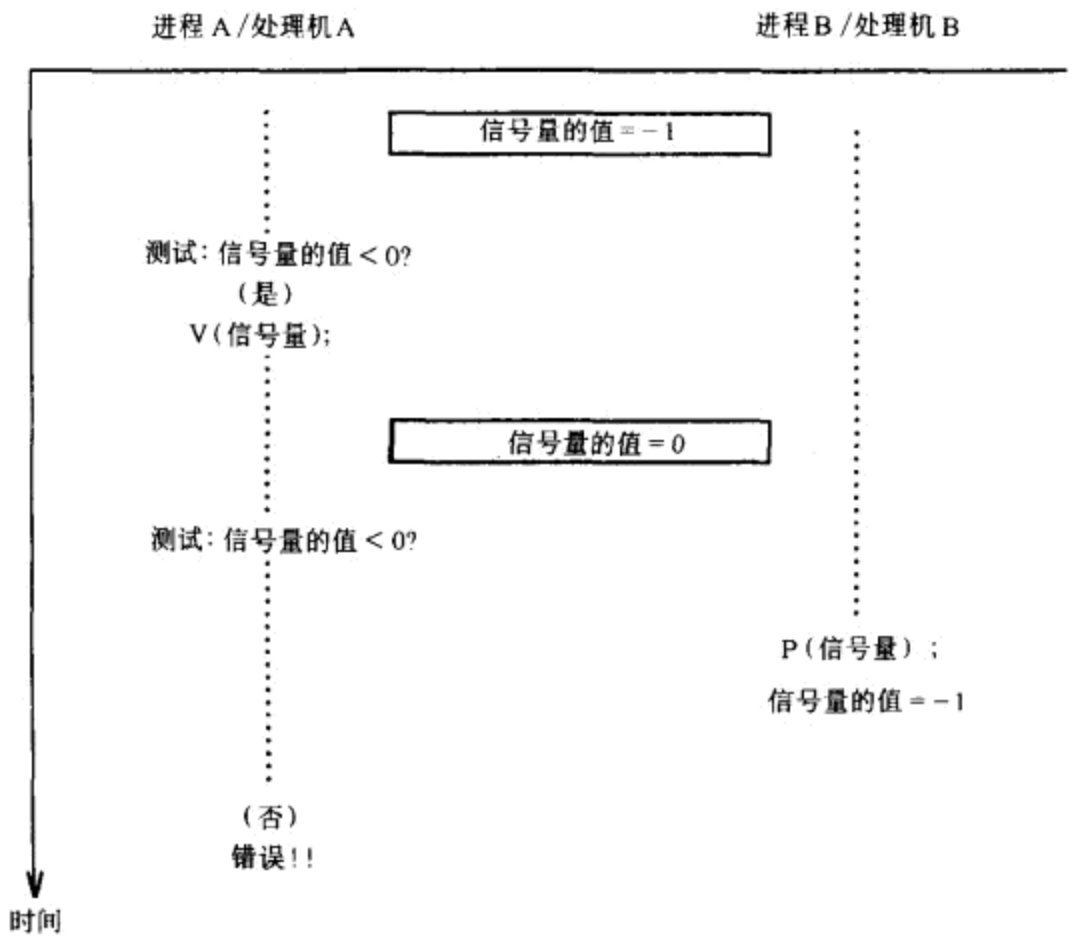


图 12-10　用 V 操作所做的失败唤醒模拟

空闲，但进程 A 也将由于执行 P 函数而睡眠。系统将招致一个额外的上下文切换的开销。另一方面，如果该锁是用睡眠锁来实现的，则进程 A 会立即获得对该资源的再使用，因为在此期间没有其他进程能锁住它。在这一情况下，睡眠锁会比信号量锁更有效。

当对多个信号量加锁时，必须使加锁次序一致以避免死锁。比如，考虑两个信号量 SA 和 SB，并考虑必须同时对这两个信号量上锁的两个内核算法。如果这两个算法按相反的次序锁住这两个信号量，就会出现死锁，如图 12-11 所示。当处理机 B 上的进程 B 锁住了信号量 SB 时，处理机 A 上的进程 A 锁住了信号量 SA。进程 A 企图锁住信号量 SB，但 P 操作

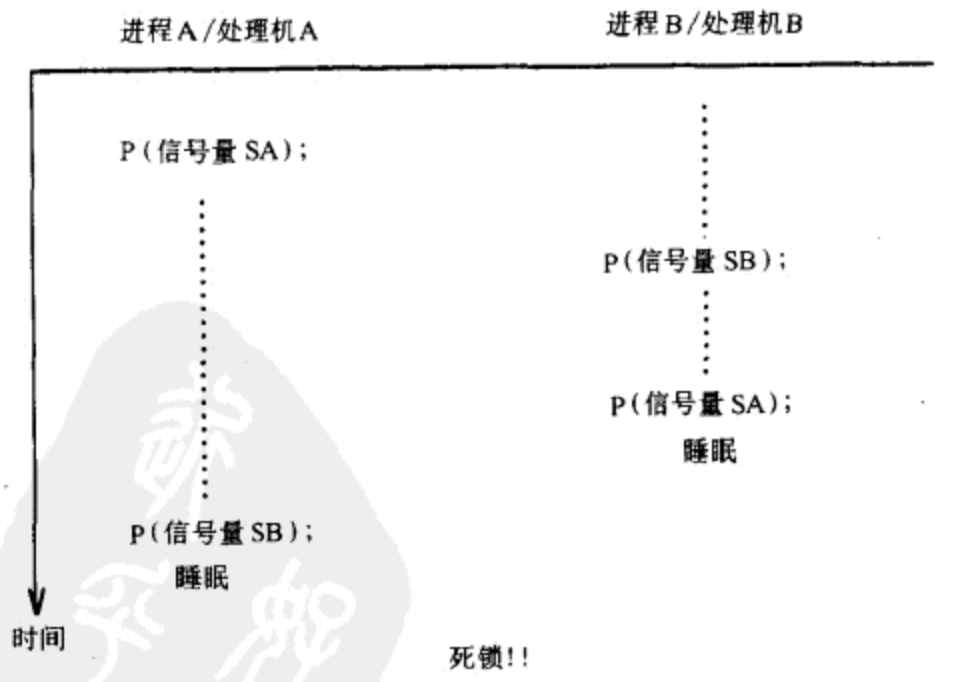


图 12-11　由于加锁次序相反而产生的死锁

使得进程 A 去睡眠，因为 SB 的值至多为 0。类似地，进程 B 企图锁住信号量 SA，但是它的 P 操作把进程 B 投入睡眠态。无论哪个进程都不能继续下去。

通过采用死锁检测算法可以避免死锁。该死锁检测算法判定是否有死锁存在，如果有，则破坏死锁条件。然而，死锁检测算法的实现会使内核代码复杂化。另一方面，考虑到内核中只有有限数目的地方需要一个进程同时锁住多个信号量，所以很容易实现在死锁发生之前避免死锁条件的内核算法。例如，如果总是按相同的次序锁住特定的信号量集，则死锁条件永远不会发生。但是，如果避免不了按相反次序锁住这些信号量的情况，则 CP 操作可防止死锁，如图 12-12 所示：若 CP 失败，则进程 B 释放它的资源以避免死锁，并且在以后的某个时间，比如当进程 A 结束对该资源的使用时重新进入该算法。

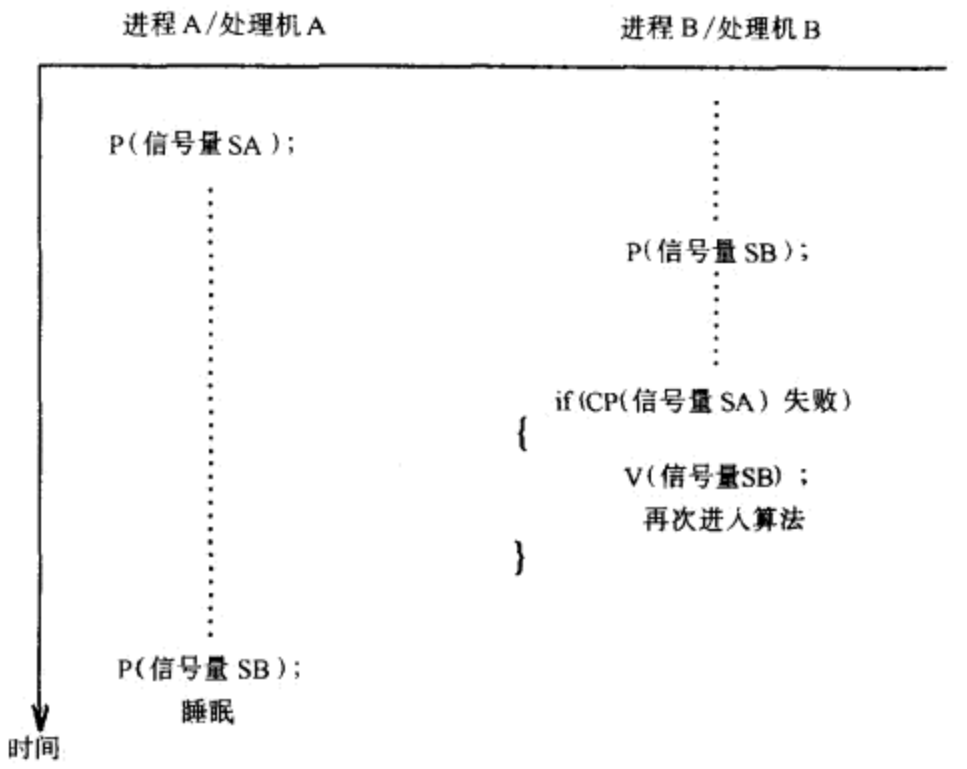


图 12-12　使用条件 P 操作避免死锁

为了不让多个进程同时使用一个资源，中断处理程序必须锁住信号量。但正如第 6 章所解释的那样，它不能去睡眠，因此不能使用 P 操作。代替地，它能执行一个循环锁（spin lock）以避免去睡眠：

```
while　(!　CP (信号量));
```

只要信号量的值小于或等于 0，该操作就循环；本处理程序不去睡眠，并且仅当信号量值变为正时循环才结束，在那时，CP 操作把信号量值减 1。

为避免死锁，内核必须封锁住执行循环锁的中断。要不然，进程会把一个信号量锁住，而在它为这个信号量解锁之前被中断，如果中断处理程序企图使用循环锁来锁住同一个信号量，则内核会自己死锁。例如，在图 12-13 中，当中断发生时信号量值至多为 0，因此在中断处理程序中 CP 操作将总是为假。如果规定当进程具有上了锁的信号量时必须封锁中断，就可以避免这种情况。

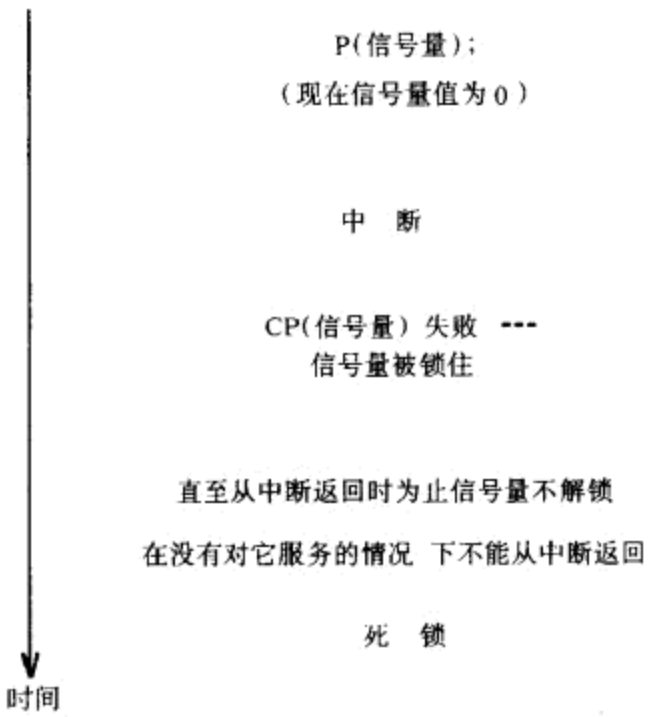


图 12-13　中断处理程序中的死锁

12.3.3　几个算法

本节回顾一下用信号量实现的四个内核算法。缓冲区分配算法说明了一个复杂的加锁情况，wait 算法说明了进程同步，驱动程序加锁方案说明了一个为设备驱动程序加锁的精巧方法，最后，处理机空转的方法表明了怎样改变一个算法以避免竞争。

1. 缓冲区算法

我们再看一下第 3 章中的缓冲区分配算法 getblk。用于缓冲区分配的三个主要数据结构是缓冲头部、缓冲区散列队列及空闲缓冲区表。内核把每个数据结构实例都与一个信号量相联系。换句话说，如果内核含有 200 个缓冲区，则每个缓冲头部都包含一个用于为缓冲区加锁的信号量；当一个进程在缓冲头部信号量上做 P 操作时，其他做 P 操作的进程去睡眠，直至第一个进程做 V 操作。每个缓冲区散列队列也有一个信号量用于锁住对散列队列的存取。单一处理机系统中不需要对散列队列加锁，因为进程从来不会去睡眠而使散列队列处在不一致状态。然而，在多处理机系统中，可能会有两个进程操纵散列队列链接表。散列队列的信号量每次仅允许一个进程操纵该链接表。类似地，空闲表也需要一个信号量。这是因为，如果不这样的话，就可能会发生多个进程讹用空闲表的情况。

图 12-14 描述了在多处理机系统上用信号量实现的算法 getblk 的第一部分（回忆一下图 3-4）。为要对一给定块搜索高速缓冲，内核用 P 操作锁住散列队列信号量。如果已经有另一个进程在这个信号量上做了 P 操作，则正在执行的进程去睡眠，直至原来的进程做 V 操作时为止。当该进程获得了对这个散列队列的排它性控制时，它去搜索适当的缓冲区。假设被搜索的缓冲区在散列队列中，则内核（进程 A）企图锁住该缓冲区。但是，如果它使用 P 操作，而且如果该缓冲区已被锁住，则它会到上了锁的散列队列中去睡眠，这样一来，即使其他进程搜索散列队列是为了找到别的缓冲区，它们也不能存取该散列队列了。所以，进程 A 不是使用 P 操作而是使用 CP 操作锁住该缓冲区的。如果 CP 成功，则它能使用该缓冲区。进程 A 通过自转循环中的 CP 锁住空闲表信号量，因为预计锁被占用的时间很短，因而以 P

操作进入睡眠并不合算。内核于是从空闲表中摘下该缓冲区，为空闲表解锁，为散列队列解锁，并且返回上了锁的缓冲区。

假设因为别的进程已经锁住缓冲区信号量而造成上述对缓冲区的 CP 操作失败，进程 A

```
算法  getblk    /* 多处理机版本 */
输入：文件系统号
      块号
输出：现在可供使用的上了锁的缓冲区
{
    while（没找到缓冲区）
    {
        P（散列队列信号量）;
        if（块在散列队列中）
        {
            if（CP（缓冲区信号量）失败）   /* 缓冲区忙 */
            {
                V（散列队列信号量）;
                P（缓冲区信号量）;   /* 睡眠直至空闲 */
                if  （CP（散列队列信号量）失败）
                {
                    V（缓冲区信号量）;
                    continue;      /* 回到 while 循环 */
                }
                else if（设备号或块号改变）
                {
                    V（缓冲区信号量）;
                    V（散列队列信号量）;
                }
            }
            while（CP（空闲表信号量）失败）;   /* 自转循环 */
            为缓冲区标记上“忙”;
            从空闲表上摘下该缓冲区;
            V（空闲表信号量）;
            V（散列队列信号量）;
            return（缓冲区）;
        }
        else  /* 缓冲区不在散列队列中 */
        /* 算法的其余部分由此继续 */
    }
}
```

图 12-14　采用信号量的缓冲区算法

释放散列队列信号量，然后用 P 操作到该缓冲区信号量上去睡眠。注意：P 在刚才引起 CP 失败的那个信号量上操作！进程 A 是否在该信号量上睡眠是没有什么关系的，在 P 操作完成之后，进程 A 控制该缓冲区。因为算法的其余部分假设缓冲区和散列队列都上了锁了，

现在进程A企图锁住散列队列[⊖]。因为此处加锁的次序（先缓冲区信号量，然后散列队列信号量）与上述说明的加锁的次序（先散列队列信号量，然后缓冲区信号量）相反，所以此处使用了CP信号量操作。如果这一锁操作失败了，则发生的处理是显而易见的。但是，如果这一锁操作成功了，则内核不能保证它有正确的缓冲区，因为可能另外的进程已经在空闲表上找到该缓冲区，并且在放弃对缓冲区信号量的控制之前，把缓冲区的内容改变为另一块的内容了。正在等待该信号量变为空闲的进程A，此时不知道它正在期待的缓冲区是不是它所感兴趣的缓冲区，因此它必须检查那个缓冲区是否仍然有效；如果无效，则它重新启动该算法。如果该缓冲区包含有效数据，则进程A完成该算法。该算法的其余部分作为练习留给读者。

2. 系统调用 wait

回忆一下第7章，进程在系统调用wait中睡眠，直至一个子进程退出。多处理机系统上的问题是确保当父进程在执行wait算法时，没有遗漏掉僵死的子进程。举例来说，如果当父进程在某处理机上执行系统调用wait，而一个子进程从另一处理机上退出了，则父进程不必为等待第二个子进程退出而睡眠。每个进程表项都含有一个被称为僵死信号量的信号量，其初始值为0，执行系统调用wait的进程就在这个信号量上睡眠，直至一个子进程退出时为止（图12-15）。当一个进程退出时，它在父进程信号量上做V操作，如果父进程正在wait中睡眠，则把父进程唤醒。如果在父进程执行wait之前子进程就退出了，则父进程找到僵死状态的进程，并且返回。如果两个进程同时执行exit与wait，但是子进程是在父进程已经检查了它的状态之后才退出的，则子进程的V操作将阻止父进程睡眠。最坏的情况是，父进程将对该循环多做一次迭代。

```
多处理机算法　wait
{
    for (;;)    /* 循环 */
    {
        搜索所有的子进程:
        if (子进程状态为僵死)
          return;
        P (僵死信号量);    /* 置初值为0 */
    }
}
```

