回车符结束，其他系统则用换行符结束。有些系统还同时采用回车符和换行符（如MS-DOS）。文件中各行的长度不一定相同。

ASCII文件的最大优势是可以显示和打印，还可以用任何文本编辑器进行编辑。再者，如果很多程序都以ASCII文件作为输入和输出，就很容易把一个程序的输出作为另一个程序的输入，如shell管道一样。（用管道实现进程间通信并非更容易，但若以一种公认的标准（如ASCII码）来表示，则更易于理解一些。）

其他与ASCII文件不同的是二进制文件。打印出来的二进制文件是无法理解的、充满混乱字符的一张表。通常，二进制文件有一定的内部结构，使用该文件的程序才了解这种结构。

如图4-3a是一个简单的可执行二进制文件，它取自某个版本的UNIX。尽管这个文件只是一个字节序列，但只有文件的格式正确时，操作系统才会执行这个文件。这个文件有五个段：文件头、正文、数据、重定位位及符号表。文件头以所谓的魔数（magic number）开始，表明该文件是一个可执行的文件（防止非这种格式的文件偶然运行）。魔数后面是文件中各段的长度、执行的起始地址和一些标志位。程序本身的正文和数据在文件头后面。这些被装入内存，并使用重定位位重新定位。符号表则用于调试。

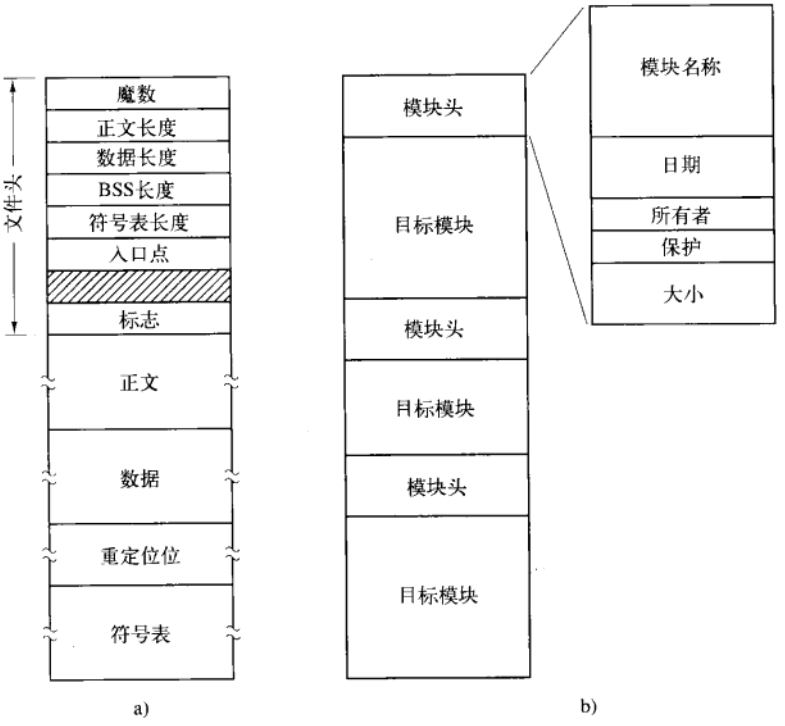


图4-3 a) 一个可执行文件；b) 一个存档文件

二进制文件的第二个例子是UNIX的存档文件，它由已编译但没有连接的库过程（模块）集合而成。每个文件以模块头开始，其中记录了名称、创建日期、所有者、保护码和文件大小。该模块头与可执行文件一样，也都是二进制数字，打印输出它们毫无意义。

所有操作系统必须能够识别它们自己的可执行文件的文件类型，其中有些操作系统还可识别更多的信息。一种老式的TOPS-20操作系统（用于DECsystem20计算机）甚至可检查可执行文件的创建时间，然后，它可以找到相应的源文件，看它在二进制文件生成后是否被修改过。如果修改过，操作系统自动重新编译这个文件。在UNIX中，就是在shell中嵌入make程序。这时操作系统要求用户必须采用固定的文件扩展名，从而确定哪个源程序生成哪个二进制文件。

如果用户执行了系统设计者没有考虑到的某种操作，这种强制类型的文件有可能会引起麻烦。比如在一个系统中，程序输出文件的扩展名是.dat（数据文件），若用户写一个格式化程序，读入.c（C程序）

文件并转换它（比如把该文件转换成标准的首行缩进），再把转换后的文件以.dat类型输出。如果用户试图用C编译器来编译这个文件，因为文件扩展名不对，C编译器会拒绝编译。若想把file.dat复制到file.c也不行，因为系统会认为这是无效的复制（防止用户错误）。

尽管对初学者而言，这类“保护”是有利的，但一些有经验的用户却感到很烦恼，因为他们要花很多精力来适应操作系统对合理和不合理操作的划分。

4.1.4 文件存取

早期操作系统只有一种文件存取方式：*顺序存取*（sequential access）。进程在这些系统中可从头顺序读取文件的全部字节或记录，但不能跳过某一些内容，也不能不按顺序读取。顺序存取文件是可以返回到起点的，需要时可多次读取该文件。在存储介质是磁带而不是磁盘时，顺序存取文件是很方便的。

当用磁盘来存储文件时，我们可以不按顺序地读取文件中的字节或记录，或者按照关键字而不是位置来存取记录。这种能够以任何次序读取其中字节或记录的文件称作*随机存取文件*（random access file）。许多应用程序需要这种类型的文件。

随机存取文件对很多应用程序而言是必不可少的，如数据库系统。如果乘客打电话预订某航班机票，订票程序必须能直接存取该航班记录，而不必先读出其他航班的成千上万个记录。

有两种方法可以指示从何处开始读取文件。一种是每次read操作都给出开始读文件的位置。另一种是用一个特殊的seek操作设置当前位置，在seek操作后，从这个当前位置顺序地开始读文件。UNIX和Windows使用的是后一种方法。

4.1.5 文件属性

文件都有文件名和数据。另外，所有的操作系统还会保存其他与文件相关的信息，如文件创建的日期和时间、文件大小等。这些附加信息称为文件*属性*（attribute），有些人称之为*元数据*（metadata）。文件的属性在不同系统中差别很大。一些常用的属性在图4-4中列出，但还存在其他的属性。没有一个系统具有所有这些属性，但每种属性都在某种系统中采用。

属　性	含　义
保护	谁可以存取文件，以什么方式存取文件
口令	存取文件需要的口令
创建者	创建文件者的ID
所有者	当前所有者
只读标志	0表示读/写；1表示只读
隐藏标志	0表示正常；1表示不在列表中显示
系统标志	0表示普通文件；1表示系统文件
存档标志	0表示已经备份；1表示需要备份
ASCII/二进制标志	0表示ASCII码文件；1表示二进制文件
随机存取标志	0表示只允许顺序存取；1表示随机存取
临时标志	0表示正常；1表示进程退出时删除该文件
加锁标志	0表示未加锁；非零表示加锁
记录长度	一个记录中的字节数
键的位置	每个记录中键的偏移量
键的长度	键字段的字节数
创建时间	文件创建的日期和时间
最后一次存取时间	文件上一次存取的日期和时间
最后一次修改时间	文件上一次修改的日期和时间
当前大小	文件的字节数
最大长度	文件可能增长到的字节数

图4-4 一些常用的文件属性

前4个属性与文件保护相关，它们指出了谁可以存取这个文件，谁不能存取这个文件。有各种不同的文件保护方案，其中一些保护方案以后会讨论。在一些系统中，用户必须给出口令才能存取文件。此时，口令也必须是文件属性之一。

标志是一些位或短的字段，用于控制或启用某些特殊属性。例如，隐藏文件不在文件列表中出现。存档标志位用于记录文件是否备份过，由备份程序清除该标志位；若文件被修改，操作系统则设置该标志位。用这种方法，备份程序可以知道哪些文件需要备份。临时标志表明当创建该文件的进程终止时，文件会被自动删除。

记录长度、键的位置和键的长度等字段只能出现在用关键字查找记录的文件里，它们提供了查找关键字所需的信息。

时间字段记录了文件的创建时间、最近一次存取时间以及最后一次修改时间，它们的作用不同。例如，目标文件生成后被修改的源文件需要重新编译生成目标文件。这些字段提供了必要的信息。

当前大小字段指出了当前的文件大小。在一些老式大型机操作系统中创建文件时，要给出文件的最大长度，以便操作系统事先按最大长度留出存储空间。工作站和和个人计算机中的操作系统则聪明多了，不需要这一点提示。

4.1.6 文件操作

使用文件的目的是存储信息并方便以后的检索。对于存储和检索，不同系统提供了不同的操作。以下是与文件有关的最常用的一些系统调用：

1) create。创建不包含任何数据的文件。该调用的目的是表示文件即将建立，并设置文件的一些属性。

2) delete。当不再需要某个文件时，必须删除该文件以释放磁盘空间。任何文件系统总有一个系统调用用来删除文件。

3) open。在使用文件之前，必须先打开文件。open调用的目的是：把文件属性和磁盘地址表装入内存，便于后续调用的快速存取。

4) close。存取结束后，不再需要文件属性和磁盘地址，这时应该关闭文件以释放内部表空间。很多系统限制进程打开文件的个数，以鼓励用户关闭不再使用的文件。磁盘以块为单位写入，关闭文件时，写入该文件的最后一块，即使这个块还没有满。

5) read。在文件中读取数据。一般地，读出数据来自文件的当前位置。调用者必须指明需要读取多少数据，并且提供存放这些数据的缓冲区。

6) write。向文件写数据，写操作一般也是从文件当前位置开始。如果当前位置是文件末尾，文件长度增加。如果当前位置在文件中间，则现有数据被覆盖，并且永远丢失。

7) append。此调用是write的限制形式，它只能在文件末尾添加数据。若系统只提供最小系统调用集合，则通常没有append。很多系统对同一操作提供了多种实现方法，这些系统中有时有append调用。

8) seek。对于随机存取文件，要指定从何处开始取数据，通常的方法是用seek系统调用把当前位置指针指向文件中特定位置。seek调用结束后，就可以从该位置开始读写数据了。

9) get attributes。进程运行常需要读取文件属性。例如，UNIX中make程序通常用于管理由多个源文件组成的软件开发项目。在调用make时，检查全部源文件和目标文件的修改时间，实现最小编译，使得全部文件都为最新版本。为达到此目的，需要查找文件的某一些属性，特别是修改时间。

10) set attributes。某些属性是可由用户设置的，在文件创建之后，用户还可以通过系统调用set attributes来修改它们。保护模式信息是一个显著的例子，大多数标志也属于此类属性。

11) rename。用户常常要改变已有文件的名字，rename系统调用用于这一目的。严格地说，设置这个系统调用不是十分必要的，因为可以先把文件复制到一个新文件名的文件中，然后删除原来的文件。

4.1.7 使用文件系统调用的一个示例程序

本节会考察一个简单的UNIX程序，它把文件从源文件处复制到目标文件处。程序清单如图4-5所示。该程序的功能很简单，甚至没有考虑出错报告处理，但它给出了有关文件的系统调用是怎样工作的一般思路。

例如，通过下面的命令行可以调用程序copyfile：

```
copyfile abc xyz
```

把文件abc复制到xyz。如果xyz已经存在，abc会覆盖它。否则，就创建它。程序调用必须提供两个参数，它们都是合法的文件名。第一个是源文件；第二个是输出文件。

在程序的开头是四个#include语句，它们把大量的定义和函数原型包含在这个程序。为了使程序遵守相应的国际标准，这些是需要的，无须作进一步的讨论。接下来一行是main函数的原型，这是ANSI C所必需的，但对我们的目的而言，它也不是重点。

接下来的第一个#define语句是一个宏定义，它把BUF_SIZE字符串定义为一个宏，其数值为4096。程序会读写若干个有4096个字节的块。类似地，给常数一个名称而且用这一名称代替常数是一种良好的编程习惯。这样的习惯不仅使程序易读，而且使程序易于维护。第二个#define 语句决定谁可以访问输出文件。

主程序名为main，它有两个参数：argc和argv。当调用这个程序时，操作系统提供这两个参数。第一个参数表示在调用该程序的命令行中包含多少个字符串，包括该程序名。它应该是3。第二个参数是指向程序参数的指针数组。在上面的示例程序中，这一数组的元素应该包含指向下列值的指针：

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

正是通过这个数组，程序访问其参数。

声明了五个变量。前面两个（in_fd和out_fd）用来保存文件描述符，即打开一个文件时返回一个小整数。后面两个（rd_count和wt_count）分别是由read和write系统调用所返回的字节计数。最后一个（buffer）是用于保存所读出的数据以及提供写入数据的缓冲区。

第一行实际语句检查argc，看它是否是3。如果不是，它以状态码1退出。任何非0的状态码均表示出错。在本程序中，状态码是惟一的出错报告处理。一个程序的产品版通常会打印出错信息。

```
/* 复制文件程序，有基本的错误检查和错误报告 */

#include <sys/types.h>                  /* 包括必要的头文件 */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI原型 */

#define BUF_SIZE 4096                   /* 使用一个4096字节大小的缓冲区 */
#define OUTPUT_MODE 0700                /* 输出文件的保护位 */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* 如果argc不等于3，语法错 */

    /* 打开输入文件并创建输出文件 */
    in_fd = open(argv[1], O_RDONLY);    /* 打开源文件 */
    if (in_fd < 0) exit(2);             /* 如果该文件不能打开，退出 */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* 创建目标文件 */
    if (out_fd < 0) exit(3);            /* 如果该文件不能被创建，退出 */

    /* 循环复制 */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* 读一块数据 */
    if (rd_count <= 0) break;           /* 如果文件结束或读时出错，退出循环 */
        wt_count = write(out_fd, buffer, rd_count); /* 写数据 */
        if (wt_count <= 0) exit(4);     /* wt_count <=0是一个错误 */
    }

    /* 关闭文件 */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* 没有读取错误 */
        exit(0);
    else
        exit(5);                        /* 有读取错发生 */
}
```

图4-5 复制文件的一个简单程序

接着我们试图打开源文件并创建目标文件。如果源文件成功打开，系统会给in_fd赋予一个小的整数，用以标识源文件。后续的调用必须引用这个整数，使系统知道需要的是哪一个文件。类似地，如果目标文件也成功地创建了，out_fd会被赋予一个标识用的值。create的第二个变量是设置保护模式。如果打开或创建文件失败，对应的文件描述符被设为-1，程序带着出错码退出。

接下来是用来复制文件的循环。一开始试图读出 4KB 数据到 buffer 中。它通过调用库过程 read来完成这项工作，该过程实际激活了read系统调用。第一个参数标识文件，第二个参数指定缓冲区，第三个参数指定读出多少字节。赋予rd_count的字节数是实际所读出的字节数。通常这个数是4096，除非文

件中只有少量字节。当到达文件尾部时，该参数的值是0。如果rd_count是零或负数，复制工作就不能再进行下去，所以执行break语句，用以中断循环（否则就无法结束了）。

调用write把缓冲区的内容输出到目标文件中去。第一个参数标识文件，第二个参数指定缓冲区，第三个参数指定写入多少字节，同read类似。注意字节计数是实际读出的字节数，不是BUF_SIZE。这一点是很重要的，因为最后一个缓冲区一般不会是4096，除非文件长度碰巧是4KB的倍数。

当整个文件处理完时，超出文件尾部的首次调用会把0值返回给rd_count，这样，程序会退出循环。此时，关闭两个文件，程序退出并附有正常完成的状态码。

尽管Windows的系统调用与UNIX的系统调用不同，但是Windows程序复制文件的命令行的一般结构与图4-5中的相当类似。我们将在第11章中考察Windows Vista的系统调用。

4.2 目录

文件系统通常提供*目录*或*文件夹*用于记录文件，在很多系统中目录本身也是文件。本节讨论目录、目录的组成、目录的特性和可以对目录进行的操作。

4.2.1 一级目录系统

目录系统的最简单形式是在一个目录中包含所有的文件。这有时称为*根目录*，但是由于只有一个目录，所以其名称并不重要。在早期的个人计算机中，这种系统很普遍，部分原因是因为只有一个用户。有趣的是，世界第一台超级计算机CDC 6600对于所有的文件也只有一个目录，尽管该机器同时被许多用户使用。这样决策毫无疑问是为了使软件设计简单。

一个单层目录系统的例子如图4-6所示。该目录中有四个文件。这一设计的优点在于简单，并且能够快速定位文件——事实上只有一个地方要查看。这种目录系统经常用于简单的嵌入式装置中，诸如电话、数码相机以及一些便携式音乐播放器等。

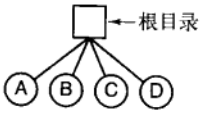


图4-6 含有四个文件的单层目录系统

4.2.2 层次目录系统

对于简单的特殊应用而言，单层目录是合适的（单层目录甚至用在了第一代个人计算机中），但是现在的用户有着成千的文件，如果所有的文件都在一个目录中，寻找文件就几乎不可能了。这样，就需要有一种方式将相关的文件组合在一起。例如，某个教授可能有一些文件，第一组文件是为了一门课程而写作的，第二组文件包含了学生为另一门课程所提交的程序，第三组文件是他构造的一个高级编译-写作系统的代码，而第四组文件是奖学金建议书，还有其他与电子邮件、短会、正在写作的文章、游戏等有关的文件。

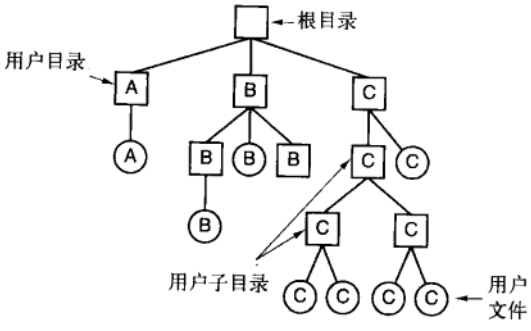


图4-7 层次目录系统

这里所需要的是层次结构（即，一个目录树）。通过这种方式，可以用很多目录把文件以自然的方式分组。进而，如果多个用户分享同一个文件服务器，如许多公司的网络系统，每个用户可以为自己的目录树拥有自己的私人根目录。这种方式如图4-7所示，其中，根目录含有目录A、B和C，分别属于不同用户，其中有两个用户为他们的项目创建了子目录。

用户可以创建任意数量的子目录，这种能力为用户组织其工作提供了强大的结构化工具。因此，几乎所有现代文件系统都是用这个方式组织的。

4.2.3 路径名

用目录树组织文件系统时，需要有某种方法指明文件名。常用的方法有两种。第一种是，每个文件都赋予一个*绝对路径名*（absolute path name），它由从根目录到文件的路径组成。例如，路径

/usr/ast/mailbox表示根目录中有子目录usr，而usr中又有子目录ast，文件mailbox就在子目录ast下。绝对路径名一定从根目录开始，且是惟一的。在UNIX中，路径各部分之间用“/”分隔。在Windows中，分隔符是“\”。在MULTICS中是“>”。这样在这三个系统中同样的路径名按如下形式写成：

```
Windows    \usr\ast\mailbox
UNIX       /usr/ast/mailbox
MULTICS    >usr>ast>mailbox
```

不管采用哪种分隔符，如果路径名的第一个字符是分隔符，则这个路径就是绝对路径。

另一种指定文件名的方法是使用*相对路径名*（relative path name）。它常和*工作目录*（working directory）（也称作*当前目录*（current directory））一起使用。用户可以指定一个目录作为当前工作目录。这时，所有的不从根目录开始的路径名都是相对于工作目录的。例如，如果当前的工作目录是/usr/ast，则绝对路径名为/usr/ast/mailbox的文件可以直接用mailbox来引用。也就是说，如果工作目录是/usr/ast，则UNIX命令

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

和命令

```
cp mailbox mailbox.bak
```

具有相同的含义。相对路径往往更方便，而它实现的功能和绝对路径完全相同。

一些程序需要存取某个特定文件，而不论当前目录是什么。这时，应该采用绝对路径名。比如，一个检查拼写的程序要读文件/usr/lib/dictionary，因为它不可能事先知道当前目录，所以就采用完整的绝对路径名。不论当前的工作目录是什么，绝对路径名总能正常工作。

当然，若这个检查拼写的程序要从目录/usr/lib中读很多文件，可以用另一种方法，即执行一个系统调用把该程序的工作目录切换到/usr/lib，然后只需用dictionary作为open的第一个参数。通过显式地改变工作目录，可以知道该程序在目录树中的确切位置，进而可以采用相对路径名。

每个进程都有自己的工作目录，这样在进程改变工作目录并退出后，其他进程不会受到影响，文件系统中也不会有改变的痕迹。对进程而言，切换工作目录是安全的，所以只要需要，就可以改变当前工作目录。但是，如果改变了库过程的工作目录，并且工作完毕之后没有修改回去，则其他程序有可能无法正常运行，因为它们关于当前目录的假设已经失效。所以库过程很少改变工作目录，若非改不可，必定要在返回之前改回到原有的工作目录。

支持层次目录结构的大多数操作系统在每个目录中有两个特殊的目录项“.”和“..”，常读作“dot”和“dotdot”。dot指当前目录，dotdot指其父目录（在根目录中例外，根目录中它指向自己）。要了解怎样使用它们，请考虑图4-8中的UNIX目录树。一个进程的工作目录是/usr/ast，它可采用“..”沿树向上。例如，可用命令

```
cp ../lib/dictionary .
```

把文件usr/lib/dictionary复制到自己的目录下。第一个路径告诉系统上溯（到usr目录），然后向下到lib目录，找到dictionary文件。

第二个参数（.）指定当前目录。当cp命令用目录名（包括“.”）作为最后一个参数时，则把全部的文件复制到该目录中。当然，对于上述复制，键入

```
cp /usr/lib/dictionary .
```

是更常用的方法。用户这里采用“.”可以避免键入两次dictionary 。无论如何，键入

```
cp /usr/lib/dictionary dictionary
```

也可正常工作，就像键入

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

一样。所有这些命令都完成同样的工作。

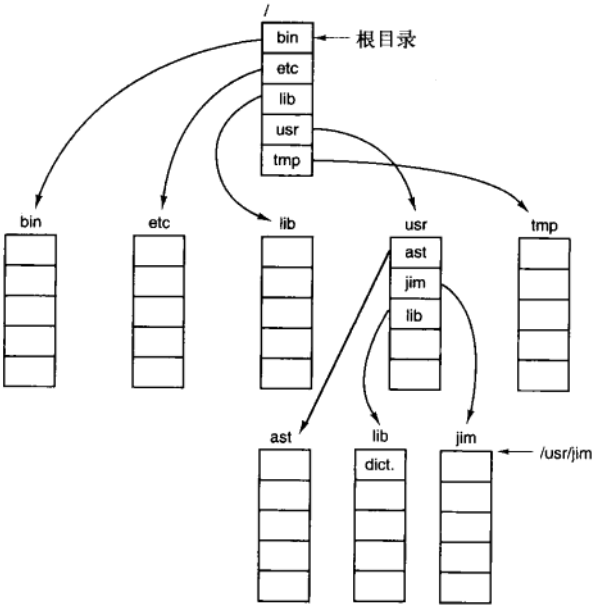


图4-8 UNIX目录树

4.2.4 目录操作

不同系统中管理目录的系统调用的差别比管理文件的系统调用的差别大。为了了解这些系统调用有哪些及它们怎样工作，下面给出一个例子（取自UNIX）。

1) create。创建目录。除了目录项“.”和“..”外，目录内容为空。目录项“.”和“..”是系统自动放在目录中的（有时通过mkdir程序完成）。

2) delete。删除目录。只有空目录可删除。只包含目录项“.”和“..”的目录被认为是空目录，这两个目录项通常不能删除。

3) opendir。目录内容可被读取。例如，为列出目录中全部文件，程序必须先打开该目录，然后读其中全部文件的文件名。与打开和读文件相同，在读目录前，必须打开目录。

4) closedir。读目录结束后，应关闭目录以释放内部表空间。

5) readdir。系统调用readdir返回打开目录的下一个目录项。以前也采用read系统调用来读目录，但这方法有一个缺点：程序员必须了解和处理目录的内部结构。相反，不论采用哪一种目录结构，readdir总是以标准格式返回一个目录项。

6) rename。在很多方面目录和文件都相似。文件可换名，目录也可以。

7) link。连接技术允许在多个目录中出现同一个文件。这个系统调用指定一个存在的文件和一个路径名，并建立从该文件到路径所指名字的连接。这样，可以在多个目录中出现同一个文件。这种类型的连接，增加了该文件 的i节点（i-node）计数器的计数（记录含有该文件的目录项数目），有时称为*硬连接*（hard link）。

8) unlink。删除目录项。如果被解除连接的文件只出现在一个目录中（通常情况），则将它从文件系统中删除。如果它出现在多个目录中，则只删除指定路径名的连接，依然保留其他路径名的连接。在UNIX中，用于删除文件的系统调用（前面已有论述）实际上就是unlink。

最主要的系统调用已在上面列出，但还有其他一些调用，如与目录相关的管理保护信息的系统调用。

关于连接文件的一种不同想法是*符号连接*。不同于使用两个文件名指向同一个内部数据结构来代表一个文件，所建立的文件名指向了命名另一个文件的小文件。当使用第一个文件时，例如打开时，文件系统沿着路径，找到在末端的名字。然后它使用该新名字启动查找进程。符号连接的优点在于它能够跨越磁盘的界限，甚至可以命名在远程计算机上的文件，不过符号连接的实现并不如硬连接那样有效率。

4.3 文件系统的实现

现在从用户角度转到实现者角度来考察文件系统。用户关心的是文件是怎样命名的、可以进行哪些操作、目录树是什么样的以及类似的界面问题。而实现者感兴趣的是文件和目录是怎样存储的、磁盘空间是怎样管理的以及怎样使系统有效而可靠地工作等。在下面几节中，我们会考察这些文件系统的实现中出现的问题，并讨论怎样解决这些问题。

4.3.1 文件系统布局

文件系统存放在磁盘上。多数磁盘划分为一个或多个分区，每个分区中有一个独立的文件系统。磁盘的0号扇区称为主引导记录（Master Boot Record，MBR），用来引导计算机。在MBR的结尾是分区表。该表给出了每个分区的起始和结束地址。表中的一个分区被标记为活动分区。在计算机被引导时，BIOS读入并执行MBR。MBR做的第一件事是确定活动分区，读入它的第一个块，称为引导块（boot block），并执行之。引导块中的程序将装载该分区中的操作系统。为统一起见，每个分区都从一个启动块开始，即使它不含有一个可启动的操作系统。不过，在将来这个分区也许会有一个操作系统的。

除了从引导块开始之外，磁盘分区的布局是随着文件系统的不同而变化的。文件系统经常包含有如图4-9所列的一些项目。第一个是超级块（superblock），超级块包含文件系统的所有关键参数，在计算机启动时，或者在该文件系统首次使用时，把超级块读入内存。超级块中的典型信息包括：确定文件系统类型用的魔数、文件系统中数据块的数量以及其他重要的管理信息。

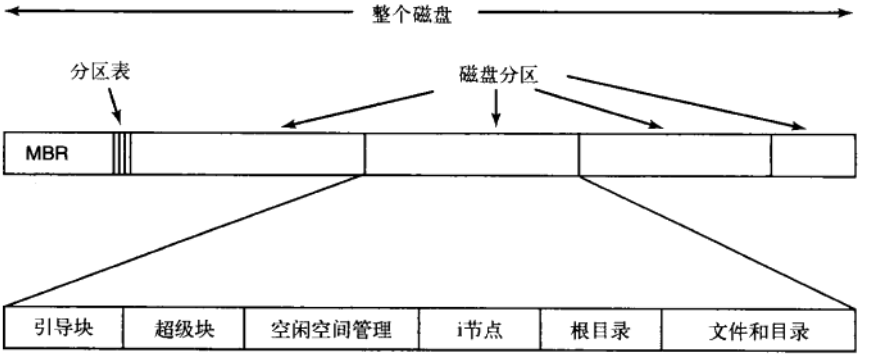


图4-9 一个可能的文件系统布局

接着是文件系统中空闲块的信息，例如，可以用位图或指针列表的形式给出。后面也许跟随的是一组i节点，这是一个数据结构数组，每个文件一个，i节点说明了文件的方方面面。接着可能是根目录，它存放文件系统目录树的根部。最后，磁盘的其他部分存放了其他所有的目录和文件。

4.3.2 文件的实现

文件存储的实现的关键问题是记录各个文件分别用到哪些磁盘块。不同操作系统采用不同的方法。这一节，我们讨论其中的一些方法。

1. 连续分配

最简单的分配方案是把每个文件作为一连串连续数据块存储在磁盘上。所以，在块大小为1KB的磁盘上，50KB的文件要分配50个连续的块。对于块大小为2KB的磁盘，将分配25个连续的块。

在图4-10a中是一个连续分配的例子。这里列出了头40块，从左面从0块开始。初始状态下，磁盘是空的。接着，从磁盘开始处（块0）开始写入长度为4块的文件A。紧接着，在文件A的结尾开始写入一个3块的文件B。

请注意，每个文件都从一个新的块开始，这样如果文件A实际上只有$3^1/_2$块，那么最后一块的结尾会浪费一些空间。在图4-10中，一共列出了7个文件，每一个都从前面文件结尾的后续块开始。加阴影是为了容易表示文件分隔，在存储中并没有实际的意义。

连续磁盘空间分配方案有两大优势。首先，实现简单，记录每个文件用到的磁盘块简化为只需记住两个数字即可：第一块的磁盘地址和文件的块数。给定了第一块的编号，一个简单的加法就可以找到任何其他块的编号。

其次，读操作性能较好，因为在单个操作中就可以从磁盘上读出整个文件。只需要一次寻找（对第一个块）。之后不再需要寻道和旋转延迟，所以，数据以磁盘全带宽的速率输入。可见连续分配实现简单且具有高的性能。