图12-15　Wait/Exit的多处理机算法

3. 驱动程序

AT&T 3B20A计算机的多处理机实现，通过在驱动程序入口点做P，V操作而避免把信号量插入到驱动程序代码中（见[Bach.84]）。回忆一下第10章，与设备驱动程序的接口仅用很少几个（实际上，大约20个）入口点就明确定义了。通过把入口点括起来的方法来保护驱动程序，比如：

⊖ 此处，该算法能通过在下一个V操作之前建立一个标志并测试标志来避免锁住散列队列。但是，这一方法解释了按相反次序锁住信号量的技术。

```
P（驱动程序信号量）;
打开驱动程序;
V（驱动程序信号量）;
```

对于一个驱动程序的所有入口点都使用同一个信号量，而对于每个驱动程序都使用不同的信号量，这样，在任一时刻，至多一个进程能执行驱动程序中的临界代码。可以按每个设备或按设备类配置信号量。例如，可以把一个信号量与一个物理终端联系起来，或者把一个信号量与所有的物理终端联系起来。前一种情况可能快些，因为存取一个终端的进程不像在后一种情况那样，要对其他的终端锁住信号量。然而，某些设备驱动程序内在地与其他设备驱动程序交互作用，在这种情况下，为一类设备指明一个信号量更易于理解。另外，3B20A 的实现给出了另一个办法，它允许特定的设备按如下方式配置：驱动程序代码运行在被指明的处理机上。

当一个设备信号量被锁住时，若该设备向系统发中断，就会产生如下问题：中断处理程序不能被引用，因为如若不然，会有讹误的危险；另一方面，内核还必须确实查明它没有丢失中断。3B20A 使中断进入队列，直至信号量被解锁并且能安全地执行中断处理程序时为止。如果必要的话，它从给驱动程序解锁的代码中调用中断处理程序。

4. 哑进程

正如在第 6 章中所说明的那样，当内核在单处理机中做上下文切换时，它是在放弃控制的进程的上下中文执行的。如果没有处于就绪状态的进程，则内核在最后运行的进程的上下文中空转。当被时钟或被其他外设中断时，它在空转的进程的上下文中处理中断。

在多处理机系统中，内核不能在最近一次在该处理机上执行的进程的上下文中空转。因为，如果一个进程在处理机 A 上进入睡眠，考虑一下当该进程被唤醒时会发生什么：它准备好运行，但即使它的上下文在处理机 A 上已经是可用的，却不能立即执行。如果这时处理机 B 选择了该进程执行，它会做上下文切换，并且恢复其执行。当由于另一个中断而使处理机 A 从空转循环中出来时，它再次在进程 A 的上下文中执行，直至它切换上下文时为止。因此，在一个短时间内，这两个处理机会向同一个地址空间写，特别地，向同一个核心栈写。

在每个处理机上创建一个哑（dummy）进程可以解决这一问题。当一个处理机无工作可做时，内核把上下文切换到哑进程，处理机在哑进程上下文中空转。哑进程仅由一个核心栈组成，它不能被调度。因为仅有一个处理机能在它的哑进程中空转，所以诸处理机不会互相讹用。

12.4　Tunis 系统

Tunis 系统有一个与 UNIX 系统的用户界面兼容的用户接口，但是，它的以并发 Euclid 语言书写的核（nucleus）是由控制系统的每一部分的核心进程组成的。因为在任一时刻仅能有一个核心进程实例运行，并且因为核心进程不操纵其他进程的数据结构，所以 Tunis 系统解决了互斥问题。通过把输入消息送入队列而激活核心进程，并以并发 Euclid 实现了管程（monitor），以防止队列的讹用。管程是这样的过程，它通过在任一时刻仅允许一个进程执行该过程体，而实现强制性互斥。因为管程强制使用模块化技术（P 和 V 在管程例行程序的入口点和出口点处出现），并且由编译程序生成同步原语，所以它们与信号量不同。Holt

指出，使用支持并发和管程概念的语言，比较容易构造出这样的系统（见［Holt 83］第190页）。然而，Tunis系统的内部结构与传统的UNIX系统根本不同。

12.5　性能局限性

本章介绍了用于实现多处理机UNIX系统的两种方法：主-从配置法与信号量法。主从配置法中仅有一个处理机能在核心态下执行；信号量法允许所有的处理机同时在核心态下执行。本章中所描述的多处理机UNIX系统的实现，可推广到任意数目的处理机，但系统吞吐量并不以线性速度随着处理机数目的增加而增大。首先，由于硬件中存储竞争的增多而出现了降级，即主存的存取需要花更长时间了。其次，在信号量方案中，对信号量的竞争增多了，进程更频繁地发现信号量上锁了，更多的进程排到队列中等待信号量变为空闲，从而进程不得不等候更长的期间以获得对信号量的存取。类似地，在主-从方案中，随着系统中处理机数目的增加，主导处理机变成了系统的瓶颈，因为仅有一个处理机能执行内核代码。虽然，仔细的硬件设计能减少竞争，并且对某些负载来说（比如，见［Beck 85］），随着处理机数目的增加也提供系统吞吐率的近乎线性的增长，但是，用现有技术建立起来的所有的多处理机系统，都只能达到一个极限，超过这个极限之后，再增加处理机也不能提高系统吞吐量了。

12.6　习题

1. 给出一个实现多处理机问题的解决方法，使得在一个多处理机配置中，任何处理机都能执行内核，但在任一时刻仅有一个处理机能执行内核。这不同于在正文中讨论的第一种解决方法，在那里，一个处理机被指定为主导处理机，以处理所有的内核服务。像这样的系统怎样才能确保仅有一个处理机处在内核中？为处理中断并且仍确保仅有一个处理机在内核中的合理策略是什么？

2. 使用共享存储区的系统调用，来测试图12-6中示出的用来实现信号量的C代码。若干彼此独立的进程会在一个信号量上执行P-V序列。你怎样演示代码中的一个错误？

3. 仿照P算法的语句行，设计一个CP（条件P）操作算法。

4. 解释为什么图12-8中的P操作算法与图12-9中的V操作算法必须封锁中断。应该在哪一点上被封锁住？

5. 如果像在

```
while (!   CP (信号量));
```

中那样把信号量用在自转循环锁中，为什么内核从来不对信号量使用无条件P操作？（提示：如果一个进程在P操作上睡眠，在自转循环锁中会发生什么？）

6. 参考第3章中的算法getblk，并针对磁盘块不在高速缓冲中的情况，描述多处理机的一个实现。

*7. 在缓冲区分配算法中，假设对缓冲区空闲表信号量有过多的竞争。通过把现在的空闲表分成两个空闲表，来实现一个减少竞争的方案。

*8. 假设终端驱动程序有一个信号量，其初值为0。如果进程的输出一齐涌向终端时，进程去睡眠。当终端又能接受数据时，它唤醒在该信号量上睡眠的每个进程。设计一个使用

P、V 操作唤醒所有进程的方案。如果需要的话，可以定义另外的标志和驱动程序加锁信号量。如果唤醒是由中断导致的，并且一个处理机不能封锁住其他处理机的中断，那么怎样才能保证这一方案是安全的？

*9. 当用信号量保护驱动程序入口点时，必须做出规定，以使进程在驱动程序中睡眠时能释放该信号量。描述一个实现方法。类似地，驱动程序应当怎样处理在驱动程序信号量被锁住时所发生的中断？

10. 回忆第 8 章中置系统时间与取系统时间的系统调用。一个系统对于不同的多处理机来说不能保证有同一的时钟速率，系统调用 time 应该怎样工作？

第 13 章　分布式 UNIX 系统

前一章探讨了紧密耦合的多处理机系统，它的特点是共享公共存储区和内核数据结构，并从一公共池中调度进程。然而，人们常常希望使每个计算机在它的环境范围内保持自主的情况下，将若干台计算机汇集起来以便共享资源。比如，个人计算机的用户想要存取存储在一台大型计算机上的文件，但仍要保持对个人计算机的控制。虽然若干实用程序，比 uucp 允许通过一个网络进行文件传输和其他应用，但对它们的使用是不透明的，因为用户知道网络的存在。此外，某些程序如文本编辑程序对远处文件也不像它们对本地文件那样地运行。然而，用户想要执行的是通常的 UNIX 系统调用集合，而且除了可能的性能方面的降低外，他们也不想知道这些系统调用已跨越了机器的边界。准确地说，系统调用 open 和 read 应当能象它们对本地系统上的文件那样，对远程系统上的文件执行。

图 13-1 示出了一个分布式系统的体系结构。图中每一个以圆圈表示的计算机是一个自主的单位，它是由 CPU、存储器及若干外围设备组成的。甚至一个不含有本地文件存储器的计算机也可以用于这个模型，也就是说，一个计算机必须有与其他机器进行通信的外围设备，但它的所有的正规文件可能是在其他机器上。更严格地讲，每台机器所拥有的物理存储器是与其他机器上的活动不相关的，这个特点是与前一章所讨论的紧密耦合的多处理机系统不同的。因此，每台机器的内核都是独立的，仅以运行于分布式环境的外部约束为条件。

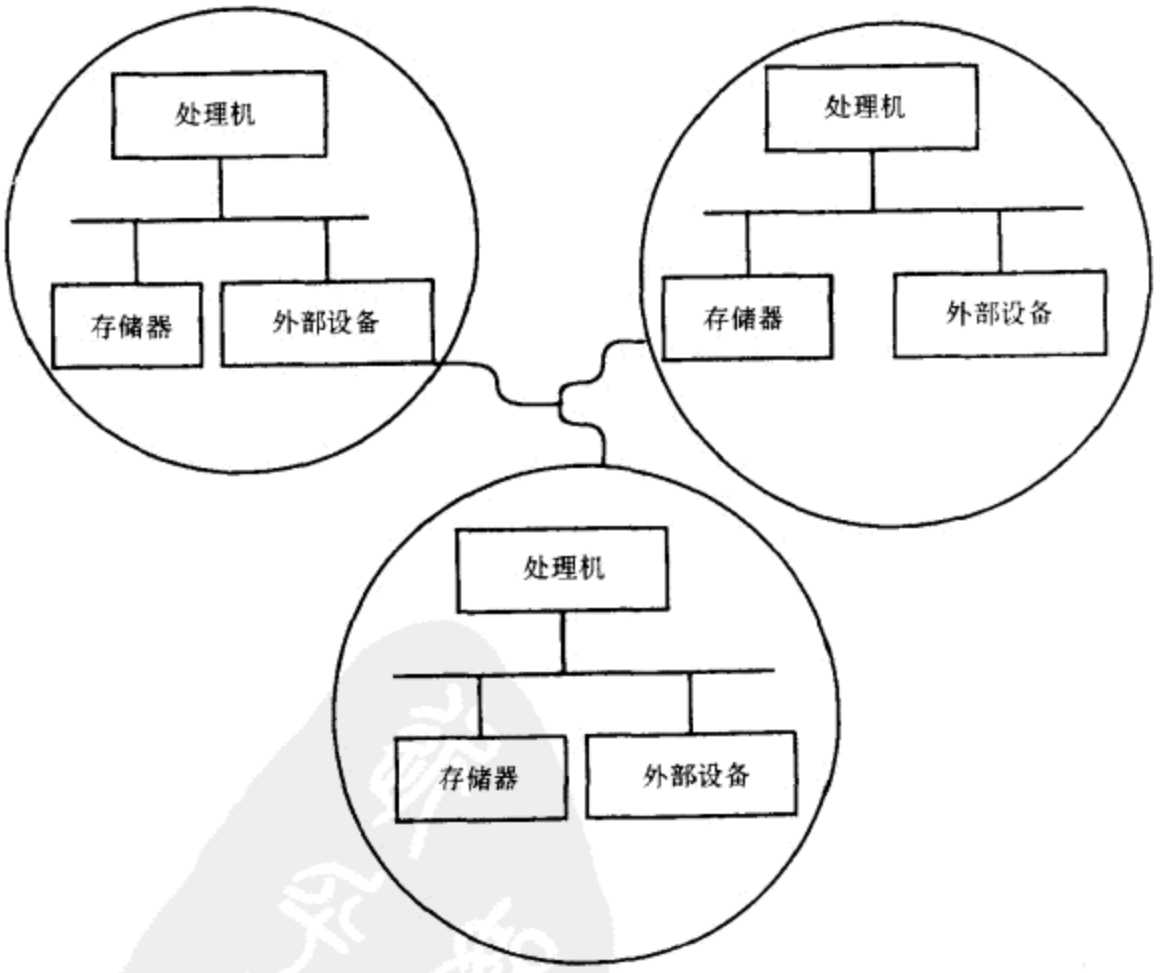


图 13-1　分布式体系结构模型

在文献中讨论了许多分布式系统的实现方法，大体上可将之归入以下几类：

• 卫星系统（satellite systems）是围绕一台机器（通常是较大的机器）的紧密的机群集，其中的卫星处理机分担中心处理机的进程负载，并将所有的系统调用提交给它处理，卫星系统的目的是为了增加系统的吞吐量，而且有可能使 UNIX 系统环境中的一个进程专用一台处理机。整个系统作为一个整体运行，除了有的时候卫星机在进程调度和本地存储器分配中具有自主性外，它不具备实际的自主性。这一点它不象其他的分布式系统模型。

•“纽卡斯尔”（Newcastle）分布式系统（此名是由一篇题为“纽卡斯尔连接”的文章得来的，见［Brownbridge 82］）通过在 C 库中识别远程文件的方法，允许存取远程文件。远程文件是由嵌在路径名中的特殊字符或放在文件系统根之前的特殊路径分量来标识的。这个方法可以在不改变内核的情况下实现，因此它比本章所叙述的其他方法要容易实现些，但其灵活性差。

• 完全透明的分布式系统允许用标准路径名来引用其他机器上的文件，由内核来识别它们是远程的文件。路径名在安装点处跨越机器边界，这一点很象在磁盘上它们在安装点处跨越文件系统。

本章考察每一种模型的体系结构，这里所介绍的并非基于某个特定的系统的实现，而是基于各种技术性文章中所刊出的信息。讨论中还假设低层协议模块和设备驱动程序负责管理诸如寻址、路由选择、流量控制以及差错检测和纠正方面的问题，因而假定每一种模型与其下面的网络无关。由于后几节中介绍的纽卡斯尔和透明模型都以与下节中介绍的卫星处理机系统的系统调用的例子相类似的方式工作，因此，这部分内容只详细地解释一次，而阐述其他模型的各节将集中讨论与它们相区别的最大的特征。

13.1　卫星处理机系统

图 13-2 示出了一个卫星处理机配置的体系结构。这样一个配置的目的是改善系统的吞吐量，而这是通过将进程从中心处理机下放，并在卫星处理机上执行它们达到的。每个卫星处理机除了那些需要用来与中心处理机通信的设备以外，都没有本地的外围设备。文件系统及所有的设备均配置在中心处理机上。在不失一般性的条件下，假设所有的用户进程都只运行在一个卫星处理机上，这就是说，进程并不在卫星处理机间迁移，一个进程一旦赋给某一处理机，它就留在那里直到它退出为止。卫星处理机内有一个简化了的操作系统，用来处理本地的系统调用、中断、存储管理、网络协议以及它用来与中心处理机通信的设备驱动程序。

当系统初启时，中心处理机上的内核将一个本地操作系统传送并装入每一个卫星处理机，这个操作系统一直在那里运行直到系统被关掉。卫星处理机上的每一个进程在中心处理机上都有一个与它相关联的存根（stub）进程（见［Birrell 84］），当卫星处理机上的一个进程要完成一次系统调用，这个系统调用需使用只有中心机才提供的服务时，卫星进程与中心处理机上它的存根进程通信，以便满足该要求。存根进程执行这个系统调用，并将结果送回到卫星处理机。类似于第 11 章中所叙述的，卫星进程和它的存根进程保持一种顾客-服务器的关系，即卫星进程是存根进程的顾客，而存根进程提供文件系统服务。这里，存根这个术语强调了远程服务进程仅为一个顾客进程服务。13.4节考虑为若干个顾客进程服务的服务