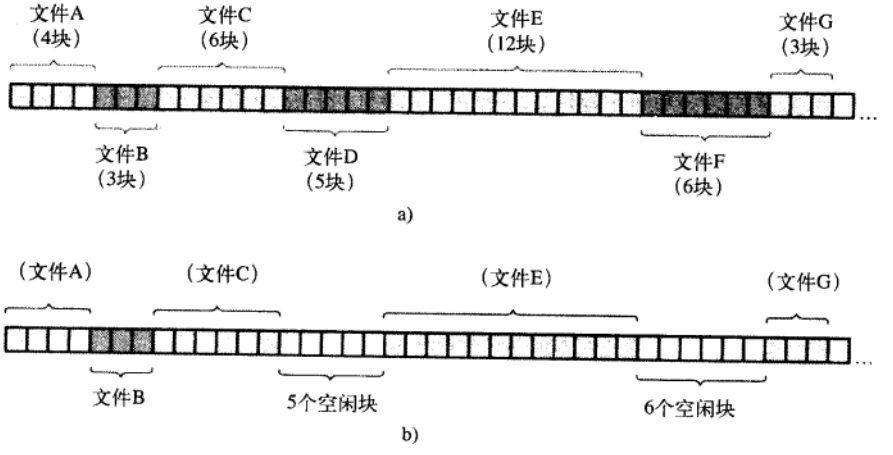


图4-10 a) 为7个文件连续分配空间；b) 删除文件D和F后磁盘的状态

但是，连续分配方案也同样有相当明显的不足之处：随着时间的推移，磁盘会变得零碎。为了了解这是如何发生的，请考察图4-10b。这里有两个文件（D和F）被删除了。当删除一个文件时，它占用的块自然就释放了，在磁盘上留下一堆空闲块。磁盘不会在这个位置挤压掉这个空洞，因为这样会涉及复制空洞之后的所有文件，可能会有上百万的块。结果是，磁盘上最终既包括文件也有空洞，如图4-10中所描述的那样。

开始时，碎片并不是问题，因为每个新的文件都在先前文件的磁盘结尾写入。但是，磁盘最终会被充满，所以要么压缩磁盘，要么重新使用空洞中的空闲空间。前者由于代价太高而不可行；后者需要维护一个空洞列表，这是可行的。但是，当创建一个新的文件时，为了挑选合适大小的空洞存入文件，就有必要知道该文件的最终大小。

设想这样一种设计的结果：为了录入一个文档，用户启动了文本编辑器或字处理软件。程序首先询问最终文件的大小会是多少。这个问题必须回答，否则程序就不能继续。如果给出的数字最后被证明小于文件的实际大小，该程序会终止，因为所使用的磁盘空洞已经满了，没有地方放置文件的剩余部分。如果用户为了避免这个问题而给出不实际的较大的数字作为最后文件的大小，比如，100 MB，编辑器可能找不到如此大的空洞，从而宣布无法创建该文件。当然，用户有权下一次使用比如50MB的数字再次启动编辑器，如此进行下去，直到找到一个合适的空洞为止。不过，这种方式看来不会使用户高兴。

然而，存在着一种情形，使得连续分配方案是可行的，而且，实际上这个办法在CD-ROM上被广泛使用着。在这里所有文件的大小都事先知道，并且在CD-ROM 文件系统的后续使用中，这些文件的大小也不再改变。在本章的后面，我们将讨论最常见的CD-ROM 文件系统。

DVD的情况有些复杂。原则上，一个90分钟的电影可以编码成一个独立的、大约4.5GB的文件。但是文件系统所使用的UDF（Universal Disk Format）格式，使用了一个30位的数来代表文件长度，从而把文件大小限制在1GB。其结果是，DVD电影一般存储在3个或4个1GB的连续文件中。这些构成一个逻辑文件（电影）的物理文件块被称作extents。

正如第1章中所提到的，在计算机科学中，随着新一代技术的出现，历史往往重复着自己。多年前，连续分配由于其简单和高性能（没有过多考虑用户友好性）被实际用在磁盘文件系统中。后来由于讨厌

在文件创建时不得不指定最终文件的大小，这个想法被放弃了。但是，随着CD-ROM、DVD以及其他一次性写光学介质的出现，突然间连续分配又成为一个好主意。所以研究那些具有清晰和简洁概念的老式系统和思想是很重要的，因为它们有可能以一种令人吃惊的方式在未来系统中获得应用。

2. 链表分配

存储文件的第二种方法是为每个文件构造磁盘块链表，如图4-11所示。每个块的第一个字作为指向下一块的指针，块的其他部分存放数据。

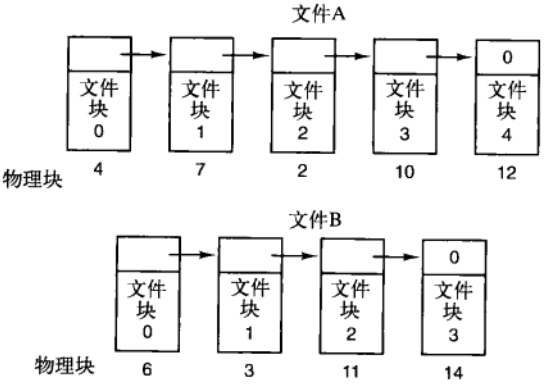


图4-11 以磁盘块的链表形式存储文件

与连续分配方案不同，这一方法可以充分利用每个磁盘块。不会因为磁盘碎片（除了最后一块中的内部碎片）而浪费存储空间。同样，在目录项中，只需要存放第一块的磁盘地址，文件的其他块就可以从这个首块地址查找到。

另一方面，在链表分配方案中，尽管顺序读文件非常方便，但是随机存取却相当缓慢。要获得块n，操作系统每一次都必须从头开始，并且要先读前面的$n-1$块。显然，进行如此多的读操作太慢了。

而且，由于指针占去了一些字节，每个磁盘块存储数据的字节数不再是2的整数次幂。虽然这个问题并不是非常严重，但是怪异的大小确实降低了系统的运行效率，因为许多程序都是以长度为2的整数次幂来读写磁盘块的。由于每个块的前几个字节被指向下一个块的指针所占据，所以要读出完整的一个块，就需要从两个磁盘块中获得和拼接信息，这就因复制引发了额外的开销。

3. 在内存中采用表的链表分配

如果取出每个磁盘块的指针字，把它放在内存的一个表中，就可以解决上述链表的两个不足。图4-12表示了图4-11所示例子的内存中表的内容。这两个图中有两个文件，文件A依次使用了磁盘块4、7、2、10和12，文件B依次使用了磁盘块6、3、11和14。利用图4-12中的表，可以从第4块开始，顺着链走到最后，找到文件A的全部磁盘块。同样，从第6块开始，顺着链走到最后，也能够找出文件B的全部磁盘块。这两个链都以一个不属于有效磁盘编号的特殊标记（如−1）结束。内存中的这样一个表格称为文件分配表（File Allocation Table，FAT）。

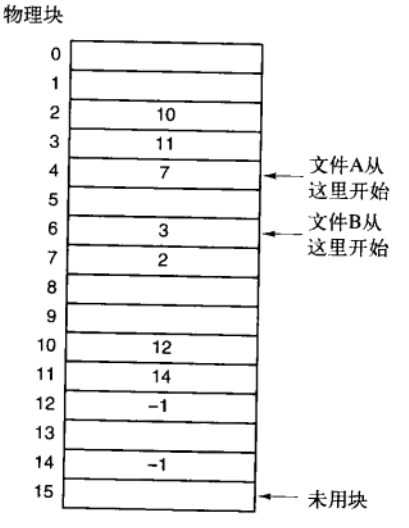


图4-12 在内存中使用文件分配表的链表分配

按这类方式组织，整个块都可以存放数据。进而，随机存取也容易得多。虽然仍要顺着链在文件中查找给定的偏移量，但是整个链表都存放在内存中，所以不需要任何磁盘引用。与前面的方法相同，不管文件有多大，在目录项中只需记录一个整数（起始块号），按照它就可以找到文件的全部块。

这种方法的主要缺点是必须把整个表都存放在内存中。对于200 GB的磁盘和1KB大小的块，这张表需要有2亿项，每一项对应于这2亿个磁盘块中的一个块。每项至少3个字节，为了提高查找速度，有时需要4个字节。根据系统对空间或时间的优化方案，这张表要占用600MB或800MB内存，不太实用。很显然FAT方案对于大磁盘而言不太合适。

4. i节点

最后一个记录各个文件分别包含哪些磁盘块的方法是给每个文件赋予一个称为i节点（index-node）的数据结构，其中列出了文件属性和文件块的磁盘地址。图4-13中是一个简单例子的描述。给定i节点，就有可能找到文件的所有块。相对于在内存中采用表的方式而言，这种机制具有很大的优势，即只有在对应文件打开时，其i节点才在内存中。如果每个i节点占有n个字节，最多k个文件同时打开，那么为了打开文件而保留i节点的数组所占据的全部内存仅仅是kn个字节。只需要提前保留少量的空间。

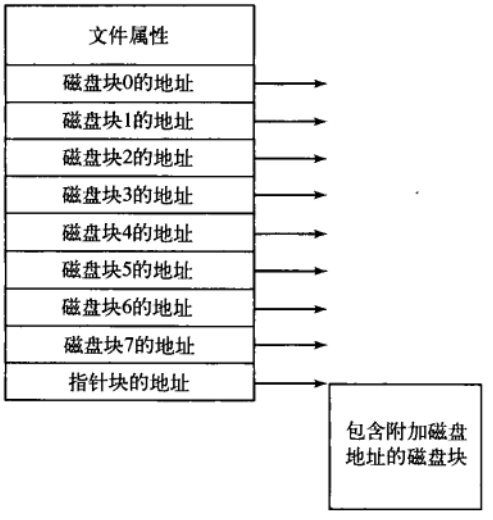


图4-13 i节点的例子

这个数组通常比上一节中叙述的文件分配表(FAT)所占据的空间要小。其原因很简单，保留所有磁盘块的链接表的表大小正比于磁盘自身的大小。如果磁盘有n块，该表需要n个表项。由于磁盘变得更大，该表格也线性随之增加。相反，i节点机制需要在内存中有一个数组，其大小正比于可能要同时打开的最大文件个数。它与磁盘是10GB、100GB还是1000GB无关。

i节点的一个问题是，如果每个i节点只能存储固定数量的磁盘地址，那么当一个文件所含的磁盘块的数目超出了i节点所能容纳的数目怎么办？一个解决方案是最后一个“磁盘地址”不指向数据块，而是指向一个包含磁盘块地址的块的地址，如图4-13所示。更高级的解决方案是：可以有两个或更多个包含磁盘地址的块，或者指向其他存放地址的磁盘块的磁盘块。在后面讨论UNIX时，我们还将涉及i节点。

4.3.3 目录的实现

在读文件前，必须先打开文件。打开文件时，操作系统利用用户给出的路径名找到相应目录项。目录项中提供了查找文件磁盘块所需要的信息。因系统而异，这些信息有可能是整个文件的磁盘地址（对于连续分配方案）、第一个块的编号（对于两种链表分配方案）或者是i节点号。无论怎样，目录系统的主要功能是把ASCII文件名映射成定位文件数据所需的信息。

与此密切相关的问题是在何处存放文件属性。每个文件系统维护诸如文件所有者以及创建时间等文件属性，它们必须存储在某个地方。一种显而易见的方法是把文件属性直接存放在目录项中。很多系统确实是这样实现的。这个办法用图4-14a说明。在这个简单设计中，目录中有一个固定大小的目录项列表，每个文件对应一项，其中包含一个（固定长度）文件名、一个文件属性结构以及用以说明磁盘块位置的一个或多个磁盘地址（至某个最大值）。

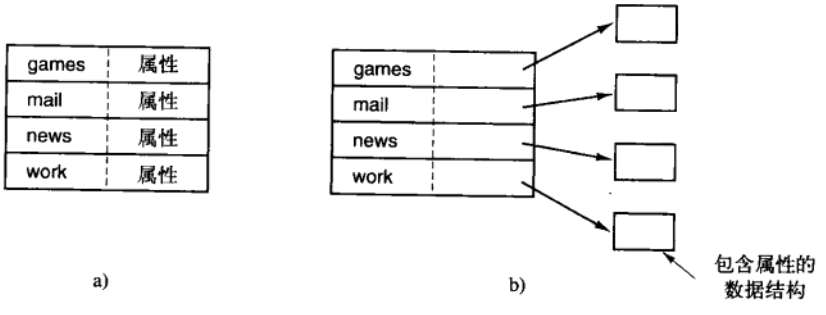


图4-14 a) 简单目录，包含固定大小的目录项，在目录项中有磁盘地址和属性；b) 每个目录项只引用i节点的目录

对于采用i节点的系统，还存在另一种方法，即把文件属性存放在i节点中而不是目录项中。在这种

情形下，目录项会更短：只有文件名和i节点号。这种方法参见图4-14b。后面我们会看到，与把属性存放到目录项中相比，这种方法更好。图4-14中的两种处理方法分别对应Windows和UNIX，在后面我们将讨论它们。

到目前为止，我们已经假设文件具有较短的、固定长度的名字。在MS-DOS中，文件有1~8个字符的基本名和1~3字符的可选扩展名。在UNIX V7中文件名有1~14个字符，包括任何扩展名。但是，几乎所有的现代操作系统都支持可变长度的长文件名。那么它们是如何实现的呢？

最简单的方法是给予文件名一个长度限制，典型值为255个字符，然后使用图4-14中的一种设计，并为每个文件名保留255个字符空间。这种处理很简单，但是浪费了大量的目录空间，因为只有很少的文件会有如此长的名字。从效率考虑，我们希望有其他的结构。

一种替代方案是放弃“所有目录项大小一样”的想法。这种方法中，每个目录项有一个固定部分，这个固定部分通常以目录项的长度开始，后面是固定格式的数据，通常包括所有者、创建时间、保护信息以及其他属性。这个固定长度的头的后面是实际文件名，可能是如图4-15a中的正序格式放置（如SPARC机器）[⊖]。在这个例子中，有三个文件，project-budget 、personnel和 foo。每个文件名以一个特殊字符（通常是0）结束，在图4-15中用带叉的矩形表示。为了使每个目录项从字的边界开始，每个文件名被填充成整数个字，如图4-15中带阴影的矩形所示。

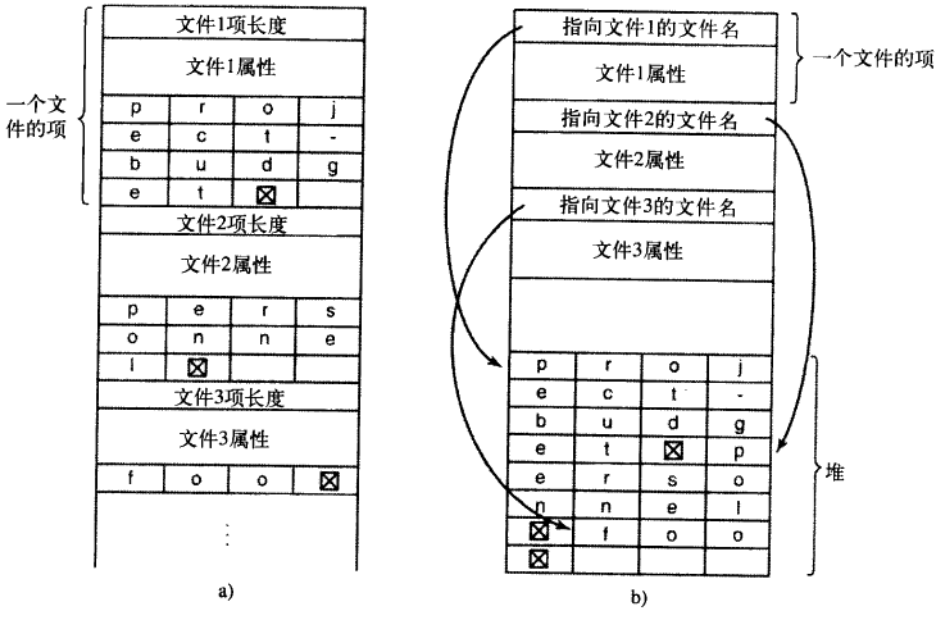


图4-15 在目录中处理长文件名的两种方法：a) 在行中；b) 在堆中

这个方法的缺点是，当移走文件后，就引入了一个长度可变的空隙，而下一个进来的文件不一定正好适合这个空隙。这个问题与我们已经看到的连续磁盘文件的问题是一样的，由于整个目录在内存中，所以只有对目录进行紧凑操作才可节省空间。另一个问题是，一个目录项可能会分布在多个页面上，在读取文件名时可能发生页面故障。

处理可变长度文件名字的另一种方法是，使目录项自身都有固定长度，而将文件名放置在目录后面的堆中，如图4-15b所示。这一方法的优点是，当一个文件目录项被移走后，另一个文件的目录项总是可以适合这个空隙。当然，必须要对堆进行管理，而在处理文件名时页面故障仍旧会发生。另一个小优

⊖ 处理机中的一串字符存放的顺序有正序（big-endian）和逆序（little-endian）之分。正序存放就是高字节存放在前低字节在后，而逆序存放就是低字节在前高字节在后。例如，十六进制数为A02B，正序存放就是A02B，逆序存放就是2BA0。——译者注

点是文件名不再需要从字的边界开始，这样，原先在图4-15a中需要的填充字符，在图4-15b中的文件名之后就不再需要了。

到目前为止，在需要查找文件名时，所有的方案都是线性地从头到尾对目录进行搜索。对于非常长的目录，线性查找就太慢了。加快查找速度的一个方法是在每个目录中使用散列表。设表的大小为n。在输入文件名时，文件名被散列到1和$n-1$之间的一个值，例如，它被n除，并取余数。其他可以采用的方法有，对构成文件名的字求和，其结果被n除，或某些类似的方法。

不论哪种方法都要对与散列码相对应的散列表表项进行检查。如果该表项没有被使用，就将一个指向文件目录项的指针放入，文件目录项紧连在散列表后面。如果该表项被使用了，就构造一个链表，该链表的表头指针存放在该表项中，并链接所有具有相同散列值的文件目录项。

查找文件按照相同的过程进行。散列处理文件名，以便选择一个散列表项。检查链表头在该位置上的所有表项，查看要找的文件名是否存在。如果名字不在该链上，该文件就不在这个目录中。

使用散列表的优点是查找非常迅速。其缺点是需要复杂的管理。只有在预计系统中的目录经常会有成百上千个文件时，才把散列方案真正作为备用方案考虑。

一种完全不同的加快大型目录查找速度的方法是，将查找结果存入高速缓存。在开始查找之前，先查看文件名是否在高速缓存中。如果是，该文件可以立即定位。当然，只有在构成查找主体的文件非常少的时候，高速缓存的方案才有效果。

4.3.4 共享文件

当几个用户同在一个项目里工作时，他们常常需要共享文件。其结果是，如果一个共享文件同时出现在属于不同用户的不同目录下，工作起来就很方便。图4-16再次给出图4-7所示的文件系统，只是C的一个文件现在也出现在B的目录下。B的目录与该共享文件的联系称为一个连接（link）。这样，文件系统本身是一个有向无环图（Directed Acyclic Graph，DAG）而不是一棵树。

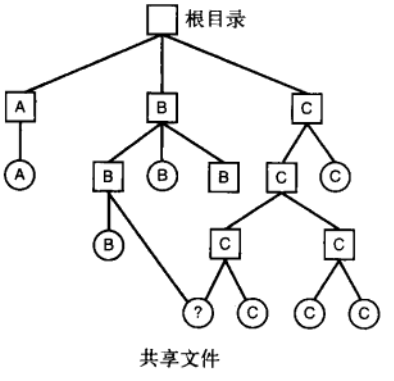


图4-16 有共享文件的文件系统

共享文件是方便的，但也带来一些问题。如果目录中包含磁盘地址，则当连接文件时，必须把C目录中的磁盘地址复制到B目录中。如果B或C随后又往该文件中添加内容，则新的数据块将只列入进行添加工作的用户的目录中。其他的用户对此改变是不知道的。所以违背了共享的目的。

有两种方法可以解决这一问题。在第一种解决方案中，磁盘块不列入目录，而是列入一个与文件本身关联的小型数据结构中。目录将指向这个小型数据结构。这是UNIX系统中所采用的方法（小型数据结构即是i节点）。

在第二种解决方案中，通过让系统建立一个类型为LINK的新文件，并把该文件放在B的目录下，使得B与C的一个文件存在连接。新的文件中只包含了它所连接的文件的路径名。当B读该连接文件时，操作系统查看到要读的文件是LINK类型，则找到该文件所连接的文件的名字，并且去读那个文件。与传统（硬）连接相对比起来，这一方法称为符号连接（symbolic linking）。

以上每一种方法都有其缺点。第一种方法中，当B连接到共享文件时，i节点记录文件的所有者是C。建立一个连接并不改变所有关系（见图4-17），但它将i节点的连接计数加1，所以系统知道目前有多少目录项指向这个文件。

如果以后C试图删除这个文件，系统将面临问题。如果系统删除文件并清除i节点，B则有一个目录项指向一个无效的i节点。如果该i节点以后分配给另一个文件，则B的连接指向一个错误的文件。系统通过i节点中的计数可知该文件仍然被引用，但是没有办法找到指向该文件的全部目录项以删除它们。指向目录的指针不能存储在i节点中，原因是有可能有无数个目录。

惟一能做的就是只删除C的目录项，但是将i节点保留下来，并将计数置为1，如图4-17c所示。而现在的状况是，只有B有指向该文件的目录项，而该文件的所有者是C。如果系统进行记账或有配额，那么C将继续为该文件付账直到B决定删除它，如果真是这样，只有到计数变为0的时刻，才会删除该文件。

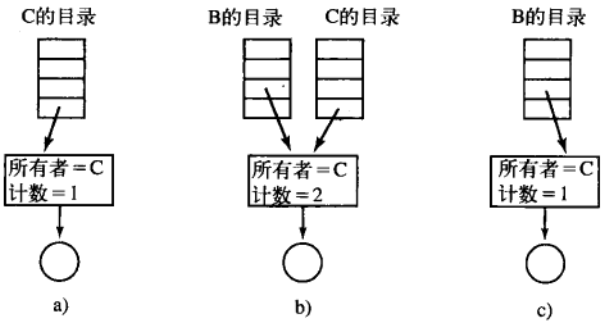


图4-17 a) 连接之前的状况；b) 创建连接之后；c) 当所有者删除文件后

对于符号连接，以上问题不会发生，因为只有真正的文件所有者才有一个指向i节点的指针。连接到该文件上的用户只有路径名，没有指向i节点的指针。当文件所有者删除文件时，该文件被销毁。以后若试图通过符号连接访问该文件将导致失败，因为系统不能找到该文件。删除符号连接根本不影响该文件。

符号连接的问题是需要额外的开销。必须读取包含路径的文件，然后要一个部分一个部分地扫描路径，直到找到i节点。这些操作也许需要很多次额外的磁盘存取。此外，每个符号连接都需要额外的i节点，以及额外的一个磁盘块用于存储路径，虽然如果路径名很短，作为一种优化，系统可以将它存储在i节点中。符号连接有一个优势，即只要简单地提供一个机器的网络地址以及文件在该机器上驻留的路径，就可以连接全球任何地方的机器上的文件。

还有另一个由连接带来的问题，在符号连接和其他方式中都存在。如果允许连接，文件有两个或多个路径。查找一指定目录及其子目录下的全部文件的程序将多次定位到被连接的文件。例如，一个将某一目录及其子目录下的文件转储到磁带上的程序有可能多次复制一个被连接的文件。进而，如果接着把磁带读进另一台机器，除非转储程序具有智能，否则被连接的文件将被两次复制到磁盘上，而不是只是被连接起来。

4.3.5 日志结构文件系统

不断进步的科技给现有的文件系统带来了更多的挑战。特别是CPU的运行速度越来越快，磁盘容量越来越大，价格也越来越便宜（但是磁盘速度并没有增快多少），同时内存容量也以指数形式增长。而没有得到快速发展的参数是磁盘的寻道时间。所以这些问题综合起来，便成为影响很多文件系统性能的一个瓶颈。为此，Berkeley设计了一种全新的文件系统，试图缓解这个问题，即日志结构文件系统（Log-structured File System，LFS）。在这一节里，我们简要说明LFS是如何工作的。如果需要了解更多相关知识，请参阅（Rosenblum 和Ousterhout，1991）。

促使设计LFS的主要原因是，CPU的运行速度越来越快，RAM内存容量变得更大，同时磁盘高速缓存也迅速地增加。进而，不需要磁盘访问操作，就有可能满足直接来自文件系统高速缓存的很大一部分读请求。所以从上面的事实可以推出，未来多数的磁盘访问是写操作，这样，在一些文件系统中使用的提前读机制（需要读取数据之前预取磁盘块），并不能获得更好的性能。

更为糟糕的情况是，在大多数文件系统中，写操作往往都是零碎的。一个50μs的磁盘写操作之前通常需要10ms的寻道时间和4ms的旋转延迟时间，可见零碎的磁盘写操作是极其没有效率的。根据这些参数，磁盘的效率降低到1%以下。

为了看看这样小的零碎写操作从何而来，考虑在UNIX文件系统上创建一个新文件。为了写这个文件，必须写该文件目录的i节点、目录块、文件的i节点以及文件本身。而这些写操作都有可能被延迟，那么如果在写操作完成之前发生死机，就可能在文件系统中造成严重的不一致性。正因为如此，i节点的写操作一般是立即完成的。

出于这一原因，LFS的设计者决定重新实现一种UNIX文件系统，该系统即使对于一个大部分由零碎的随机写操作组成的任务，同样能够充分利用磁盘的带宽。其基本思想是将整个磁盘结构化为一个日志。每隔一段时间，或是有特殊需要时，被缓冲在内存中的所有未决的写操作都被放到一个单独的段中，

作为在日志末尾的一个邻接段写入磁盘。一个单独的段可能会包括i节点、目录块、数据块或者都有。每一个段的开始都是该段的摘要，说明该段中都包含哪些内容。如果所有的段平均在1MB左右，那么就几乎可以利用磁盘的完整带宽。

在LFS的设计中，同样存在着i节点，且具有与UNIX中一样的结构，但是i节点分散在整个日志中，而不是放在磁盘的某一个固定位置。尽管如此，当一个i节点被定位后，定位一个块就用通常的方式来完成。当然，由于这种设计，要在磁盘中找到一个i节点就变得比较困难了，因为i节点的地址不能像在UNIX中那样简单地通过计算得到。为了能够找到i节点，必须要维护一个由i节点编号索引组成的i节点图。在这个图中的表项*i*指向磁盘中的第*i*个i节点。这个图保存在磁盘上，但是也保存在高速缓存中，因此，大多数情况下这个图的最常用部分还是在内存中。

总而言之，所有的写操作最初都被缓冲在内存中，然后周期性地把所有已缓冲的写作为一个单独的段，在日志的末尾处写入磁盘。要打开一个文件，则首先需要从i节点图中找到文件的i节点。一旦i节点定位之后就可以找到相应的块的地址。所有的块都放在段中，在日志的某个位置上。

如果磁盘空间无限大，那么有了前面的讨论就足够了。但是，实际的硬盘空间是有限的，这样最终日志将会占用整个磁盘，到那个时候将不能往日志中写任何新的段。幸运的是，许多已有的段包含了很多不再需要的块，例如，如果一个文件被覆盖了，那么它的i节点就会指向新的块，但是旧的磁盘块仍然在先前写入的段中占据着空间。

为了解决这个问题，LFS有一个*清理线程*，该清理线程周期地扫描日志进行磁盘压缩。该线程首先读日志中的第一个段的摘要，检查有哪些i节点和文件。然后该线程查看当前i节点图，判断该i节点是否有效以及文件块是否仍在使用中。如果没有使用，则该信息被丢弃。如果仍然使用，那么i节点和块就进入内存等待写回到下一个段中。接着，原来的段被标记为空闲，以便日志可以用它来存放新的数据。用这种方法，清理线程遍历日志，从后面移走旧的段，然后将有效的数据放入内存等待写到下一个段中。由此，整个磁盘成为一个大的环形的缓冲区，写线程将新的段写到前面，而清理线程则将旧的段从后面移走。

日志的管理并不简单，因为当一个文件块被写回到一个新段的时候，该文件的i节点（在日志的某个地方）必须首先要定位、更新，然后放到内存中准备写回到下一个段中。i节点图接着必须更新以指向新的位置。尽管如此，对日志进行管理还是可行的，而且性能分析的结果表明，这种由管理而带来的复杂性是值得的。在上面所引用文章中的测试数据表明，LFS在处理大量的零碎的写操作时性能上优于UNIX，而在读和大块写操作的性能方面并不比UNIX文件系统差，甚至更好。

4.3.6 日志文件系统

虽然基于日志结构的文件系统是一个很吸引人的想法，但是由于它们和现有的文件系统不相匹配，所以还没有被广泛应用。尽管如此，它们内在的一个思想，即面对出错的鲁棒性，却可以被其他文件系统所借鉴。这里的基本想法是保存一个用于记录系统下一步将要做什么的日志。这样当系统在完成它们即将完成的任务前崩溃时，重新启动后，可以通过查看日志，获取崩溃前计划完成的任务，并完成它们。这样的文件系统被称为*日志文件系统*，并已经被实际应用。微软（Microsoft）的NTFS文件系统、Linux ext3和ReiserFS文件系统都使用日志。接下来，我们会对这个话题进行简短介绍。

为了看清这个问题的实质，考虑一个简单、普通并经常发生的操作：移除文件。这个操作（在UNIX中）需要三个步骤完成：

1) 在目录中删除文件；

2) 释放i节点到空闲i节点池；

3) 将所有磁盘块归还空闲磁盘块池。

在Windows中，也需要类似的步骤。不存在系统崩溃时，这些步骤执行的顺序不会带来问题；但是当存在系统崩溃时，就会带来问题。假如在第一步完成后系统崩溃。i节点和文件块将不会被任何文件获得，也不会被再分配；它们只存在于废物池中的某个地方，并因此减少了可利用的资源。如果崩溃发生在第二步后，那么只有磁盘块会丢失。

如果操作顺序被更改，并且i节点最先被释放，这样在系统重启后，i节点可以被再分配，但是旧的目录入口将继续指向它，因此指向错误文件。如果磁盘块最先被释放，这样一个在i节点被清除前的系

统崩溃将意味着一个有效的目录入口指向一个i节点，它所列出的磁盘块当前存在于空闲块存储池中并可能很快被再利用。这将导致两个或更多的文件分享同样的磁盘块。这样的结果都是不好的。