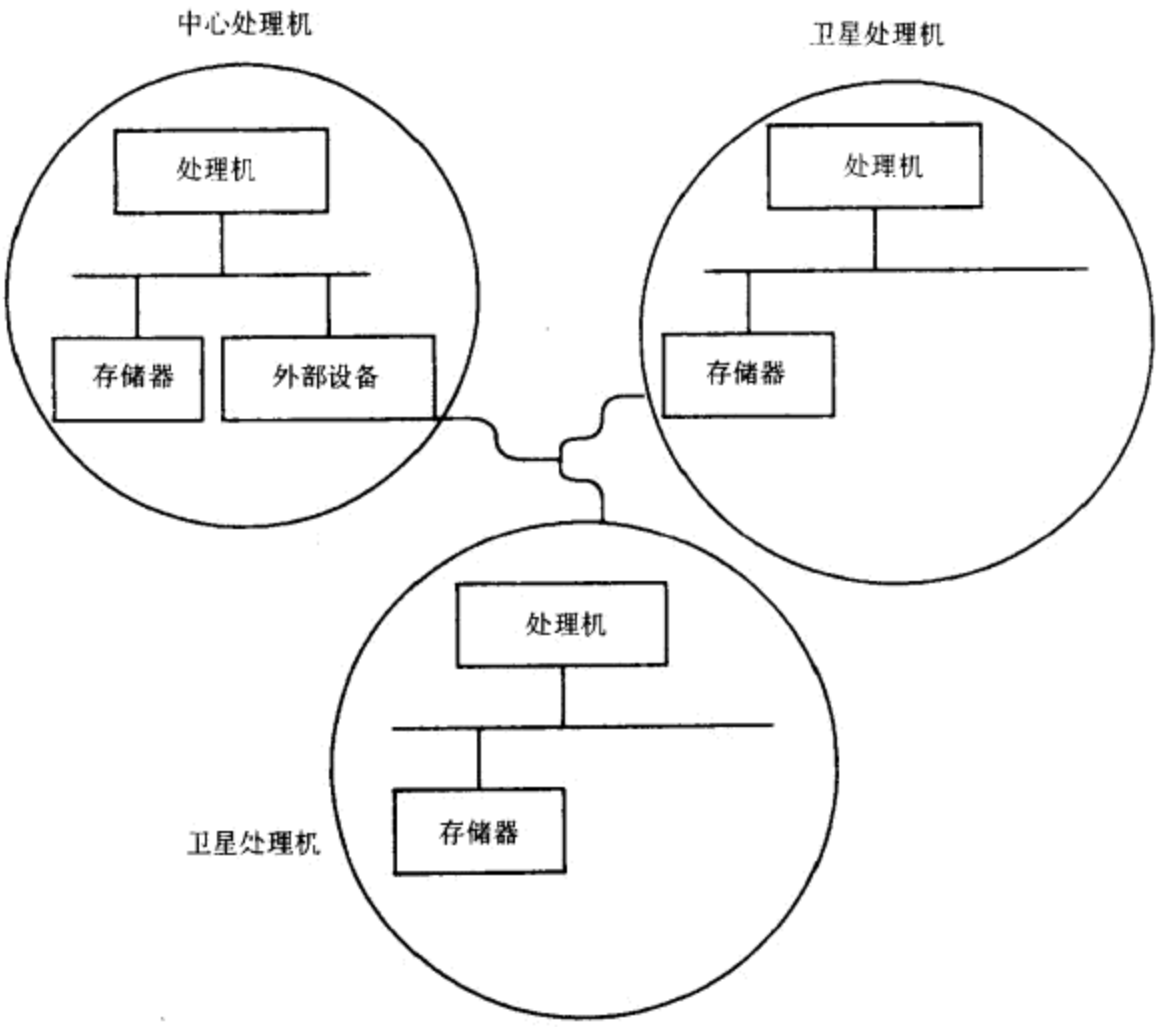


图 13-2 卫星处理机系统配置

进程。为方便起见，我们将运行于一个卫星处理机上的进程称为卫星进程（satellite process）。

当卫星进程要完成一次能在本地处理的系统调用时，该卫星处理机的内核不需要向存根进程发送请求。比如，为使一个进程得到更多的存储空间，它可以在本地执行系统调用sbrk。然而，如果它需要从中心处理机得到服务，例如当它想以系统调用 open 打开一个文件时，它将系统调用的参数以及该进程的环境经适当编码形成一个报文（message），并将此报文发送给存根进程(图13-3)。该报文是由规定存根进程应代表顾客进程完成的系统调用

报文格式

系统调用标记	系统调用参数	环境数据	..... 路径名或数据流 .....

响应

系统调用返回值	错误码	软中断信号序号	..... 数 据 流 .....

图 13-3 报文格式

类型的标记、该系统调用的参数以及诸如用户 ID、用户组 ID 这样的环境数据所组成的，这些对每个系统调用都可以是不同的。报文中其余的部分包含变长度的数据，比如文件路径名或对系统调用 write 要写的数据等。

存根进程一直等待着从卫星进程来的请求。当它接收到一个请求时，它对报文进行解码，确定它应当发出什么系统调用，然后执行该系统调用，并将系统调用的执行结果编码，形成一个对卫星进程的响应。这个响应中包含应返回给调用进程的作为该系统调用结果的返回值、一个用来报告存根进程中错误的错误码、一个软中断信号以及一个变长度数据数组，这个数组可以用来存放，比如说，从一个文件中读的数据。卫星进程在那个系统调用中睡眠，直到它接收到这一响应，然后进行解码，并将结果返回给用户。以上所说的就是处理系统调用的一般方案，本节的其余部分将详细地考察某些特殊的系统调用。

为了解释卫星系统是如何工作的，下面让我们来考虑以下几个系统调用：getppid，open，write，fork，exit 和 signal。系统调用 getppid 是很简单的，因为它只要求在卫星机和中心处理机间传送简单的请求和响应。卫星处理机上的内核形成一个有着指示这次系统调用是 getppid 的标记的报文，并将这一请求送往中心处理机。中心处理机上的存根进程从卫星处理机读这个报文，解码后得到系统调用的类型，执行系统调用 getppid，并找到其父进程标识号，然后，它形成一个响应，将其写回卫星进程。卫星进程则一直在对通信链路进行读，等待着这个响应。当卫星进程从存根进程接收到应答时，它将结果返回给原来请求系统调用 getppid 的进程。另一种方法是：如果卫星进程保留诸如父进程标识号这样的数据，它完全不需要与它的存根通信。

对于系统调用 open，卫星进程发送一个 open 的报文给存根进程，报文中含有文件名及其他参数。假定存根成功地完成了系统调用 open，它在中心处理机上分配了一个索引节点和文件表项，在它的 u 区用户文件描述符表中对一个表项赋值，并将文件描述符返回给卫星进程。在此期间，卫星进程一直在读通信链路，等待着从存根进程送回的响应。卫星进程没有记录这一打开文件的有关信息的内核数据结构，所以系统调用 open 返回的文件描述符是其存根进程的用户文件描述符表的索引。图 13-4 给出了系统调用 open 之后的结果。

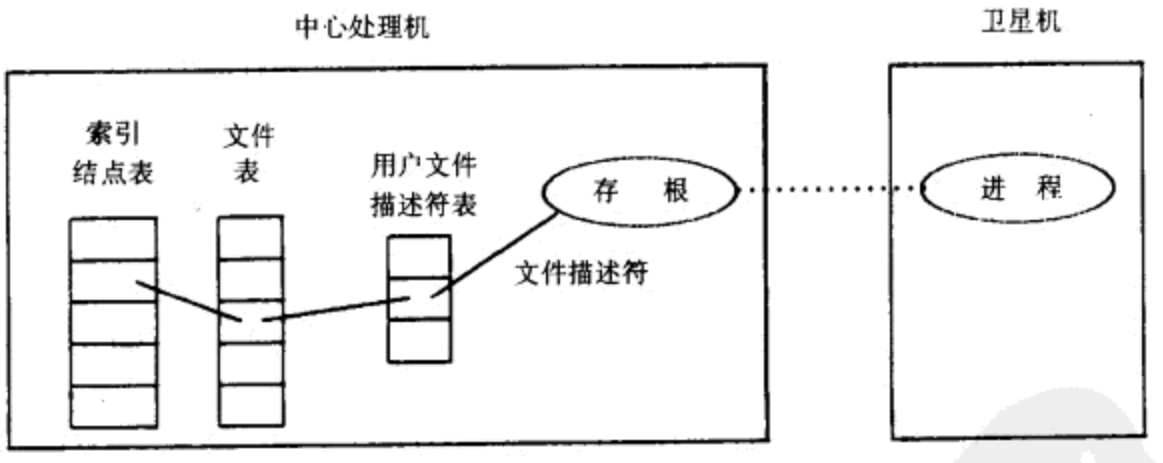


图 13-4　为卫星进程执行系统调用 open

对于系统调用 write，卫星进程形成一个报文，该报文中含有 write 标记、文件描述符以及数据字节数。然后，它从卫星进程用户空间拷贝数据，并写到通信链路上去。存根进程对报文进行解码，从通信链路上读数据，并将之写到适当的文件中去。为此，它要从文件描述符找到文件表项和索引节点，而所有这些全都在中心处理机上进行。当这一切都完成以后，

存根进程送给卫星进程一个确认报文，其中包括成功地写的字节数。系统调用 read 是类似的，如果存根进程并未返回所要求的字节数，例如在从终端或管道中读时，则它通知卫星进程。系统调用 read 和 write 均可能需要通过网络传送多个数据报文，这一点依赖于要传送的数据量及网络数据包的大小。

系统调用 fork 是中心处理机上唯一需要做内部修改的系统调用。当中心处理机上的一个进程执行系统调用 fork 时，内核选择一个卫星机来执行这个系统调用，向这个卫星机上的一个特殊的服务进程发送报文，通知它即将传送和装入一个进程了。假设该服务进程接受这个 fork 请求，它执行一次 fork 来产生一个新进程，将一个进程表项和 u 区初始化。中心处理机将执行系统调用 fork 的进程映象的拷贝传送并装入卫星处理机，以便改写在卫星处理机上刚产生的进程的地址空间，然后派生一个本地的存根进程与新的卫星进程通信，并且发送一个报文给该卫星处理机，来初始化新进程的程序计数器。这个存根进程（在中心处理机上）是执行系统调用 fork 进程的子进程，卫星进程从技术上说是服务进程的子进程，然而，逻辑上它是执行系统调用 fork 进程的子进程。在 fork 完成后，服务进程与这个子进程没有逻辑关系，服务进程的唯一目的是协助产生子进程。因为系统的紧密耦合性（卫星处理机没有自主性），卫星进程和存根进程具有相同的进程 ID。图 13-5 说明了这些进程之间的关系：实线表示父子关系；虚线表示对等进程间的通信关系。后者或者是父进程对服务进程；或者是子进程对它的存根。

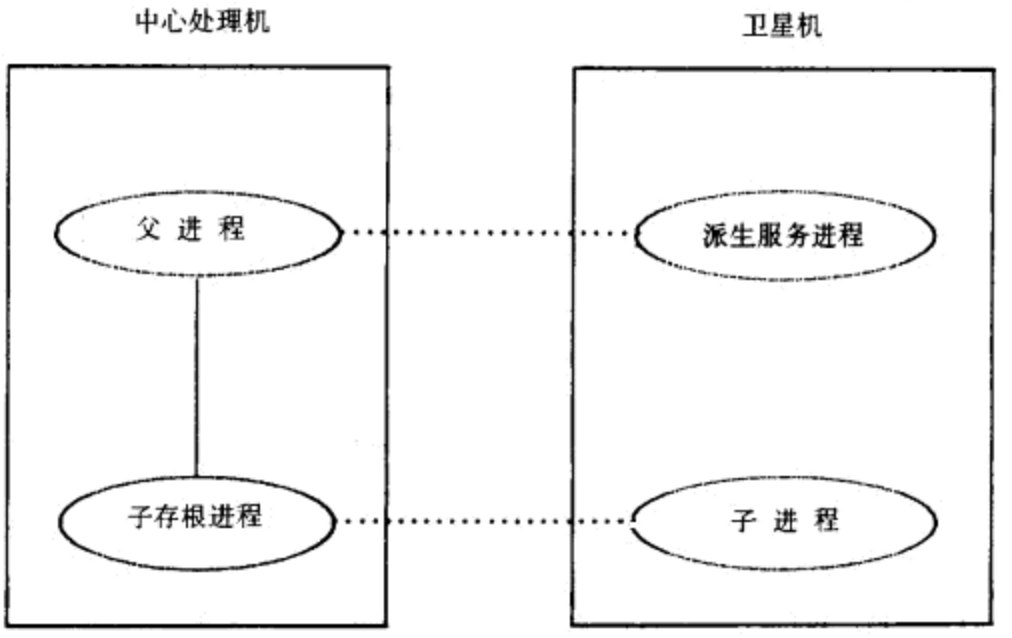


图 13-5　在中心处理机上派生子进程

当卫星处理机上的一个进程调用系统调用 fork 时，它发送一个报文给它在中心处理机上的存根进程，然后也经历类似的操作序列。该存根进程找到一个新的卫星处理机，并且这样来安排传送和装入老进程的映象：它发送报文给父卫星进程要求读其进程映象，而卫星进程以将其进程映象写到通信链路作为响应。这样，存根进程读到该进程的映象，并将它写到子卫星处理机中去。当这个卫星处理机完成了传送和装入时，存根进程执行系统调用 fork，在中心处理机上产生一个子存根进程，并将程序计数器的内容写到子卫星进程去，以使它知道从哪儿开始执行。很明显，如果子进程被分配在与它的父进程同一个卫星机上会获得性能的优化，但是此处的设计允许子进程在除了它们被派生的那个卫星机以外的其他卫星处理机

上运行。图 13-6 画的是在 fork 之后进程的关系。当卫星进程执行系统调用 exit 时，它向存根进程发送一个 exit 报文，因而存根进程也退出。然而，存根进程不能发起一个退出序列。

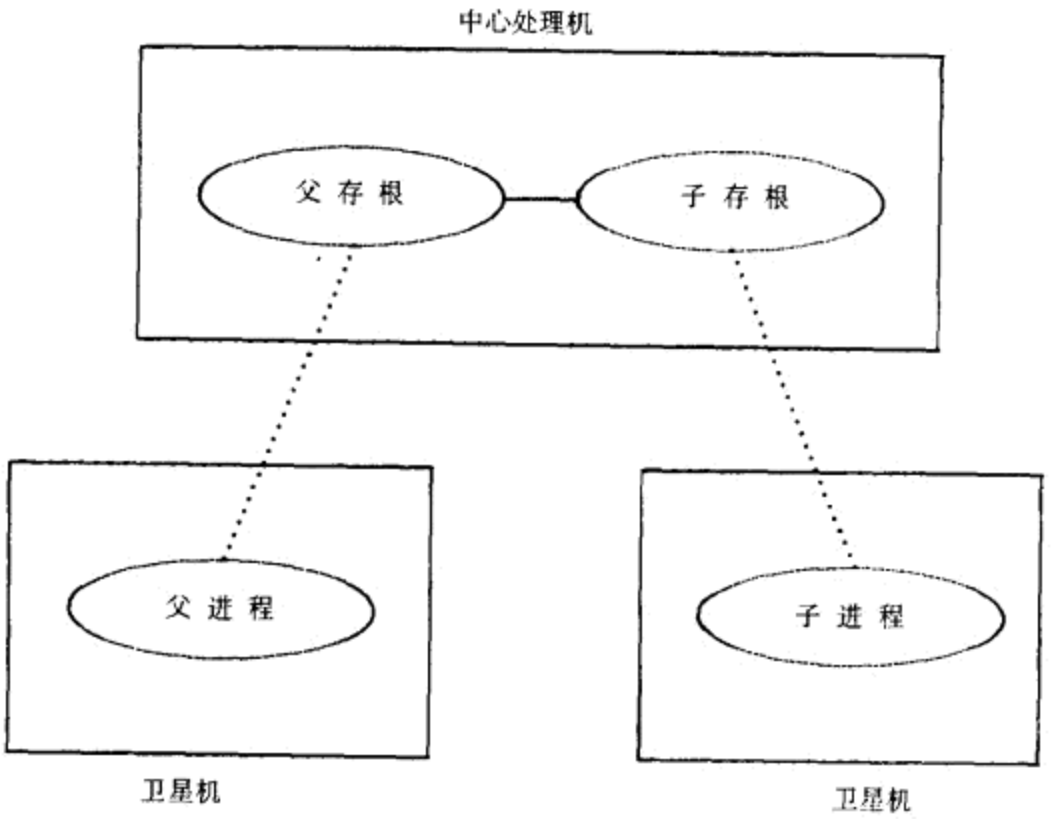


图 13-6　在卫星处理机上派生子进程

进程必须以与它在单一处理机上相同的方式来响应软中断信号，也就是说，根据它睡眠的优先级，或是在完成系统调用后再检查有无软中断信号，或是立即从睡眠中唤醒并异常地终止系统调用。由于存根进程为卫星进程处理系统调用，因而，它必须与卫星进程一致地响应软中断信号。如果在单一处理机上某软中断信号导致一个系统调用异常地终止，那么存根进程应以同样的方式动作。类似地，如果某信号导致一进程退出，那么卫星进程退出，并发送一个 exit 报文给存根进程，接着存根进程也自然地退出。

当某卫星进程执行系统调用 signal 时，它将通常处理这个系统调用的信息存储在本地表格中，并发送一个报文给存根进程，通知它是否应当忽略某特定软中断信号。下面我们将看到，对于存根进程，到底一个进程是捕俘一个信号还是执行缺省的操作是无区别的。进程对软中断信号的响应基于以下三个因素的组合（图 13-7）：该信号是否恰好在进程执行系统调用中间时出现；该进程是否已经调用过系统调用 signal 来忽略这一信号；以及该信号是在那个卫星处理机上或其他处理机上发出的。下面让我们来考查各种可能性。

假设当存根进程代表卫星进程执行一个系统调用时，卫星进程正在睡眠。如果软中断信号是从另一个处理机发出的，则存根进程比卫星进程早看到这个信号，这里有三种情况：

（1）如果存根进程睡眠在一个软中断信号出现时不能被唤醒的事件上，它就会完成这个系统调用，然后在发送给卫星进程的报文中给出适当的结果，并且指示它已接收到了哪一种信号。

（2）如果进程忽略这个信号，那么存根进程继续执行该系统调用的算法，不进行通常对被忽略信号的动作——执行算法 longjmp，以便从可中断的睡眠中醒来。当存根进程对卫星

进程应答时，它也不指示它曾经接收过一个信号。

（3）如果存根进程由于接收到一个信号执行了 longjmp 而从系统调用中返回，那么它通知卫星进程该系统调用被中断，并且指示该软中断信号的序号。

```
算法　sighandle　　/* 处理软中断信号的算法 */
输入：无
输出：无
{
    if(是存根进程)
    {
        if(忽视软中断信号)
            return;
        if(在系统调用中间)
            对存根进程设置软中断信号;
        else
            发送 signal 报文到卫星进程;
    }
    else　　/* 是卫星进程 */
    {
        /* 无论是否是在系统调用中间 */
        发送 signal 报文到存根进程;
    }
}
算法　satellite_end_of_syscall　/* 卫星系统调用结束 */
输入：无
输出：无
{
    if(系统调用被中断)
        发送报文到卫星进程说明被中断及软中断信号的序号;
    else　　/* 系统调用来被中断 */
        发送系统调用结果（包括指示有软中断信号到来的标志）;
}
```