日志文件系统则先写一个日志项，列出三个将要完成的动作。然后日志项被写入磁盘（并且为了良好地实施，可能从磁盘读回来验证它的完整性）。只有当日志项已经被写入，不同的操作才可以进行。当所有的操作成功完成后，擦除日志项。如果系统这时崩溃，系统恢复后，文件系统可以通过检查日志来查看是不是有未完成的操作。如果有，可以重新运行所有未完成的操作（这个过程在系统崩溃重复发生时执行多次），直到文件被正确地删除。

为了让日志文件系统工作，被写入日志的操作必须是*幂等的*，它意味着只要有必要，它们就可以重复执行很多次，并不会带来破坏。像操作“更新位表并标记i节点*k*或者块*n*是空闲的”可以重复任意次。同样地，查找一个目录并且删除所有叫foobar的项也是幂等的。在另一方面，把从i节点*k*新释放的块加入空闲表的末端不是幂等的，因为它们可能已经被释放并存放在那里了。更复杂的操作如“查找空闲块列表并且如果块*n*不在列表就将块*n*加入”是幂等的。日志文件系统必须安排它们的数据结构和可写入日志的操作以使它们都是幂等的。在这些条件下，崩溃恢复可以被快速安全地实施。

为了增加可信性，一个文件系统可以引入数据库中*原子事务*（atomic transaction）的概念。使用这个概念，一组动作可以被界定在开始事务和结束事务操作之间。这样，文件系统就会知道它必须完成所有被界定的操作，或者什么也不做，但是没有其他的选择。

NTFS有一个扩展的日志文件系统，并且它的结构几乎不会因系统崩溃而受到破坏。自1993年NTFS第一次随Windows NT一起发行以来就在不断地发展。Linux上有日志功能的第一个文件系统是ReiserFS，但是因为它和后来标准化的ext2文件系统不相匹配，它的推广受到阻碍。相比之下，ext3——一个不像ReiserFS那么有野心的工程，也具有日志文件功能并且和之前的ext2系统可以共存。

4.3.7 虚拟文件系统

即使在同一台计算机上同一个操作系统下，也会使用很多不同的文件系统。一个Windows可能有一个主要的NTFS文件系统，但是也有继承的FAT-32或者FAT-16驱动，或包含旧的但仍被使用的数据的分区，并且不时地也可能需要一个CD-ROM或者DVD（每一个包含它们特有的文件系统）。Windows通过指定不同的盘符来处理这些不同的文件系统，比如“C:”、“D:”等。当一个进程打开一个文件，盘符是显式或者隐式存在的，所以Windows知道向哪个文件系统传递请求，不需要尝试将不同类型文件系统整合为统一模式。

相比之下，所有现代的UNIX系统做了一个很认真的尝试，即将多种文件系统整合到一个统一的结构中。一个Linux系统可以用ext2作为根文件系统，ext3分区装载在/home下，另一块采用ReiserFS文件系统的硬盘装载在/home下，以及一个ISO 9660的CD-ROM临时装载在/mnt下。从用户的观点来看，那只有一个文件系统层级。它们事实上是多种（不相容的）文件系统，对于用户和进程是不可见的。

但是，多种文件系统的存在，在实际应用中是明确可见的，而且因为先前Sun公司（Kleiman, 1986）所做的工作，绝大多数UNIX操作系统都使用*虚拟文件系统*（Virtual File System，VFS）概念尝试将多种文件系统统一成一个有序的框架。关键的思想就是抽象出所有文件系统都共有的部分，并且将这部分代码放在单独的一层，该层调用底层的实际文件系统来具体管理数据。大体上的结构在图4-18中有阐述。以下的介绍不是单独针对Linux和FreeBSD或者其他版本的UNIX，而是给出了一种普遍的关于UNIX下文件系统的描述。

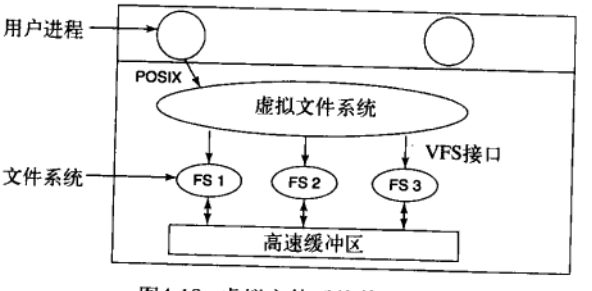


图4-18 虚拟文件系统的位置

所有和文件相关的系统调用在最初的处理上都指向虚拟文件系统。这些来自用户进程的调用，都是标准的POSIX系统调用，比如open、read write和lseek等。因此，虚拟文件系统对用户进程有一个“更高层“接口，它就是著名的POSIX接口。

VFS也有一个对于实际文件系统的“更低层”接口，就是在图4-18中被标记为VFS接口的部分。这个接口包含许多功能调用，这样VFS可以使每一个文件系统完成任务。因此，当创造一个新的文件系统和VFS一起工作时，新文件系统的设计者就必须确定它提供VFS所需要的功能调用。关于这个功能的一个明显的例子就是从磁盘中读某个特定的块，把它放在文件系统的高速缓冲中，并且返回指向它的指针。因此，VFS有两个不同的接口：上层给用户进程的接口和下层给实际文件系统的接口。

尽管VFS下大多数的文件系统体现了本地磁盘的划分，但并不总是这样。事实上，Sun建立虚拟文件系统最原始的动机是支持使用NFS（Network File System，网络文件系统）协议的远程文件系统。VFS设计是只要实际的文件系统提供VFS需要的功能，VFS就不需知道或者关心数据具体存储在什么地方或者底层的文件系统是什么样的。

大多数VFS应用本质上都是面向对象的，即便它们用C语言而不是C++编写。有几种通常支持的主要的对象类型，包括超块（描述文件系统）、v节点（描述文件）和目录（描述文件系统目录）。这些中的每一个都有实际文件系统必须支持的相关操作。另外，VFS有一些供它自己使用的内部数据结构，包括用于跟踪用户进程中所有打开文件的装载表和文件描述符的数组。

为了理解VFS是如何工作的，让我们按时间的先后举一个例子。当系统启动时，根文件系统在VFS中注册。另外，当装载其他文件系统时，不管在启动时还是在操作过程中，它们也必须在VFS中注册。当一个文件系统注册时，它做的最基本的工作就是提供一个包含VFS所需要的函数地址的列表，可以是一个长的调用矢量（表），或者是许多这样的矢量（如果VFS需要），每个VFS对象一个。因此，只要一个文件系统在VFS注册，VFS就知道如何从它那里读一个块——它从文件系统提供的矢量中直接调用第4个（或者任何一个）功能。同样地，VFS也知道如何执行实际文件系统提供的每一个其他的功能：它只需调用某个功能，该功能所在的地址在文件系统注册时就提供了。

装载文件系统后就可以使用它了。比如，如果一个文件系统装载在/usr并且一个进程调用它：

```
open("/usr/include/unistd.h", O_RDONLY)
```

当解析路径时，VFS看到新的文件系统被装载在/usr，并且通过搜索已经装载文件的超块表来确定它的超块。做完这些，它可以找到它所装载的文件的根目录，在那里查找路径include/unistd.h。然后VFS创建一个v节点并调用实际文件系统，以返回所有的在文件i节点中的信息。这个信息被和其他信息一起复制到v节点中（在RAM中），而这些信息中最重要的是指向包含调用v节点操作的功能表的指针，比如read、write和close等。

当v节点被创建以后，VFS在文件描述符表中为调用进程创建一个入口，并且将它指向一个新的v节点（为了简单，文件描述符实际上指向另一个包含当前文件位置和指向v节点的指针的数据结构，但是这个细节对于我们这里的陈述并不重要）。最后，VFS向调用者返回文件描述符，所以调用者可以用它去读、写或者关闭文件。

随后，当进程用文件描述符进行一个读操作，VFS通过进程表和文件描述符表确定v节点的位置，并跟随指针指向功能表（所有这些都是被请求文件所在的实际文件系统中的地址）。这样就调用了处理read的功能，在实际文件系统中的代码运行并得到所请求的块。VFS并不知道数据是来源于本地硬盘，还是来源于网络中的远程文件系统、CD-ROM、USB存储棒或者其他介质。所有有关的数据结构在图4-19中展示。从调用者进程号和文件描述符开始，进而是v节点，读功能指针，然后是对实际文件系统的入口函数定位。

通过这种方法，加入新的文件系统变得相当直接。为了加入一个文件系统，设计者首先获得一个VFS期待的功能调用的列表，然后编写文件系统实现这些功能。或者，如果文件系统已经存在，它们必须提供VFS需要的包装功能，通常通过建造一个或者多个内在的指向实际文件系统的调用来实现。

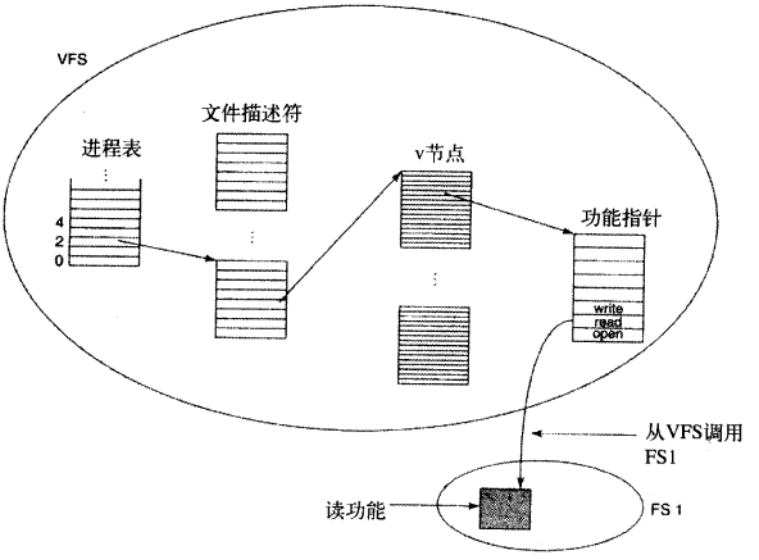


图4-19 VFS和实际文件系统进行读操作所使用的数据结构和代码的简化视图

4.4 文件系统管理和优化

要使文件系统工作是一件事，使真实世界中的文件系统有效、鲁棒地工作是另一回事。本节中，我们将考察有关管理磁盘的一些问题。

4.4.1 磁盘空间管理

文件通常存放在磁盘上，所以对磁盘空间的管理是系统设计者要考虑的一个主要问题。存储n个字节的文件可以有两种策略：分配n个字节的连续磁盘空间，或者把文件分成很多个连续（或并不一定连续）的块。在存储管理系统中，分段处理和分页处理之间也要进行同样的权衡。

正如我们已经见到的，按连续字节序列存储文件有一个明显问题，当文件扩大时，有可能需要在磁盘上移动文件。内存中分段也有同样的问题。不同的是，相对于把文件从磁盘的一个位置移动到另一个位置，内存中段的移动操作要快得多。因此，几乎所有的文件系统都把文件分割成固定大小的块来存储，各块之间不一定相邻。

1. 块大小

一旦决定把文件按固定大小的块来存储，就会出现一个问题：块的大小应该是多少？按照磁盘组织方式，扇区、磁道和柱面显然都可以作为分配单位（虽然它们都与设备相关，这是一种负面因素）。在分页系统中，页面大小也是主要讨论的问题之一。

拥有大的块尺寸意味着每个文件，甚至一个1字节的文件，都要占用一整个柱面，也就是说小的文件浪费了大量的磁盘空间。另一方面，小的块尺寸意味着大多数文件会跨越多个块，因此需要多次寻道与旋转延迟才能读出它们，从而降低了性能。因此，如果分配的单元太大，则浪费了空间；如果太小，则浪费时间。

做出一个好的决策需要知道有关文件大小分配的信息。Tanenbaum 等人（2006）给出了1984年及2005年在一所大型研究型大学（VU）的计算机系以及一个政治网站（www.electoral-vote.com）的商业网络服务器上研究的文件大小分配数据。结果显示在图4-20，其中，对于每个2的幂文件大小，在3个数据集里每一数据集中的所有小于等于这个值的文件所占的百分比被列了出来。例如，在2005年，59.13%的VU的文件是4KB或更小，且90.84%的文件是64KB或更小，其文件大小的中间值是2475字节。一些人可能会因为这么小的尺寸而感到吃惊。

文件大小	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

文件大小	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

图4-20 小于某个给定值（字节）的文件的百分比

我们能从这些数据中得出什么结论呢？如果块大小是1KB，则只有30%~50%的文件能够放在一个块内，但如果块大小是4KB，这一比例将上升到60%~70%。那篇论文中的其他数据显示，如果块大小是4KB，则93%的磁盘块会被10%最大的文件使用。这意味着在每个小文件末尾浪费一些空间几乎不会有任何关系，因为磁盘被少量的大文件（视频）给占用了，并且小文件所占空间的总量根本就无关紧要，甚至将那90%最小的文件所占的空间翻一倍也不会引人注目。

另一方面，分配单位很小意味着每个文件由很多块组成，每读一块都有寻道和旋转延迟时间，所以，读取由很多小块组成的文件会非常慢。

举例说明，假设磁盘每道有1MB，其旋转时间为8.33ms，平均寻道时间为5ms。以毫秒（ms）为单位，读取一个k个字节的块所需要的时间是寻道时间、旋转延迟和传送时间之和：

$$5+ 4.165 + (k/ 1\ 000\ 000) \times 8.33$$

图4-21的虚线表示一个磁盘的数据率与块大小之间的函数关系。要计算空间利用率，则要对文件的平均大小做出假设。为简单起见，假设所有文件都是4KB。尽管这个数据稍微大于在VU测量得到的数据，但是学生们大概应该有比公司数据中心更小的文件，所以这样整体上也许更好些。图4-21中的实线表示作为盘块大小函数的空间利用率。

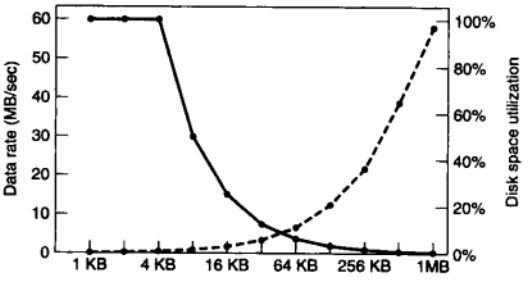


图4-21 虚线（左边标度）给出磁盘数据率，实线（右边标度）给出磁盘空间利用率（所有文件大小均为4KB）

可以按下面的方式理解这两条曲线。对一个块的访问时间完全由寻道时间和旋转延迟所决定，所以若要花费9ms的代价访问一个盘块，那么取的数据越多越好。因此，数据率随着磁盘块的增大而增大（直到传输花费很长的时间以至于传输时间成为主导因素）。

现在考虑空间利用率。对于4KB文件和1KB、2KB或4KB的磁盘块，分别使用4、2、1块的文件，没有浪费。对于8KB块以及4KB文件，空间利用率降至50%，而16KB块则降至25%。实际上，很少有文件的大小是磁盘块整数倍的，所以一个文件的最后一个磁盘块中总是有一些空间浪费。

然而，这些曲线显示出性能与空间利用率天生就是矛盾的。小的块会导致低的性能但是高的空间利用率。对于这些数据，不存在合理的折中方案。在两条曲线的相交处的大小大约是64KB，但是数据（传输）速率只有6.6MB/s并且空间利用率只有大约7%，两者都不是很好。从历史观点上来说，文件系统将大小设在1~4KB之间，但现在随着磁盘超过了1TB，还是将块的大小提升到64KB并且接受浪费的磁

盘空间，这样也许更好。磁盘空间几乎不再会短缺了。

在考察Windows NT的文件使用情况是否与UNIX的文件使用情况存在微小差别的实验中，Vogels在康奈尔大学对文件进行了测量（Vogels，1999）。他观察到NT的文件使用情况比UNIX的文件使用情况复杂得多。他写道：

当我们在notepad文本编辑器中输入一些字符后，将内容保存到一个文件中将触发26个系统调用，包括3个失败的open企图、1个文件重写和4个打开和关闭序列。

尽管如此，他观察到了文件大小的中间值（以使用情况作为权重）：只读的为1KB，只写的为2.3KB，读写的文件为4.2KB。考虑到数据集测量技术以及年份上的差异，这些结果与VU的结果是相当吻合的。

2. 记录空闲块

一旦选定了块大小，下一个问题就是怎样跟踪空闲块。有两种方法被广泛采用，如图4-22所示。第一种方法是采用磁盘块链表，每个块中包含尽可能多的空闲磁盘块号。对于1KB大小的块和32位的磁盘块号，空闲表中每个块包含有255个空闲块的块号（需要有一个位置存放指向下一个块的指针）。考虑500GB的磁盘，拥有488×10^6个块。为了在255块中存放全部这些地址，需要190万个块。通常情况下，采用空闲块存放空闲表，这样存储器基本上是空的。

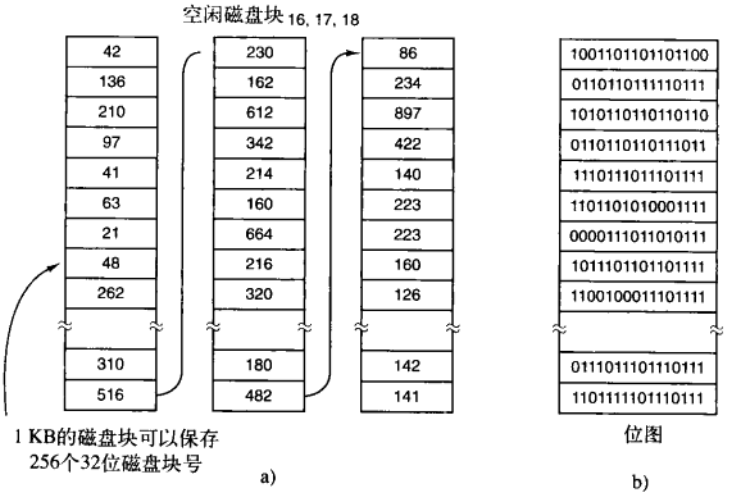


图4-22 a) 把空闲表存放在链表中；b) 位图

另一种空闲磁盘空间管理的方法是采用位图。n个块的磁盘需要n位位图。在位图中，空闲块用1表示，已分配块用0表示（或者反之）。对于500GB磁盘的例子，需要488×10^6位表示，即需要60 000个1KB块存储。很明显，位图方法所需空间较少，因为每块只用一个二进制位标识，相反在链表方法中，每一块要用到32位。只有在磁盘快满时（即几乎没有空闲块时）链表方案需要的块才比位图少。

如果空闲块倾向于成为一个长的连续分块的话，则空闲列表系统可以改成记录分块而不是单个的块。一个8、16、32位的计数可以与每一个块相关联，来记录连续空闲块的数目。在最好的情况下，一个基本上空的磁盘可以用两个数表达：第一个空闲块的地址，以及空闲块的计数。另一方面，如果磁盘产生了很严重的碎片，记录分块会比记录单独的块效率要低，因为不仅要存储地址，而且还要存储计数。

这个情形说明了操作系统设计者经常遇到的一个问题。有许多数据结构与算法可以用来解决一个问题，但选择其中最好的则需要数据，而这些数据是设计者无法预先拥有的，只有在系统被部署完毕并被大量使用后才会获得。更有甚者，有些数据可能就是无法获取。例如，1984年与1995年我们在VU测量的文件大小、网站的数据以及在康奈尔大学的数据，是仅有的4个数据样本。尽管有总比什么都没有好，我们仍旧不清楚是否这些数据也可以代表家用计算机、公司计算机、政府计算机及其他。经过一些努力我们也许可以获取一些其他种类计算机的样本，但即使那样，（就凭这些数据来）推断那种测量适用于

所有计算机也是愚蠢的。

现在回到空闲表方法，只需要在内存中保存一个指针块。当文件创建时，所需要的块从指针块中取出。现有的指针块用完时，从磁盘中读入一个新的指针块。类似地，当删除文件时，其磁盘块被释放，并添加到内存的指针块中。当这个块填满时，就把它写入磁盘。

在某些特定情形下，这个方法产生了不必要的磁盘I/O。考虑图4-23a中的情形，内存中的指针块只有两个表项了。如果释放了一个有三个磁盘块的文件，该指针块就溢出了，必须将其写入磁盘，这就产生了图4-23b的情形。如果现在写入含有三个块的文件，满的指针块不得不再次读入，这将回到图4-23a的情形。如果有三个块的文件只是作为临时文件被写入，当它被释放时，就需要另一个磁盘写操作，以便把满的指针块写回磁盘。总之，当指针块几乎为空时，一系列短期的临时文件就会引起大量的磁盘I/O。

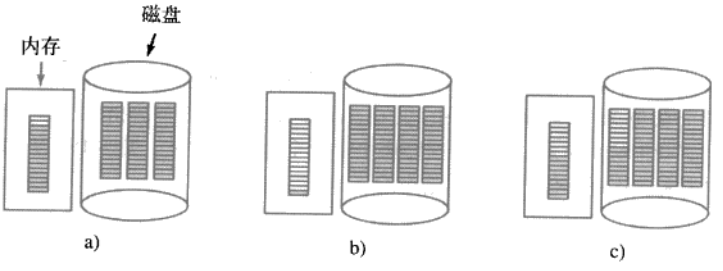


图4-23 a) 在内存中一个被指向空闲磁盘块的指针几乎充满的块，以及磁盘上三个指针块；b) 释放一个有三个块的文件的结果；c) 处理该三个块的文件的替代策略
（带阴影的表项代表指向空闲磁盘块的指针）

一个可以避免过多磁盘I/O的替代策略是，拆分满了的指针块。这样，当释放三个块时，不再是从图4-23a变化到图4-23b，而是从图4-23a变化到图4-23c。现在，系统可以处理一系列临时文件，而不需进行任何磁盘I/O。如果内存中指针块满了，就写入磁盘，半满的指针块从磁盘中读入。这里的思想是：保持磁盘上的大多数指针块为满的状态（减少磁盘的使用），但是在内存中保留一个半满的指针块。这样，它可以既处理文件的创建又同时处理文件的删除操作，而不会为空闲表进行磁盘I/O。

对于位图，在内存中只保留一个块是有可能的，只有在该块满了或空了的情形下，才到磁盘上取另一块。这样处理的附加好处是，通过在位图的单一块上进行所有的分配操作，磁盘块会较为紧密地聚集在一起，从而减少了磁盘臂的移动。由于位图是一种固定大小的数据结构，所以如果内核是（部分）分页的，就可以把位图放在虚拟内存内，在需要时将位图的页面调入。

3. 磁盘配额

为了防止人们贪心而占有太多的磁盘空间，多用户操作系统常常提供一种强制性磁盘配额机制。其思想是系统管理员分给每个用户拥有文件和块的最大数量，操作系统确保每个用户不超过分给他们的配额。下面将介绍一种典型的机制。

当用户打开一个文件时，系统找到文件属性和磁盘地址，并把它们送入内存中的打开文件表。其中一个属性告诉文件所有者是谁。任何有关该文件大小的增长都记到所有者的配额上。

第二张表包含了每个用户当前打开文件的配额记录，即使是其他人打开该文件也一样。这张表如图4-24所示，该表的内容是从被打开文件的所有者的磁

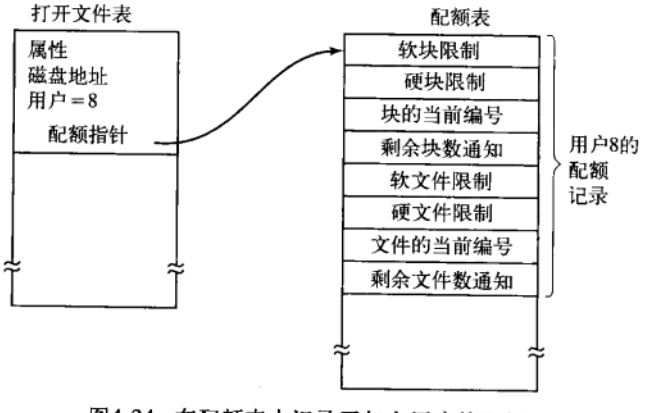


图4-24 在配额表中记录了每个用户的配额

盘配额文件中提取出来的。当所有文件关闭时，该记录被写回配额文件。

当在打开文件表中建立一新表项时，会产生一个指向所有者配额记录的指针，以便很容易找到不同的限制。每一次往文件中添加一块时，文件所有者所用数据块的总数也增加，引发对配额硬限制和软限制检查。可以超出软限制，但硬限制不可以超出。当已达到硬限制时，再往文件中添加内容将引发错误。同时，对文件数目也存在着类似的检查。

当用户试图登录时，系统核查配额文件，查看该用户文件数目或磁盘块数目是否超过软限制。如果超过了任一限制，则显示一个警告，保存的警告计数减1。如果该计数已为0，表示用户多次忽略该警告，因而将不允许该用户登录。要想再得到登录的许可，就必须与系统管理员协商。

这一方法具有一种性质，即只要用户在退出系统前消除所超过的部分，他们就可以在一次终端会话期间超过其软限制；但无论什么情况下都不能超过硬限制。

4.4.2 文件系统备份

比起计算机的损坏，文件系统的破坏往往要糟糕得多。如果由于火灾、闪电电流或者一杯咖啡泼在键盘上而弄坏了计算机，确实让人伤透脑筋，而且又要花上一笔钱，但一般而言，更换非常方便。只要去计算机商店，便宜的个人计算机在短短一个小时之内就可以更换（当然，如果这发生在大学里面，则发出订单需3个委员会的同意，5个签字要花90天的时间）。

不管是硬件或软件的故障，如果计算机的文件系统被破坏了，恢复全部信息会是一件困难而又费时的工作，在很多情况下，是不可能的。对于那些丢失了程序、文档、客户文件、税收记录、数据库、市场计划或者其他数据的用户来说，这不啻为一次大的灾难。尽管文件系统无法防止设备和介质的物理损坏，但它至少应能保护信息。直接的办法是制作备份。但是备份并不如想象得那么简单。让我们开始考察。

许多人都认为不值得把时间和精力花在备份文件这件事上，直到某一天磁盘突然崩溃，他们才意识到事态的严重性。不过现在很多公司都意识到了数据的价值，常常把数据转到磁带上存储，并且每天至少做一次备份。现在磁带的容量大至几十甚至几百GB，而每个GB仅仅需要几美分。其实，做备份并不像人们说得那么烦琐，现在就让我们来看一下相关的要点。

做磁带备份主要是要处理好两个潜在问题中的一个：

1) 从意外的灾难中恢复。

2) 从错误的操作中恢复。

第一个问题主要是由磁盘破裂、火灾、洪水等自然灾害引起的。事实上这些情形并不多见，所以许多人也就不以为然。这些人往往也是以同样的原因忽略了自家的火灾保险。

第二个原因主要是用户意外地删除了原本还需要的文件。这种情况发生得很频繁，使得Windows的设计者们针对“删除”命令专门设计了特殊目录——“回收站”，也就是说，在人们删除文件的时候，文件本身并不真正从磁盘上消失，而是被放置到这个特殊目录下，待以后需要的时候可以还原回去。文件备份更主要是指这种情况，这就允许几天之前，甚至几个星期之前的文件都能从原来备份的磁带上还原。

为文件做备份既耗时间又费空间，所以需要做得又快又好，这一点很重要。基于上述考虑我们来看看下面的问题。首先，是要备份整个文件系统还是仅备份一部分呢？在许多安装配置中，可执行程序（二进制代码）放置在文件系统树的受限制部分，所以如果这些文件能直接从厂商提供的CD-ROM盘上重新安装的话，也就没有必要为它们做备份。此外，多数系统都有专门的临时文件目录，这个目录也不需要备份。在UNIX系统中，所有的特殊文件（也就是I/O设备）都放置在/dev目录下，对这个目录做备份不仅没有必要而且还十分危险——因为一旦进行备份的程序试图读取其中的文件，备份程序就会永久挂起。简而言之，合理的做法是只备份特定目录及其下的全部文件，而不是备份整个文件系统。

其次，对前一次备份以来没有更改过的文件再做备份是一种浪费，因而产生了增量转储的思想。最简单的增量转储形式就是周期性地（每周一次或每月一次）做全面的转储（备份），而每天只对当天更改的数据做备份。稍微好一点的做法只备份自最近一次转储以来更改过的文件。当然了，这种做法极大地缩减了转储时间，但操作起来却更复杂，因为最近的全面转储先要全部恢复，随后按逆序进行增量转储。为了方便，人们往往使用更复杂的增量转储模式。

第三，既然待转储的往往是海量数据，那么在将其写入磁带之前对文件进行压缩就很有必要。可是

对许多压缩算法而言，备份磁带上的单个坏点就能破坏解压缩算法，并导致整个文件甚至整个磁带无法阅读。所以是否要对备份文件流进行压缩必须慎重考虑。

第四，对活动文件系统做备份是很难的。因为在转储过程中添加、删除或修改文件和目录可能会导致文件系统的不一致性。不过，既然转储一次需要几个小时，那么在晚上大部分时间让文件系统脱机是很有必要的，虽然这种做法有时会令人难以接受。正因如此，人们修改了转储算法，记下文件系统的瞬时状态，即复制关键的数据结构，然后需要把将来对文件和目录所做的修改复制到块中，而不是处处更新它们（Hutchinson等人，1999）。这样，文件系统在抓取快照的时候就被有效地冻结了，留待以后空闲时再备份。

第五，即最后一个问题，做备份会给一个单位引入许多非技术性问题。如果当系统管理员下楼去取打印文件，而毫无防备地把备份磁带搁置在办公室里的时候，就是世界上最棒的在线保安系统也会失去作用。这时，一个间谍所要做的只是潜入办公室、将一个小磁带放入口袋，然后绅士般地离开。再见吧保安系统。即使每天都做备份，如果碰上一场大火烧光了计算机和所有的备份磁带，那做备份又有什么意义呢？由于这个原因，所以备份磁带应该远离现场存放，不过这又带来了更多的安全风险（因为，现在必须保护两个地点了）。关于此问题和管理中的其他实际问题，请参考（Nemeth等人，2000）。接下来我们只讨论文件系统备份所涉及的技术问题。

转储磁盘到磁带上有两种方案：物理转储和逻辑转储。*物理转储*是从磁盘的第0块开始，将全部的磁盘块按序输出到磁带上，直到最后一块复制完毕。此程序很简单，可以确保万无一失，这是其他任何实用程序所不能比的。

不过有几点关于物理转储的评价还是值得一提的。首先，未使用的磁盘块无须备份。如果转储程序能够得到访问空闲块的数据结构，就可以避免该程序备份未使用的磁盘块。但是，既然磁带上的第k块并不代表磁盘上的第k块，那么要想略过未使用的磁盘块就需要在每个磁盘块前边写下该磁盘块的号码（或其他等效数据）。

第二个需要关注的是坏块的转储。制造大型磁盘而没有任何瑕疵几乎是不可能的，总是有一些坏块存在。有时进行低级格式化后，坏块会被检测出来，标记为坏的，并被应对这种紧急状况的在每个轨道末端的一些空闲块所替换。在很多情况下，磁盘控制器处理坏块的替换过程是透明的，甚至操作系统也不知道。

然而，有时格式化后块也会变坏，在这种情况下操作系统可以检测到它们。通常，可以通过建立一个包含所有坏块的“文件”来解决这个问题——只要确保它们不会出现在空闲块池中并且决不会被分配。不用说，这个文件是完全不能够读取的。

如果磁盘控制器将所有坏块重新映射，并对操作系统隐藏的话，物理转储工作还是能够顺利进行的。另一方面，如果这些坏块对操作系统可见并映射到在一个或几个坏块文件或者位图中，那么在转储过程中，物理转储程序绝对有必要能访问这些信息，并避免转储之，从而防止在对坏块文件备份时的无止境磁盘读错误发生。

物理转储的主要优点是简单、极为快速（基本上是以磁盘的速度运行）。主要缺点是，既不能跳过选定的目录，也无法增量转储，还不能满足恢复个人文件的请求。正因如此，绝大多数配置都使用逻辑转储。

*逻辑转储*从一个或几个指定的目录开始，递归地转储其自给定基准日期（例如，最近一次增量转储或全面系统转储的日期）后有所更改的全部文件和目录。所以，在逻辑转储中，转储磁带上会有一连串精心标识的目录和文件，这样就很容易满足恢复特定文件或目录的请求。

既然逻辑转储是最为普遍的形式，就让我们以图4-25为例来仔细研究一个通用算法。该算法在UNIX系统上广为使用。在图中可以看到一棵由目录（方框）和文件（圆圈）组成的文件树。被阴影覆盖的项目代表自基准日期以来修改过，因此需要转储，无阴影的则不需要转储。

该算法还转储通向修改过的文件或目录的路径上的所有目录（甚至包括未修改的目录），原因有二。其一是为了将这些转储的文件和目录恢复到另一台计算机的新文件系统中。这样，转储程序和恢复程序就可以在计算机之间进行文件系统的整体转移。

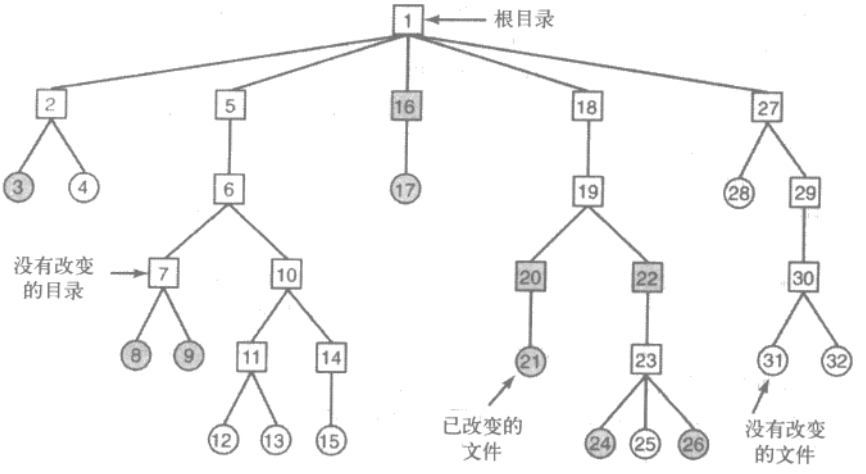


图4-25 待转储的文件系统，其中方框代表目录，圆圈代表文件。被阴影覆盖的项目表示自上次转储以来修改过。每个目录和文件都被标上其i节点号

转储被修改文件之上的未修改目录的第二个原因是为了可以对单个文件进行增量恢复（很可能是对愚蠢操作所损坏文件的恢复）。设想如果星期天晚上转储了整个文件系统，星期一晚上又做了一次增量转储。在星期二，/usr/jhs/proj/nr3目录及其下的全部目录和文件被删除了。星期三一大早用户又想恢复/usr/jhs/proj/nr3/plans/summary文件。但因为没有设置，所以不可能单独恢复summary文件。必须首先恢复nr3和plans这两个目录。为了正确获取文件的所有者、模式、时间等各种信息，这些目录当然必须再次备份到转储磁带上，尽管自上次完整转储以来它们并没有修改过。

逻辑转储算法要维持一个以i节点号为索引的位图，每个i节点包含了几位。随着算法的执行，位图中的这些位会被设置或清除。算法的执行分为四个阶段。第一阶段从起始目录（本例中为根目录）开始检查其中的所有目录项。对每一个修改过的文件，该算法将在位图中标记其i节点。算法还标记并递归检查每一个目录（不管是否修改过）。

第一阶段结束时，所有修改过的文件和全部目录都在位图中标记了，如图4-26a所示（以阴影标记）。理论上说来，第二阶段再次递归地遍历目录树，并去掉目录树中任何不包含被修改过的文件或目录的目录上的标记。本阶段的执行结果如图4-26b所示。注意，i节点号为10、11、14、27、29和30的目录此时已经被去掉标记，因为它们所包含的内容没有做任何修改。它们因而也不会被转储。相反，i节点号为5和6的目录尽管没有被修改过也要被转储，因为到新的机器上恢复当日的修改时需要这些信息。为了提高算法效率，可以将这两阶段的目录树遍历合二为一。

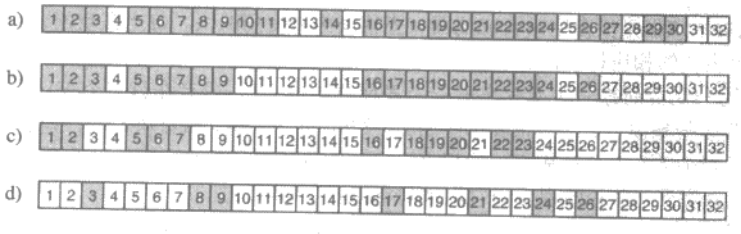


图4-26 逻辑转储算法所使用的位图

现在哪些目录和文件必须被转储已经很明确了，就是图4-26b中所标记的部分。第三阶段算法将以节点号为序，扫描这些i节点并转储所有标记的目录，如图4-26c所示。为了进行恢复，每个被转储的目录都用目录的属性（所有者、时间等）作为前缀。最后，在第四阶段，在图4-26d中被标记的文件也被转储，同样，由其文件属性作为前缀。至此，转储结束。

从转储磁带上恢复文件系统很容易办到。首先要在磁盘上创建一个空的文件系统，然后恢复最近一次的完整转储。由于磁带上最先出现目录，所以首先恢复目录，给出文件系统的框架；然后恢复文件本身。在完整转储之后的是增量转储，重复这一过程，以此类推。

尽管逻辑转储十分简单，还是有几点棘手之处。首先，既然空闲块列表并不是一个文件，那么在所有被转储的文件恢复完毕之后，就需要从零开始重新构造。这一点可以办到，因为全部空闲块的集合恰好是包含在全部文件中的块集合的补集。

另一个问题是关于连接。如果一个文件被连接到两个或多个目录中，要注意在恢复时只对该文件恢复一次，然后要恢复所有指向该文件的目录。

还有一个问题就是：UNIX文件实际上包含了许多“空洞”。打开文件，写几个字节，然后找到文件中一个偏移了一定距离的地址，又写入更多的字节，这么做是合法的。但两者之间的这些块并不属于文件本身，从而也不应该在其上实施转储和恢复操作。核心文件通常在数据段和堆栈段之间有一个数百兆字节的空洞。如果处理不得当，每个被恢复的核心文件会以“0”填充这些区域，这可能导致该文件与虚拟地址空间一样大（例如，2^{32}字节，更糟糕可能会达到2^{64}字节）。

最后，无论属于哪一个目录（它们并不一定局限于/dev目录下），特殊文件、命名管道以及类似的文件都不应该转储。关于文件系统备份的更多信息，请参考（Chervenak 等人，1998; Zwicky，1991）。

磁带密度不会像磁盘密度那样改进得那么快。这会逐渐导致备份一个很大的磁盘需要多个磁带的状况。当磁带机器人可以自动换磁带时，如果这种趋势继续下去，作为一种备份介质，磁带会最终变得太小。在那种情况下，备份一个磁盘的惟一的方式是在另一个磁盘上。对每一个磁盘直接做镜像是一种方式。一个更加复杂的方案，称为RAID，将会在第5章讨论。

4.4.3 文件系统的一致性

影响文件系统可靠性的另一个问题是文件系统的一致性。很多文件系统读取磁盘块，进行修改后，再写回磁盘。如果在修改过的磁盘块全部写回之前系统崩溃，则文件系统有可能处于不一致状态。如果一些未被写回的块是i节点块、目录块或者是包含有空闲表的块时，这个问题尤为严重。

为了解决文件系统的不一致问题，很多计算机都带有一个实用程序以检验文件系统的一致性。例如，UNIX有fsck，而Windows用scandisk。系统启动时，特别是崩溃之后的重新启动，可以运行该实用程序。下面我们介绍在UNIX中这个fsck实用程序是怎样工作的。scandisk有所不同，因为它工作在另一种文件系统上，不过运用文件系统的内在冗余进行修复的一般原理仍然有效。所有文件系统检验程序可以独立地检验每个文件系统（磁盘分区）的一致性。

一致性检查分为两种：块的一致性检查和文件的一致性检查。在检查块的一致性时，程序构造两张表，每张表中为每个块设立一个计数器，都初始化为0。第一个表中的计数器跟踪该块在文件中的出现次数，第二个表中的计数器跟踪该块在空闲表中的出现次数。

接着检验程序使用原始设备读取全部的i节点，忽略文件的结构，只返回所有的磁盘块，从0开始。由i节点开始，可以建立相应文件中采用的全部块的块号表。每当读到一个块号时，该块在第一个表中的计数器加1。然后，该程序检查空闲表或位图，查找全部未使用的块。每当在空闲表中找到一个块时，就会使它在第二个表中的相应计数器加1。

如果文件系统一致，则每一块或者在第一个表计数器中为1，或者在第二个表计数器中为1，如图4-27a所示。但是当系统崩溃后，这两张表可能如图4-27b所示，其中，磁盘块2没有出现在任何一张表中，这称为块丢失。尽管块丢失不会造成实际的损害，但它的确浪费了磁盘空间，减少了磁盘容量。块丢失问题的解决很容易：文件系统检验程序把它们加到空闲表中即可。

有可能出现的另一种情况如图4-27c所示。其中，块4在空闲表中出现了2次（只在空闲表是真正意义上的一张表时，才会出现重复，在位图中，不会发生这类情况）。解决方法也很简单：只要重新建立空闲表即可。

最糟的情况是，在两个或多个文件中出现同一个数据块，如图4-27d中的块5。如果其中一个文件被删除，块5会添加到空闲表中，导致一个块同时处于使用和空闲两种状态。若删除这两个文件，那么在空闲表中这个磁盘块会出现两次。

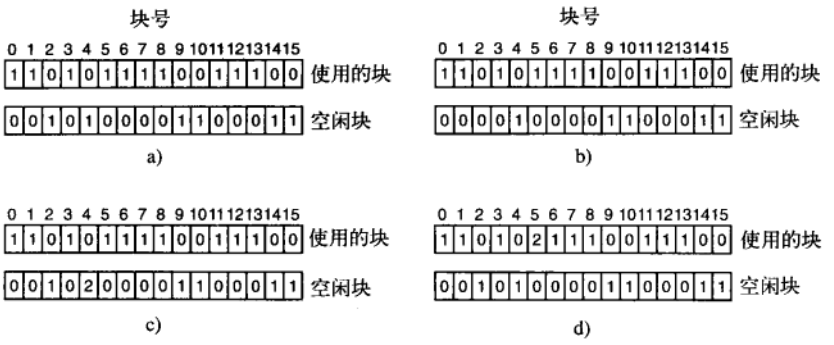


图4-27 文件系统状态：a) 一致；b) 块丢失；c) 空闲表中有重复块；d) 重复数据块

文件系统检验程序可以采取相应的处理方法是，先分配一空闲块，把块5中的内容复制到空闲块中，然后把它插到其中一个文件之中。这样文件的内容未改变（虽然这些内容几乎可以肯定是不对的），但至少保持了文件系统的一致性。这一错误应该报告，由用户检查文件受损情况。

除检查每个磁盘块计数的正确性之外，文件系统检验程序还检查目录系统。此时也要用到一张计数器表，但这时是一个文件（而不是一个块）对应于一个计数器。程序从根目录开始检验，沿着目录树递归下降，检查文件系统中的每个目录。对每个目录中的每个文件，将文件使用计数器加1。要注意，由于存在硬连接，一个文件可能出现在两个或多个目录中。而遇到符号连接是不计数的，不会对目标文件的计数器加1。

在检验程序全部完成后，得到一张由i节点号索引的表，说明每个文件被多少个目录包含。然后，检验程序将这些数字与存储在文件i节点中的连接数目相比较。当文件创建时，这些计数器从1开始，随着每次对文件的一个（硬）连接的产生，对应计数器加1。如果文件系统一致，这两个计数应相等。但是，有可能出现两种错误，即i节点中的连接计数太大或者太小。

如果i节点的连接计数大于目录项个数，这时即使所有的文件都从目录中删除，这个计数仍是非0，i节点不会被删除。该错误并不严重，却因为存在不属于任何目录的文件而浪费了磁盘空间。为改正这一错误，可以把i节点中的连接计数设成正确值。

另一种错误则是潜在的灾难。如果同一个文件连接两个目录项，但其i节点连接计数只为1，如果删除了任何一个目录项，对应i节点连接计数变为0。当i节点计数为0时，文件系统标志该i节点为“未使用”，并释放其全部块。这会导致其中一个目录指向一未使用的i节点，而很有可能其块马上就被分配给其他文件。解决方法同样是把i节点中连接计数设为目录项的实际个数值。

由于效率上的考虑，以上的块检查和目录检查经常被集成到一起（即仅对i节点扫描一遍）。当然也有一些其他检查方法。例如，目录是有明确格式的，包含有i节点数目和ASCII文件名，如果某个目录的i节点编号大于磁盘中i节点的实际数目，说明这个目录被破坏了。

再有，每个i节点都有一个访问权限项。一些访问权限是合法的，但是很怪异，比如0007，它不允许文件所有者及所在用户组的成员进行访问，而其他的用户却可以读、写、执行此文件。在这类情况下，有必要报告系统已经设置了其他用户权限高于文件所有者权限这一情况。拥有1000多个目录项的目录也很可疑。为超级用户所拥有，但放在用户目录下，且设置了SETUID位的文件，可能也有安全问题，因为任何用户执行这类文件都需要超级用户的权限。可以列出一长串特殊的情况，尽管这些情况合法，但报告给用户却是有必要的。

以上讨论了防止因系统崩溃而破坏用户文件的问题，某一些文件系统也防止用户自身的误操作。如果用户想输入

```
rm *.o
```

删除全部以.o结尾的文件（编译器生成的目标文件），但不幸键入了

```
rm * .o
```

（注意，星号后面有一空格），则rm命令会删除全部当前目录中的文件，然后报告说找不到文件.o。在MS-DOS和一些其他系统中，文件的删除仅仅是在对应目录或i节点上设置某一位，表示文件被删除，并没有把磁盘块返回到空闲表中，直到确实需要时才这样做。所以，如果用户立即发现了操作错误，可以运行特定的一个“撤销删除”（即恢复）实用程序恢复被删除的文件。在Windows中，删除的文件被转移到回收站目录中（一个特别的目录），稍后若需要，可以从那里还原文件。当然，除非文件确实从回收站目录中删除，否则不会释放空间。

4.4.4 文件系统性能

访问磁盘比访问内存慢得多。读内存中一个32位字大概要10ns。从硬盘上读的速度大约超过100MB/s，对32位字来说，大约要慢4倍，还要加上5~10ms寻道时间，并等待所需的扇面抵达磁头下。如果只需要一个字，内存访问则比磁盘访问快百万数量级。考虑到访问时间的这个差异，许多文件系统采用了各种优化措施以改善性能。本节我们将介绍其中三种方法。

1. *高速缓存*

最常用的减少磁盘访问次数技术是*块高速缓存*（block cache）或者*缓冲区高速缓存*（buffer cache）。在本书中，高速缓存指的是一系列的块，它们在逻辑上属于磁盘，但实际上基于性能的考虑被保存在内存中。

管理高速缓存有不同的算法，常用的算法是：检查全部的读请求，查看在高速缓存中是否有所需要的块。如果存在，可执行读操作而无须访问磁盘。如果该块不在高速缓存中，首先要把它读到高速缓存，再复制到所需地方。之后，对同一个块的请求都通过高速缓存完成。

高速缓存的操作如图4-28所示。由于在高速缓存中有许多块（通常有上千块），所以需要有某种方法快速确定所需要的块是否存在。常用方法是将设备和磁盘地址进行散列操作，然后，在散列表中查找结果。具有相同散列值的块在一个链表中连接在一起，这样就可以沿着冲突链查找其他块。

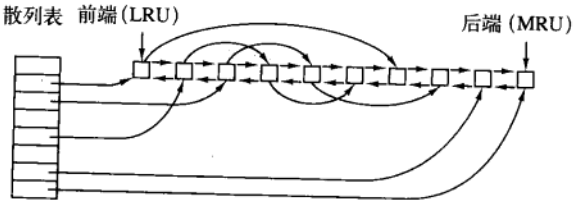


图4-28 缓冲区高速缓存数据结构

如果高速缓存已满，则需要调入新的块，因此，要把原来的某一块调出高速缓存（如果要调出的块在上次调入以后修改过，则要把它写回磁盘）。这种情况与分页非常相似，所有常用的页面置换算法在第3章中已经介绍，例如FIFO算法、第二次机会算法、LRU算法等，它们都适用于高速缓存。与分页相比，高速缓存的好处在于对高速缓存的引用不很频繁，所以按精确的LRU顺序在链表中记录全部的块是可行的。

在图4-28中可以看到，除了散列表中的冲突链之外，还有一个双向链表把所有的块按照使用时间的先后次序链接起来，近来使用最少的块在该链表的前端，而近来使用最多的块在该链表的后端。当引用某个块时，该块可以从双向链表中移走，并放置到该表的尾部去。用这种方法，可以维护一种准确的LRU顺序。

但是，这又带来了意想不到的难题。现在存在一种情形，使我们有可能获得精确的LRU，但是碰巧该LRU却又不符合要求。这个问题与前一节讨论的系统崩溃和文件一致性有关。如果一个关键块（比如i节点块）读进了高速缓存并做过修改，但是没有写回磁盘，这时，系统崩溃会导致文件系统的不一致。如果把i节点块放在LRU链的尾部，在它到达链首并写回磁盘前，有可能需要相当长的一段时间。

此外，某一些块，如i节点块，极少可能在短时间内被引用两次。基于这些考虑需要修改LRU方案，并应注意如下两点：

1) 这一块是否不久后会再次使用？

2) 这一块是否关系到文件系统的一致性？

考虑以上两个问题时，可将块分为i节点块、间接块、目录块、满数据块、部分数据块等几类。把有可能最近不再需要的块放在LRU链表的前部，而不是LRU链表的后端，于是它们所占用的缓冲区可以很快被重用。对很快就可能再次使用的块，比如正在写入的部分满数据块，可放在链表的尾部，这样它们能在高速缓存中保存较长的一段时间。

第二个问题独立于前一个问题。如果关系到文件系统一致性（除数据块之外，其他块基本上都是这样）的某块被修改，都应立即将该块写回磁盘，不管它是否被放在LRU链表尾部。将关键块快速写回磁盘，将大大减少在计算机崩溃后文件系统被破坏的可能性。用户的文件崩溃了，该用户会不高兴，但是如果整个文件系统都丢失了，那么这个用户会更生气。

尽管用这类方法可以保证文件系统一致性不受到破坏，但我们仍然不希望数据块在高速缓存中放很久之后才写入磁盘。设想某人正在用个人计算机编写一本书。尽管作者让编辑程序将正在编辑的文件定期写回磁盘，所有的内容只存在高速缓存中而不在磁盘上的可能性仍然非常大。如果这时系统崩溃，文件系统的结构并不会被破坏，但他一整天的工作就会丢失。

即使只发生几次这类情况，也会让人感到不愉快。系统采用两种方法解决这一问题。在UNIX系统中有一个系统调用sync，它强制性地把全部修改过的块立即写回磁盘。系统启动时，在后台运行一个通常名为update的程序，它在无限循环中不断执行sync调用，每两次调用之间休眠30s。于是，系统即使崩溃，也不会丢失超过30秒的工作。

虽然目前Windows有一个等价于sync的系统调用——FlushFileBuffers，不过过去没有。相反，Windows采用一个在某种程度上比UNIX方式更好（有时更坏）的策略。其做法是，只要被写进高速缓存，就把每个被修改的块写进磁盘。将缓存中所有被修改的块立即写回磁盘称为*通写高速缓存*（write-through cache）。同非通写高速缓存相比，通写高速缓存需要更多的磁盘I/O。

若某程序要写满1KB的块，每次写一个字符，这时可以看到这两种方法的区别。UNIX在高速缓存中保存全部字符，并把这个块每30秒写回磁盘一次，或者当从高速缓存删除这一块时，写回磁盘。在通写高速缓存里，每写入一字符就要访问一次磁盘。当然，多数程序有内部缓冲，通常情况下，在每次执行write系统调用时并不是只写入一个字符，而是写入一行或更大的单位。

采用这两种不同的高速缓存策略的结果是：在UNIX系统中，若不调用sync就移动（软）磁盘，往往会导致数据丢失，在被毁坏的文件系统中也经常如此。而在通写高速缓存中，就不会出现这类情况。选择不同策略的原因是，在UNIX开发环境中，全部磁盘都是硬盘，不可移动。而第一代Windows 文件源自MS-DOS，是从软盘世界中发展起来的。由于UNIX方案有更高的效率它成为当然的选择（但可靠性更差），随着硬盘成为标准，它目前也用在Windows的磁盘上。但是，NTFS使用其他方法（日志）改善其可靠性，这在前面已经讨论过。

一些操作系统将高速缓存与页缓存集成。这种方式特别是在支持内存映射文件的时候很吸引人。如果一个文件被映射到内存上，则它其中的一些页就会在内存中，因为它们被要求按页进入。这些页面与在高速缓存中的文件块几乎没有不同。在这种情况下，它们能被以同样的方式来对待，也就是说，用一个缓存来同时存储文件块与页。

2. 块提前读

第二个明显提高文件系统性能的技术是：在需要用到块之前，试图提前将其写入高速缓存，从而提高命中率。特别地，许多文件都是顺序读的。如果请求文件系统在某个文件中生成块k，文件系统执行相关操作且在完成之后，会在用户不察觉的情形下检查高速缓存，以便确定块$k+1$是否已经在高速缓存。如果还不在，文件系统会为块$k+1$安排一个预读，因为文件系统希望在需要用到该块时，它已经在高速缓存或者至少马上就要在高速缓存中了。

当然，块提前读策略只适用于顺序读取的文件。对随机存取文件，提前读丝毫不起作用。相反，它还会帮倒忙，因为读取无用的块以及从高速缓存中删除潜在有用的块将会占用固定的磁盘带宽（如果有“脏”块的话，还需要将它们写回磁盘，这就占用了更多的磁盘带宽）。那么提前读策略是否值得采用呢？文件系统通过跟踪每一个打开文件的访问方式来确定这一点。例如，可以使用与文件相关联的某个位协助跟踪该文件到底是“顺序存取方式”还是“随机存取方式”。在最初不能确定文件属于哪种存取方式时，先将该位设置成顺序存取方式。但是，查找一完成，就将该位清除。如果再次发生顺序读取，就再次设置该位。这样，文件系统可以通过合理的猜测，确定是否应该采取提前读的策略。即便弄错了一次也不会产生严重后果，不过是浪费一小段磁盘的带宽罢了。

3. 减少磁盘臂运动

高速缓存和块提前读并不是提高文件系统性能的惟一方法。另一种重要技术是把有可能顺序存取的

块放在一起，当然最好是在同一个柱面上，从而减少磁盘臂的移动次数。当写一个输出文件时，文件系统就必须按照要求一次一次地分配磁盘块。如果用位图来记录空闲块，并且整个位图在内存中，那么选择与前一块最近的空闲块是很容易的。如果用空闲表，并且链表的一部分存在磁盘上，要分配紧邻着的空闲块就困难得多。

不过，即使采用空闲表，也可以采用块簇技术。这里用到一个小技巧，即不用块而用连续块簇来跟踪磁盘存储区。如果一个扇区有512个字节，有可能系统采用1KB的块（2个扇区），但却按每2块（4个扇区）一个单位来分配磁盘存储区。这和2KB的磁盘块并不相同，因为在高速缓存中它依然使用1KB的块，磁盘与内存数据之间传送也是以1KB为单位进行，但在一个空闲的系统上顺序读取文件，寻道的次数可以减少一半，从而使文件系统的性能大大改善。若考虑旋转定位则可以得到这类方案的变体。在分配块时，系统尽量把一个文件中的连续块存放在同一柱面上。

在使用i节点或任何类似i节点的系统中，另一个性能瓶颈是，读取一个很短的文件也需要两次磁盘访问：一次是访问i节点，另一次是访问块。通常情况下，i节点的放置如图4-29a所示。其中，全部i节点都放在靠近磁盘开始位置，所以i节点和它指向的块之间的平均距离是柱面数的一半，这将需要较长的寻道时间。

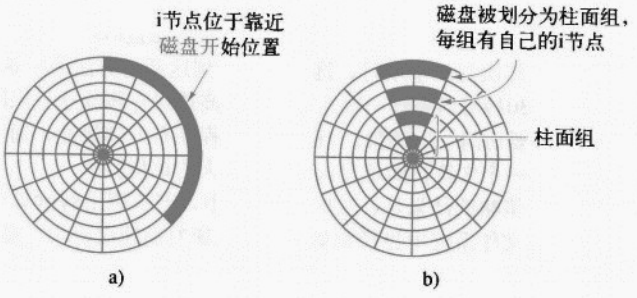


图4-29 a) i节点放在磁盘开始位置；b) 磁盘分为柱面组，每组有自己的块和i节点

一个简单的改进方法是，在磁盘中部而不是开始处存放i节点，此时，在i节点和第一块之间的平均寻道时间减为原来的一半。另一种做法是：将磁盘分成多个柱面组，每个柱面组有自己的i节点、数据块和空闲表（McKusick 等人，1984），见图4-29b。在文件创建时，可选取任一i节点，但首先在该i节点所在的柱面组上查找块。如果在该柱面组中没有空闲的块，就选用与之相邻的柱面组的一个块。

4.4.5 磁盘碎片整理

在初始安装操作系统后，从磁盘的开始位置，一个接一个地连续安装了程序与文件。所有的空闲磁盘空间放在一个单独的、与被安装的文件邻近的单元里。但随着时间的流逝，文件被不断地创建与删除，于是磁盘会产生很多碎片，文件与空穴到处都是。结果是，当创建一个新文件时，它使用的块会散布在整个磁盘上，造成性能的降低。

磁盘性能可以通过如下方式恢复：移动文件使它们相邻，并把所有的（至少是大部分的）空闲空间放在一个或多个大的连续的区域内。Windows有一个程序defrag就是从事这个工作的。Windows的用户应该定期使用它。

磁盘碎片整理程序会在一个在分区末端的连续区域内有适量空闲空间的文件系统上很好地运行。这段空间会允许磁盘碎片整理程序选择在分区开始端的碎片文件，并复制它们所有的块放到空闲空间内。这个动作在磁盘开始处释放出一个连续的块空间，这样原始或其他的文件可以在其中相邻地存放。这个过程可以在下一大块的磁盘空间上重复，并继续下去。

有些文件不能被移动，包括页文件、休眠文件以及日志，因为移动这些文件所需的管理成本要大于移动它们的价值。在一些系统中，这些文件是固定大小的连续的区域，因此它们不需要进行碎片整理。这类文件缺乏灵活性会造成一些问题，一种情况是，它们恰好在分区的末端附近并且用户想减小分区的大小。解决这种问题的惟一的方法是把它们一起删除，改变分区的大小，然后再重新建立它们。

Linux文件系统（特别是ext2和ext3）由于其选择磁盘块的方式，在磁盘碎片整理上一般不会遭受像Windows那样的困难，因此很少需要手动的磁盘碎片整理。

4.5 文件系统实例

在这一节，我们将讨论文件系统的几个实例，包括从相对简单的文件系统到十分复杂的文件系统。现代流行的UNIX文件系统和Windows Vista自带文件系统在本书的第10章和第11章有详细介绍，在此就不再讨论了。但是我们有必要来看看这些文件系统的前身。

4.5.1 CD-ROM文件系统

作为第一个文件系统实例，让我们来看看用于CD-ROM的文件系统。因为这些文件系统是为一次性写介质设计的，所以非常简单。例如，该文件系统不需要记录空闲块，这是因为一旦光盘生产出来后，CD-ROM上的文件就不能被删除或者创建了。下面我们来看看主要的CD-ROM文件系统类型以及对这个文件系统的两种扩展。

在CD-ROM出现一些年后，引进了CD-R（可记录CD）。不像CD-ROM，CD-R可以在初次刻录之后加文件，但只能简单地加在CD-R的最后面。文件不能删除（尽管可以更新目录来隐藏已存在的文件）。因而对于这种“只能添加”的文件系统，其基本的性质不会改变。特别地，所有的空闲空间放在了CD末端连续的一大块内。