图 13-7　在卫星系统上处理软中断信号的算法

卫星进程检查它接收的响应，看是否曾发生过软中断信号。若是，则在从系统调用返回之前以通常的方式处理这些信号。用这样的方法，使一个进程准确地像它在单一处理机时那样表现——或者它不从内核返回就退出；或者它调用用户软中断信号处理程序；或者它忽略该信号并从系统调用中返回。

举例来说，设想一个卫星进程从一个连接到中心处理机的终端上读信息，当存根进程执行系统调用 read 时它睡眠（图 13-8）。如果此时用户按下 break 键，则存根进程所在处理机的内核送一 SIGINT 软中断信号给存根进程。如果存根进程正在睡眠，等待着用户输入，则它立即被唤醒，并终止系统调用 read 的执行。在它给卫星进程的响应中，存根进程设置一错误码（指示系统调用被中断）以及 SIGINT 信号的序号。卫星进程检查这个响应，因为该

报文表明已有一个 SIGINT 信号被发送了，所以它将这个信号通知它自己。这样，在从系统调用 read 返回以前，卫星处理机的内核检查是否曾有信号发生过，就会发现由存根进程送回的 SIGINT 信号，因而以通常的方式处理它。如果这个 SIGINT 信号造成卫星进程退出的话，那么系统调用 exit 负责杀死存根进程。如果卫星进程正在捕俘 SIGINT 信号，那么它调用用户的信号捕俘程序，以后在从系统调用返回时，给用户返回一个错误。然而，如果存根

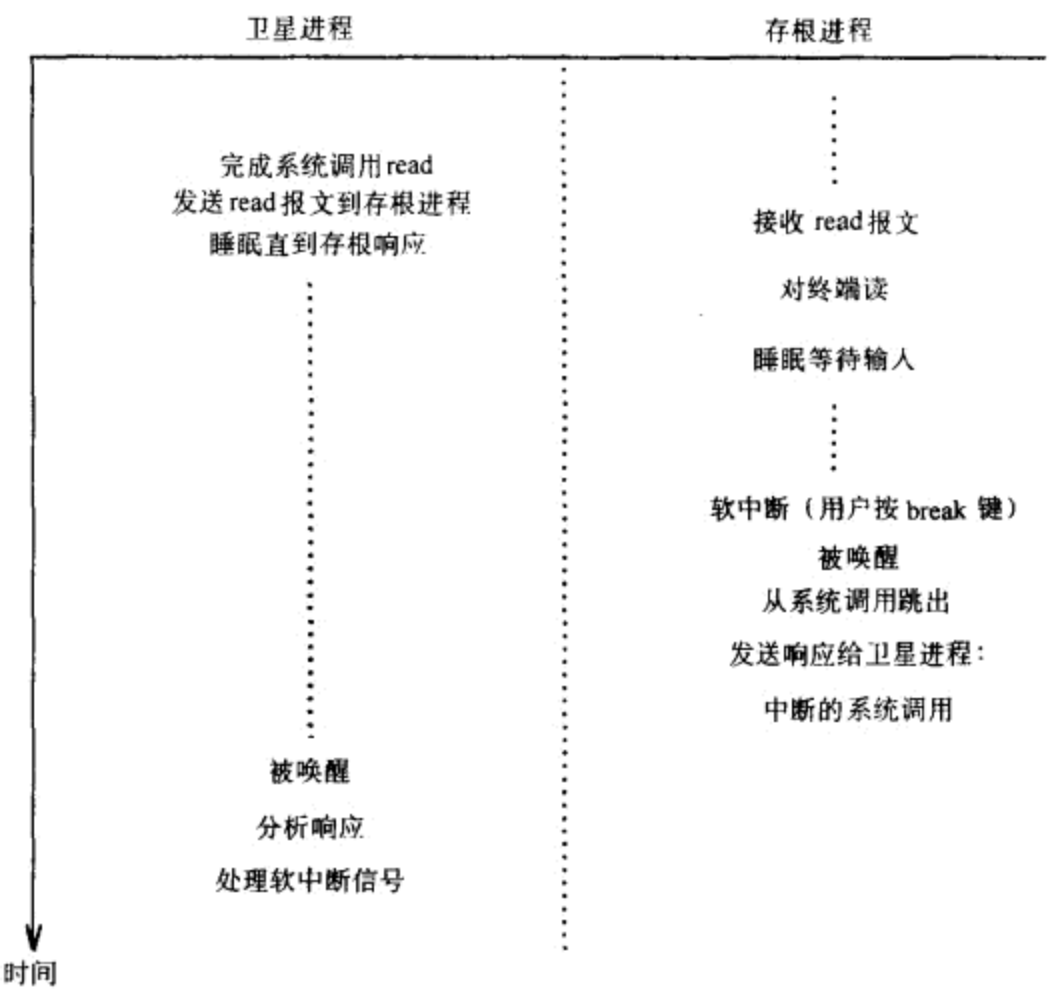


图 13-8　在系统调用期间中断

进程代表卫星进程执行的是系统调用 stat 时，它接收到软中断信号后并不停止该系统调用的执行（由于 stat 不需要无限期地等待一个资源，故 stat 是保证可以从所有的睡眠中被唤醒的），存根进程完成这个系统调用，并将软中断信号的序号送回给卫星进程，卫星进程通知它自己这一信号，并且当从系统调用返回时就会发现这个信号。

如果进程正处在系统调用执行中间，在卫星处理机上产生一个软中断信号，卫星进程无法知道存根进程是很快返回还是永远睡眠下去，那么，卫星进程发送一个特殊的报文给存根进程，通知它该信号的出现。中心处理机的内核读到这个报文，并将该信号送给存根进程。这时，存根进程就如前一段中所描述的那样反应了。卫星进程不能直接地把报文送给存根进程，因为存根进程正处在一个系统调用中间，并且它没有正在读通信链路。中心处理机的内核辩认这个特殊的报文，并将该信号通知相应的存根进程。

让我们再用上面解释的 read 的例复述一下。卫星进程不知道存根进程是正在等待着输入，还是正在做其他的处理，所以它送给存根进程一个 signal 报文。如果存根进程正在一可被中断的优先级上睡眠，则它立即被唤醒并终止该系统调用，否则，它正常地完成该系统调用。

最后，让我们考虑当一个进程不在系统调用中间时一个软中断信号到来时的情形。如果信号是在另一个处理机上产生的，则存根进程将首先收到这个信号，而且，不管卫星进程想如何

处理这个信号，它都送给卫星进程一个特殊的 signal 报文。卫星机的内核对这个报文进行解码，并将此信号发给进程，由进程以通常的方式来响应它。如果这个软中断信号就在该卫星处理机上产生，则卫星进程进行常规的处理，并且也不需要与存根进程进行特殊的通信。

当卫星进程发给其他进程一个软中断信号时，它为系统调用 kill 形成一个报文，并将它发送到存根进程，存根进程再在本地执行系统调用 kill。这时，在其他卫星处理机上的某些进程也应收到这个信号，它们的存根接收这个信号，并做上面所叙述的那些处理。

13.2　纽卡斯尔连接

前节研究的是一种紧密耦合的系统配置，在这种系统中，卫星处理机中所有的文件子系统调用都被陷入和转送到一个远程的处理机去执行。下面，我们将视野扩展到更加松散耦合的系统，在这种系统中每台机器都要存取其他机器上的文件。比如，在一个由若干台个人计算机和工作站构成的网络中，用户可能想要存取存储在一台大型机上的文件。下面两节将考虑这样的一些系统的配置，其中本地系统能执行所有的系统调用，然而文件子系统的系统调用还可以存取其他机器上的文件。

这些系统通常用以下两种方法中之一来识别远程的文件。有些系统将一特殊字符嵌入文件路径名：在该特殊字符之前的分量名用来识别一台机器，路径名的其余部分识别该机的一个文件。例如，路径名

```
"sftig!/fs1/mjb/rje"
```

用来标识在“sftig”机器上的文件“/fs1/mjb/rje”。这种文件命名机构沿用了在 UNIX 系统间传送文件的 uucp 程序的习惯用法。另外一种命名机构是在路径名前附加一个特殊的前辍，如

```
/../sftig/fs1/mjb/rje
```

其中的“/..”通知分析程序这是引用一个远程的文件，第二个分量名给出远程机器名。后面的这种命名法使用了 UNIX 系统习惯的文件名语法，因此用户软件不需要进行变换来处理前一种方案中所出现的“不规则构造名”问题（见 [Pike 85]）。

本节的其余部分描述一个仿照纽卡斯尔连接的系统。在这个系统中，内核不参与确定一个文件是否是远程的，相反地，提供内核接口的 C 库程序检查一次文件存取是远程的，并完成适当的动作。对上述两种命名法，C 库程序分析路径名的第一个分量，决定一个文件是远程的文件。当然，这个方法偏离了原来库程序不分析路径名的通常的实现方法。图 13-9 示出了文件服务请求是怎样形成的。如果文件名是本地的，则本地内核以常规的方式处理这次请求。然而，让我们考虑以下系统调用的执行：

```
open ( "/../sftig/fs1/mjb/rje/file", O_RDONLY);
```

C 库程序中的 open 程序分析路径名中前两个分量，并辩认出该文件应在远程机器“sftig”上。它保持一个数据结构，记录该进程过去是否与“sftig”机已建立了通信联系。若尚未建立联系，它与远程机器上的文件服务者进程建立一个通信的链路。当一个进程第一次发出远程的请求时，远程服务器确认一下该请求的合法性，必要时映射用户 ID 和用户组 ID，然后

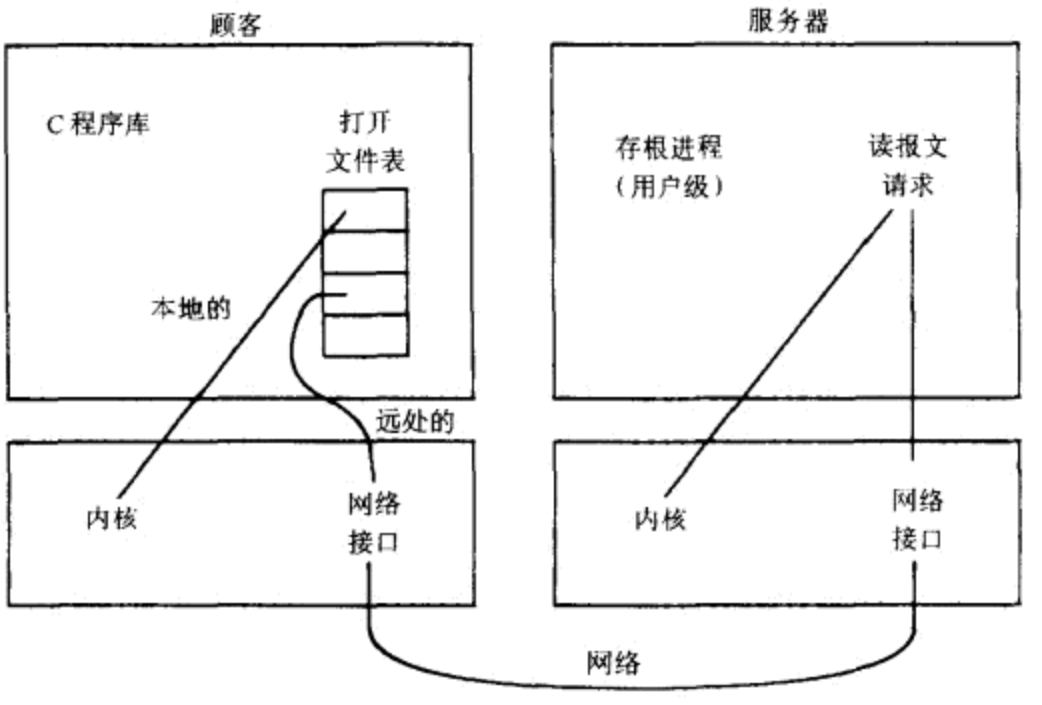


图 13-9　文件服务请求的形成

产生一个存根进程，作为该顾客进程的代理。

执行顾客进程请求的存根进程对文件应当有与该顾客用户在远程机器上对这些文件同样的存取权限，也就是说，用户“mjb”应当以控制存取本地文件同样的许可权来存取远程的文件。遗憾的是，“mjb”的用户 ID 在远处机器上可能是一个不同的用户 ID。为此，或者各台机器的系统管理员必须为网络上所有的用户分配一个唯一的标识号，或者在必须请求网络服务时做用户 ID 的变换。若做不到上述两点，存根进程就应以“其他用户”的许可权在远程机器上执行了。

对远程文件允许使用超级用户的存取权限是一个更加棘手的情况。从一方面说，一个超级用户顾客在远程机器上不该具有超级用户的权利，因为如果是这样的话，一个用户在远程机器上就可能绕过安全性的措施。但从另一方面说，当没有超级用户的能力时，各种程序可能就不能运行。举例来说，让我们回忆第 7 章中用来创建一个新目录的程序 mkdir，它是作为一个有超级用户权限的 setuid 程序运行的。这样，如果远程系统不能辩认出远程的超级用户的许可权，它就会不允许一个顾客创建一个新目录。因此，创建一个远程的目录问题成为实现系统调用 mkdir 的强有力的理由，而该系统调用应自动地建立所有必要的目录联结关系。然而，作为超级用户来执行存取远程文件的 setuid 程序则仍然是一个必须处理的普遍问题。这个问题的最佳解决方法是：为远程的超级用户访问文件提供一组分开的存取权限。然而，遗憾的是，这会要求改变磁盘索引结点的数据结构，用来保存新的许可权字段，而这将在现存系统中导致太多的混乱。

当系统调用 open 成功地返回时，本地的库程序在用户级库数据结构中做适当的记号，其中包括网络地址、存根进程 ID、存根文件描述符和其他有关的信息。系统调用 read 和 write 的库程序要检验该文件描述符，辩认原来的文件引用是否是对远程文件的，若是，给存根发送一个报文。对所有需要在远程机器上提供服务的系统调用，顾客进程与其存根进程通信。如果一个进程对远程机器上两个文件进行存取，则它使用一个存根。然而如果它在两个远程机器上存取文件，则它要使用两个存根，每台机器上一个。当通过存根执行一个系统调用时，该进程要形成一个包括有系统调用号、路径名和其他有关信息的报文，类似于我们

在卫星处理机中所叙述的报文类型。

操纵当前目录是更加复杂的问题。当一进程改变当前目录到一远程的目录时，库程序发送一报文到存根，由存根改变它的当前目录，而库程序要记住当前目录是远程的目录。对于所有不以“斜杠”字符开始的路径名，库程序要将此路径名送往远程机器，在远程机器上由存根进程从当前目录来判定其路径名。如果当前目录就是本地的，则库程序简单地将该路径名传递给内核。对系统调用 chroot 要改变到一个远程的目录的处理也是类似的，然而本地的内核可能未发现该进程曾执行过 chroot。更严格地说，一个进程能忽略一个改变到远程目录的 chroot，因为只有库程序对它保留一个记录。习题 13.10 让你考虑在一个安装点上“..”的情形。

当一进程派生子进程时，库程序 fork 发送给每一个存根一个 fork 报文，这些存根进程派生子进程，并将这些子进程的标识号送回给父顾客进程。然后，顾客进程请求（内核）执行系统调用 fork，当它返回到子进程时，库程序存储与子存根进程有关的地址信息，最后，本地子进程与远程的子存根进程进行对话。对系统调用 fork 这样的处理使得这些存根进程易于掌握已打开文件和当前目录的线索。当一个有远程文件的进程退出时，库程序发送给远程的存根一个报文，后者以退出作为响应。后面有几个习题将详细地探讨系统调用 exec 和 exit。

纽卡斯尔设计的优点是：进程可以透明地存取远程的文件，而不需要改变内核。然而这个设计也有些缺点。比如，系统的性能可能降低。由于较大的 C 库。每个进程甚至在它并不引用远程文件时也要占用较多的存储空间，而且库程序重复内核的功能，并占用更多的空间。较大的进程在系统调用 exec 时要花费较长的时间来开始操作，并且导致对存储器的较多的竞争，引起在一个系统上更大程度地换页和对换操作。本地请求由于要花费较长的时间才能进入内核，因而执行得更慢，远程请求也可能很慢，因为它们必须通过网络发送请求，因而在用户级需要做更多的处理。这些额外的用户级的处理造成了更多的进程上下文切换、换页和对换的机会。最后，为了能对远程文件进行存取，程序必须以新的 C 库重新编译，老的程序及厂商提供的目标模块除非重新编译，否则不能运行。下一节中将叙述的方案没有这些缺点。

13.3　透明型分布式文件系统

这里，透明分布这个术语的含义是：一个机器上的用户可以在不意识到他们跨越了机器边界的情况下存取另一台机器上的文件，这就类似于在一台机器上跨越一个安装点从一个文件系统到另一个文件系统。用来存取远程机器上的文件的路径名看上去和存取本地文件的路径名一样，其中不包含什么区分符。图 13-10 示出了机器 B 上的目录“/usr/src”被安装在机器 A 的目录“/usr/src”上的一种配置情形，这种配置便于那些想要共享一份系统原代码——通常被安排在“/usr/src”目录下——的系统。在机器 A 上的用户可以用正规的文件名语法存取机器 B 上的文件，比如文件“/usr/src/cmd/login.c”，由内核内部地决定一个文件是远程的还是本地的。机器 B 上的用户能存取其本地文件，他们并不知道机器 A 上的用户也能存取这些文件，但他们不能存取机器 A 上的文件。当然其他配置情况也是可能的，如全部的远程系统都被安装在本地系统的根目录下，使得用户可以存取所有系统上的所有的文件。