1. ISO 9660文件系统

最普遍的一种CD-ROM文件系统的标准是1988年被采纳的名为ISO 9660的国际标准。实际上现在市场上的所有CD-ROM都支持这个标准，有的则带有一些扩展（下面会对此进行讨论）。这个标准的一个目标就是使CD-ROM独立于机器所采用的字节顺序和使用的操作系统，即在所有的机器上都是可读的。因此，在该文件系统上加上了一些限制，使得最弱的操作系统（如MS-DOS）也能读取该文件系统。

CD-ROM没有和磁盘一样的同心柱面，而是沿一个连续的螺旋线来顺序存储信息（当然，跨越螺旋线查找也是可能的）。螺旋上的位序列被划分成大小为2352字节的逻辑块（也称为逻辑扇区）。这些块有的用来进行引导，有的用来进行错误纠正或者其他一些用途。每个逻辑块的有效部分是2048字节。当用于存放音乐时，CD中有导入部分、导出部分以及轨道间的间隙，但是用于存储数据的CD-ROM则没有这些。通常，螺旋上的逻辑块是按分钟或者秒进行分配的。通过转换系数1秒＝75块，则可以转换得到相应的线性块号。

ISO 9660支持的CD-ROM集可以有多达$2^{16}-1$个CD。每个单独的CD-ROM还可分为多个逻辑卷（分区）。下面我们重点考虑单个没有分区CD-ROM时的ISO 9660。

每个CD-ROM有16块作为开始，这16块的用途在ISO 9660标准中没有定义。CD-ROM制造商可以在这一区域里放入引导程序，使计算机能够从CD-ROM引导，或者用于其他目的。接下来的一块存放基本卷描述符（primary volume descriptor），基本卷描述符包含了CD-ROM的一些基本信息。这些信息包括系统标识符（32字节）、卷标识符（32字节）、发布标识符（128字节）和数据预备标识符（128字节）。制造商可以在上面的几个域中填入需要的信息，但是为了跨平台的兼容性，不能使用大写字母、数字以及很少一部分标点符号。

基本卷描述符还包含了三个文件的名字，这三个文件分别用来存储概述、版权声明和文献信息。除此之外，还包含有一些关键数字信息，例如逻辑块的大小（通常为2048，但是在某些情况下可以是4096、8192或者更大）、CD-ROM所包含的块数目以及CD-ROM的创建日期和过期日期。基本卷描述符也包含了根目录的目录表项，说明根目录在CD-ROM的位置（即从哪一块开始）。从这个根目录，系统就能找到其他文件所在的位置。

除基本卷描述符之外，CD-ROM还包含有一个补充卷描述符（supplementary volume descriptor）。它和基本卷描述符包含类似的信息，在这里不再详细讨论。

根目录和所有的其他目录包含可变数目的目录项，目录中的最后一个目录项有一位用于标记该目录项是目录中的最后一个。目录项本身也是长度可变的。每一个目录项由10到12个域构成，其中一些域是ASCII域，另外一些是二进制数字域。二进制域被编码两次，一个用于低地址结尾格式（例如在Pentium

上所用的），一个用于高地址结尾格式（例如在SPARC上所用的）。因此，一个16位的数字需要4个字节，一个32位的数字需要8个字节。

这样冗余编码的目的主要是为了能在标准发展的同时照顾到各个方面的利益。如果该标准仅规定低地址结尾，那么在产品中使用高地址结尾的厂家就会觉得自己受到歧视，就不会接受这个标准。所以我们可以准确地用冗余的字节/小时数来衡量一张CD-ROM的情感因素。

ISO 9660目录项的格式如图4-30所示。因为目录项是长度可变的，所以，第一个域就说明这一项的长度。这一字节被定义为高位在左，以避免混淆。

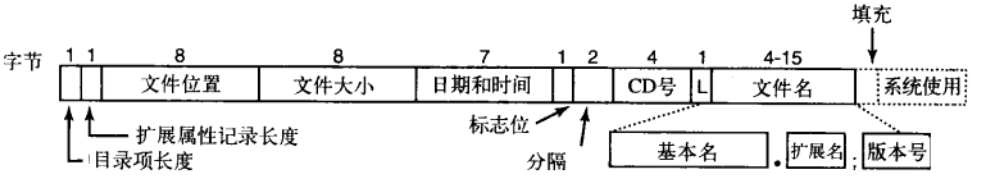


图4-30 ISO 9660的目录项

目录项可能包含有扩展属性。如果使用了这个特性，则第二个字节就说明扩展属性的长度。

接下来是文件本身的起始块。文件是以连续块的方式存储的，所以一个文件的位置完全可以由起始块的位置和大小来确定。起始块的下一个域就是文件大小。

CD-ROM的日期和时间被记录在下一个域中，其中分隔的字节分别表示年、月、日、小时、分钟、秒和时区。年份是从1900年开始计数的，这意味着CD-ROM将会遇到2156年问题，因为在2155年之后将会是1900年。如果定义初始的日期为1988年（标准通过的那一年）的话，那么这个问题就可以推迟88年产生，也就是2244年。

标志位域包含一些其他的位，包括一个用来在打开目录时隐藏目录项（来自MS-DOS的特性）的标志位，一个用以区分该项是文件还是目录的标志位，一个用以标志是否使用扩展属性的标志位，以及一个用来标志该项是否为目录中最后一项的标志位。其他一些标志位也在这个域中，但是在此我们不再讨论。下一个域说明了在ISO 9660的最简版本中是否使用文件分隔块，我们也不做讨论。

再下一个域标明了该文件放在哪一个CD-ROM上。一个CD-ROM的目录项可以引用在同一CD-ROM集中的另外一个CD-ROM上的文件。用这样的方法就可以在第一张CD-ROM上建立一个主目录，该主目录列出了在这个CD-ROM集合中的其他所有CD-ROM上的文件。

图4-30中标有L的域给出了文件名的大小（以字节为单位）。之后的域就是文件名本身。一个文件名由基本名、一个点、扩展名、分号和二进制版本号（1或2个字节）构成。基本名和扩展名可以使用大写字母、数字0~9和下划线。禁止使用其他字符以保证所有的机器都能处理这个文件名。基本名最多可以为8个字符，而扩展名最多可以为3个字符。这样做是为了保证能和MS-DOS兼容。只要文件的版本号不同，则相同的文件名可以在同一个目录中出现多次。

最后两个域不是必需的。填充域用来保证每一个目录项都是偶数个字节，以2字节为边界对齐下一项的数字域。如果需要填充的话，就用0代替。最后一个域是系统使用域，该域的功能和大小没有定义，仅仅只要求该域为偶数个字节。不同的系统对该域有不同的用途。例如，Macintosh系统就把此域用来保存Finder标志。

一个目录中的项除了前两项之外，其余的都按字母顺序排列。第一项表示当前目录本身，第二项表示当前目录的父目录。这和UNIX的.目录和..目录相似。而文件本身不需要按其目录项在目录中的顺序来排列。

对于目录中目录项的数目没有特定的限制；但是对于目录的嵌套深度有限制，最大的目录嵌套深度为8。为了使得有关的实现简化一些，这个限制是任意设置的。

ISO 9660定义了三个级别。级别1的限制最多，限制文件名使用上面提到的8+3个字符的表示法，而且所有的文件必须是连续的（这些我们在前面介绍过）。进而，目录名被限制在8个字符而且不能有扩展名。这个级别的使用，使得CD-ROM可以在所有的机器上读出。

级别2放宽了对长度的限制。它允许文件和目录名多达31个字符，但是字符集还是一样的。

级别3使用和级别2同样的限制，但是文件不需要是连续的。在这个级别上，一个文件可以由几个段（extents）构成，每一个段可以由若干连续分块构成。同一个分块可以在一个文件中出现多次，也可以出现在两个或者更多的文件中。如果相当大的一部分数据在几个文件中重复，级别3则通过要求数据不能出现多次来进行空间上的优化。

2. Rock Ridge扩展

正如我们上面所看到的，ISO 9660在很多方面有限制。在这个标准公布不久，UNIX工作者开始在这个标准上进行扩展，使得在CD-ROM上能实现UNIX文件系统。这个扩展被命名为Rock Ridge，这个名字来源于Gene Wilder的电影《Blazing Saddles》中一个小镇，也许委员会的成员之一喜欢这个电影，便以此命名。

该扩展使用了系统使用域，使得Rock Ridge CD-ROM可以在所有计算机上可读。其他所有的域仍然保持ISO 9660标准中的定义。所有其他不识别Rock Ridge扩展的系统只需要忽略这个域，把盘当作普通的CD-ROM来识别即可。

该扩展分为下面几个域：

1) PX——POSIX属性。

2) PN——主设备号和次设备号。

3) SL——符号链接。

4) NM——替代名。

5) CL——子位置。

6) PL——父位置。

7) RE——重定位。

8) TF——时间戳。

PX域包含了标准UNIX 的rwxrwxrwx所有者、同组用户和其他用户权限位。也包含了包含在模式字中的其他位，如SETUID位和SETGID位等。

为了能在CD-ROM上表示原始设备，需要PN域来表示。该域包含了和文件相关的主设备号和次设备号。这样，/dev目录的内容就可以在写到CD-ROM上之后在目标系统上重新正确地构造。

SL域是符号链接，它允许在一个文件系统上的文件可以引用另一个文件系统上的文件。

最重要的域是NM域。它允许同一个文件可以关联第二个名字。这个名字不受ISO 9660字符集和长度的限制，这样使得在CD-ROM上可以表示任意的UNIX文件。

接下来的三个域一起用来消除ISO 9660中的对目录嵌套深度为8的限制。使用这几个域可以指明一个目录被重定位了，而且可以标明其层次结构。这对于消除深度限制非常有用。

最后，TF域包含了每个UNIX的i节点中的三个时间戳：文件创建时间、文件修改时间和文件最后访问的时间。有了这些扩展，就可以将一个UNIX文件系统复制到CD-ROM上，并且能够在不同的系统上正确恢复。

3. Joliet扩展

UNIX委员会不是惟一对ISO 9660进行扩展的小组，微软也发现了这个标准有太多的限制（尽管这些限制最初都是由于微软自己的MS-DOS引起的）。所以微软也做了一些扩展，名为Joliet。这个扩展设计的目的是，为了能够将Windows文件系统复制到CD-ROM上，并且能够恢复（与为UNIX设计Rock Ridge的思路一样）。实际上所有能在Windows上运行的、使用CD-ROM的程序都支持Joliet，包括可写CD的刻录程序。通常这些程序都让用户选择是使用ISO 9660标准还是Joliet标准。

Joliet提供的主要扩展为：

1) 长文件名。

2) Unicode字符集。

3) 比8层更深的目录嵌套深度。

4) 带扩展名的目录。

第一个扩展允许文件名多达64字符。第二个扩展允许文件名使用Unicode字符集，这个扩展对那些不使用拉丁字符集的国家非常重要，如日本、以色列和希腊。因为Unicode字符是2个字节的，所以Joliet最长的文件名可以达到128字节。

和Rock Ridge一样，Joliet同样消除了对目录嵌套深度的限制。目录可以根据需要达到一定的嵌套深度。最后，目录名也可以有扩展名。目前还不清楚为什么有这个扩展，因为大多数的Windows目录从来没有扩展名，但或许有一天会用到。

4.5.2 MS-DOS文件系统

MS-DOS文件系统是第一个IBM PC系列所采用的文件系统。它也是Windows 98与Windows ME所采用的主要的文件系统。Windows 2000、Windows XP与Windows Vista上也支持它，虽然除了软盘以外，它现在已经不再是新的PC的标准了。但是，它和它的扩展（FAT-32）一直被许多嵌入式系统所广泛使用。大部分的数码相机使用它。许多MP3播放器只能使用它。流行的苹果公司的iPod使用它作为默认的文件系统，尽管知识渊博的骇客可以重新格式化iPod并安装一个不同的文件系统。使用MS-DOS文件系统的电子设备的数量现在要远远多于过去，并且当然远远多于使用更现代的NTFS文件系统的数量。因此，我们有必要看一看其中的一些细节。

要读文件时，MS-DOS程序首先要调用open系统调用，以获得文件的句柄。open系统调用识别一个路径，可以是绝对路径或者是相对于现在工作目录的路径。路径是一个分量一个分量地查找的，直到查到最终的目录并读进内存。然后开始搜索要打开的文件。

尽管MS-DOS的目录是可变大小的，但它使用固定的32字节的目录项，MS-DOS的目录项的格式如图4-31所示。它包含文件名、属性、建立日期和时间、起始块和具体的文件大小。在每个分开的域中，少于8+3个字符的文件名左对齐，在右边补空格。属性域是一个新的域，包含用来指示一个文件是只读的、存档的、隐藏的还是一个系统文件的位。不能写只读文件，这样避免了文件意外受损。存档位没有对应的操作系统的功能（即MS-DOS不检查和设置它）。存档位主要的用途是使用户级别的存档程序在存档一个文件后清理这一位，其他程序在修改了这个文件之后设置这一位。以这种方式，一个备份程序可以检查每个文件的这一位来确定是否需要备份该文件。设置隐藏位能够使一个文件在目录列表中不出现，其作用是避免初级用户被一些不熟悉的文件搞糊涂了。最后，系统位也隐藏文件。另外，系统文件不可以用del命令删除，在MS-DOS的主要组成部分中，系统位都被设置。

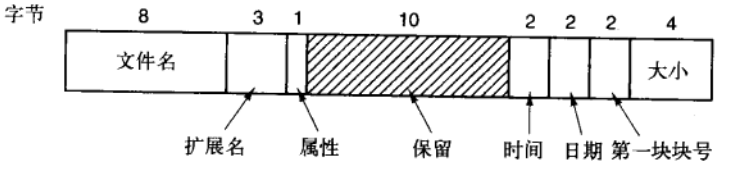


图4-31 MS-DOS的目录项

目录项也包含了文件建立和最后修改的日期和时间。时间只是精确到±2s，因为它只是用2个字节的域来存储，只能存储65 536个不同的值（一天包含86 400秒）。这个时间域被分为秒（5个位）、分（6个位）和小时（5个位）。以日为单位计算的日期使用三个子域：日（5个位），月（4个位），年-1980（7个位）。用7个位的数字表示年，时间的起始为1980年，最高的表示年份是2107年。所以MS-DOS有内在的2108年问题。为了避免灾难，MS-DOS的用户应该尽快开始在2108年之前转变工作。如果把MS-DOS使用组合的日期和时间域作为32位的秒计数器，它就能准确到秒，可把灾难推迟到2116年。

MS-DOS按32位的数字存储文件的大小，所以理论上文件大小能够大至4GB。尽管如此，其他的约束（下面论述）将最大文件限制在2GB或者更小。让人吃惊的是目录项中的很大一部分空间（10 字节）没有使用。

MS-DOS通过内存里的文件分配表来跟踪文件块。目录表项包含了第一个文件块的编号，这个编号用作内存里有64K个目录项的FAT的索引。沿着这条链，所有的块都能找到。FAT的操作在图4-12中有描述。

FAT文件系统总共有三个版本：FAT-12，FAT-16和FAT-32，取决于磁盘地址包含有多少二进制位。

其实，FAT-32只用到了地址空间中的低28位，它更应该叫FAT-28。但使用2的幂的这种表述听起来要匀整得多。

在所有的FAT中，都可以把磁盘块大小调整到512字节的倍数（不同的分区可能采用不同的倍数），合法的块大小（微软称之为簇大小）在不同的FAT中也会有所不同。第一版的MS-DOS使用块大小为512字节的FAT-12，分区大小最大为$2^{12}\times$ 512字节（实际上只有4086×512字节，因为有10个磁盘地址被用作特殊的标记，如文件的结尾、坏块等）。根据这些参数，最大的磁盘分区大小约为2MB，而内存里的FAT表中有4096个项，每项2字节（16位）。若使用12位的目录项则会非常慢。

这个系统在软盘条件下工作得很好，但当硬盘出现时，它就出现问题了。微软通过允许其他的块大小如（1KB，2KB,4KB）来解决这个问题。这个修改保留了FAT-12表的结构和大小，但是允许可达16MB的磁盘分区。

由于MS-DOS支持在每个磁盘驱动器中划分四个磁盘分区，所以新的FAT-12文件系统可在最大64MB 的磁盘上工作。除此之外，还必须引入新的内容。于是就引进了FAT-16，它有16位的磁盘指针，而且允许8KB、16KB和32KB的块大小（32 768是用16位可以表示的2的最大幂）。FAT-16表需要占据内存128KB的空间。由于当时已经有更大的内存，所以它很快就得到了应用，并且取代了FAT-12系统。FAT-16能够支持的最大磁盘分区是2GB（64K个项，每个项32KB），支持最大8GB的磁盘，即4个分区，每个分区2GB。

对于商业信函来说，这个限制不是问题，但对于存储采用DV标准的数字视频来说，一个2GB的文件仅能保存9分钟多一点的视频。结果就是无论磁盘有多大，PC的磁盘也只能支持四个分区，能存储在磁盘中的最长的视频大约是38分钟。这一限制也意味着，能够在线编辑的最大的视频少于19分钟，因为同时需要输入和输出文件。

随着Windows 95第2版的发行，引入了FAT-32文件系统，它具有28位磁盘地址。在Windwos 95下的MS-DOS也被改造，以适应FAT-32。在这个系统中，分区理论上能达到$2^{28}\times 2^{15}$字节，但实际上是限制在2TB（2048GB），因为系统在内部的512字节长的扇区中使用了一个32位的数字来记录分区的大小，这样$2^{9}\times 2^{32}$是2TB。对应不同的块大小以及所有三种FAT类型的最大分区都在图4-32中表示出来。

块大小	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

图4-32 对应不同的块大小的最大分区（空格表示禁止这种组合）

除了支持更大的磁盘之外，FAT-32文件系统相比FAT-16文件系统有另外两个优点。首先，一个用FAT-32的8GB磁盘可以是一个分区，而使用FAT-16则必须是四个分区，对于Windows用户来说，就是“C:”、“D:”、“E:”和“F:”逻辑磁盘驱动器。用户可以自己决定哪个文件放在哪个盘以及记录的内容放在什么地方。

FAT-32相对于FAT-16的另外一个优点是，对于一个给定大小的硬盘分区，可以使用一个小一点的块大小。例如，对于一个2GB的硬盘分区，FAT-16必须使用32KB的块，否则仅有的64K个磁盘地址就不能覆盖整个分区。相反，FAT-32处理一个2GB的硬盘分区的时候就能够使用4KB的块。使用小块的好处是大部分文件都小于32KB。如果块大小是32KB，那么一个10 字节的文件就占用32KB的空间，如果文件平均大小是8KB，使用32KB的块大小，3/4的磁盘空间会被浪费，这不是使用磁盘的有效方法。而8KB的文件用4KB的块没有空间的损失，却会有更多的RAM被FAT系统占用。把4KB的块应用到一个2GB的磁盘分区，会有512K个块，所以FAT系统必须在内存里包含512K个项（占用了2MB的RAM）。

MS-DOS使用FAT来跟踪空闲磁盘块。当前没有分配的任何块都会标上一个特殊的代码。当MS-DOS需要一个新的磁盘块时，它会搜索FAT以找到一个包含这个代码的项。所以不需要位图或者空闲表。

4.5.3 UNIX V7 文件系统

即使是早期版本的UNIX也有一个相当复杂的多用户文件系统，因为它是从MULTICS继承下来的。下面我们将会讨论V7文件系统，这是为PDP-11创建的一个文件系统，它也使得UNIX闻名于世。我们将

在第10章通过Linux讨论现代UNIX的文件系统。

文件系统从根目录开始形成树状，加上链接，形成了一个有向无环图。文件名可以多达14个字符，能够容纳除了/和NUL之外的任何ASCII字符，NUL也表示成数字数值0。

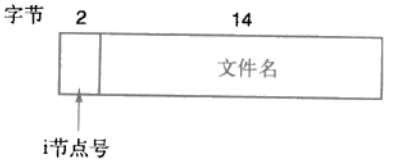


图4-33 UNIX V7的目录表项

UNIX目录中为每个文件保留了一项。每项都很简单，因为UNIX使用i节点，如图4-13中所示。一个目录项包含了两个域，文件名（14个字节）和i节点的编号（2个字节），如图4-33所示。这些参数决定了每个文件系统的文件数目为64K。

就像图4-13中的i节点一样，UNIX的i节点包含一些属性。这些属性包括文件大小、三个时间（创建时间，最后访问时间，最后修改时间）、所有者、所在组、保护信息以及一个计数（用于记录指向i节点的目录项的数量）。最后一个域是为了链接而设的。当一个新的链接加到一个i节点上，i节点里的计数就会加1。当移走一个连接时，该计数就减1。当计数为0时，就收回该i节点，并将对应的磁盘块放进空闲表。

对于特别大的文件，可以通过图4-13所示的方法来跟踪磁盘块。前10个磁盘地址是存储在i节点自身中的，所以对于小文件来说，所有必需的信息恰好是在i节点中。而当文件被打开时，i节点将被从磁盘取到内存中。对于大一些的文件，i节点内的其中一个地址是称为一次间接块（single indirect block）的磁盘块地址。这个块包含了附加的磁盘地址。如果还不够的话，在i节点中还有另一个地址，称为二次间接块（double indirect block）。它包含一个块的地址，在这个块中包含若干个一次间接块。每一个这样的一次间接块指向数百个数据块。如果这样还不够的话，可以使用三次间接块（triple indirect block）。整个情况参见图4-34。

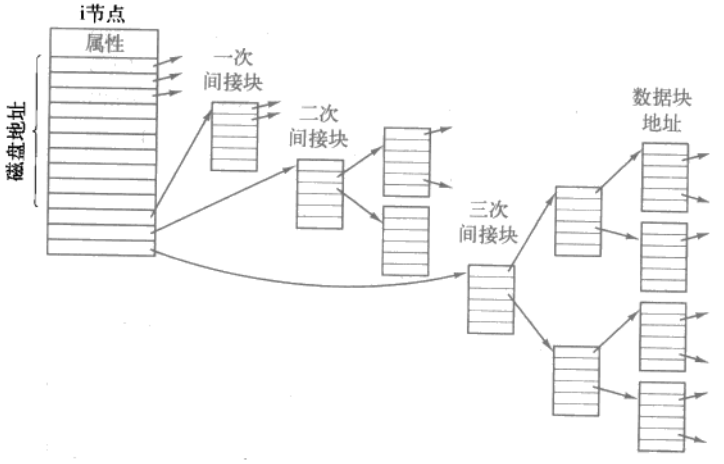


图4-34 一个UNIX的i节点

当打开某个文件时，文件系统必须要获得文件名并且定位它所在的磁盘块。让我们来看一下怎样查找路径名/usr/ast/mbox。以UNIX为例，但对所有的层次目录系统来说，这个算法是大致相同的。首先，文件系统定位根目录。在UNIX系统中，根目录的i节点存放于磁盘上固定的位置。从这个i节点，系统将可以定位根目录，虽然根目录可以放在磁盘上的任何位置，但假定它放在磁盘块1的位置。

接下来，系统读根目录并且在根目录中查找路径的第一个分量usr，以获取/usr目录的i节点号。由i节点号来定位i节点是很直接的，因为每个i节点在磁盘上都有固定的位置。根据这个i节点，系统定位/usr目录并在其中查找下一个分量ast。一旦找到ast的项，便找到了/usr/ast目录的i节点。依据这个i节点，可以定位该目录并在其中查找mbox。然后，这个文件的i节点被读入内存，并且在文件关闭之前会一直保留在内存中。图4-35显示了查找的过程。

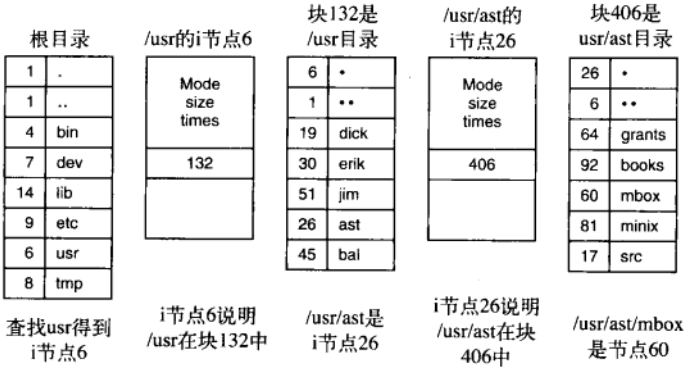


图4-35 查找/usr/ast/mbox的过程

相对路径名的查找同绝对路径的查找方法相同，只不过是从当前工作目录开始查找而不是从根目录开始。每个目录都有.和..项，它们是在目录创建的时候同时创建的。.表项是当前目录的i节点号，而..表项是父目录（上一层目录）的i节点号。这样，查找../dick/prog.c的过程就成为在工作目录中查找..，寻找父目录的i节点号，并查询dick目录。不需要专门的机制处理这些名字。目录系统只要把这些名字看作普通的ASCII字符串即可，如同其他的名字一样。这里惟一的巧妙之处是..在根目录中指向自身。

4.6 有关文件系统的研究

文件系统总是比操作系统的其他部分吸引了更多的研究，现在也是这样。当标准的文件系统被完全理解后，现在还有很多后续的关于优化高速缓存管理的研究（Burnett等人，2002；Ding等人，2007；Gnaidy等人，2004；Kroeger和Long，2001；Pai等人，2000；以及Zhou等人，2001）。后续的工作还有关于新类型的文件系统，例如用户级别的文件系统（Mazières，2001），闪存文件系统（Gal等人，2005），日志文件系统（Prabhakaran等人，2005；以及Stein等人，2001），版本控制（versioning）文件系统（Cornell等人，2004），对等（peer-to-peer）文件系统（Muthitacharoen等人，2002），以及其他。Google文件系统也不寻常，因为它有极好的容错性能（Ghemawat等人，2003）。文件系统内不同的查询方法也是很有意义（Padioleau和Ridoux，2003）。

另一个受到关注的领域是起源（provenance）——追踪数据的历史，包括它们来自哪里，谁拥有它们，以及它们是如何转换的（Muniswarmy-Reddy等人，2006；以及Shah等人，2007）。这个信息可以以不同的方式加以运用。备份也一直受到关注（Cox等人，2002；以及Rycroft等人，2006），如同恢复的相关主题一样（Keeton等人，2006）。与备份有关的还有，设法保持数据几十年，并仍旧可以使用（Baker等人，2006；Maniatis等人，2003）。可靠性与安全性也是需要解决的问题（Greenan和Miller，2006；Wires和Feeley，2007；Wright等人，2007；以及Yang等人，2006）。最后，性能始终是一个值得研究的主题（Caudill和Gavrikovska，2006；Chiang和Huang，2007；Stein，2006；Wang等人，2006a；以及Zhang和Ghose，2007）。

4.7 小结

从外部看，文件系统是一组文件和目录，以及对文件和目录的操作。文件可以被读写，目录可以被创建和删除，并可将文件从一个目录移到另一个目录中。大多数现代操作系统都支持层次目录系统，其中，目录中还有子目录，子目录中还可以有子目录，如此无限下去。

而在内部看，文件系统又是另一番景象。文件系统的设计者必须考虑存储区是如何分配的，系统如何记录哪个块分给了哪个文件。可能的方案有连续文件、链表、文件分配表和i节点等。不同的系统有不同的目录结构。属性可以存在目录中或存在别处（比如，在i节点中）。磁盘空间可以通过位图的空闲表来管理。通过增量转储以及用程序修复故障文件系统的方法，可以提高文件系统的可靠性。文件系统的性能非常重要，可以通过多种途径提高性能，包括高速缓存、预读取以及尽可能仔细地将一个文件中