由于安装本地文件系统与提供存取远程文件系统的能力有类似性，系统调用 mount 被

修改为可安装远程的文件系统。内核保留一个扩展了的安装表，当执行一个远程的系统调用 mount 时，内核建立一个与远程机器的网络连接，并将该连接的信息存储在安装表中。

对于路径名中含有“..”（点-点）的情形，一个有趣的问题出现了：如果进程改变目录到了一个远程的文件系统，这之后使用“..”应使进程回到本地文件系统，而不是允许它存取远程的被安装的目录以上的那些文件。让我们再回到图 13-10 的例，如果一个在机器 A 上的进程的当前目录是目录“/usr/src/cmd”（在远程），它执行

```
cd../../..
```

时，它的新的当前目录应是机器 A 的根目录，而非机器 B 的根目录。远程内核中的算法 namei 检验所有的“..”序列，看调用的进程是否是一个顾客进程的代理进程，若是，检查一下当前工作目录，看该顾客进程是否把该目录当成是一个远程安装的文件系统的根目录。

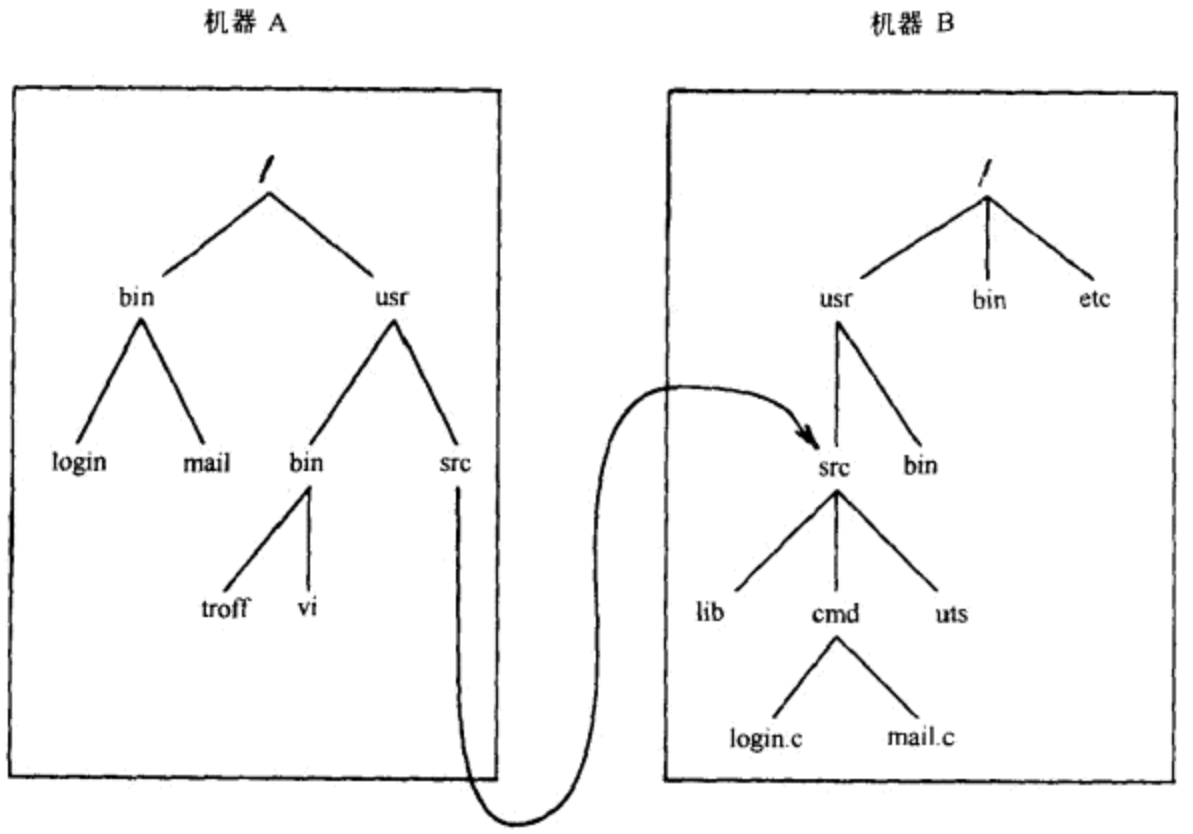


图 13-10　在远处安装后的文件系统

与远程机器的通信采取两种形式之一：远程过程调用或远程系统调用。在使用远程过程调用的系统中，每个处理索引节点的内核辩认一下某特定的索引节点是否指的是一个远程的文件，若是，发送给远程机器一个报文以便完成一个特定的索引节点的操作。这种机制与第 5 章最后所说明的抽象文件系统自然地吻合。这样，一个存取远程文件的系统调用会引起若干个报文穿过网络，这依赖于其中参与了多少次内部的索引节点操作。由于网络等待时间的存在，造成了相应长的响应时间。作为一种极端情形，远程的操作包含对索引节点的锁标志、引用数等的操作。因此，对上述纯理论的模型的各种优化方案已经实现，诸如将几个索引节点的操作结合到一个单一的报文中去、高速缓冲重要数据等（见［Sandberg 85］）。

考虑一个进程要打开远程文件“/usr/src/cmd/login.c”的情形，其中 src 是安装点。当内核在算法 namei 和 iget 中分析该路径名时，它发现该文件是远程文件，因而对远程机器发

送一个请求，以便送回一个上锁的索引节点。当收到成功的响应时，本地内核分配一个相应于远程文件的内存索引节点，然后它通过发送给远程机器的另一个报文来检验文件的属性，看是否具有必要的许可权（例如允许读）。它接着执行第 5 章中给出的算法 open，必要时向远程机器发送报文，直到它完成该算法，并对该索引节点解锁为止。图 13-11 给出了在系统调用 open 结束时内核数据结构间的关系。

对于系统调用 rcad，顾客所在机器的内核对本地索引节点上锁，发送报文去锁住远程的索引节点，发送报文以便读数据，将数据拷贝到本地存储器中，发送报文去对远程索引节点解锁，最后对本地索引节点解锁。这个机制与现存的单一处理机内核编码的语义是一致的，然而，频繁地使用网络（每个系统调用可能若干次）会损害性能。为减少网络交通量，可以

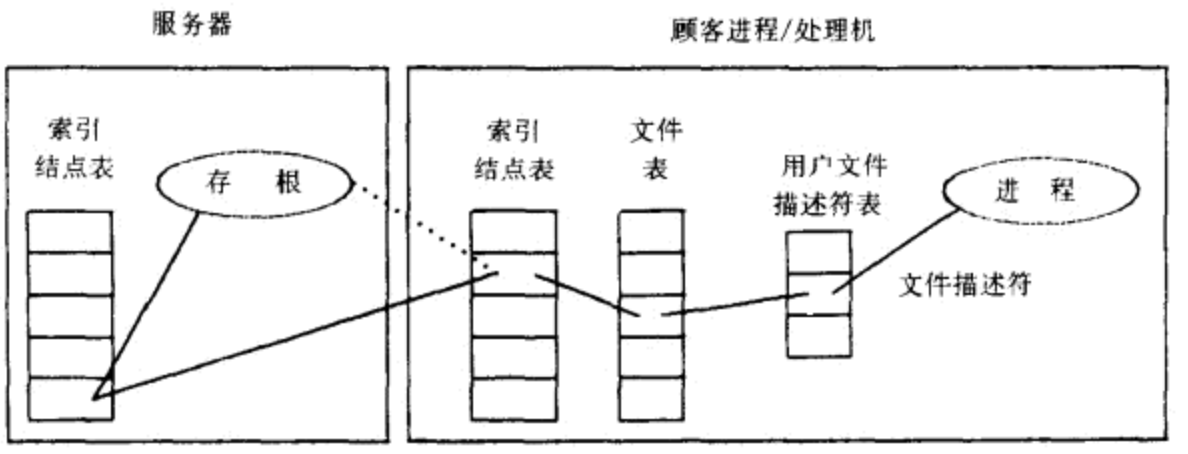


图 13-11　打开一个远处的文件

将若干个操作结合到一个报文中去。在 read 这个例中，顾客可以仅送一个“read”报文给服务者，而服务器知道它在完成这个读操作过程中必须对它的索引节点先上锁、后解锁。如以前提到的，用远程的高速缓冲来实现可以进一步减少网络交通量，然而必须十分小心以保持文件系统调用的语义。

在以远程系统调用设计的方案中，当本地的内核像前面那样的辨认出一个系统调用指的是一远程的文件时，将该系统调用的参数发送到远程系统，由远程系统执行这个系统调用，并将结果返回给顾客。顾客所在的机器接收到远程系统调用的结果后，通过 longjmp 从系统调用返回。多数系统调用能够仅用一个网络报文就能执行，造成相当好的系统响应，然而有几个内核操作这种模型对它们不适用。举例来说，内核为一进程接收到各种各样软中断信号时建立一个“内存映象”文件（第 7 章）。创建一个内存映象文件并不是相应于一次系统调用，而是要求几次索引节点的操作，如建立一个文件、检验存取权限以及完成几次写操作。

对于系统调用 open，远程系统调用的报文由路径名的后部（路径名中标识远程的路径名分量之后的字符串）和各种标志位组成。让我们重复使用早先使用过的那个例，即一个进程欲打开文件“/usr/src/cmd/login.c”，内核将路径名“cmd/login.c”送往远程机器。报文中也含有某些用于标识的信息，如用户 ID 和用户组 ID，这些信息对决定该进程在远程机器上文件存取能力是必需的。当远程机器给出响应，说明系统调用 open 成功时，本地内核分配一个未用的、本地的内存索引节点，将它标志为“远程的”，保存用来识别远程机器及远程索引节点的信息，并以通常的方式分配一个新文件表表项。本地机器上的索引节点是远程机器上的实际索引节点的一个虚构物，这就造成与远程过程调用模型（图 13-11）同样的结

构。当一个进程以其文件描述符发出一个系统调用，存取一个远程的文件时，本地内核检查它的索引节点，辨认出这个文件是远程的文件，形成一个报文来包装这个系统调用，并把报文发送到远程机器。报文中含有远程索引节点的索引值，因而存根进程能识别那个远程的文件。

对所有的系统调用，本地内核可以执行一些特殊的管理响应的编码，并最终通过算法 longjmp 从系统调用中返回，因为这之后的为单一处理机设计的本地处理可能是不相干的。因此，为支持远程系统调用模型，内核算法的语义可能改变了。然而，网络交通量保持在最小值，允许系统尽可能快的响应。

13.4　无存根进程的透明分布式模型

在透明型分布式系统模型中使用存根进程能使远程系统易于掌握那些远程文件的线索，但是，远程系统上的进程表却会因存在众多的、且大部分时间是空闲的存根进程而被搞得混乱。其他可选的方案是：在远程机器上使用特殊的服务进程来处理远程的请求（见［Sandberg 85］和［Cole 85］）。远程系统中设有一个服务进程池，当远程请求到来时，临时地分配它们来处理每个远程的请求。某个服务进程并不记住各系统调用的用户上下文（如用户 ID），因为它可能为若干个进程处理系统调用。结果是：从顾客进程发出的每个报文中都必须包含有关它的环境的数据，如用户 ID、当前目录、软中断信号的处理等。而存根进程是在建立时刻或在系统调用的正常执行过程中取得这些数据的。

当某进程执行系统调用 open 打开一个远程的文件时，远程的内核分配一个索引节点以便以后引用该文件。本地的机器在用户文件描述符表、文件表和索引节点表中也有通常的打开文件所具有的那些表项，而且在索引节点表项中还要标识远程机器及索引节点。对于需要使用文件描述符的系统调用，如 read，内核发送一个报文，在报文中标识以前分配的远程索引节点和传递某些专用的信息，例如用户 ID、文件尺寸的最大值等。当远程机器分派了一个服务进程时，它与顾客进程的通信就类似于我们以前所描述的，只是顾客与服务进程之间的连接仅在系统调用期间存在。

使用服务进程而非存根进程使得流量控制处理、软中断信号处理和存取远程的设备变得更加困难。如果一台远程机器收到一大批从许多机器来的请求，若它没有足够多的服务进程，则它必须将请求排入队列，这样，就需要一个比其下面的网络已经提供的协议更高层次的协议。然而，在存根模型中，一个存根进程不会被请求所淹没，因为对顾客的所有事务处理都是同步的，也就是说，一个顾客最多有一个待处理的请求。

用服务进程模型来处理使一个系统调用中断的软中断信号也是十分复杂的，因为远程机器必须正确地找到正在执行这个系统调用的服务进程。甚至可能那个系统调用仍在等待服务，如果所有的服务进程都在忙碌。类似地，若服务进程将系统调用的结果返回给调用进程时，响应方传递的 signal 报文正在网络的途中，就会发生竞争条件。每个报文必须要加上标签，使远程的系统能对它进行定位，并在必要时中断服务进程。然而，使用存根进程模型时，正在为顾客的系统调用服务的进程是自动地被识别的，因而当软中断信号到来时，很容易确定是否它已经完成了一个系统调用的处理。

最后，如果一个进程发出了一个引起服务进程无限期地睡眠的系统调用（如从一个远程的终端读信息），则该服务进程不能再处理其他请求了，从效果上看，它被从服务进程池中

删除了。如果许多进程都存取远程的设备，而且服务进程的数目又有一个上限，则这可能是一个严重的瓶颈。当使用存根进程模型时，这种情况不可能发生，因为存根进程是按每个顾客进程来分配的，习题 13.14 探讨了使用服务进程来存取远程设备中的另一个问题。

尽管使用进程的存根有很多优点，但它对进程表表项的要求太高，以致在实践中多数的系统使用服务进程池来处理远程的请求。

13.5　本章小结

本章叙述了使进程能存取存储在远程机器上的文件的三种方案，把远程的文件系统当作本地文件系统的一种扩展。图 13-12 说明了三者的体系结构之间的区别。这些系统与前一章讨论的多处理机系统不同，因为这些处理机不共享物理存储器。卫星处理机方案由一组共享

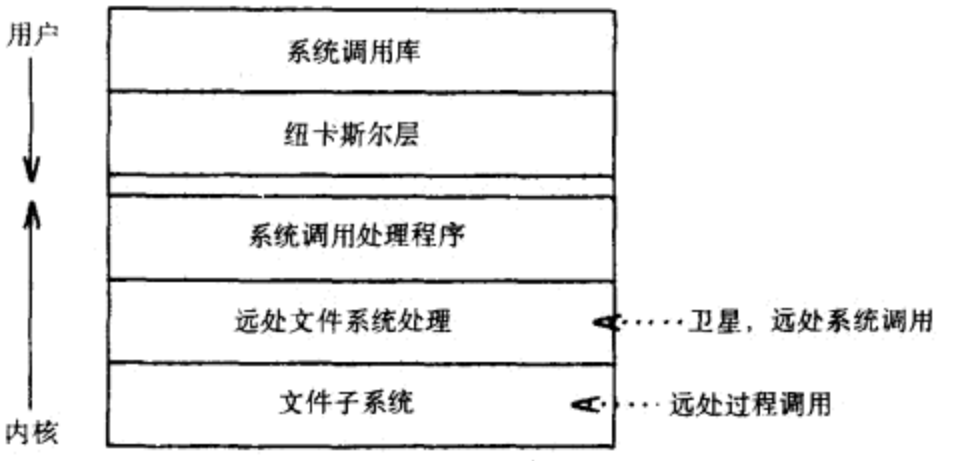


图 13-12　远处文件存取的概念性核心层次

中心处理机文件资源的紧密耦合的处理机组成。纽卡斯尔连接给出一种透明的远程文件存取的表观，然而，远程存取是由 C 库的一种特殊的实现提供的，而不是由内核提供的。其结果是，为使用纽卡斯尔连接，程序必须重新编译，这常常是一个严重的缺陷。另外，远程的文件是由标识存储该文件的远程机器的特殊字符序列来标志的，这又成为一个限制可移植性的因素。

透明型分布式系统使用系统调用 mount 的变体来提供对远程文件系统的存取，这一点很象通常的系统调用 mount，将本地文件系统扩展到新安装的磁盘上。本地系统的索引节点指出它们指的是远程文件，本地内核向远程内核发送报文，说明内核的算法（哪一个系统调用）、它的参数及远程的索引节点。有两种支持远程的透明型分布式操作的设计方法：其一是远程过程调用模型，其中的报文指示远程机器执行索引节点操作；其二是远程系统调用模型，其中的报文指示远程机器执行系统调用。最后，本章考察了用存根进程或一个通用的服务进程池来对远程的请求进行服务时所存在的问题。

13.6　习题

* 1. 叙述在卫星处理机系统上系统调用 exit 的一种实现方案。这与作为收到一个不能捕俘的软中断信号的结果——进程退出的情形有何不同？内核应如何转储“内存映象”文件？

2. 进程是不能忽略 SIGKILL 软中断信号的，说明在卫星系统上当一个进程收到这个信号时会发生什么情况。

*3. 叙述在卫星处理机系统上实现系统调用 exec 的一种方案。

*4. 中心处理机为了均衡执行的负载应如何分配进程到卫星处理机上去?

*5. 如果一卫星处理机没有足够大的存储空间来接受向它传送并装入的进程映象时会发生什么情况? 它应如何通过网络来处理对换和换页?

6. 考虑一个系统，它允许用识别路径名的特殊的前序来存取远程服务机上的文件。假设一进程执行

```
execl ( "/../sftig/bin/sh", "sh", 0);
```

时，可执行的文件是在远程机器上，而它应在本地机器上执行。说明为执行这个 execl，本地系统如何将远程的可执行文件取回本地系统。

7. 如果系统管理员想往一个纽卡斯尔系统中增加新机器，通知 C 库模块的最好方法是什么?