的块紧密地放置在一起等方法。日志结构文件系统通过大块单元写入的操作也可以改善性能。

文件系统的例子有ISO 9660、MS-DOS以及UNIX。它们之间在怎样记录每个文件所使用的块、目录结构以及对空闲磁盘空间管理等方面都存在着差别。

习题

1. 在早期的 UNIX系统中，可执行文件（a.out）以一个非常特别的魔数开始，这个数不是随机选择的。这些文件都有文件头，后面是正文段和数据段。为什么要为可执行文件挑选一个非常特别的魔数，而其他类型文件的第一个字反而有一个或多或少是随机选择的魔数？
2. 在图4-4中，一个属性是记录长度。为什么操作系统要关心这个属性？
3. 在UNIX中open系统调用绝对需要吗？如果没有会产生什么结果？
4. 在支持顺序文件的系统中总有一个文件回绕操作，支持随机存取文件的系统是否也需要该操作？
5. 某一些操作系统提供系统调用rename给文件重命名，同样也可以通过把文件复制到新文件并删除原文件而实现文件重命名。请问这两种方法有何不同？
6. 在有些系统中有可能把部分文件映射进内存中。如此一来系统应该施加什么限制？这种部分映射如何实现？
7. 有一个简单操作系统只支持单一目录结构，但是允许该目录中有任意多个文件，且带有任意长度的名字。这样可以模拟层次文件系统吗？如何进行？
8. 在UNIX和Windows中，通过使用一个特殊的系统调用把文件的“当前位置”指针移到指定字节，从而实现了随机访问。请提出一个不使用该系统调用完成随机存取的替代方案。
9. 考虑图4-8中的目录树，如果当前工作目录是/usr/jim，则相对路径名为../ast/x的文件的绝对路径名是什么？
10. 正如书中所提到的，文件的连续分配会导致磁盘碎片，因为当一个文件的长度不等于块的整数倍时，文件中的最后一个磁盘块中的空间会浪费掉。请问这是内碎片还是外碎片？并将它与先前一章的有关讨论进行比较。
11. 一种在磁盘上连续分配并且可以避免空洞的方案是，每次删除一个文件后就紧缩一下磁盘。由于所有的文件都是连续的，复制文件时需要寻道和旋转延迟以便读取文件，然后全速传送。在写回文件时要做同样的工作。假设寻道时间为5ms，旋转延迟为4 ms，传送速率为8MB/s，而文件平均长度是8 KB，把一个文件读入内存并写回到磁盘上的一个新位置需要多长时间？运用这些数字，计算紧缩16GB磁盘的一半需要多长时间？
12. 基于前一个问题的答案，紧缩磁盘有什么作用吗？
13. 某些数字消费设备需要存储数据，比如存放文件等。给出一个现代设备的名字，该设备需要文件存储，并且对文件运用连续分配空间的方法是不错的方法。
14. MS-DOS如何在文件中实现随机访问？
15. 考虑图4-13中的i节点。如果它含有用4个字节表示的10个直接地址，而且所有的磁盘块大小是1024KB，那么文件最大可能有多大？
16. 有建议说，把短文件的数据存在i节点之内会提高效率并且节省磁盘空间。对于图4-13中的i节点，在i节点之内可以存放多少字节的数据？
17. 两个计算机科学系的学生Carolyn和Elinor正在讨论i节点。Carolyn认为存储器容量越来越大，价格越来越便宜，所以当打开文件时，直接取i节点的副本，放到内存i节点表中，建立一个新i节点将更简单、更快，没有必要搜索整个i节点来判断它是否已经存在。Elinor则不同意这一观点。他们两个人谁对？
18. 说明硬连接优于符号链接的一个优点，并说明符号链接优于硬连接的一个优点。
19. 空闲磁盘空间可用空闲块表或位图来跟踪。假设磁盘地址需要D位，一个磁盘有B个块，其中有F个空闲。在什么条件下，空闲块表采用的空间少于位图？设D为16位，请计算空闲磁盘空间的百分比。
20. 一个空闲块位图开始时和磁盘分区首次初始化类似，比如：1000 0000 0000 0000（首块被根目录使用），系统总是从最小编号的盘块开始寻找空闲块，所以在有6块的文件A写入之后，该位图为 1111 1110 0000 0000。请说明在完成如下每一个附加动作之后位图的状态：

    a) 写入有5块的文件B。

    b) 删除文件A。

c) 写入有8块的文件C。

d) 删除文件B。

21. 如果因为系统崩溃而使存放空闲磁盘块信息的空闲块表或位图完全丢失，会发生什么情况？有什么办法从这个灾难中恢复吗，还是与该磁盘彻底再见？分别就UNIX和FAT-16文件系统讨论你的答案。
22. Oliver Owl在大学计算中心的工作是更换用于通宵数据备份的磁带，在等待每盘磁带完成的同时，他在写一篇毕业论文，证明莎士比亚戏剧是由外星访客写成的。由于仅有一个系统，所以只能在正在做备份的系统上运行文本编辑程序。这样的安排有什么问题吗？
23. 在教材中我们详细讨论过增量转储。在Windows中很容易说明何时要转储一个文件，因为每个文件都有一个存档位。在UNIX中没有这个位，那么UNIX备份程序怎样知道哪个文件需要转储？
24. 假设图4-25中的文件21自上次转储之后没有修改过，在什么情况下图4-26中的四张位图会不同？
25. 有人建议每个UNIX文件的第一部分最好和其i节点放在同一个磁盘块中，这样做有什么好处？
26. 考虑图4-27。对某个特殊的块号，计数器的值在两个表中有没有可能都是数值2？这个问题如何纠正？
27. 文件系统的性能与高速缓存的命中率有很大的关系（即在高速缓存中找到所需块的概率）。从高速缓存中读取数据需要1ms，而从磁盘上读取需要40ms，若命中率为h，给出读取数据所需平均时间的计算公式。并画出h从0到1.0变化时的函数曲线。
28. 考虑图4-21背后的思想，目前磁盘平均寻道时间为8ms，旋转速率为15 000rpm，每道为262 144字节。对大小各为1KB、2KB和4KB的磁盘块，传送速率各是多少？
29. 某个文件系统使用2KB的磁盘块，而中间文件大小值为1KB。如果所有的文件都是正好1KB大，那么浪费掉的磁盘空间的比例是多少？你认为一个真正的文件系统所浪费的空间比这个数值大还是小？请说明理由。
30. MS-DOS的FAT-16表有64K个表项，假设其中的一位必须用于其他用途，这样该表就只有32 768个表项了。如果没有其他修改，在这个条件下最大的MS-DOS文件有多大？
31. MS-DOS中的文件必须在内存中的FAT-16表中竞争空间。如果某个文件使用了k个表项，其他任何文件就不能使用这k个表项，这样会对所有文件的总长度带来什么限制？
32. 一个UNIX系统使用1KB磁盘块和4字节磁盘地址。如果每个i节点中有10个直接表项以及一个一次间接块、一个二次间接块和一个三次间接块，那么文件的最大尺寸是多少？
33. 对于文件 /usr/ast/courses/os/handout.t，若要调入其i节点需要多少个磁盘操作？假设其根目录的i节点在内存中，其他路径都不在内存中。并假设所有的目录都在一个磁盘块中。
34. 在许多UNIX系统中，i节点存放在磁盘的开始之处。一种替代设计方案是，在文件创建时分配i节点，并把i节点存放在该文件首个磁盘块的开始之处。请讨论这个方案的优缺点。
35. 编写一个将文件字节倒写的程序，这样最后一个字节成为第一个字节，而第一个字节成为最后一个字节。程序必须适合任何长度的文件，并保持适当的效率。
36. 编写一个程序，该程序从给定的目录开始，从此点开始沿目录树向下，记录所找到的所有文件的大小。在完成这一切之后，该程序应该打印出文件大小分布的直方图，以该直方图的区间宽度为参数（比如，区间宽度为1024，那么大小为0~1023的文件同在一个区间宽度，大小为1024~2047的文件同在下一个区间宽度，如此类推）。
37. 编写一个程序，扫描UNIX文件系统中的所有目录，并发现和定位有两个或更多硬连接计数的i节点。对于每个这样的文件，列出指向该文件的所有文件的名称。
38. 编写UNIX 的新版ls程序。这个版本将一个或多个目录名作为变量，并列出每个目录中所有的文件，一个文件一行。每个域应该对其类型进行合理的格式化。仅列出第一个磁盘地址（若该地址存在的话）。

第5章 输入/输出

除了提供抽象(例如，进程（和线程）、地址空间和文件）以外，操作系统还要控制计算机的所有I/O（输入/输出）设备。操作系统必须向设备发送命令，捕捉中断，并处理设备的各种错误。它还应该在设备和系统的其他部分之间提供简单且易于使用的接口。如果有可能，这个接口对于所有设备都应该是相同的，这就是所谓的设备无关性。I/O部分的代码是整个操作系统的重要组成部分。操作系统如何管理I/O是本章的主题。

本章的内容是这样组织的：首先介绍I/O硬件的基本原理，然后介绍一般的I/O软件。I/O软件可以分层构造，每层都有明确的任务。我们将对这些软件层进行研究，看一看它们做些什么，以及如何在一起配合工作。

在此之后将详细介绍几种I/O设备：磁盘、时钟、键盘和显示器。对于每一种设备我们都将从硬件和软件两方面加以介绍。最后，我们还将介绍电源管理。

5.1 I/O硬件原理

不同的人对于I/O硬件的理解是不同的。对于电子工程师而言，I/O硬件就是芯片、导线、电源、电机和其他组成硬件的物理部件。对程序员而言，则只注意I/O硬件提供给软件的接口，如硬件能够接收的命令、它能够完成的功能以及它能够报告的错误。本书主要介绍怎样对I/O设备编程，而不是如何设计、制造和维护硬件，因此，我们的讨论限于如何对硬件编程，而不是其内部的工作原理。然而，很多I/O设备的编程常常与其内部操作密切相关。在下面三节中，我们将介绍与I/O硬件编程有关的一般性背景知识。这些内容可以看成是对1.4节介绍性材料的复习和扩充。

5.1.1 I/O设备

I/O设备大致可以分为两类：*块设备*（block device）和*字符设备*（character device）。块设备把信息存储在固定大小的块中，每个块有自己的地址。通常块的大小在512字节至32 768字节之间。所有传输以一个或多个完整的（连续的）块为单位。块设备的基本特征是每个块都能独立于其他块而读写。硬盘、CD-ROM和USB盘是最常见的块设备。

如果仔细观察，块可寻址的设备与其他设备之间并没有严格的界限。磁盘是公认的块可寻址的设备，因为无论磁盘臂当前处于什么位置，它总是能够寻址其他柱面并且等待所需要的磁盘块旋转到磁头下面。现在考虑一个用来对磁盘进行备份的磁带机。磁带包含按顺序排列的块。如果给出命令让磁带机读第*N*块，它可以首先向回倒带，然后再前进直到第*N*块。该操作与磁盘的寻道相类似，只是花费的时间更长。不过，重写磁带中间位置的块有可能做得到，也有可能做不到。即便有可能把磁带当作随机访问的块设备来使用，也是有些勉为其难的，毕竟通常并不这样使用磁带。

另一类I/O设备是字符设备。字符设备以字符为单位发送或接收一个字符流，而不考虑任何块结构。字符设备是不可寻址的，也没有任何寻道操作。打印机、

设　备	数 据 率
键盘	10B/s
鼠标	100B/s
56K调制解调器	7KB/s
扫描仪	400 KB/s
数字便携式摄像机	3.5 MB/s
802.11g无线网络	6.75MB/s
52倍速CD-ROM	7.8MB/s
快速以太网	12.5 MB/s
袖珍闪存卡	40MB/s
火线（IEEE 1394）	50MB/s
USB 2.0	60MS/s
SONET OC-12网络	78MB/s
SCSI Ultra 2磁盘	80MB/s
千兆以太网	125MB/s
SATA磁盘驱动器	300MB/s
Ultrium磁带	320MB/s
PCI总线	528MB/s

图5-1 某些典型的设备、网络和总线的数据率

网络接口、鼠标（用作指点设备）、老鼠（用作心理学实验室实验），以及大多数与磁盘不同的设备都可看作是字符设备。

这种分类方法并不完美，有些设备就没有包括进去。例如，时钟既不是块可寻址的，也不产生或接收字符流。它所做的工作就是按照预先规定好的时间间隔产生中断。内存映射的显示器也不适用于此模型。但是，块设备和字符设备的模型具有足够的一般性，可以用作使处理I/O设备的某些操作系统软件具有设备无关性的基础。例如，文件系统只处理抽象的块设备，而把与设备相关的部分留给较低层的软件。

I/O设备在速度上覆盖了巨大的范围，要使软件在跨越这么多数量级的数据率下保证性能优良，给软件造成了相当大的压力。图5-1列出了某些常见设备的数据率，这些设备中大多数随着时间的推移而变得越来越快。

5.1.2 设备控制器

I/O设备一般由机械部件和电子部件两部分组成。通常可以将这两部分分开处理，以提供更加模块化和更加通用的设计。电子部件称作*设备控制器*（device controller）或*适配器*（adapter）。在个人计算机上，它经常以主板上的芯片的形式出现，或者以插入（PCI）扩展槽中的印刷电路板的形式出现。机械部件则是设备本身。这一安排如图1-6所示。

控制器卡上通常有一个连接器，通向设备本身的电缆可以插入到这个连接器中。很多控制器可以操作2个、4个甚至8个相同的设备。如果控制器和设备之间采用的是标准接口，无论是官方的ANSI、IEEE或ISO标准还是事实上的标准，各个公司都可以制造各种适合这个接口的控制器或设备。例如，许多公司都生产符合IDE、SATA、SCSI、USB或火线（IEEE 1394）接口的磁盘驱动器。

控制器与设备之间的接口通常是一个很低层次的接口。例如，磁盘可以按每个磁道10 000个扇区，每个扇区512字节进行格式化。然而，实际从驱动器出来的却是一个串行的位（比特）流，它以一个*前导符*（preamble）开始，接着是一个扇区中的4096位，最后是一个校验和，也称为*错误校正码*（Error-Correcting Code，ECC）。前导符是在对磁盘进行格式化时写上去的，它包括柱面数和扇区号、扇区大小以及类似的数据，此外还包含同步信息。

控制器的任务是把串行的位流转换为字节块，并进行必要的错误校正工作。字节块通常首先在控制器内部的一个缓冲区中按位进行组装，然后在对校验和进行校验并证明字节块没有错误后，再将它复制到主存中。

在同样低的层次上，监视器的控制器也是一个位串行设备。它从内存中读入包含待显示字符的字节，并产生用来调制CRT电子束的信号，以便将结果写到屏幕上。该控制器还产生信号使CRT电子束在完成一行扫描后做水平回扫，并且产生信号使CRT电子束在整个屏幕扫描结束后做垂直回扫。如果没有CRT控制器，那么操作系统程序员只能对显像管的模拟扫描直接进行编程。有了控制器，操作系统就可以用几个参数（这些参数包括每行的字符数或像素数、每屏的行数等）对其初始化，并让控制器实际驱动电子束。平板TFT显示器的工作原理与此不同，但是也同样复杂。

5.1.3 内存映射I/O

每个控制器有几个寄存器用来与CPU进行通信。通过写入这些寄存器，操作系统可以命令设备发送数据、接收数据、开启或关闭，或者执行某些其他操作。通过读取这些寄存器，操作系统可以了解设备的状态，是否准备好接收一个新的命令等。

除了这些控制寄存器以外，许多设备还有一个操作系统可以读写的数据缓冲区。例如，在屏幕上显示像素的常规方法是使用一个视频RAM，这一RAM基本上只是一个数据缓冲区，可供程序或操作系统写入数据。

于是，问题就出现了：CPU如何与设备的控制寄存器和数据缓冲区进行通信？存在两个可选的方法。在第一个方法中，每个控制寄存器被分配一个I/O*端口*（I/O port）号，这是一个8位或16位的整数。所有I/O端口形成I/O*端口空间*（I/O port space），并且受到保护使得普通的用户程序不能对其进行访问（只有操作系统可以访问）。使用一条特殊的I/O指令，例如

```
IN REG, PORT
```

CPU可以读取控制寄存器PORT的内容并将结果存入到CPU寄存器REG中。类似地，使用

```
OUT PORT, REG
```

CPU可以将REG的内容写入到控制寄存器中。大多数早期计算机，包括几乎所有大型主机，如IBM 360及其所有后续机型，都是以这种方式工作的。

在这一方案中，内存地址空间和I/O地址空间是不同的，如图5-2a所示。指令

```
IN R0, 4
```

和

```
MOV R0, 4
```

在这一设计中完全不同。前者读取I/O端口4的内容并将其存入R0，而后者则读取内存字4的内容并将其存入R0。因此，这些例子中的4引用的是不同且不相关的地址空间。

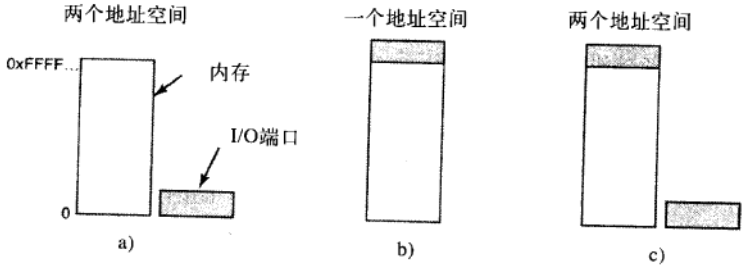


图5-2 a) 单独的I/O和内存空间；b) 内存映射I/O；c) 混合方案

第二个方法是PDP-11引入的，它将所有控制寄存器映射到内存空间中，如图5-2b所示。每个控制寄存器被分配惟一的一个内存地址，并且不会有内存被分配这一地址。这样的系统称为内存映射I/O（memory-mapped I/O）。通常分配给控制寄存器的地址位于地址空间的顶端。图5-2c所示是一种混合的方案，这一方案具有内存映射I/O的数据缓冲区，而控制寄存器则具有单独的I/O端口。Pentium处理器使用的就是这一体系结构。在IBM PC兼容机中，除了0到64K−1的I/O端口之外，640K到1M−1的地址保留给设备的数据缓冲区。

这些方案是怎样工作的？在各种情形下，当CPU想要读入一个字的时候，不论是从内存中读入还是从I/O端口中读入，它都要将需要的地址放到总线的地址线上，然后在总线的一条控制线上置起一个READ信号。还要用到第二条信号线来表明需要的是I/O空间还是内存空间。如果是内存空间，内存将响应请求。如果是I/O空间，I/O设备将响应请求。如果只有内存空间（如图5-2b所示的情形），那么每个内存模块和每个I/O设备都会将地址线和它所服务的地址范围进行比较，如果地址落在这一范围之内，它就会响应请求。因为绝对不会有地址既分配给内存又分配给I/O设备，所以不会存在歧义和冲突。

这两种寻址控制器的方案具有不同的优缺点。我们首先来看一看内存映射I/O的优点。第一，如果需要特殊的I/O指令读写设备控制寄存器，那么访问这些寄存器需要使用汇编代码，因为在C或C++中不存在执行IN或OUT指令的方法。调用这样的过程增加了控制I/O的开销。相反，对于内存映射I/O，设备控制寄存器只是内存中的变量，在C语言中可以和任何其他变量一样寻址。因此，对于内存映射I/O，I/O设备驱动程序可以完全用C语言编写。如果不使用内存映射I/O，就要用到某些汇编代码。

第二，对于内存映射I/O，不需要特殊的保护机制来阻止用户进程执行I/O操作。操作系统必须要做的全部事情只是避免把包含控制寄存器的那部分地址空间放入任何用户的虚拟地址空间之中。更为有利的是，如果每个设备在地址空间的不同页面上拥有自己的控制寄存器，操作系统只要简单地通过在其页表中包含期望的页面就可以让用户控制特定的设备而不是其他设备。这样的方案可以使不同的设备驱动程序放置在不同的地址空间中，不但可以减小内核的大小，而且可以防止驱动程序之间相互干扰。

第三，对于内存映射I/O，可以引用内存的每一条指令也可以引用控制寄存器。例如，如果存在一条指令TEST可以测试一个内存字是否为0，那么它也可以用来测试一个控制寄存器是否为0，控制寄存器为0可以作为信号，表明设备空闲并且可以接收一条新的命令。汇编语言代码可能是这样的：

```
LOOP:   TEST PORT_4     //检测端口4是否为0
        BEQ READY       //如果为0，转向READY
        BRANCH LOOP     //否则，继续测试
READY:
```

如果不是内存映射I/O，那么必须首先将控制寄存器读入CPU，然后再测试，这样就需要两条指令而不是一条。在上面给出的循环的情形中，就必须加上第四条指令，这样会稍稍降低检测空闲设备的响应度。

在计算机设计中，实际上任何事情都要涉及权衡，此处也不例外。内存映射I/O也有缺点。首先，现今大多数计算机都拥有某种形式的内存字高速缓存。对一个设备控制寄存器进行高速缓存可能是灾难性的。在存在高速缓存的情况下考虑上面给出的汇编代码循环。第一次引用PORT_4将导致它被高速缓存，随后的引用将只从高速缓存中取值并且不会再查询设备。之后当设备最终变为就绪时，软件将没有办法发现这一点。结果，循环将永远进行下去。

对内存映射I/O，为了避免这一情形，硬件必须针对每个页面具备选择性禁用高速缓存的能力。操作系统必须管理选择性高速缓存，所以这一特性为硬件和操作系统两者增添了额外的复杂性。

其次，如果只存在一个地址空间，那么所有的内存模块和所有的I/O设备都必须检查所有的内存引用，以便了解由谁做出响应。如果计算机具有单一总线，如图5-3a所示，那么让每个内存模块和I/O设备查看每个地址是简单易行的。

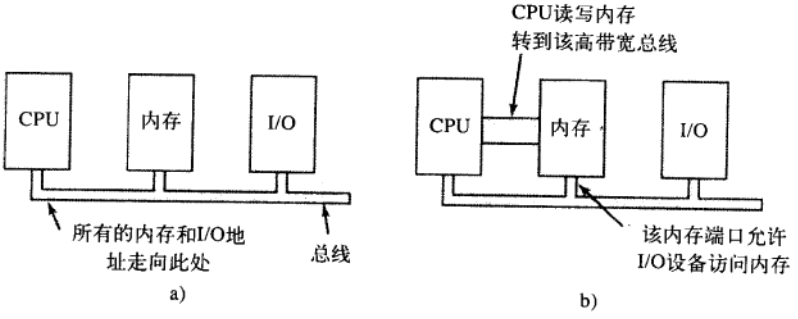


图5-3 a) 单总线体系结构；b) 双总线内存体系结构

然而，现代个人计算机的趋势是包含专用的高速内存总线，如图5-3b所示。顺便提一句，在大型机中也可以发现这一特性。装备这一总线是为了优化内存性能，而不是为了慢速的I/O设备而做的折中。Pentium系统甚至可以有多种总线（内存、PCI、SCSI、USB、ISA），如图1-12所示。

在内存映射的机器上具有单独的内存总线的麻烦是I/O设备没有办法查看内存地址，因为内存地址旁路到内存总线上，所以没有办法响应。此外，必须采取特殊的措施使内存映射I/O工作在具有多总线的系统上。一种可能的方法是首先将全部内存引用发送到内存，如果内存响应失败，CPU将尝试其他总线。这一设计是可以工作的，但是需要额外的硬件复杂性。

第二种可能的设计是在内存总线上放置一个探查设备，放过所有潜在地指向所关注的I/O设备的地址。此处的问题是，I/O设备可能无法以内存所能达到的速度处理请求。

第三种可能的设计是在PCI桥芯片中对地址进行过滤，这正是图1-12中Pentium结构上所使用的。该芯片中包含若干个在引导时预装载的范围寄存器。例如，640K到1M−1可能被标记为非内存范围。落在标记为非内存的那些范围之内的地址将被转发给PCI总线而不是内存。这一设计的缺点是需要在引导时判定哪些内存地址不是真正的内存地址。因而，每一设计都有支持它和反对它的论据，所以折中和权衡是不可避免的。

5.1.4 直接存储器存取

无论一个CPU是否具有内存映射I/O，它都需要寻址设备控制器以便与它们交换数据。CPU可以从I/O控制器每次请求一个字节的数据，但是这样做浪费CPU的时间，所以经常用到一种称为*直接存储器存取*（Direct Memory Access，DMA）的不同方案。只有硬件具有DMA控制器时操作系统才能使用

DMA，而大多数系统都有DMA控制器。有时DMA控制器集成到磁盘控制器和其他控制器之中，但是这样的设计要求每个设备有一个单独的DMA控制器。更加普遍的是，只有一个DMA控制器可利用（例如，在主板上），由它调控到多个设备的数据传送，而这些数据传送经常是同时发生的。

无论DMA控制器在物理上处于什么地方，它都能够独立于CPU而访问系统总线，如图5-4所示。它包含若干个可以被CPU读写的寄存器，其中包括一个内存地址寄存器、一个字节计数寄存器和一个或多个控制寄存器。控制寄存器指定要使用的I/O端口、传送方向（从I/O设备读或写到I/O设备）、传送单位（每次一个字节或每次一个字）以及在一次突发传送中要传送的字节数。

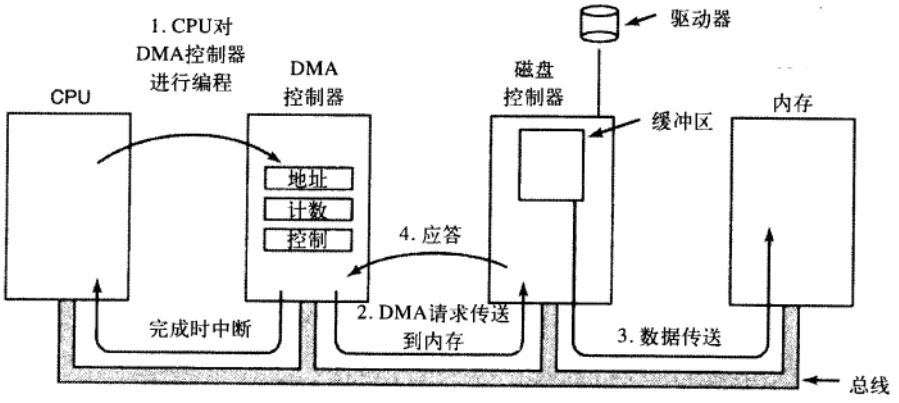


图5-4 DMA传送操作

为了解释DMA的工作原理，让我们首先看一下没有使用DMA时磁盘如何读。首先，控制器从磁盘驱动器串行地、一位一位地读一个块（一个或多个扇区），直到将整块信息放入控制器的内部缓冲区中。接着，它计算校验和，以保证没有读错误发生。然后控制器产生一个中断。当操作系统开始运行时，它重复地从控制器的缓冲区中一次一个字节或一个字地读取该块的信息，并将其存入内存中。

使用DMA时，过程是不同的。首先，CPU通过设置DMA控制器的寄存器对它进行编程，所以DMA控制器知道将什么数据传送到什么地方（图5-4中的第1步）。DMA控制器还要向磁盘控制器发出一个命令，通知它从磁盘读数据到其内部的缓冲区中，并且对校验和进行检验。如果磁盘控制器的缓冲区中的数据是有效的，那么DMA就可以开始了。

DMA控制器通过在总线上发出一个读请求到磁盘控制器而发起DMA传送（第2步）。这一读请求看起来与任何其他读请求是一样的，并且磁盘控制器并不知道或者并不关心它是来自CPU还是来自DMA控制器。一般情况下，要写的内存地址在总线的地址线上，所以当磁盘控制器从其内部缓冲区中读取下一个字的时候，它知道将该字写到什么地方。写到内存是另一个标准总线周期（第3步）。当写操作完成时，磁盘控制器在总线上发出一个应答信号到DMA控制器（第4步）。于是，DMA控制器步增要使用的内存地址，并且步减字节计数。如果字节计数仍然大于0，则重复第2步到第4步，直到字节计数到达0。此时，DMA控制器将中断CPU以便让CPU知道传送现在已经完成了。当操作系统开始工作时，用不着将磁盘块复制到内存中，因为它已经在内存中了。

DMA控制器在复杂性方面的区别相当大。最简单的DMA控制器每次处理一路传送，如上面所描述的。复杂一些的DMA控制器经过编程可以一次处理多路传送，这样的控制器内部具有多组寄存器，每一通道一组寄存器。CPU通过用与每路传送相关的参数装载每组寄存器而开始。每路传送必须使用不同的设备控制器。在图5-4中，传送每一个字之后，DMA控制器要决定下一次要为哪一设备提供服务。DMA控制器可能被设置为使用轮转算法，它也可能具有一个优先级规划设计，以便让某些设备受到比其他设备更多的照顾。假如存在一个明确的方法分辨应答信号，那么在同一时间就可以挂起对不同设备控制器的多个请求。出于这样的原因，经常将总线上不同的应答线用于每一个DMA通道。

许多总线能够以两种模式操作：每次一字模式和块模式。某些DMA控制器也能够以这两种模式操作。在前一个模式中，操作如上所述：DMA控制器请求传送一个字并且得到这个字。如果CPU也想使

用总线，它必须等待。这一机制称为周期窃取（cycle stealing），因为设备控制器偶尔偷偷溜入并且从CPU偷走一个临时的总线周期，从而轻微地延迟CPU。在块模式中，DMA控制器通知设备获得总线，发起一连串的传送，然后释放总线。这一操作形式称为突发模式（burst mode）。它比周期窃取效率更高，因为获得总线占用了时间，并且以一次总线获得的代价能够传送多个字。突发模式的缺点是，如果正在进行的是长时间突发传送，有可能将CPU和其他设备阻塞相当长的周期。

在我们一直讨论的模型——有时称为飞越模式（fly-by mode）中，DMA控制器通知设备控制器直接将数据传送到主存。某些DMA控制器使用的其他模式是让设备控制器将字发送给DMA控制器，DMA控制器然后发起第2个总线请求将该字写到它应该去的任何地方。采用这种方案，每传送一个字需要一个额外的总线周期，但是更加灵活，因为它可以执行设备到设备的复制甚至是内存到内存的复制（通过首先发起一个到内存的读，然后发起一个到不同内存地址的写）。

大多数DMA控制器使用物理内存地址进行传送。使用物理地址要求操作系统将预期的内存缓冲区的虚拟地址转换为物理地址，并且将该物理地址写入DMA控制器的地址寄存器中。在少数DMA控制器中使用的一个替代方案是将虚拟地址写入DMA控制器，然后DMA控制器必须使用MMU来完成虚拟地址到物理地址的转换。只有在MMU是内存的组成部分（有可能，但罕见）而不是CPU的组成部分的情况下，才可以将虚拟地址放到总线上。

我们在前面提到，在DMA可以开始之前，磁盘首先要将数据读入其内部的缓冲区中。你也许会产生疑问：为什么控制器从磁盘读取字节后不立即将其存储在主存中？换句话说，为什么需要一个内部缓冲区？有两个原因。首先，通过进行内部缓冲，磁盘控制器可以在开始传送之前检验校验和。如果校验和是错误的，那么将发出一个表明错误的信号并且不会进行传送。

第二个原因是，一旦磁盘传送开始工作，从磁盘读出的数据就是以固定速率到达的，而不论控制器是否准备好接收数据。如果控制器要将数据直接写到内存，则它必须为要传送的每个字取得系统总线的控制权。此时，若由于其他设备使用总线而导致总线忙（例如在突发模式中），则控制器只能等待。如果在前一个磁盘字还未被存储之前下一个磁盘字到达，控制器只能将它存放在某个地方。如果总线非常忙，控制器可能需要存储很多字，而且还要完成大量的管理工作。如果块被放入内部缓冲区，则在DMA启动前不需要使用总线，这样，控制器的设计就可以简化，因为对DMA到内存的传送没有严格的时间要求。（事实上，有些老式的控制器是直接存取内存的，其内部缓冲区设计得很小，但是当总线很忙时，一些传送有可能由于超载运行错误而被终止。）

并不是所有的计算机都使用DMA。反对的论据是主CPU通常要比DMA控制器快得多，做同样的工作可以更快（当限制因素不是I/O设备的速度时）。如果CPU没有其他工作要做，让（快速的）CPU等待（慢速的）DMA控制器完成工作是无意义的。此外，去除DMA控制器而让CPU用软件做所有的工作还可以节约金钱，这一点在低端（嵌入式）计算机上十分重要。

5.1.5 重温中断

我们在1.4.5节中简要介绍了中断，但是还有更多的内容要介绍。在一台典型的个人计算机系统中，中断结构如图5-5所示。在硬件层面，中断的工作如下所述。当一个I/O设备完成交给它的工作时，它就产生一个中断（假设操作系统已经开放中断），它是通过在分配给它的一条总线信号线上置起信号而产生中断的。该信号被主板上的中断控制器芯片检测到，由中断控制器芯片决定做什么。