*8. 在执行系统调用 exec 过程中，内核重写一进程的地址空间，其中包括在纽卡斯尔实现方案中用来记录所引用的远程文件 C 库中的表格。在 exec 之后，该进程应当仍能以这些文件的老文件描述符来存取它们。试说明一种实现方法。

*9. 如第 13.2 节所述，在纽卡斯尔系统上执行系统调用 exit 造成了发送一个报文给存根进程，该报文导致它的退出。而这是在 C 库一级完成的。如果本地进程接收到一个软中断信号，引起它从内核中退出时，会发生什么情况?

*10. 在纽卡斯尔系统中，远程文件是由特殊的前序标志的。系统应如何允许一用户使用".."(父目录) 分量来在一个远程的安装点上溯一层目录?

11. 回忆第 7 章中关于各种各样的软中断信号导致了一个进程在其当前目录下转储内存映象文件。如果当前目录是在远程文件系统中会发生什么情况? 在纽卡斯尔系统上会发生什么情况?

*12. 若某人在远处机器上杀死了所有的存根或服务者进程，本地进程将如何得到这一好消息?

*13. 讨论在透明型分布式系统中，系统调用 link 和 exec 的实现方法。其中 link 可能有两个远程的路径名，exec 有若干个内部的读操作。考虑两种设计方案：远程过程调用和远程系统调用。

*14. 当一服务进程（非存根进程）存取一个设备时，它可能会睡眠直到设备驱动程序唤醒它。假设有固定数目的服务进程，想象一种情况：由于所有的服务者都睡眠在一设备驱动程序中，一个系统将不能满足从本地机器来的任何更多的请求了。试设计一种安全的方案，其中并非所有的服务进程可以因等待设备的 I/O 而睡眠。这样，系统调用不会因所有的服务进程当前都繁忙而失败返回。

*15. 当一用户注册到一个系统时，终端行规则程序将该终端是一个控制终端的信息保存起来，通知该进程组。用这样的方法，当用户在终端按 break 键时，这些进程接收到 SIGINT 软中断信号。考虑一种系统的配置，其中所有的终端从物理上说是连接到一台机器的，但用户从逻辑上说可以注册到其他机器（图 13-13），更准确地说，某系统为一个远程的终端派生一个 getty 进程。如果有一服务进程池处理远程系统调用，则一个服务进程在设备打开程序中睡眠，等待连接的到来。当服务进程完成系统调用 open 时，它回到进程池中去，断开它与终端的连接。如果用户按 break 键，该 SIGINT 软中断信号是怎样被送到在顾客机上执行的进程组中的进程去的?

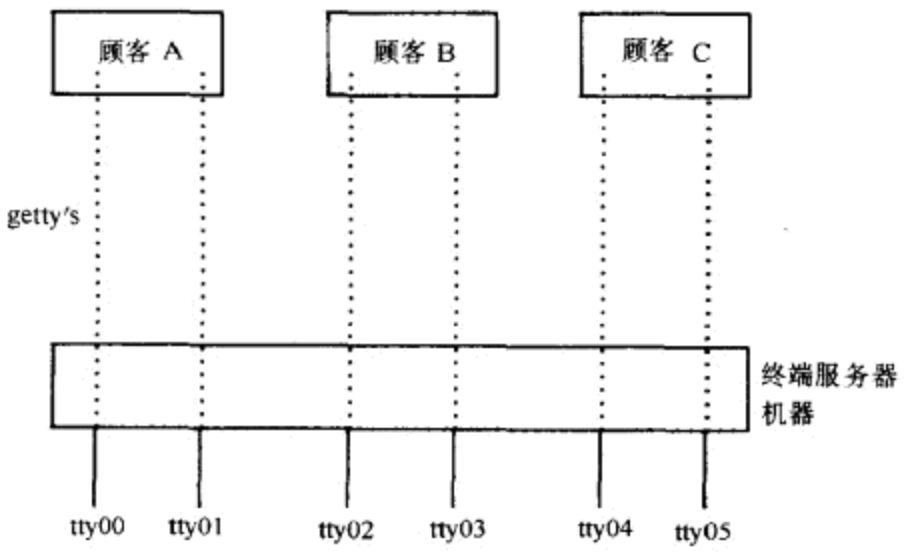


图 13-13　终端服务器的一种配置方案

* 16. 共享存储区的特征本质上说是一个本地机器的操作。从逻辑上说，不同机器上的进程对一块公共的物理存储器进行存取是可能的，不管该存储器是本地的还是远程的。试说明一种实现方案。

* 17. 第 9 章考察的请求调页和对换的算法假设使用了一台本地的对换设备。为了支持远程的对换设备，对这些算法必须作些什么修改？

* 18. 设想一台远程的机器故障（或网络故障），并且局域网协议能发现这一事实。试为本地系统设计一个通过向一远程的服务系统发出请求来恢复的机制。反过来，为一个丧失它与顾客机的连接的服务系统设计恢复机制。

* 19. 当一进程存取一远程文件时，路径名可能跨越若干台机器逐渐展开，至完全被判定止。以路径名“/usr/src/uts/3b2/os”为例，“/usr”可能在机器 A 上，机器 B 的根目录可能被安装在“/usr/src”，机器 C 的根目录又可能被安装在“/usr/src/uts/3b2”。我们把穿过几个机器才到达最终目的地的路径称为多段路径（multihop）。然而，若在 A 和 C 间存在直接的网络连接时，在这两台机器间通过机器 B 传送数据是低效率的。试说明在纽卡斯尔和透明型分布式模型中为这种多段路径提供的一种设计方案。

附录A 系统调用

本附录是UNIX系统调用的简短一览表。对于这些系统调用的完整的说明书可参考UNIX系统Ⅴ用户程序员手册。然而，当阅读本书中的各种不同的程序示例时，本处的说明已足够了。

所规定的文件名是以空白字符终结的字符串，其中每个个别的分量由斜杠字符分隔开。所有的系统调用出错时返回－1，由外部变量errno指示具体的错误。除非特别地规定，系统调用成功时返回0。某些系统调用是若干个函数的人口点：这表明对这些函数的汇编语言接口是相同的。此处的表沿用UNIX系统手册的习惯，但是程序员不必关心一个系统调用的人口点是处理一个还是多个系统调用的。

access

```
access (filename, mode)
char * filename;
int mode;
```

access根据mode的值检查调用进程对文件filename是否具有读、写或执行的许可权。mode的值是位模式4（读许可权）、2（写许可权）和1（执行许可权）的一种组合。所检查的是真正用户标识号，而不是有效用户标识号。

acct

```
acct (filename)
char * filename;
```

倘若filename不为0，acct使系统能进行记帐，否则不记帐。

alarm

```
unsigned alarm (seconds)
unsigned seconds;
```

alarm为调用进程在所指示的seconds秒后安排一个SIGALRM闹钟软中断信号。它返回的是从调用时刻开始直到该信号出现所剩余的时间量。

brk

```
int brk (end_data_seg)
char * end_data_seg
```

brk将一进程的数据区的最高地址设置为end_data_seg。另一个系统调用sbrk使用这个系统调用的人口点，并根据所规定的增量改变一个进程的数据区的最高地址。

chdir

```
chdir (filename)
```

```
char * filename;
```

chdir 将调用进程的当前目录移到 filename 所指示的目录上。

chmod

```
chmod (filename, mode)
char * filename;
int mode;
```

chmod 将所指示的文件 filename 的存取许可权改变为所规定的 mode，它是以下位（八进制表示）的组合：

04000	设置用户标识（setuid）位
02000	设置用户组标识位
01000	驻留（sticky）位
00400	所有者读
00200	所有者写
00100	所有者执行
00040	同组用户读
00020	同组用户写
00010	同组用户执行
00004	其他用户读
00002	其他用户写
00001	其他用户执行

chown

```
chown (filename, owner, group)
char * filename;
int owner, group;
```

chown 将所指示的文件 filename 的所有者和用户组 ID 改变为由 owner 和 group 所规定的所有者和用户组 ID。

chroot

```
chroot (filename)
char * filename;
```

chroot 将调用进程私用的改变了的根设置为 filename。

close

```
close (fildes)
int fildes;
```

close 关闭一个文件描述符 fildes，该文件描述符或是从先前的一个系统调用 open，creat，dup，pipe，fchtl 得到的，或者从系统调用 fork 继承的。

creat

```
creat (filename, mode)
char * filename;
```

```
int mode;
```

creat 以 filename 所指示的文件名和 mode 所规定的存取许可权方式创建一个新文件。除了驻留位（sticky bit）是清除的以及由系统调用 umask 设置的屏蔽位是清除的以外，mode 与系统调用 access 所规定的相同。creat 返回一个文件描述符以便在其他的系统调用中使用。

dup

```
dup (fildes)
int fildes;
```

dup 复制所规定的文件描述符，返回可用的最低序号的文件描述符。老的和新的文件描述符使用同一文件指针和共享其他属性。

exec

```
execve (filename, argv, envp)
char * filename;
char * argv [];
char * envp [];
```

execve 执行程序文件 filename，覆盖正在执行的进程的地址空间。argv 是所执行程序的一个字符串参数数组，envp 是一个成为新进程的环境的字符串数组。

exit

```
exit (status)
int status;
```

exit 引起调用进程的退出，并将 status 和低 8 位状态报告给等待它的父进程。内核也可以在响应某些软中断信号时内部地退出。

fcntl

```
fcntl (fildes, cmd, arg)
int fildes, cmd, arg;
```

fcntl 支持由文件描述符 filldes 标识的已打开文件上的一组各种各样的操作。对命令 cmd 和参数 arg 的解释如下（常数清单定义在文件“/usr/include/fcntl.h”中）：

F_DUPFD	返回大于或等于 arg 的最低序号的文件描述符。
F_SETFD	以 arg 的低位设置“执行后关闭”标志（若为 1，文件在系统调用 exec 后是关闭的）。
F_GETFD	返回“执行后关闭”标志的值。
F_SETFL	设置文件状态标志（O_NDELAY 是不因等待 I/O 而睡眠；O_APPEND 是附加欲写的数据到文件尾）。
F_GETEL	取文件状态标志。

```
Struct flock
    Short l_type;       /* 锁操作类型：F_RDLCK 加读锁；F_WRLCK 加写锁；F_UNLCK 解锁 */
    Short l_whence;     /* 锁偏移量从文件头开始（0）；锁偏移量从当前文件指针开始（1）；锁偏移量从文件尾开始（2） */
    long l_start;       /* 按照 l_whence 进行解释的字节偏移量 */
    long l_len;         /* 欲上锁的字节数，若为 0，从 l_start 到文件尾上锁 */
    long l_pid;         /* 对文件上锁的进程 ID */
    long l_sysid;       /* 对文件上锁的进程的系统 ID */
```

F_GETLK　对参数 arg 中规定的锁，读取它阻碍应用的第一个锁，然后改写 arg 参数。如果这样的锁不存在，把 arg 中 l_type 修改为 F_UNLCK

F_SETLK　对由 arg 规定的文件上锁或解锁，如果不能上锁则返回 -1

F_SETLKW　对由 arg 规定的文件上锁或解锁，如果不能上锁则睡眠

在一个文件中几个读锁可以重叠，但没有任何锁可以和一个写锁重叠。

fork

```
fork ()
```

fork 创建一个新进程。除了父进程从 fork 的返回值是子进程 ID，而子进程的返回值是 0 外，子进程是其父进程的一个逻辑拷贝。

getpid

```
getpid ()
```

getpid 返回调用进程的进程 ID。使用这个入口点的其他调用有：返回调用进程的进程组 ID 的 getpgrp；返回调用进程的父进程 ID 的 getppid。

getuid

```
getuid ()
```

getuid 返回调用进程的真正用户 ID。使用这个系统调用入口点的其他调用有：返回有效用户 ID 的 geteuid；返回用户组 ID 的 getgid；返回调用进程的有效用户组 ID 的 getegid。

ioctl

```
ioctl (fildes, cmd, arg)
int fildes, cmd;
```

ioctl 在文件描述符为 fildes 的打开设备上执行设备专用的操作。cmd 规定在该设备上欲执行的命令，arg 是一个参数，它的类型依赖于命令。

kill

```
kill (pid, sig)
int pid, sig;
```

kill 送软中断信号给由 pid 标识的进程。其中 pid 的值及含义如下：

正值　送信号给进程 ID 为 pid 的进程。

0　送信号给进程组 ID 等于发送者的进程组 ID 的那些进程。

-1　若发送者的有效用户 ID 是超级用户，送信号给所有进程；否则，送信号给真正用户 ID 等于发送者有效用户 ID 的所有进程。

小于 -1　送信号给进程组 ID 为 pid 的所有进程。

link

```
link (filename1, filename2)
char * filename1, * filename2;
```

link 将另一文件名 filename2 赋给文件 filename1。该文件可通过任一文件名存取。

lseek

```
lseek (fildes, offset, origin)
```

```
int fildes, origin;
long offset;
```

lseek 移动文件描述符为 fildes 的读/写指针的位置，并返回该新值。该指针的值依赖于参数 origin 如下：

0　　将指针设置到距文件头 offset 字节处。
1　　将当前指针值增加 offset。
2　　将指针设置到文件大小加上 offset 字节处。

mknod

```
mknod (filename, modes, dev)
char * filename;
int modes, dev;
```

mknod 根据 modes 的类型创建一个名为 filename 的特殊文件、目录或 FIFO 文件。对 modes 规定如下：

010000　　FIFO（有名管道）文件
020000　　字符设备特殊文件
040000　　目录文件
060000　　块设备特殊文件

modes 的低 12 位与前述的系统调用 chmod 中的 mode 有相同的含义。如果该文件是块设备或字符设备特殊文件，则 dev 给出该设备的主设备号和次设备号。

mount

```
mount (specialfile, dir, rwflag)
char * specialfile, * dir;
int rwflag;
```

mount 安装由 specialfile 所规定的文件系统到目录 dir 上。倘若 rwflag 的低位为 1，文件系统被安装成只读的。

msgctl

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/msg.h>

msgctl (id, cmd, buf)
int id, cmd;
struct msqid_ds * buf;
```

msgctl 允许进程根据命令 cmd 的值设置或查询消息队列 id 的状态、或删除该队列。结构 msgid_ds 定义如下：

```
struct ipc_perm {
        ushort      uid;      /* 当前用户 ID */
        ushort      gid;      /* 当前进程组 ID */
        ushort      cuid;     /* 创建用户 ID */
```

```
        ushort      cgid;      /*  创建进程组 ID  */
        ushort      mode;      /*  存取许可权  */
        short       pad1;      /*  由系统使用  */
        long        pad2;      /*  由系统使用  */
    }
    struct msqid_ds {
      structipc_perm msg_perm;      /*  许可权结构  */
      short          pad1 [7];      /*  由系统使用  */
      ushort         msg_qnum;      /*  队列上消息数  */
      ushort         msg_qbytes;    /*  队列上最大字节数  */
      ushort         msg_lspid;     /*  最后发送消息操作的 PID  */
      ushort         msg_lrpid;     /*  最后接收消息操作的 PID  */
      time_t         msg_stime;     /*  最后发送消息时间  */
      time_t         msg_rtime;     /*  最后接收消息时间  */
      time_t         msg_ctime;     /*  最后更改时间  */
    };
```

命令及其含义如下：

IPC_STAT	将与 id 相关联的消息队列头标读入 buf
IPC_SET	以 buf 中相应的值设置 msg_perm.uid、msg_perm.gid、msg_perm.mode（低 9 位）和 msg_qbytes 的值。
IPC－RMID	删除 id 的消息队列。

msgget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgget (key.flag)
key_t key;
int flag;
```

msgget 返回一个名字为 key 的消息队列的标识符。key 可以规定所返回的队列标识符指的是一个私用的队列（IPC_PRIVATE），在这种情况下创建一个新的消息队列。flag 规定是否该队列应被创建（IPC_CREAT），或是否该队列的创建应是互斥的（IPC_EXCL）。在后者情况下，倘若该队列已存在，msgget 失败返回。

msgsnd 和 msgrcv

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgsnd (id, msgp, size, flag)
int id, size, flag;
struct msgbuf * msgp;

msgrcv (id, msgp, size, type, flag)
int id, size, type, flag;
struct msgbuf * msgmp;
```

msgsnd 发送在缓冲 msgp 中的 size 个字节的一个消息到消息队列 id 上去，msgbuf 定义为：

```
struct msgbuf {
        long mtype;
        char mtext [];
};
```

若在标志 flag 中未设置 IPC _ NOWAIT 位，则当该消息队列中的字节数超过一最大值时，或系统范围的消息数超过一最大值时，调用 msgsnd 的进程睡眠。若 IPC _ NOWAIT 是设置的，在这些情况下，msgsnd 立即返回。

msgrcv 从由 id 标识的队列中接收消息。假若类型 type 是 0，接收该队列上的第一个消息；若为正，接收类型 type 的第一个消息；若为负，接收小于或等于 type 绝对值的最低类型的第一个消息。size 指明用户想要接收的消息正文的最大尺寸。若在 flag 中设置了 MSG _ NOERROR，内核截断所接收的消息，倘若其大小大于 size 的话。否则，它返回一个错误。若在 flag 中未设置 MSG _ NOERROR，调用 msgsnd 的进程睡眠，直到满足 type 的一个消息被发送。若 IPC _ NOWAIT 设置了，它立即返回。msgrcv 返回消息正文中的字节数。

```
nice
    nice (increment)
    int increment;
```

nice 将增量 increment 加到进程的 nice 值上。较高的 nice 值使该进程具有较低的调度优先权。

```
open
    # include <fcntl.h>

    open (filename, flag, mode)
    char * filename;
    int flag, mode;
```

open 根据 flag 的值打开所指定的文件 filename。flag 的值是以下位的组合（前面三位中必须恰好使用一位）：

O _ RDONLY	只读方式打开
O _ WRONLY	只写方式打开
O _ RDWR	读和写方式打开
O _ NDELAY	当置位时，对设备特殊文件的系统调用 open 不等待载波信号就返回。对有名管道，open 立即返回，而不等待另一进程打开该有名管道（若 O _ WRONLY 被置位，以错误返回）。
O _ APPEND	导致所有的写操作都添加数据到文件尾。
O _ CREAT	若文件不存在时，创建它。mode 与系统调用 creat 中一样规定许可权方式。若文件已存在，此标志无意义。
O _ TRUNC	将文件长度截为 0。
O _ EXCL	若此位及 O _ CREAT 被设置且文件存在时，系统调用 open 失败返回。这是一种被称为互斥的打开。

系统调用 open 返回一个文件描述符，以便其他系统调用使用。

```
pause
```

```
pause ()
```

pause 暂停调用进程的执行，直至它接收到一个软中断信号。

```
pipe
    pipe (fildes)
    int fildes [2];
```

pipe 返回一个读文件描述符和一个写文件描述符（分别为 fildes [**0**] 和 fildes [**1**]）。数据通过一个管道以先进先出的次序发送，数据不能读两次。

```
plock
    # include <sys/lock.h>

    plock (op)
    int op;
```

plock 根据 op 的值将进程的区锁入内存以及解锁。op 规定如下：

PROCLOCK	将正文区和数据区锁入内存。
TXTLOCK	将正文区锁入内存。
DATLOCK	将数据区锁入内存。
UNLOCK	对所有区解锁。

```
profil
    profil (buf, size, offset, scale)
    char * buf;
    int size, offset, scale;
```

profil 请求内核给出该进程的执行直方图。buf 是该进程中的一个数组，它累计在该进程的不同地址执行的频度计数。size 是数组 buf 的大小，offset 是在该进程中应被统计的起始地址，scale 是一个比例因子。

```
ptrace
    ptrace (cmd, pid, addr, data)
    int cmd, pid, addr, data;
```

ptrace 根据命令 cmd 的值，允许一个进程跟踪另一进程 pid 的执行。各命令意义如下：

0	允许子进程被跟踪（由子进程调用）。
1，2	返回被跟踪进程 pid 在地址 addr 处的字。
3	返回被跟踪进程的 u 区中偏移量为 addr 处的字。
4，5	将 data 的值写入被跟踪进程地址 addr 处。
6	将 data 的值写入被跟踪进程 u 区偏移量为 addr 处。
7	使被跟踪进程继续执行。
8	使被跟踪进程退出。
9	与机器有关——在程序状态字中设置标志位以单步执行。

```
read
    read (fildes, buf, size)
    int fildes,
```

```
        char * buf;
        int size;
```

read 从文件 fildes 中最多读 size 字节的数据到用户缓冲区 buf。它返回它所读的字节数。对设备特殊文件或管道文件，倘若无可用的数据且在文件打开时设置了 O _ NDELAY，read 立即返回。

semctl

```
        # include 〈sys/types.h〉
        # include 〈sys/ipc.h〉
        # include 〈sys/sem.h〉

        semctl (id, num, cmd, arg)
        int id, num, cmd;
        union semun {
                int val;
                struct semid _ ds * buf;
                ushort * array;
    } arg;
```

semctl 对由 id 所指示的信号量队列执行所规定的命令 cmd，命令如下：

GETVAL	返回索引值为 num 的信号量的值。
SETVAL	将索引值为 num 的信号量值设置为 arg.val。
GETPID	返回最后在索引值为 num 的信号量上执行信号量操作的进程标识号。
GETNCNT	返回等待信号量值变为正值的进程数。
GETZCNT	返回等待信号量值变为 0 的进程数。
GETALL	将所有信号量值返回到数组 arg.array 中去。
SETALL	根据数组 a:g.array 设置所有信号量值。
IPC _ STAT	将 id 标识的信号量头标数据结构读入 arg.buf
IPC _ SET	根据 arg.buf 设置 sem _ perm.uid，sem _ perm.gid 和 sem _ perm.mode（低 9 位）。
IPC _ RMID	删除与 id 相关联的信号量。

num 给出欲处理的信号量集合中的信号量序号。结构 semid _ ds 定义为：

```
    struct semid _ ds {
        struct ipc _ perm   sem _ perm;   /*  许可权结构  */
        int *               pad;          /*  由系统使用  */
        ushort              sem _ nsems;  /*  集合中信号量数目  */
        time _ t            sem _ otime;  /*  最后信号量操作时间  */
        time _ t            sem _ otime;  /*  最后修改时间  */
    };
```

结构 ipc _ perm 与系统调用 msgctl 中所定义的相同。

semget

```
        # include 〈sys/types.h〉
        # include 〈sys/ipc.h〉
        # include 〈sys/sem.h〉
```

```
semget (key, nsems, flag)
key_t key;
int nsems, flag;
```

semget 根据关键字 key 创建一个信号量数组。key 和 flag 与它们在系统调用 msgget 中有相同的含义。

```
semop
    semop (id, ops, num)
    int id, num;
    struct sembuf * *ops;
```

semop 对以 id 标识的信号量集合执行结构 ops 数组中的信号量操作集合。num 是 ops 中的项数。sembuf 的结构是：

```
struct sembuf {
    short    sem_num;    /* 信号量序号 */
    short    sem_op;     /* 信号量操作 */
    short    sem_flg;    /* 标志 */
};
```

sem_num 为某特定的操作规定在信号量数组中的索引值，而 sem_flg 规定该操作的标志。对信号量的操作值 sem_op 有：

负值	假若信号量值与信号量操作值 sem_op 之和大于或等于 0，将 sem_op 加到信号量值上；否则根据标志睡眠。
正值	将 sem_op 加到信号量值上。
0	倘若信号量值为 0，继续；否则根据标志睡眠。

如果在信号量操作标志 sem_flg 中为一个特定的操作设置了 IPC_NOWAIT，系统调用 semop 对那些它会睡眠的情形应立即地返回。倘若 SEM_UNDO 标志被置位，该操作从这些值的一个运行总和中减去这个操作值。当该进程退出时，这个总和被加到该信号量值上。semop 返回在信号量操作 ops 中最后一个信号量操作在本次调用时刻的值。

```
setpgrp
    setpgrp ()
```

setpgrp 设置调用进程的进程组 ID 为它的进程 ID，并返回该新值。

```
setuid
    setuid (uid)
    int uid;

    setgid (gid)
    int gid;
```

setuid 设置调用进程的真正和有效用户 ID。如果调用者的有效用户 ID 为超级用户，则系统调用 setuid 重新设置真正和有效用户 ID；否则，如果它的真正用户 ID 等于 uid，setuid 重新

设置有效用户 ID 为 uid。最后，如果它的保存的用户 ID（通过在系统调用 exec 中执行一个 setuid 程序来设置的）等于 uid，则 setuid 重新设置其有效用户 ID 为 uid。系统调用 setgid 以同样的方式对真正和有效用户组 ID 操作。

shmctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

shmctl (id, cmd, buf)
int id, cmd;
struct shmid_ds * buf;
```

shmctl 对以 id 标识的共享存储区完成各种控制操作。结构 shmid_ds 定义为：

```
struct shmid_ds {
    struct ipc_perm   shm_perm;    /*  许可权结构  */
    int               shm_segsz;   /*  段大小  */
    int *             pad1;        /*  由系统使用  */
    ushort            shm_lpid;    /*  最后操作的进程 ID  */
    ushort            shm_cpid;    /*  创建者的进程 ID  */
    ushort            shm_nattch;  /*  当前附接数  */
    short             pad2;        /*  由系统使用  */
    time_t            shm_atime;   /*  最后附接时间  */
    time_t            shm_dtime;   /*  最后断接时间  */
    time_t            shm_ctime;   /*  最后修改时间  */
};
```

所进行的操作有：

IPC_STAT　　将以 id 标识的共享存储区头标的值读入 buf。

IPC_SET　　根据 buf 中的值设置 shm_perm.id，shm_perm.gid 和 shm_perm.mode（低 9 位）。

IPC_RMID　　删除由 id 标识的共享存储区。

shmget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

shmget (key, size, flag)
key_t key;
int size, flag;
```

shmget 存取或创建一个 size 字节的共享存储区。其中的参数 key 和 flag 与系统调用 msgget 中这些参数的含义相同。

shmop

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>

shmat (id, addr, flag)
int id, flag;
char * addr;

shmdt (addr)
char * addr;
```

shmat 将以 id 标识的共享存储区附接到一个进程的地址空间去。若 addr 为 0，内核选择一个适当的地址来附接该区；否则，它试图在所规定的地址 addr 上来附接该区。倘若在标志 flag 中设置了 SHM_RND 位，内核在必要时舍入这个地址。shmat 返回该区被附接的地址。

shmdt 断接先前附接在地址 addr 处的共享存储区。

signal

```
#include <signal.h>

signal (sig, function)
int sig;
void (*func) ();
```

signal 允许调用进程控制软中断信号的处理。sig 的值是：

SIGHUP	挂起（hangup）
SIGINT	键盘按 delete 键或 break（interrupt）
SIGQUIT	键盘按 quit 键（quit）
SIGILL	非法指令
SIGTRAP	跟踪陷井
SIGIOT	IOT 指令
SIGEMT	EMT 指令
SIGFPE	浮点运算溢出
SIGKILL	要求终止进程
SIGBUS	总线错
SIGSEGV	段违例
SIGSYS	系统调用参数错
SIGPIPE	向无读者管道上写
SIGALRM	闹钟
SIGTERM	软件终结
SIGUSR1	用户定义信号
SIGUSR2	第二个用户定义信号
SIGCLD	子进程死
SIGPWR	电源故障

对 function 的解释如下：

SIG_DFL	缺省操作。对除 SIGPWR 和 SIGCLD 外的所有信号的缺省操作是进程终止。对信号 SIGQUIT，SIGILL，SIGTRAP，SIGIOT，SIGEMT，SIGFPE，SIGBUS，SIGSEGV 和 SIGSYS 它产生一内存映象文件。
SIG_IGN	忽略该信号的出现。
function	在该进程中的一个函数的地址。在内核返回用户态时，它以软中断信号的序号作为参数调用该函数。对除了信号 SIGILL，SIGTRAP 和 SIGPWR 以外的所有信号，内核自动地重新设置软中断信号处理程序的值为 SIG_DFL。一个进程不能捕俘 SIGKILL 信号。

```
stat
    stat (filename, statbuf)
    char * filename;
    struct stat * statbuf;

    fstat (fd, statbuf)
    int fd;
    struct stat * statbuf;
```

stat 返回关于所规定文件 filename 的状态信息。fstat 对文件描述符为 fd 的打开文件完成同样的操作。结构 statbuf 是：

```
struct stat {
    dev_t   st_dev;     /* 含有该文件的设备的设备号 */
    ino_t   st_ino;     /* 索引节点号 */
    ushort  st_mode;    /* 文件类型及许可权 */
    short   st_nlink;   /* 文件的联结数 */
    ushort  st_uid;     /* 文件所有者的用户 ID */
    ushort  st_gid;     /* 文件的进程组 ID */
    dev_t   st_rdev;    /* 主和次设备号 */
    off_t   st_size;    /* 文件大小（以字节计） */
    time_t  st_atime;   /* 最后存取时间 */
    time_t  st_mtime;   /* 最后修改时间 */
    time_t  st_ctime;   /* 最后状态变更时间 */
};
```

```
stime
    stime (tptr)
    long * tptr;
```

stime 根据由 tptr 所指向的值设置系统时间和日期。时间值规定是从格林威治时间 1970 年 1 月 1 日零点零分零秒开始，以秒为单位计算的。

```
sync
    sync ()
```

sync 将系统缓冲中的文件系统数据倾泻到磁盘上去。

```
time
    time (tloc)
    long * tloc;
```

time 返回从格林威治时间 1970 年 1 月 1 日零时零分零秒开始的秒数。如果 tloc 不为 0，它也含有返回的值。

```
times
    # include <sys/types.h>
```

```
#include <sys/times.h>

times (tbuf)
struct tms * tbuf;
```

times 返回从过去的一个任意的确定时间以来所消逝的时间，该时间是以时钟滴答为单位计算的，并以下述的记帐信息填入 tbuf：

```
struct tms {
    time_t  tms_utime;      /*  花费在用户态的 CPU 时间   */
    time_t  tms_stime;      /*  花费在核心态的 CPU 时间   */
    time_t  tms_cutime;     /*  子进程用户态时间总和   */
    time_t  tms_sutime;     /*  子进程核心态时间总和   */
};
```

```
ulimit
    ulimit (cmd, limit)
    int cmd;
    long limit;
```

ulimit 允许一个进程根据命令 cmd 的值设置各种限制，命令值及含义如下：

1　　返回该进程能写的最大文件尺寸（以 512 字节的块为单位）。

2　　将文件尺寸最大值设置为 limit 值。

3　　返回数据区中可能的最高的地址（系统调用 brk 中参数的可能的最大值）。

```
umask
    umask (mask)
    int mask;
```

umask 设置创建文件时许可权方式的屏蔽位为 mask，并返回其老值。当创建一文件时，如果在 mask 中某位被置位，许可权方式的相应位为 0。

```
umount
    umount (specialfile)
    char * specialfile;
```

umount 拆卸在块设备特殊文件 specialfile 中的文件系统。

```
uname
    #include <sys/utsname.h>

    uname (name)
    struct utsname * name;
```

uname 根据以下的结构返回系统专用的信息；

```
struct utsname {
    char   sysname[9];      /*   名字   */
```

```
    char   nodename [9];     /*  网络节点名   */
    char   release [9];      /*  系统版本信息   */
    char   version [9];      /*  更多的版本信息   */
    char   machine [9];      /*  硬件   */
};

unlink
    unlink (filename)
    char * filename;
```

unlink 删除所指示的文件 filename 的目录项。

```
ustat
    # include <sys/types.h>
    # include <ustat.h>

    ustat (dev, ubuf)
    int dev;
    struct ustat * ubuf;
```

ustat 返回有关由 dev（主设备号和次设备号）标识的文件系统的统计信息，结构 ustat 定义为：

```
struct ustat {
    daddr_t  f_tfree;          /*  空闲块的数目   */
    ino_t    f_tinode;         /*  空闲索引结点的数目   */
    char     f_fname [6];      /*  文件系统名   */
    char     f_fpack [6];      /*  文件系统包装名   */
};

utime
    # include <sys/types.h>

    utime (filename, times)
    char * filename;
    struct utimbuf * times;
```

utime 根据 times 中的值设置所规定的文件 filename 的存取和修改时间。若 times 为 0，使用当前时间，否则，times 指向以下结构：

```
struct utimbuf {
    time_t  axtime;      /*  存取时间   */
    time_t  modtime;     /*  修改时间   */
};
```

全部时间都是从格林威治时间 1970 年 1 月 1 日零点开始计算的。

```
wait
    wait (wait_stat)
    int * wait_stat;
```

wait引起调用进程去睡眠，直到它发现一个子进程已退出或一个进程在跟踪态中睡眠。假如wait _ stat不为0。当它由此系统调用返回时，它指向一个含有状态信息的地址，其中只有低16位写有状态信息。如果系统调用wait是因它发现一个已退出的子进程而返回的，状态信息的低8位为0，高8位中含有子进程作为系统调用exit的一个参数传递的值的低8位。如果子进程是因一软中断信号而退出的，其高8位为0，低8位含有软中断信号的序号。此外，倘若内存映象文件被转储，其0200位被设置。如果系统调用wait因发现一被跟踪进程而返回，其高8位（16位中的）含有引起它停止的软中断信号序号，低8位含有0177。

write

```
write (fd, buf, count)
int fd, count;
char * buf;
```

write将用户地址buf中count字节的数据写到文件描述符为fd的文件中去。

参 考 文 献

[Babaoglu 81] Babaoglu, O., and W.Joy, "Converting a Swap－Based System to do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the 8th Symposium on Operating Systems Principles*, *ACM Operating Systems Review*, Vol.15 (5), Dec. 1981, pp.78-86.

[Bach 84] Bach, M. J., and S. J. Buroff, "Multiprocessor UNIX Systems," *AT&T Bell Laboratories Technical Journal*, Oct. 1984, Vol 63, No. 8, Part 2, pp.1733-1750.

[Barak 80] Barak, A. B. and A. Shapir, "UNIX with Satellite Processors," *Software-Practice and Experience*, Vol. 10, 1980, pp. 383-392.

[Beck 85] Beck, B. and B. Kasten, "VLSI Assist in Building a Multiprocessor UNIX System," *Proceedings of the USENIX Association Summer Conference*, June 1985, pp. 255-275.

[Berkeley 83] *UNIX Programmer's Manual*, 4.2 *Berkeley Software Distribution*, *Virtual VAX-11 Version*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, August 1983.

[Birrell 84] Birrell, A.D. and B.J.Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 39-59.

[Bodenstab 84] Bodenstab, D. E., T. F. Houghton, K. A.Kelleman, G. Ronkin, and E. P. Schan, "UNIX Operating System Porting Experiences," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Oct. 1984, pp. 1769-1790.

[Bourne 78] Bourne, S. R., "The UNIX Shell," *The Bell System Technical Journal*, July-August 1978, Vol. 57, No. 6, Part 2, pp. 1971-1990.

[Bourne 83] Bourne, S. R., *The UNIX System*, Addison-Wesley, Reading, MA, 1983.

[Brownbridge 82] Brownbridge, D. R., L. F. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!" in *Software-Practice and Experience*, Vol. 12, 1982, pp. 1147-1162.

[Bunt 76] Bunt, R.B., "Scheduling Techniques for Operating Systems," *Computer*, Oct.1976, pp. 10-17.