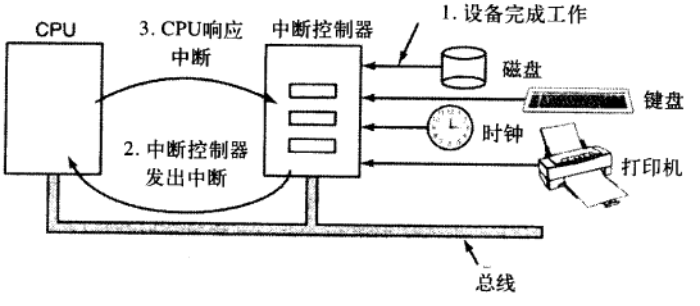


图5-5 中断是怎样发生的。设备与中断控制器之间的连接实际上使用的是总线上的中断线而不是专用连线

如果没有其他中断悬而未决，中断控制器将立刻对中断进行处理。如果有另一个中断正在处理中，或者另一个设备在总线上具有更高优先级的一条中断请求线上同时发出中断请求，该设备将暂时不被理睬。在这种情况下，该设备将继续在总线上置起中断信号，直到得到CPU的服务。

为了处理中断，中断控制器在地址线上放置一个数字表明哪个设备需要关注，并且置起一个中断CPU的信号。

中断信号导致CPU停止当前正在做的工作并且开始做其他的事情。地址线上的数字被用做指向一个称为中断向量（interrupt vector）的表格的索引，以便读取一个新的程序计数器。这一程序计数器指向相应的中断服务过程的开始。一般情况下，陷阱和中断从这一点上看使用相同的机制，并且常常共享相同的中断向量。中断向量的位置可以硬布线到机器中，也可以在内存中的任何地方通过一个CPU寄存器（由操作系统装载）指向其起点。

中断服务过程开始运行后，它立刻通过将一个确定的值写到中断控制器的某个I/O端口来对中断做出应答。这一应答告诉中断控制器可以自由地发出另一个中断。通过让CPU延迟这一应答直到它准备好处理下一个中断，就可以避免与多个几乎同时发生的中断相牵涉的竞争状态。说句题外的话，某些（老式的）计算机没有集中的中断控制器，所以每个设备控制器请求自己的中断。

在开始服务程序之前，硬件总是要保存一定的信息。哪些信息要保存以及将其保存到什么地方，不同的CPU之间存在巨大的差别。作为最低限度，必须保存程序计数器，这样被中断的进程才能够重新开始。在另一个极端，所有可见的寄存器和很多内部寄存器或许也要保存。

将这些信息保存到什么地方是一个问题。一种选择是将其放入内部寄存器中，在需要时操作系统可以读出这些内部寄存器。这一方法的问题是，中断控制器之后无法得到应答，直到所有可能的相关信息被读出，以免第二个中断重写内部寄存器保存状态。这一策略在中断被禁止时将导致长时间的死机，并且可能丢失中断和丢失数据。

因此，大多数CPU在堆栈中保存信息。然而，这种方法也有问题。首先，使用谁的堆栈？如果使用当前堆栈，则它很可能是用户进程的堆栈。堆栈指针甚至可能不是合法的，这样当硬件试图在它所指的地址处写某些字时，将导致致命错误。此外，它可能指向一个页面的末端。若干次内存写之后，页面边界可能被超出并且产生一个页面故障。在硬件中断处理期间如果发生页面故障将引起更大的问题：在何处保存状态以处理页面故障？

如果使用内核堆栈，将存在更多的堆栈指针是合法的并且指向一个固定的页面的机会。然而，切换到核心态可能要求改变MMU上下文，并且可能使高速缓存和TLB的大部分或全部失效。静态地或动态地重新装载所有这些东西将增加处理一个中断的时间，因而浪费CPU的时间。

精确中断和不精确中断

另一个问题是由下面这样的事实引起的：现代CPU大量地采用流水线并且有时还采用超标量（内部并行）。在老式的系统中，每条指令完成执行之后，微程序或硬件将检查是否存在悬而未决的中断。如果存在，那么程序计数器和PSW将被压入堆栈中而中断序列将开始。在中断处理程序运行之后，相反的过程将会发生，旧的PSW和程序计数器将从堆栈中弹出并且先前的进程继续运行。

这一模型使用了隐含的假设，这就是如果一个中断正好在某一指令之后发生，那么这条指令前的所有指令（包括这条指令）都完整地执行过了，而这条指令后的指令一条也没有执行。在老式的机器上，这一假设总是正确的，而在现代计算机上，这一假设则未必是正确的。

首先，考虑图1-6a的流水线模型。在流水线满的时候（通常的情形），如果出现一个中断，那么会发生什么情况？许多指令正处于各种不同的执行阶段，当中断出现时，程序计数器的值可能无法正确地反映已经执行过的指令和尚未执行的指令之间的边界。事实上，许多指令可能部分地执行了，不同的指令完成的程度或多或少。在这种情况下，程序计数器更有可能反映的是将要被取出并压入流水线的下一条指令的地址，而不是刚刚被执行单元处理过的指令的地址。

在如图1-7b所示的超标量计算机上，事情更加糟糕。指令可能分解成微操作，而微操作有可能乱序执行，这取决于内部资源（如功能单元和寄存器）的可用性。当中断发生时，某些很久以前启动的指令可能还没开始执行，而其他最近启动的指令可能几乎要完成了。当中断信号出现时，可能存在许多指令

处于不同的完成状态，它们与程序计数器之间没有什么关系。

将机器留在一个明确状态的中断称为*精确中断*（precise interrupt）（Walker和Cragon，1995）。精确中断具有4个特性：

1) PC（程序计数器）保存在一个已知的地方。

2) PC所指向的指令之前的所有指令已经完全执行。

3) PC所指向的指令之后的所有指令都没有执行。

4) PC所指向的指令的执行状态是已知的。

注意，对于PC所指向的指令之后的那些指令来说，此处并没有禁止它们开始执行，而只是要求在中断发生之前必须撤销它们对寄存器或内存所做的任何修改。PC所指向的指令有可能已经执行了，也有可能还没有执行，然而，必须清楚适用的是哪种情况。通常，如果中断是一个I/O中断，那么指令就会还没有开始执行。然而，如果中断实际上是一个陷阱或者页面故障，那么PC一般指向导致错误的指令，所以它以后可以重新开始执行。图5-6a所示的情形描述了精确中断。程序计数器（316）之前的所有指令都已经完成了，而它之后的指令都还没有启动（或者已经回退以撤销它们的作用）。

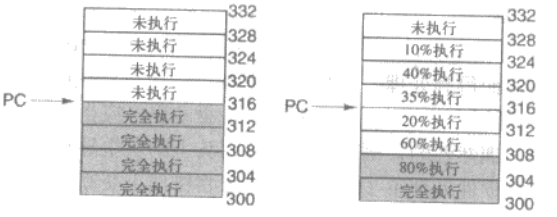


图5-6 a) 精确中断；b) 不精确中断

不满足这些要求的中断称为*不精确中断*（imprecise interrupt），不精确中断使操作系统编写者过得极为不愉快，现在操作系统编写者必须断定已经发生了什么以及还要发生什么。图5-6b描述了不精确中断，其中邻近程序计数器的不同指令处于不同的完成状态，老的指令不一定比新的指令完成得更多。具有不精确中断的机器通常将大量的内部状态“吐出”到堆栈中，从而使操作系统有可能判断出正在发生什么事情。重新启动机器所必需的代码通常极其复杂。此外，在每次中断发生时将大量的信息保存在内存中使得中断响应十分缓慢，而恢复则更加糟糕。这就导致具有讽刺意味的情形：由于缓慢的中断使得非常快速的超标量CPU有时并不适合实时工作。

有些计算机设计成某些种类的中断和陷阱是精确的，而其他的不是。例如，可以让I/O中断是精确的，而归因于致命编程错误的陷阱是不精确的，由于在被0除之后不需要尝试重新开始运行的进程，所以这样做也不错。有些计算机具有一个位，可以设置它强迫所有的中断都是精确的。设置这一位的不利之处是，它强迫CPU仔细地将正在做的一切事情记入日志并且维护寄存器的影子副本，这样才能够在任意时刻生成精确中断。所有这些开销都对性能具有较大的影响。

某些超标量计算机（例如Pentium系列）具有精确中断，从而使老的软件正确工作。为精确中断付出的代价是CPU内部极其复杂的中断逻辑，以便确保当中断控制器发出信号想要导致一个中断时，允许直到某一点之前的所有指令完成而不允许这一点之后的指令对机器状态产生任何重要的影响。此处付出的代价不是在时间上，而是在芯片面积和设计复杂性上。如果不是因为向后兼容的目的而要求精确中断的话，这一芯片面积就可以用于更大的片上高速缓存，从而使CPU的速度更快。另一方面，不精确中断使得操作系统更为复杂而且运行得更加缓慢，所以断定哪一种方法更好是十分困难的。

5.2 I/O软件原理

在讨论了I/O硬件之后，下面我们来看一看I/O软件。首先我们将看一看I/O软件的目标，然后从操作系统的观点来看一看I/O实现的不同方法。

5.2.1 I/O软件的目标

在设计I/O软件时一个关键的概念是*设备独立性*（device independence）。它的意思是应该能够编写出这样的程序：它可以访问任意I/O设备而无需事先指定设备。例如，读取一个文件作为输入的程序应该能够在硬盘、CD-ROM、DVD或者USB盘上读取文件，无需为每一种不同的设备修改程序。类似地，用户应该能够键入这样一条命令

```
sort <input> output
```

并且无论输入来自任意类型的存储盘或者键盘，输出送往任意类型的存储盘或者屏幕，上述命令都可以工作。尽管这些设备实际上差别很大，需要非常不同的命令序列来读或写，但这一事实所带来的问题将由操作系统负责处理。

与设备独立性密切相关的是统一命名（uniform naming）这一目标。一个文件或一个设备的名字应该是一个简单的字符串或一个整数，它不应依赖于设备。在UNIX系统中，所有存储盘都能以任意方式集成到文件系统层次结构中，因此，用户不必知道哪个名字对应于哪台设备。例如，一个USB盘可以安装（mount）到目录/usr/ast/backup下，这样复制一个文件到/usr/ast/backup/monday就是将文件复制到USB盘上。用这种方法，所有文件和设备都采用相同的方式——路径名进行寻址。

I/O软件的另一个重要问题是错误处理（error handling）。一般来说，错误应该尽可能地在接近硬件的层面得到处理。当控制器发现了一个读错误时，如果它能够处理那么就应该自己设法纠正这一错误。如果控制器处理不了，那么设备驱动程序应当予以处理，可能只需重读一次这块数据就正确了。很多错误是偶然性的，例如，磁盘读写头上的灰尘导致读写错误时，重复该操作，错误经常就会消失。只有在低层软件处理不了的情况下，才将错误上交高层处理。在许多情况下，错误恢复可以在低层透明地得到解决，而高层软件甚至不知道存在这一错误。

另一个关键问题是同步（synchronous）（即阻塞）和异步（asynchronous）（即中断驱动）传输。大多数物理I/O是异步的——CPU启动传输后便转去做其他工作，直到中断发生。如果I/O操作是阻塞的，那么用户程序就更加容易编写——在read系统调用之后，程序将自动被挂起，直到缓冲区中的数据准备好。正是操作系统使实际上是中断驱动的操作变为在用户程序看来是阻塞式的操作。

I/O软件的另一个问题是缓冲（buffering）。数据离开一个设备之后通常并不能直接存放到其最终的目的地。例如，从网络上进来一个数据包时，直到将该数据包存放在某个地方并对其进行检查，操作系统才知道要将其置于何处。此外，某些设备具有严格的实时约束（例如，数字音频设备），所以数据必须预先放置到输出缓冲区之中，从而消除缓冲区填满速率和缓冲区清空速率之间的相互影响，以避免缓冲区欠载。缓冲涉及大量的复制工作，并且经常对I/O性能有重大影响。

此处我们将提到的最后一个概念是共享设备和独占设备的问题。有些I/O设备（如磁盘）能够同时让多个用户使用。多个用户同时在同一磁盘上打开文件不会引起什么问题。其他设备（如磁带机）则必须由单个用户独占使用，直到该用户使用完，另一个用户才能拥有该磁带机。让两个或更多的用户随机地将交叉混杂的数据块写入相同的磁带是注定不能工作的。独占（非共享）设备的引入也带来了各种各样的问题，如死锁。同样，操作系统必须能够处理共享设备和独占设备以避免问题发生。

5.2.2 程序控制I/O

I/O可以以三种根本不同的方式实现。在本小节中我们将介绍第一种（程序控制I/O），在后面两小节中我们将研究另外两种（中断驱动I/O和使用DMA的I/O）。I/O的最简单形式是让CPU做全部工作，这一方法称为程序控制I/O（programmed I/O）。

借助于例子来说明程序控制I/O是最简单的。考虑一个用户进程，该进程想在打印机上打印8个字符的字符串“ABCDEFGH”。它首先要在用户空间的一个缓冲区中组装字符串，如图5-7a所示。

然后，用户进程通过发出系统调用打开打印机来获得打印机以便进行写操作。如果打印机当前被另一个进程占用，该系统调用将失败并返回一个错误代码，或者将阻塞直到打印机可用，具体情况取决于操作系统和调用参数。一旦拥有打印机，用户进程就发出一个系统调用通知操作系统在打印机上打印字符串。

然后，操作系统（通常）将字符串缓冲区复制到内核空间中的一个数组（如p）中，在这里访问更加容易（因为内核可能必须修改内存映射才能到达用户空间）。然后操作系统要查看打印机当前是否可用。如果不可用，就要等待直到它可用。一旦打印机可用，操作系统就复制第一个字符到打印机的数据寄存器中，在这个例子中使用了内存映射I/O。这一操作将激活打印机。字符也许还不会出现在打印机上，因为某些打印机在打印任何东西之前要先缓冲一行或一页。然而，在图5-7b中，我们看到第一个字符已经打印出来，并且系统已经将“B”标记为下一个待打印的字符。

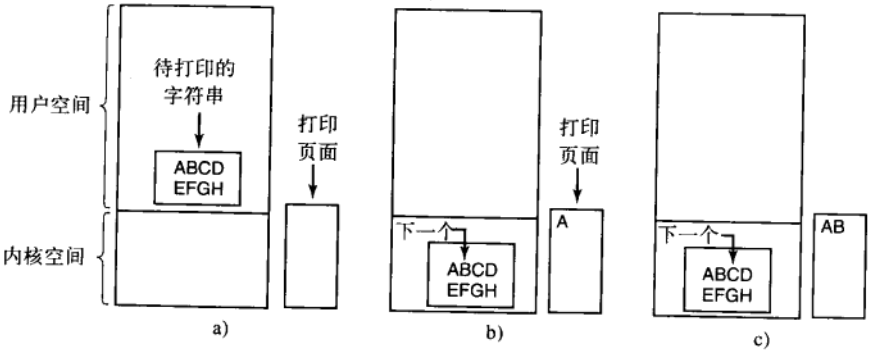


图5-7 打印一个字符串的步骤

一旦将第一个字符复制到打印机，操作系统就要查看打印机是否就绪准备接收另一个字符。一般而言，打印机都有第二个寄存器，用于表明其状态。将字符写到数据寄存器的操作将导致状态变为非就绪。当打印机控制器处理完当前字符时，它就通过在其状态寄存器中设置某一位或者将某个值放到状态寄存器中来表示其可用性。

这时，操作系统将等待打印机状态再次变为就绪。打印机就绪事件发生时，操作系统就打印下一个字符，如图5-7c所示。这一循环继续进行，直到整个字符串打印完。然后，控制返回到用户进程。

操作系统相继采取的操作总结在图5-8中。首先，数据被复制到内核空间。然后，操作系统进入一个密闭的循环，一次输出一个字符。在该图中，清楚地说明了程序控制I/O的最根本的方面，这就是输出一个字符之后，CPU要不断地查询设备以了解它是否就绪准备接收另一个字符。这一行为经常称为*轮询*（polling）或*忙等待*（busy waiting）。

```
copy_from_user(buffer, p, count);                /* p是内核缓冲区 */
for (i = 0; i < count; i++) {                    /* 对每个字符循环 */
    while (*printer_status_reg != READY) ;       /* 循环直到就绪 */
    *printer_data_register = p[i];               /* 输出一个字符 */
}
return_to_user();
```

图5-8 使用程序控制I/O将一个字符串写到打印机

程序控制I/O十分简单但是有缺点，即直到全部I/O完成之前要占用CPU的全部时间。如果“打印”一个字符的时间非常短（因为打印机所做的全部事情就是将新的字符复制到一个内部缓冲区中），那么忙等待还是不错的。此外，在嵌入式系统中，CPU没有其他事情要做，忙等待也是合理的。然而，在更加复杂的系统中，CPU有其他工作要做，忙等待将是低效的，需要更好的I/O方法。

5.2.3 中断驱动I/O

现在我们考虑在不缓冲字符而是在每个字符到来时便打印的打印机上进行打印的情形。如果打印机每秒可以打印100个字符，那么打印每个字符将花费10ms。这意味着，当每个字符写到打印机的数据寄存器中之后，CPU将有10ms搁置在无价值的循环中，等待允许输出下一个字符。这10ms时间足以进行一次上下文切换并且运行其他进程，否则就浪费了。

这种允许CPU在等待打印机变为就绪的同时做某些其他事情的方式就是使用中断。当打印字符串的系统调用被发出时，如我们前面所介绍的，字符串缓冲区被复制到内核空间，并且一旦打印机准备好接收一个字符时就将第一个字符复制到打印机中。这时，CPU要调用调度程序，并且某个其他进程将运行。请求打印字符串的进程将被阻塞，直到整个字符串打印完。系统调用所做的工作如图5-9a所示。

当打印机将字符打印完并且准备好接收下一个字符时，它将产生一个中断。这一中断将停止当前进程并且保存其状态。然后，打印机中断服务过程将运行。图5-9b所示为打印机中断服务过程的一个粗略的版本。如果没有更多的字符要打印，中断处理程序将采取某个操作将用户进程解除阻塞。否则，它将

输出下一个字符，应答中断，并且返回到中断之前正在运行的进程，该进程将从其停止的地方继续运行。

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

b)

图5-9 使用中断驱动I/O将一个字符串写到打印机：a) 当打印系统调用被发出时执行的代码；b) 打印机的中断服务过程

5.2.4 使用DMA的I/O

中断驱动I/O的一个明显缺点是中断发生在每个字符上。中断要花费时间，所以这一方法将浪费一定数量的CPU时间。这一问题的一种解决方法是使用DMA。此处的思路是让DMA控制器一次给打印机提供一个字符，而不必打扰CPU。本质上，DMA是程序控制I/O，只是由DMA控制器而不是主CPU做全部工作。这一策略需要特殊的硬件（DMA控制器），但是使CPU获得自由从而可以在I/O期间做其他工作。使用DMA的代码概要如图5-10所示。

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

b)

图5-10 使用DMA打印一个字符串：a) 当打印系统调用被发出时执行的代码；b) 中断服务过程

DMA重大的成功是将中断的次数从打印每个字符一次减少到打印每个缓冲区一次。如果有许多字符并且中断十分缓慢，那么采用DMA可能是重要的改进。另一方面，DMA控制器通常比主CPU要慢很多。如果DMA控制器不能以全速驱动设备，或者CPU在等待DMA中断的同时没有其他事情要做，那么采用中断驱动I/O甚至采用程序控制I/O也许更好。

5.3 I/O软件层次

I/O软件通常组织成四个层次，如图5-11所示。每一层具有一个要执行的定义明确的功能和一个的定义明确的与邻近层次的接口。功能与接口随系统的不同而不同，所以下面的讨论并不针对一种特定的机器。我们将从底层开始讨论每一层。

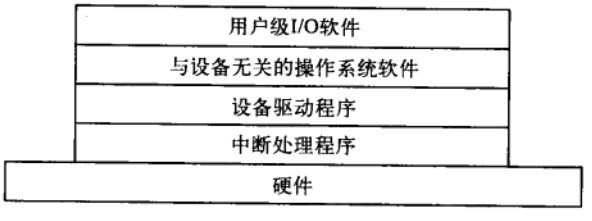


图5-11 I/O软件系统的层次

5.3.1 中断处理程序

虽然程序控制I/O偶尔是有益的，但是对于大多数I/O而言，中断是令人不愉快的事情并且无法避免。应当将其深深地隐藏在操作系统内部，以便系统的其他部分尽量不与它发生联系。隐藏它们的最好办法是将启动一个I/O操作的驱动程序阻塞起来，直到I/O操作完成且产生一个中断。驱动程序阻塞自己的手段有：在一个信号量上执行down操作、在一个条件变量上执行wait操作、在一个消息上执行receive操作或者某些类似的操作。

当中断发生时，中断处理程序将做它必须要做的全部工作以便对中断进行处理。然后，它可以将启动中断的驱动程序解除阻塞。在一些情形中，它只是在一个信号量上执行up操作；其他情形中，是对管程中的条件变量执行signal操作；还有一些情形中，是向被阻塞的驱动程序发一个消息。在所有这些情形中，中断最终的结果是使先前被阻塞的驱动程序现在能够继续运行。如果驱动程序构造为内核进程，

具有它们自己的状态、堆栈和程序计数器，那么这一模型运转得最好。

当然，现实没有如此简单。对一个中断进行处理并不只是简单地捕获中断，在某个信号量上执行up操作，然后执行一条IRET指令从中断返回到先前的进程。对操作系统而言，还涉及更多的工作。我们将按一系列步骤给出这一工作的轮廓，这些步骤是硬件中断完成之后必须在软件中执行的。应该注意的是，细节是非常依赖于系统的，所以下面列出的某些步骤在一个特定的机器上可能是不必要的，而没有列出的步骤可能是必需的。此外，确实发生的步骤在某些机器上也可能有不同的顺序。

1) 保存没有被中断硬件保存的所有寄存器（包括PSW）。

2) 为中断服务过程设置上下文，可能包括设置TLB、MMU和页表。

3) 为中断服务过程设置堆栈。

4) 应答中断控制器，如果不存在集中的中断控制器，则再次开放中断。

5) 将寄存器从它们被保存的地方（可能是某个堆栈）复制到进程表中。

6) 运行中断服务过程，从发出中断的设备控制器的寄存器中提取信息。

7) 选择下一次运行哪个进程，如果中断导致某个被阻塞的高优先级进程变为就绪，则可能选择它现在就运行。

8) 为下一次要运行的进程设置MMU上下文，也许还需要设置某个TLB。

9) 装入新进程的寄存器，包括其PSW。

10) 开始运行新进程。

由此可见，中断处理远不是无足轻重的小事。它要花费相当多的CPU指令，特别是在存在虚拟内存并且必须设置页表或者必须保存MMU状态（例如*R*和*M*位）的机器上。在某些机器上，当在用户态与核心态之间切换时，可能还需要管理TLB和CPU高速缓存，这就要花费额外的机器周期。

5.3.2 设备驱动程序

在本章前面的内容中，我们介绍了设备控制器所做的工作。我们注意到每一个控制器都设有某些设备寄存器用来向设备发出命令，或者设有某些设备寄存器用来读出设备的状态，或者设有这两种设备寄存器。设备寄存器的数量和命令的性质在不同设备之间有着根本性的不同。例如，鼠标驱动程序必须从鼠标接收信息，以识别鼠标移动了多远的距离以及当前哪一个键被按下。相反，磁盘驱动程序可能必须要了解扇区、磁道、柱面、磁头、磁盘臂移动、电机驱动器、磁头定位时间以及所有其他保证磁盘正常工作的机制。显然，这些驱动程序是有很大区别的。

因而，每个连接到计算机上的I/O设备都需要某些设备特定的代码来对其进行控制。这样的代码称为*设备驱动程序*（device driver），它一般由设备的制造商编写并随同设备一起交付。因为每一个操作系统都需要自己的驱动程序，所以设备制造商通常要为若干流行的操作系统提供驱动程序。

每个设备驱动程序通常处理一种类型的设备，或者至多处理一类紧密相关的设备。例如，SCSI磁盘驱动程序通常可以处理不同大小和不同速度的多个SCSI磁盘，或许还可以处理SCSI CD-ROM。而另一方面，鼠标和游戏操纵杆是如此的不同，以至于它们通常需要不同的驱动程序。然而，对于一个设备驱动程序控制多个不相关的设备并不存在技术上的限制，只是这样做并不是一个好主意。

为了访问设备的硬件（意味着访问设备控制器的寄存器），设备驱动程序通常必须是操作系统内核的一部分，至少对目前的体系结构是如此。实际上，有可能构造运行在用户空间的驱动程序，使用系统调用来读写设备寄存器。这一设计使内核与驱动程序相隔离，并且使驱动程序之间相互隔离，这样做可以消除系统崩溃的一个主要源头——有问题的驱动程序以这样或那样的方式干扰内核。对于建立高度可靠的系统而言，这绝对是正确的方向。MINIX 3就是一个这样的系统，其中设备驱动程序就作为用户进程而运行。然而，因为大多数其他桌面操作系统要求驱动程序运行在内核中，所以我们在这里只考虑这样的模型。

因为操作系统的设计者知道由外人编写的驱动程序代码片断将被安装在操作系统的内部，所以需要有一个体系结构来允许这样的安装。这意味着要有一个定义明确的模型，规定驱动程序做什么事情以及如何与操作系统的其余部分相互作用。设备驱动程序通常位于操作系统其余部分的下面，如图5-12所示。

操作系统通常将驱动程序归类于少数的类别之一。最为通用的类别是块设备（block device）和字符设备（character device）。块设备（例如磁盘）包含多个可以独立寻址的数据块，字符设备（例如键盘和打印机）则生成或接收字符流。

大多数操作系统都定义了一个所有块设备都必须支持的标准接口，并且还定义了另一个所有字符设备都必须支持的标准接口。这些接口由许多过程组成，操作系统的其余部分可以调用它们让驱动程序工作。典型的过程是那些读一个数据块（对块设备而言）或者写一个字符串（对字符设备而言）的过程。

在某些系统中，操作系统是一个二进制程序，包含需要编译到其内部的所有驱动程序。这一方案多年以来对UNIX系统而言是标准规范，因为UNIX系统主要由计算中心运行，I/O设备几乎不发生变化。如果添加了一个新设备，系统管理员只需重新编译内核，将新的驱动程序增加到新的二进制程序中。

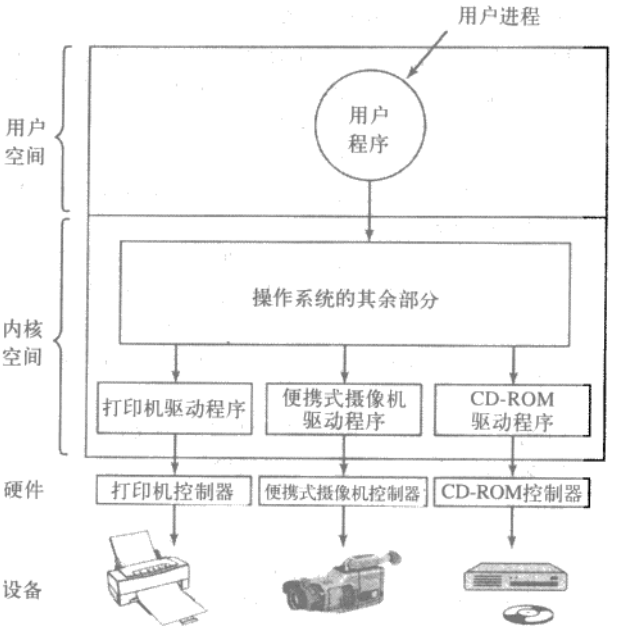


图5-12 设备驱动程序的逻辑定位。实际上，驱动程序和设备控制器之间的所有通信都通过总线

随着个人计算机的出现，这一模型不再起作用，因为个人计算机有太多种类的I/O设备。即便拥有源代码或目标模块，也只有很少的用户有能力重新编译和重新连接内核，何况他们并不总是拥有源代码或目标模块。为此，从MS-DOS开始，操作系统转向驱动程序在执行期间动态地装载到系统中的另一个模型。不同的操作系统以不同的方式处理驱动程序的装载工作。

设备驱动程序具有若干功能。最明显的功能是接收来自其上方与设备无关的软件所发出的抽象的读写请求，并且目睹这些请求被执行。除此之外，还有一些其他的功能必须执行。例如，如果需要的话，驱动程序必须对设备进行初始化。它可能还需要对电源需求和日志事件进行管理。

许多设备驱动程序具有相似的一般结构。典型的驱动程序在启动时要检查输入参数，检查输入参数的目的是搞清它们是否是有效的，如果不是，则返回一个错误。如果输入参数是有效的，则可能需要进行从抽象事项到具体事项的转换。对磁盘驱动程序来说，这可能意味着将一个线性的磁盘块号转换成磁盘几何布局的磁头、磁道、扇区和柱面号。

接着，驱动程序可能要检查设备当前是否在使用。如果在使用，请求将被排入队列以备稍后处理。如果设备是空闲的，驱动程序将检查硬件状态以了解请求现在是否能够得到处理。在传输能够开始之前，可能需要接通设备或者启动马达。一旦设备接通并就绪，实际的控制就可以开始了。

控制设备意味着向设备发出一系列命令。依据控制设备必须要做的工作，驱动程序处在确定命令序列的地方。驱动程序在获知哪些命令将要发出之后，它就开始将它们写入控制器的设备寄存器。驱动程序在把每个命令写到控制器之后，它可能必须进行检测以了解控制器是否已经接收命令并且准备好接收下一个命令。这一序列继续进行，直到所有命令被发出。对于某些控制器，可以为其提供一个在内存中的命令链表，并且告诉它自己去读取并处理所有命令而不需要操作系统提供进一步帮助。

命令发出之后，会牵涉两种情形之一。在多数情况下，设备驱动程序必须等待，直到控制器为其做某些事情，所以驱动程序将阻塞其自身直到中断到来解除阻塞。然而，在另外一些情况下，操作可以无延迟地完成，所以驱动程序不需要阻塞。在字符模式下滚动屏幕只需要写少许字节到控制器的寄存器中，由于不需要机械运动，所以整个操作可以在几纳秒内完成，这便是后一种情形的例子。