[Christian 83] Christian, K., *The UNIX Operating System*, John Wiley & Sons Inc., New York, NY, 1983.

[Coffman 73] Coffman, E.G., and P.J. Denning, *Operating Systems Theory*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1973.

[Cole 85] Cole, C.T., P.B.Flinn, and A.B. Atlas, "An Implementation of an Extended File System for UNIX," *Proceedings of the USENIX Conference*, Summer 1985, pp. 131-149.

[Denning 68] Denning, P.J., "The Working Set Model for Program Behavior," *Communications of the ACM*, Volume 11, No.5, May 1968, pp. 323-333.

[Dijkstra 68] Dijkstra, E.W., "Cooperating Sequential Processes," in *Programming Languages*, ed. F. Genuys, Academic Press, New York NY, 1968.

[Felton 84] Felton, W. A., G.L.Miller, and J. M. Milner, "A UNIX Implementation for System/370," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Oct. 1984, pp. 1751-1767.

[Henry 84] Henry, G. J., "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, Oct. 1984, Vol 63, No. 8, Part 2, pp.1845-1858.

[Holley 79] Holley, L.H., R.P. Parmelee, C.A. Salisbury, and D.N.Saul, "VM/370 Asymmetric Multiprocessing," *IBM Systems Journal*, Vol. 18, No. 1, 1979, pp. 47-70.

[Holt 83] Holt, R.C., *Concurrent Euclid*, *the UNIX System*, *and Tunis*, Addison-Wesley, Reading, MA. 1983.

[Horning 73] Horning, J. J., and B. Randell, "Process Structuring," *Computing Surveys*, Vol. 5, No. 1, March 1973, pp.5-30.

[Hunter 84] Hunter, C.B. and E. Farquhar, "Introduction to the NS16000 Architecture," *IEEE Micro*, April 1984, pp. 26-47.

[Johnson 78] Johnson, S.C. and D.M.Ritchie, "Portability of C Programs and the UNIX System," *The Bell System Technical Journal*, Vol. 57, No.6, Part 2, July-August, 1978, pp.2021-2048.

[Kavaler 83] Kavaler, P.and A. Greenspan ,"Extending UNIX to Local-Area Networks," *Mini-Micro Systems*, Sept. 1983, pp. 197-202.

[Kernighan 78] Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kernighan 84] Kernighan, B.W., and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Killian 84] Killian, T.J. ,"Processes as Files," *Proceedings of the USENIX Conference*, *Summer* 1984, pp.203-207.

[Levy 80] Levy, H.M., and R.H.Eckhouse, *Computer Programming and Architecture*: *The VAX-11*, Digital Press, Bedford, MA, 1980.

[Levy 82] Levy, H.M., and P.H. Lipman, "Virtual Memory Management in the VAX/VMS Operating System," *Computer*, Vol. 15, No.3, March 1982, pp. 35-41.

[Lu 83] Lu, P.M., W. A. Dietrich, et.al., "Architecture of a VLSI MAP for BELLMAC-32 Microprocessor," *Proc. of IEEE Spring Compcon*, Feb. 28, 1983, pp.213-217.

[Luderer 81] Luderer, G.W.R., H.Che, J.P. Haggerty, P.A. Kirslis, and W.T. Marshall, "A Distributed UNIX System Based on a Virtual Circuit Switch," *Proceedings of the Eighth Symposium on Operating Systems Principles*, Asilomar, California, December 14-16, 1981.

[Lycklama 78a] Lycklama, H.and D.L.Bayer, "The MERT Operating System," *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July-August 1978, pp. 2049-2086.

[Lycklama 78b] Lycklama, H. and C. Christensen, "A Minicomputer Satellite Processor System," *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July-August 1978, pp. 2103-2114.

[McKusick 84] McKusick, M.K., W.N. Joy, S.J. Leffler, and R.S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2 (3), August 1984, pp. 181-197.

[Mullender 84] Mullender, S.J. and A.S. Tanenbaum, "Immediate Files," *Software-Practice and Experience*, Vol. 14 (4), April 1984, pp. 365-368.

[Nowitz 80] Nowitz, D.A. and M.E. Lesk, "Implementation of a Dial-Up Network of UNIX Systems," *IEEE Proceedings of Fall* 1980 *COMPCON*, Washington, D.C., pp. 483-486.

[Organick 72] Organick, E.J., *The Multics System*: *An Examination of Its Structure*, The MIT Press, Cambridge, MA, 1972.

[Peachey 84] Peachey, D.R., R.B. Bunt, C.L. Williamson, and T.B. Brecht, "An Experimental Investigation of Scheduling Strategies for UNIX," *Performance Evaluation Review*, 1984 *SIGMETRICS Conference on Measurement and Evaluation of Computer Systems*, Vol. 12 (3), August 1984, pp. 158-166.

[Peterson 83] Peterson, James L. and A. Silberschatz, *Operating System Concepts*, Addison-Wesley, Reading,

MA, 1983.

[Pike 84] Pike, R., "The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Laboratories Technical Journal*, Oct. 1984, Vol 63, No. 8, Part 2, pp. 1607-1632.

[Pike 85] Pike, R., and P. Weinberger, "The Hideous Name," *Proceedings of the USENIX Conference, Summer* 1985, pp. 563-568.

[Postel 80] Postel, J. (ed.), "DOD Standard Transmission Control Protocol," *ACM Computer Communication Review*, Vol. 10, No. 4, Oct. 1980, pp. 52-132.

[Postel 81] Postel, J., C.A.Sunshine, and D. Cohen, "The ARPA Internet Protocol," *Computer Networks*, Vol. 5, No. 4, July 1981, pp.261-271.

[Raleigh 76] Raleigh, T.M., "Introduction to Scheduling and Switching under UNIX," *Proceedings of the Digital Equipment Computer Users Society*, Atlanta, Ga., May 1976, pp. 867-877.

[Richards 69] Richards, M., "BCPL: A Tool for Compiler Writing and Systems Programming," *Proc. AFIPS SJCC* 34, 1969, pp.557-566.

[Ritchie 78a] Ritchie, D. M. and K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal*, July-August 1978, Vol.57, No.6, Part 2, pp.1905-1930.

[Ritchie 78b] Ritchie, D.M., "A Retrospective," *The Bell System Technical Journal*, July-August 1978, Vol.57, No.6, Part 2, pp.1947-1970.

[Ritchie 81] Ritchie, D.M. and K. Thompson, "Some Further Aspects of the UNIX Time-Sharing System," *Mini-Micro Software*, Vol.6, No.3, 1981, pp. 9-12.

[Ritchie 84a] Ritchie, D.M., "The Evolution of the UNIX Time-sharing System," *AT&T Bell Laboratories Technical Journal*, Oct. 1984, Vol 63, No. 8, Part 2, pp. 1577-1594.

[Ritchie 84b] Ritchie, D.M., "A Stream Input Output System," *AT&T Bell Laboratories Technical Journal*, Oct. 1984, Vol 63, No. 8, Part 2, pp. 1897-1910.

[Rochkind 85] Rochkind, M.J., *Advanced UNIX Programming*, Prentice-Hall, 1985.

[Saltzer 66] Saltzer, J.H., *Traffic Control in a Multiplexed Computer System*, Ph.D. Thesis, MIT, 1966.

[Sandberg 85] Sandberg, R., D.Goldberg, S.Kleiman, D.Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem" *Proceedings of the USENIX Conference*, Summer 1985, pp.119-131.

[SVID 85] *System V Interface Definition*, Spring 1985, Issue 1, AT&T Customer Information Center, Indianapolis, IN.

[SystemV 84a] *UNIX System V User Reference Manual*.

[SystemV 84b] *UNIX System V Administrator's Manual*.

[Thompson 74] Thompson, K. and D.M. Ritchie, "The UNIX Time-Sharing System," *Communications of the ACM*, Vol.17, No.7, July, 1974, pp. 365-375 (revised and reprinted in [Ritchie 78a]).

[Thompson 78] Thompson, K., "UNIX Implementation," *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July-August, 1978, pp.1931-1946.

[Weinberger 84] Weinberger, P.J., "Cheap Dynamic Instruction Counting," *The AT&T Bell Laboratories Technical Journal*, Vol. 63, No.6, Part 2, October 1984, pp. 1815-1826.

索　引

本书以UNIX系统V为背景，全面、系统地介绍了UNIX操作系统内核的内部数据结构和算法。本书首先对系统内核结构做了简要介绍，然后分章节描述了文件系统、进程调度和存储管理，并在此基础上讨论了UNIX系统的高级问题，如驱动程序接口、进程间通讯与网络等。

本书可作为大学计算机科学系高年级学生和研究生的教材或参考书，也为从事UNIX系统研究与实用程序开发人员提供了一本极有价值的参考资料。

本书版权登记号：图字：01－2000－0303

图书在版编目（CIP）数据

UNIX操作系统设计/(美)贝奇（Bach.M.J）著；陈葆珏等译.—北京：机械工业出版社，2000.4
(计算机科学丛书)
书名原文：The Design of the UNIX Operating System
ISBN 7－111－07850－0

Ⅰ.U… Ⅱ.①贝…②陈… Ⅲ.操作系统，UNIX-系统设计 Ⅳ.TP316.81

中国版本图书馆CIP数据核字（2000）第03582号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：陈 谊
北京第二外国语学院印刷厂印刷·新华书店北京发行所发行
2000年4月第1版·2000年8月第2次印刷
787mm×1092mm 1/16·23印张
印数：6 001-11 000册
定价：35.00元

译 者 序

UNIX 操作系统自 1974 年问世以来，迅速在世界范围内推广。目前，它不仅是小型机、高档微型机、工作站系统的主流操作系统，而且已进入中、大型计算机领地，成为“事实上”的标准操作系统，并被国际标准化组织 ISO 等考虑和采纳作为分布处理系统的本地操作系统参考模型。值得一提的是，90 年代以来随着运行在 PC 机上的 UNIX 系统的变体——Linux 的出现和迅速普及，使 UNIX 系统更具生命力。

当前，介绍 UNIX 系统的书籍很多，然而论述 UNIX 系统内部结构的专著却屈指可数。本书是其中最引人注目的一本。本书作者 Maurice J.Bach 多年来在 AT&T 公司的贝尔实验室工作，对 UNIX 系统的设计思想有深刻了解，又有讲授 UNIX 系统的丰富经验。作者在回顾 UNIX 操作系统的发展演变的基础上，描述了 UNIX 系统 V 内核内部的数据结构和算法，并对其做了深入浅出的分析。在每章之后，还给出了大量富有启发性和实际意义的题目。因而，本书不仅可用作大学高年级和研究生操作系统课程的教科书和参考书，也为从事 UNIX 操作系统的研究人员或 UNIX 实用程序开发人员提供了极有价值的参考资料。

在本书翻译过程中，我们尽量保持原著的特色，对书中大量的以 C 伪码形式描述的算法，仍保持 C 语言的结构和格式。因而，阅读本书的读者应具备一定的 C 语言基础。另外，对书中的若干明显错误，我们也一一作了修正。有错误或不妥之处，恳请指正。

本书是由北京大学计算机系的几位同志合作翻译的：第 1、2、3、4、12 章由柳纯录翻译；第 5、6、8 章由冯雪山翻译；第 7、9 章由王旭翻译；第 10、11、13 章由陈葆珏翻译。全书由陈葆珏修改、定稿。特别值得一提的是，本书的翻译从一开始就得到了杨芙清教授的支持和帮助。她还在百忙之中为本书做了校阅，特此表示衷心的感谢。

译者

于北京

前　言

UNIX 系统是由 Ken Thompson 和 Dennis Ritchie 于 1974 年在《ACM 通讯》中的一篇文章中首次提出的[Thompson 74]。从那时起，UNIX 系统得到迅速传播并在计算机工业中得到广泛采用。越来越多的计算机厂家在他们的机器上提供对 UNIX 系统的支持。UNIX 系统在大学里尤其普遍，它通常被用于操作系统的研究及实例分析。

许多专著和文章曾讨论了系统的各个部分，其中有《贝尔系统技术杂志》1978 年和 1984 年的两个专刊[BSTJ 78][BLTJ 84]。还有许多书介绍了 UNIX 系统的用户接口，特别是如何使用电子邮件、如何准备文件及如何使用称为“shell”的命令解释程序等；《The UNIX Programming Environment》[Kernighan 84]（该书已由机械工业出版社引进出版，中译本名为《UNIX 编程环境》）和《Advanced UNIX Programming》[Rochkind 85]等书讨论了程序设计环境。本书则着重描述构成操作系统基础（称为内核）的内部算法和数据结构以及它们与程序员接口之间的关系。因此，本书适用于几种环境。首先，它可用作高年级本科生或一年级研究生的操作系统课程的教材。使用本书的同时若能参考系统源代码则将获益匪浅，但也可以独立地学习本书。其次，系统程序员可将本书作为参考书，从而能更好地理解内核是如何工作的，并可以将 UNIX 系统中采用的算法与其他操作系统的算法加以比较。最后，UNIX 系统上的程序员能够更深入地了解他们的程序是如何与系统相互作用的，从而编出更有效、更高级的程序。

本书的内容及组织形式取自我在 AT&T 贝尔实验室在 1983 年和 1984 年期间讲授的一门课程。尽管这门课集中于阅读系统源代码，但我发现一旦掌握了算法的基本思想，源代码的阅读和理解就会容易得多。在本书里，我已努力使算法的描述尽可能地简单，从而反映出算法所描述的系统的简单性和精巧性。因此，本书并不是用英文逐行地翻译系统；它描述了各种算法的主要流程，更重要的是，它描述了各种算法是如何相互作用的。算法用类似 C 语言的伪码来表示，从而有助于读者理解自然语言的描述；算法的名字对应于内核内部的过程名。书中的各种插图描绘了系统对各种数据结构进行操作时它们之间的关系。在稍后的一些章中，采用了许多小的 C 语言程序来说明一些系统的概念，这些程序用户是容易明白的。为节省篇幅和清晰起见，这些例子一般不检查错误条件，而这一点在写程序时是一定要做的。我已经在系统 V 上运行了这些程序；除了某些演示系统 V 的特殊特点的程序以外，这些程序也应该能在 UNIX 系统的其他版本上运行。

原来为课程所准备的许多习题已放在每章的最后，它们是本书的重要组成部分。有些习题是直接了当的，用于说明正文中引入的概念。有些习题比较困难，用来帮助读者在一个较深的层次上理解系统。最后，还有些习题具有研究性质，设计这些题目是为了提出问题以供研究探讨。难度大的题目都标有 * 号。

本书对 UNIX 系统的描述基于 AT&T 所支持的系统 V，第 2 版。还包括了一些第 3 版的新特点。这是我最熟悉的系统，但我还尽力描述了其他版本对 UNIX 系统的有意义的贡献，特别是 BSD 对系统的修改。本书回避了与特殊的硬件特性有关的问题，力图以通用的

术语描述内核硬件的接口，并忽略特定机器的特殊特点。但是，当与机器有关的问题对理解内核的实现十分重要时，本书则讨论得相对详细一些。至少，对这些问题的探讨会突出操作系统中最依赖于机器的部分。

本书的读者必须具有用高级语言进行程序设计的经验，这是理解本书内容的必备条件，最好还有汇编语言的经验。建议读者具有用 UNIX 系统工作的经验，并了解 C 语言[Kernighan 78]。但是，在编写此书时，我努力使没有这种背景的读者也能理解本书的内容。本书的附录含有系统调用的简单描述，它们足以使读者理解书中的表达方式，但并不作为完整的参考手册。

本书的内容按如下方式组织。第 1 章，系统概貌，简要地描述了用户所看到的系统的特点，并给出了系统结构。第 2 章描述了内核结构的一般概貌，并引入一些基本概念。其余的章节按系统结构所表示的组成部分，描述其中各个成分。这些章可分为三部分：文件系统、进程控制和高级问题。本书先讨论文件系统，因为其概念比进程控制容易一些。这样，第 3 章描述了系统缓冲区高速缓存机制，这是文件系统的基础。第 4 章给出文件系统内部使用的一些算法和数据结构。这些算法使用了第 3 章中解释的算法，并讨论了管理用户文件所需要的内务操作。第 5 章说明提供文件系统用户接口的系统调用；这些系统调用使用了第 4 章的算法来存取用户文件。

第 6 章转向进程控制，其中定义了进程的上下文，讨论了控制进程上下文的内部内核原语。特别地讨论了系统调用接口，中断处理及上下文切换。第 7 章给出了控制进程上下文的系统调用。第 8 章讨论了进程调度问题。第 9 章的内容是存储管理，其中包括对换和请求调页系统。

第 10 章讨论了通用驱动程序接口，特别地讨论了磁盘驱动程序和终端驱动程序。尽管从逻辑上说设备是文件系统的一部分，但是，因为进程控制的问题要在终端驱动程序中出现，所以，对设备的讨论推迟到这一章。这一章也是通向本书其余章节中所给出的更高级的问题的桥梁。第 11 章讨论进程间通信和网络问题，其中包括系统 V 的消息、共享存储区及信号量，还有 BSD 的套接字。第 12 章解释了紧密耦合的多处理机 UNIX 系统。第 13 章研究了松散耦合的分布式系统。

前九章的内容可以在一学期的操作系统课程中完成。其余各章的内容可以在高级讨论班中进行讨论，并同时作各种课题研究。

至此，本人要作几点说明。可以确切地说，本书没有作系统性能方面的讨论，也没有提出任何用于系统安装的配置参数。这些数据会因机器类型、硬件配置、系统版本和实现、以及应用类型等的不同而不同。同时，我有意地尽量避免预测 UNIX 操作系统的未来发展。所讨论的高级问题并不意味着 AT&T 就要提供这些特别的特性，甚至也不意味着那些特殊的领域正在开发研究中。

Maurice J. Bach

目　录