在前一种情况下，阻塞的驱动程序可以被中断唤醒。在后一种情况下，驱动程序根本就不会休眠。无论是哪一种情况，操作完成之后驱动程序都必须检查错误。如果一切顺利，驱动程序可能要将数据（例如刚刚读出的一个磁盘块）传送给与设备无关的软件。最后，它向调用者返回一些用于错误报告的状态信息。如果还有其他未完成的请求在排队，则选择一个启动执行。如果队列中没有未完成的请求，则该驱动程序将阻塞以等待下一个请求。

这一简单的模型只是现实的粗略近似，许多因素使相关的代码比这要复杂得多。首先，当一个驱动程序正在运行时，某个I/O设备可能会完成操作，这样就会中断驱动程序。中断可能会导致一个设备驱动程序运行，事实上，它可能导致当前驱动程序运行。例如，当网络驱动程序正在处理一个到来的数据包时，另一个数据包可能到来。因此，驱动程序必须是重入的（reentrant），这意味着一个正在运行的驱动程序必须预料到在第一次调用完成之前第二次被调用。

在一个热可插拔的系统中，设备可以在计算机运行时添加或删除。因此，当一个驱动程序正忙于从某设备读数据时，系统可能会通知它用户突然将设备从系统中删除了。在这样的情况下，不但当前I/O传送必须中止并且不能破坏任何核心数据结构，而且任何对这个现已消失的设备的悬而未决的请求都必须适当地从系统中删除，同时还要为它们的调用者提供这一坏消息。此外，未预料到的新设备的添加可能导致内核重新配置资源（例如中断请求线），从驱动程序中撤除旧资源，并且在适当位置填入新资源。

驱动程序不允许进行系统调用，但是它们经常需要与内核的其余部分进行交互。对某些内核过程的调用通常是允许的。例如，通常需要调用内核过程来分配和释放硬接线的内存页面作为缓冲区。还可能需要其他有用的调用来管理MMU、定时器、DMA控制器、中断控制器等。

5.3.3 与设备无关的I/O软件

虽然I/O软件中有一些是设备特定的，但是其他部分I/O软件是与设备无关的。设备驱动程序和与设备无关的软件之间的确切界限依赖于具体系统（和设备），因为对于一些本来应按照与设备无关方式实现的功能，出于效率和其他原因，实际上是由驱动程序来实现的。图5-13所示的功能典型地由与设备无关的软件实现。

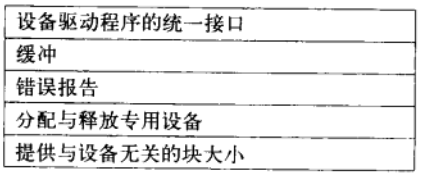


图5-13 与设备无关的I/O软件的功能

与设备无关的软件的基本功能是执行对所有设备公共的I/O功能，并且向用户层软件提供一个统一的接口。下面我们将详细介绍上述问题。

1. 设备驱动程序的统一接口

操作系统的一个主要问题是如何使所有I/O设备和驱动程序看起来或多或少是相同的。如果磁盘、打印机、键盘等接口方式都不相同，那么每次在一个新设备出现时，都必须为新设备修改操作系统。必须为每个新设备修改操作系统决不是一个好主意。

设备驱动程序与操作系统其余部分之间的接口是这一问题的一个方面。图5-14a所示为这样一种情形：每个设备驱动程序有不同的与操作系统的接口。这意味着，可供系统调用的驱动程序函数随驱动程序的不同而不同。这可能还意味着，驱动程序所需要的内核函数也是随驱动程序的不同而不同的。综合起来看，这意味着为每个新的驱动程序提供接口都需要大量全新的编程工作。

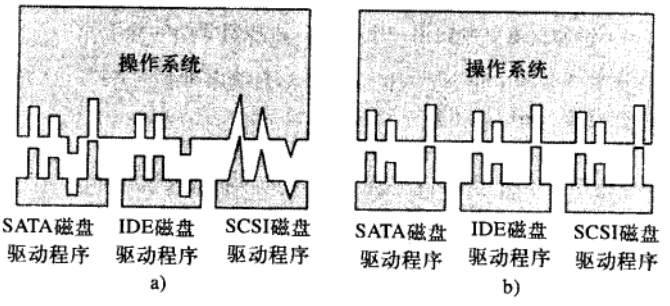


图5-14 a) 没有标准的驱动程序接口；b) 具有标准的驱动程序接口

相反，图5-14b所示为一种不同的设计，在这种设计中所有驱动程序具有相同的接口。这样一来，倘若符合驱动程序接口，那么添加一个新的驱动程序就变得容易多了。这还意味着驱动程序的编写人员知道驱动程序的接口应该是什么样子的。实际上，虽然并非所有的设备都是绝对一样的，但是通常只存在少数设备类型，而它们的确大体上是相同的。

这种设计的工作方式如下。对于每一种设备类型，例如磁盘或打印机，操作系统定义一组驱动程序必须支持的函数。对于磁盘而言，这些函数自然地包含读和写，除此之外还包含开启和关闭电源、格式化以及其他与磁盘有关的事情。驱动程序通常包含一张表格，这张表格具有针对这些函数指向驱动程序自身的指针。当驱动程序装载时，操作系统记录下这张函数指针表的地址，所以当操作系统需要调用一个函数时，它可以通过这张表格发出间接调用。这张函数指针表定义了驱动程序与操作系统其余部分之间的接口。给定类型（磁盘、打印机等）的所有设备都必须服从这一要求。

如何给I/O设备命名是统一接口问题的另一个方面。与设备无关的软件要负责把符号化的设备名映射到适当的驱动程序上。例如，在UNIX系统中，像/dev/disk0这样的设备名惟一确定了一个特殊文件的i节点，这个i节点包含了*主设备号*（major device number），主设备号用于定位相应的驱动程序。i节点还包含*次设备号*（minor device number），次设备号作为参数传递给驱动程序，用来确定要读或写的具体单元。所有设备都具有主设备号和次设备号，并且所有驱动程序都是通过使用主设备号来选择驱动程序而得到访问。

与设备命名密切相关的是设备保护。系统如何防止无权访问设备的用户访问设备呢？在UNIX和Windows中，设备是作为命名对象出现在文件系统中的，这意味着针对文件的常规的保护规则也适用于I/O设备。系统管理员可以为每一个设备设置适当的访问权限。

2. *缓冲*

无论对于块设备还是对于字符设备，由于种种原因，缓冲也是一个重要的问题。作为例子，我们考虑一个想要从调制解调器读入数据的进程。让用户进程执行read系统调用并阻塞自己以等待字符的到来，这是对到来的字符进行处理的一种可能的策略。每个字符的到来都将引起中断，中断服务过程负责将字符递交给用户进程并且将其解除阻塞。用户进程把字符放到某个地方之后可以对另一个字符执行读操作并且再次阻塞。这一模型如图5-15a所示。

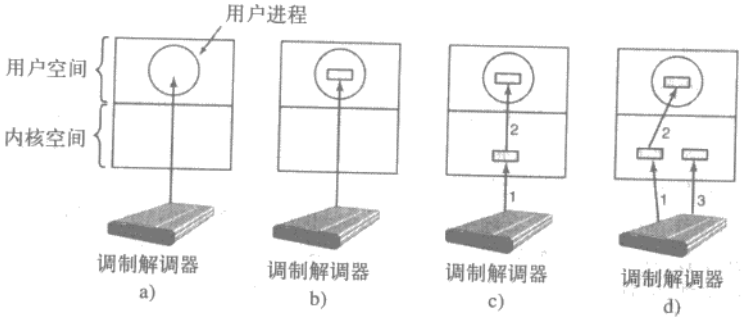


图5-15 a) 无缓冲的输入；b) 用户空间中的缓冲；c) 内核空间中的缓冲接着复制到用户空间；d) 内核空间中的双缓冲

这种处理方式的问题在于：对于每个到来的字符，都必须启动用户进程。对于短暂的数据流量让一个进程运行许多次效率会很低，所以这不是一个良好的设计。

图5-15b所示为一种改进措施。此处，用户进程在用户空间中提供了一个包含n个字符的缓冲区，并且执行读入n个字符的读操作。中断服务过程负责将到来的字符放入该缓冲区中直到缓冲区填满，然后唤醒用户进程。这一方案比前一种方案的效率要高很多，但是它也有一个缺点：当一个字符到来时，如果缓冲区被分页而调出了内存会出现什么问题呢？解决方法是将缓冲区锁定在内存中，但是如果许多进程都在内存中锁定页面，那么可用页面池就会收缩并且系统性能将下降。

另一种方法是在内核空间中创建一个缓冲区并且让中断处理程序将字符放到这个缓冲区中，如图

5-15c所示。当该缓冲区被填满的时候，将包含用户缓冲区的页面调入内存（如果需要的话），并且在一次操作中将内核缓冲区的内容复制到用户缓冲区中。这一方法的效率要高很多。

然而，即使这种方案也面临一个问题：正当包含用户缓冲区的页面从磁盘调入内存的时候有新的字符到来，这样会发生什么事情？因为缓冲区已满，所以没有地方放置这些新来的字符。一种解决问题的方法是使用第二个内核缓冲区。第一个缓冲区填满之后，在它被清空之前，使用第二个缓冲区，如图5-15d所示。当第二个缓冲区填满时，就可以将它复制给用户（假设用户已经请求它）。当第二个缓冲区正在复制到用户空间的时候，第一个缓冲区可以用来接收新的字符。以这样的方法，两个缓冲区轮流使用：当一个缓冲区正在被复制到用户空间的时候，另一个缓冲区正在收集新的输入。像这样的缓冲模式称为*双缓冲*（double buffering）。

广泛使用的另一种形式的缓冲是循环缓冲区（circular buffer）。它由一个内存区域和两个指针组成。一个指针指向下一个空闲的字，新的数据可以放置到此处。另一个指针指向缓冲区中数据的第一个字，该字尚未被取走。在许多情况下，当添加新的数据时（例如刚刚从网络到来），硬件将推进第一个指针，而操作系统在取走并处理数据时推进第二个指针。两个指针都是环绕的，当它们到达顶部时将回到底部。

缓冲对于输出也是十分重要的。例如，对于没有缓冲区的调制解调器，我们考虑采用图5-15b的模型输出是如何实现的。用户进程执行write系统调用以输出n个字符。系统在此刻有两种选择。它可以将用户阻塞直到写完所有字符，但是这样做在低速的电话线上可能花费非常长的时间。它也可以立即将用户释放并且在进行I/O的同时让用户做某些其他计算，但是这会导致一个更为糟糕的问题：用户进程怎样知道输出已经完成并且可以重用缓冲区？系统可以生成一个信号或软件中断，但是这样的编程方式是十分困难的并且被证明是竞争条件。对于内核来说更好的解决方法是将数据复制到一个内核缓冲区中，与图5-15c相类似（但是是另一个方向），并且立刻将调用者解除阻塞。现在实际的I/O什么时候完成都没有关系了，用户一旦被解除阻塞立刻就可以自由地重用缓冲区。

缓冲是一种广泛采用的技术，但是它也有不利的方面。如果数据被缓冲太多次，性能就会降低。例如，考虑图5-16中的网络。其中，一个用户执行了一个系统调用向网络写数据。内核将数据包复制到一个内核缓冲区中，从而立即使用户进程得以继续进行（第1步）。在此刻，用户程序可以重用缓冲区。

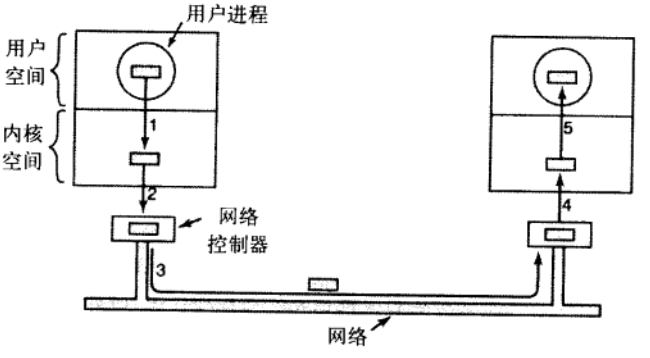


图5-16 可能涉及多次复制一个数据包的网络

当驱动程序被调用时，它将数据包复制到控制器上以供输出（第2步）。它不是将数据包从内核内存直接输出到网线上，其原因是一旦开始一个数据包的传输，它就必须以均匀的速度继续下去，驱动程序不能保证它能够以均匀的速度访问内存，因为DMA通道与其他I/O设备可能正在窃取许多周期。不能及时获得一个字将毁坏数据包，而通过在控制器内部对数据包进行缓冲就可以避免这一问题。

当数据包复制到控制器的内部缓冲区中之后，它就会被复制到网络上（第3步）。数据位被发送之后立刻就会到达接收器，所以在最后一位刚刚送出之后，该位就到达了接收器，在这里数据包在控制器中被缓冲。接下来，数据包复制到接收器的内核缓冲区中（第4步）。最后，它被复制到接收进程的缓冲区中（第5步）。然后接收器通常会发回一个应答。当发送者得到应答时，它就可以自由地发送下一个数据包。然而，应该清楚的是，所有这些复制操作都会在很大程度上降低传输速率，因为所有这些步骤必须有序地发生。

3. 错误报告

错误在I/O上下文中比在其他上下文中要常见得多。当错误发生时，操作系统必须尽最大努力对它们进行处理。许多错误是设备特定的并且必须由适当的驱动程序来处理，但是错误处理的框架是设备无关的。

一种类型的I/O错误是编程错误，这些错误发生在一个进程请求某些不可能的事情时，例如写一个输入设备（键盘、扫描仪、鼠标等）或者读一个输出设备（打印机、绘图仪等）。其他的错误包括提供了一个无效的缓冲区地址或者其他参数，以及指定了一个无效的设备（例如，当系统只有两块磁盘时指定了磁盘3），如此等等。在这些错误上采取的行动是直截了当的：只是将一个错误代码报告返回给调用者。

另一种类型的错误是实际的I/O错误，例如，试图写一个已经被破坏的磁盘块，或者试图读一个已经关机的便携式摄像机。在这些情形中，应该由驱动程序决定做什么。如果驱动程序不知道做什么，它应该将问题向上传递，返回给与设备无关的软件。

软件要做的事情取决于环境和错误的本质。如果是一个简单的读错误并且存在一个交互式的用户可利用，那么它就可以显示一个对话框来询问用户做什么。选项可能包括重试一定的次数，忽略错误，或者杀死调用进程。如果没有用户可利用，惟一的实际选择或许就是以一个错误代码让系统调用失败。

然而，某些错误不能以这样的方式来处理。例如，关键的数据结构（如根目录或空闲块列表）可能已经被破坏，在这种情况下，系统也许只好显示一条错误消息并且终止。

4. 分配与释放专用设备

某些设备，例如CD-ROM刻录机，在任意给定的时刻只能由一个进程使用。这就要求操作系统对设备使用的请求进行检查，并且根据被请求的设备是否可用来接受或者拒绝这些请求。处理这些请求的一种简单方法是要求进程在代表设备的特殊文件上直接执行open操作。如果设备是不可用的，那么open就会失败。于是就关闭这样的一个专用设备，然后将其释放。

一种代替的方法是对于请求和释放专用设备要有特殊的机制。试图得到不可用的设备可以将调用者阻塞，而不是让其失败。阻塞的进程被放入一个队列。迟早被请求的设备会变得可用，这时就可以让队列中的第一个进程得到该设备并且继续执行。

5. 与设备无关的块大小

不同的磁盘可能具有不同的扇区大小。应该由与设备无关的软件来隐藏这一事实并且向高层提供一个统一的块大小，例如，将若干个扇区当作一个逻辑块。这样，高层软件就只需处理抽象的设备，这些抽象设备全都使用相同的逻辑块大小，与物理扇区的大小无关。类似地，某些字符设备（如调制解调器）一次一个字节地交付它们的数据，而其他的设备（如网络接口）则以较大的单位交付它们的数据。这些差异也可以被隐藏起来。

5.3.4 用户空间的I/O软件

尽管大部分I/O软件都在操作系统内部，但是仍然有一小部分在用户空间，包括与用户程序连接在一起的库，甚至完全运行于内核之外的程序。系统调用（包括I/O系统调用）通常由库过程实现。当一个C程序包含调用

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

时，库过程write将与该程序连接在一起，并包含在运行时出现在内存中的二进制程序中。所有这些库过程的集合显然是I/O系统的组成部分。

虽然这些过程所做的工作不过是将这些参数放在合适的位置供系统调用使用，但是确有其他I/O过程实际实现真正的操作。输入和输出的格式化是由库过程完成的。一个例子是C语言中的printf，它以一个格式串和可能的一些变量作为输入，构造一个ASCII字符串，然后调用write以输出这个串。作为printf的一个例子，考虑语句

```
printf("The square of %3d is %6d\n", i, i*i);
```

该语句格式化一个字符串，该字符串是这样组成的：先是14个字符的串“The square of ”（注意of后有一个空格），随后是i值作为3个字符的串，然后是4个字符的串“is”（注意前后各有一个空格），然后是i^2值作为6个字符的串，最后是一个换行。

对输入而言，类似过程的一个例子是scanf，它读取输入并将其存放到一些变量中，采用与printf同样语法的格式串来描述这些变量。标准的I/O库包含许多涉及I/O的过程，它们都是作为用户程序的一部分运行的。

并非所有的用户层I/O软件都是由库过程组成的。另一个重要的类别是假脱机系统。假脱机(spooling)是多道程序设计系统中处理独占I/O设备的一种方法。考虑一种典型的假脱机设备：打印机。尽管在技术上可以十分容易地让任何用户进程打开表示该打印机的字符特殊文件，但是假如一个进程打开它，然后很长时间不使用，则其他进程都无法打印。

另一种方法是创建一个特殊进程，称为守护进程(daemon)，以及一个特殊目录，称为假脱机目录(spooling directory)。一个进程要打印一个文件时，首先生成要打印的整个文件，并且将其放在假脱机目录下。由守护进程打印该目录下的文件，该进程是允许使用打印机特殊文件的惟一进程。通过保护特殊文件来防止用户直接使用，可以解决某些进程不必要地长期空占打印机的问题。

假脱机不仅仅用于打印机，还可以在其他情况下使用。例如，通过网络传输文件常常使用一个网络守护进程。要发送一个文件到某个地方，用户可以将该文件放在一个网络假脱机目录下。稍后，由网络守护进程将其取出并且发送出去。这种假脱机文件传输方式的一个特定用途是USENET新闻系统，该网络由世界上使用因特网进行通信的成千上万台计算机组成，针对许多话题存在着几千个新闻组。要发送一条新闻消息，用户可以调用新闻程序，该程序接收要发出的消息，然后将其存放在假脱机目录中，待以后发送到其他计算机上。整个新闻系统是在操作系统之外运行的。

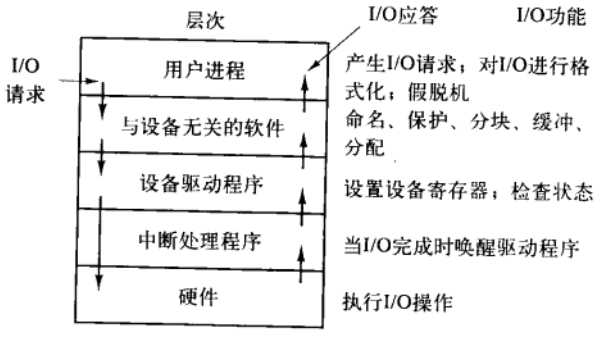


图5-17 I/O系统的层次以及每一层的主要功能

图5-17对I/O系统进行了总结，给出了所有层次以及每一层的主要功能。从底部开始，这些层是硬件、中断处理程序、设备驱动程序、与设备无关的软件，最后是用户进程。

图5-17中的箭头表明了控制流。例如，当一个用户程序试图从一个文件中读一个块时，操作系统被调用以实现这一请求。与设备无关的软件在缓冲区高速缓存中查找有无要读的块。如果需要的块不在其中，则调用设备驱动程序，向硬件发出一个请求，让它从磁盘中获取该块。然后，进程被阻塞直到磁盘操作完成。

当磁盘操作完成时，硬件产生一个中断。中断处理程序就会运行，它要查明发生了什么事情，也就是说此刻需要关注哪个设备。然后，中断处理程序从设备提取状态信息，唤醒休眠的进程以结束此次I/O请求，并且让用户进程继续运行。

5.4 盘

现在我们开始研究某些实际的I/O设备。我们将从盘开始，盘的概念简单，但是非常重要。然后，我们将研究时钟、键盘和显示器。

5.4.1 盘的硬件

盘具有多种多样的类型。最为常用的是磁盘（硬盘和软盘），它们具有读写速度同样快的特点，这使得它们成为理想的辅助存储器（用于分页、文件系统等）。这些盘的阵列有时用来提供高可靠性的存储器。对于程序、数据和电影的发行而言，各种光盘（CD-ROM、可刻录CD以及DVD）也非常重要。在下面各小节中，我们首先描述这些设备的硬件，然后描述其软件。

1. *磁盘*

磁盘被组织成柱面，每一个柱面包含若干磁道，磁道数与垂直堆叠的磁头个数相同。磁道又被分成若干扇区，软盘上大约每条磁道有8～32个扇区，硬盘上每条磁道上扇区的数目可以多达几百个。磁头数大约是1～16个。

老式的磁盘只有少量的电子设备，它们只是传送简单的串行位流。在这些磁盘上，控制器做了大部分的工作。在其他磁盘上，特别是在IDE（Integrated Drive Electronics，集成驱动电子设备）和SATA

（Serial ATA，串行ATA）盘上，磁盘驱动器本身包含一个微控制器，该微控制器承担了大量的工作并且允许实际的控制器发出一组高级命令。控制器经常做磁道高速缓存、坏块重映射以及更多的工作。

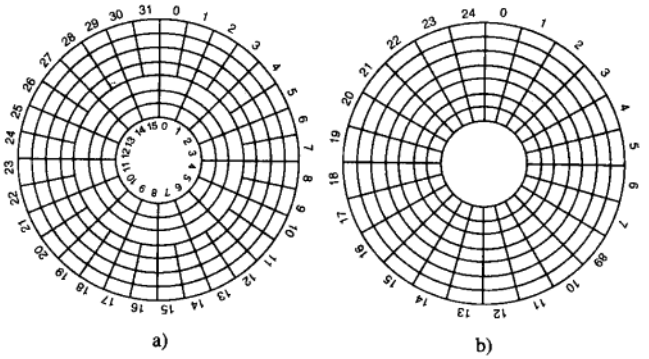
对磁盘驱动程序有重要意义的一个设备特性是：控制器是否可以同时控制两个或多个驱动器进行寻道，这就是**重叠寻道**（overlapped seek）。当控制器和软件等待一个驱动器完成寻道时，控制器可以同时启动另一个驱动器进行寻道。许多控制器也可以在一个驱动器上进行读写操作，与此同时再对另一个或多个其他驱动器进行寻道，但是软盘控制器不能在两个驱动器上同时进行读写操作。（读写数据要求控制器在微秒级时间尺度传输数据，所以一次传输就用完了控制器大部分的计算能力。）对于具有集成控制器的硬盘而言情况就不同了，在具有一个以上这种硬盘驱动器的系统上，它们能够同时操作，至少在磁盘与控制器的缓冲存储器之间进行数据传输的限度之内是这样。然而，在控制器与主存之间可能同时只有一次传输。同时执行两个或多个操作的能力极大地降低了平均存取时间。

图5-18比较了最初的IBM PC标准存储介质的参数与20年后制造的磁盘的参数，从中可以看出过去20年磁盘发生了多大的变化。有趣的是，可以注意到并不是所有的参数都具有同样程度的改进。平均寻道时间改进了7倍，传输率改进了1300倍，而容量的改进则高达50 000倍。这一格局主要是因为磁盘中运动部件的改进相对和缓渐进，而记录表面则达到了相当高的位密度。

参数	IBM 360KB软盘	WD 18300硬盘
柱面数	40	10 601
每柱面磁道数	2	12
每磁道扇区数	9	281（平均）
每磁盘扇区数	720	35 742 000
每扇区字节数	512	512
磁盘容量	360KB	18.3GB
寻道时间（相邻柱面）	6ms	0.8ms
寻道时间（平均情况）	77ms	6.9ms
旋转时间	200ms	8.33ms
电动机停止/启动时间	250ms	20ms
传输1个扇区的时间	22ms	17μs

图5-18 最初的IBM PC 360KB软盘参数与西部数据公司WD 18300硬盘参数

在阅读现代硬盘的说明书时，要清楚的事情是标称的几何规格以及驱动程序软件使用的几何规格与物理格式几乎总是不同的。在老式的磁盘上，每磁道扇区数对所有柱面都是相同的。而现代磁盘则被划分成环带，外层的环带比内层的环带拥有更多的扇区。图5-19a所示为一个微小的磁盘，它具有两个环带，外层的环带每磁道有32个扇区，内层的环带每磁道有16个扇区。一个实际的磁盘（例如WD 18300）常常有16个环带，从最内层的环带到最外层的环带，每个环带的扇区数增加大约4%。



a) b)

图5-19 a) 具有两个环带的磁盘的物理几何规格；b) 该磁盘的一种可能的虚拟几何规格

为了隐藏每个磁道有多少扇区的细节，大多数现代磁盘都有一个虚拟几何规格呈现给操作系统。软件在工作时仿佛存在着x个柱面、y个磁头、每磁道z个扇区，而控制器则将对(x, y, z)的请求重映射到实际的柱面、磁头和扇区。对于图5-19a中的物理磁盘，一种可能的虚拟几何规格如图5-19b所示。在两种情形中磁盘拥有的扇区数都是192，只不过公布的排列与实际的排列是不同的。

对于PC机而言，上述三个参数的最大值常常是（65 535，16，63），这是因为需要与最初IBM PC的限制向后兼容。在IBM PC机器上，使用16位、4位和6位的字段来设定这些参数，其中柱面和扇区从1开始编号，磁头从0开始编号。根据这些参数以及每个扇区512字节可知，磁盘最大可能的容量是31.5GB。为突破这一限制，所有现代磁盘现在都支持一种称为*逻辑块寻址*（logical block addressing，LBA）的系统，在这样的系统中，磁盘扇区从0开始连续编号，而不管磁盘的几何规格如何。

2. RAID

在过去十多年里，CPU的性能一直呈现出指数增长，大体上每18个月翻一番。但是磁盘的性能就不是这样了。20世纪70年代，小型计算机磁盘的平均寻道时间是50～100毫秒，现在的寻道时间略微低于10毫秒。在大多数技术产业（如汽车业或航空业）中，在20年之内有5～10倍的性能改进就将是重大的新闻（想象300 MPG的轿车[⊖]），但是在计算机产业中，这却是一个窘境。因此，CPU性能与磁盘性能之间的差距随着时间的推移将越来越大。

正如我们已经看到的，为了提高CPU的性能，越来越多地使用了并行处理。在过去许多年，很多人也意识到并行I/O是一个很好的思想。Patterson等人在他们1988年写的文章中提出，使用六种特殊的磁盘组织可能会改进磁盘的性能、可靠性或者同时改进这两者（Patterson 等人，1988）。这些思想很快被工业界所采纳，并且导致称为RAID的一种新型I/O设备的诞生。Patterson等人将RAID定义为Redundant Array of Inexpensive Disk（*廉价磁盘冗余阵列*），但是工业界将I重定义为Independent（独立）而不是Inexpensive（廉价），或许这样他们就可以收取更多的费用？因为反面角色也是需要的（如同RISC对CISC，这也是源于Patterson），此处的“坏家伙”是SLED（Single Large Expensive Disk，*单个大容量昂贵磁盘*）。

RAID背后的基本思想是将一个装满了磁盘的盒子安装到计算机（通常是一个大型服务器）上，用RAID控制器替换磁盘控制器卡，将数据复制到整个RAID上，然后继续常规的操作。换言之，对操作系统而言一个RAID应该看起来就像是一个SLED，但是具有更好的性能和更好的可靠性。由于SCSI盘具有良好的性能、较低的价格并且在单个控制器上能够容纳多达7个驱动器（对宽型SCSI而言是15个），很自然地大多数RAID由一个RAID SCSI控制器加上一个装满了SCSI盘的盒子组成，而对操作系统而言这似乎就是一个大容量磁盘。以这样的方法，不需要软件做任何修改就可以使用RAID，对于许多系统管理员来说这可是一大卖点。

除了对软件而言看起来就像是一个磁盘以外，所有的RAID都具有同样的特性，那就是将数据分布在全部驱动器上，这样就可以并行操作。Patterson等人为这样的操作定义了几种不同的模式，它们现在被称为0级RAID到5级RAID。此外，还有少许其他的辅助层级，我们就不讨论了。“层级”这一术语多少有一些用词不当，因为此处不存在分层结构，它们只是可能的六种不同组织形式而已。

0级RAID如图5-20a所示。它将RAID模拟的虚拟单个磁盘划分成条带，每个条带具有k个扇区，其中扇区$0 \sim k-1$为条带0，扇区$k \sim 2k-1$为条带1，以此类推。如果$k = 1$，则每个条带是一个扇区；如果$k = 2$，则每个条带是两个扇区；以此类推。0级RAID结构将连续的条带以轮转方式写到全部驱动器上，图5-20a所示为具有四个磁盘驱动器的情形。

像这样将数据分布在多个驱动器上称为*划分条带*（striping）。例如，如果软件发出一条命令，读取一个由四个连续条带组成的数据块，并且数据块起始于条带边界，那么RAID控制器就会将该命令分解为四条单独的命令，每条命令对应四块磁盘中的一块，并且让它们并行操作。这样我们就运用了并行I/O而软件并不知道这一切。

0级RAID对于大数据量的请求工作性能最好，数据量越大性能就越好。如果请求的数据量大于驱动器数乘以条带大小，那么某些驱动器将得到多个请求，这样当它们完成了第一个请求之后，就会开始处

⊖ MPG是Miles Per Gallon的缩写，即每加仑燃油可以跑多少英里。各国政府对车辆燃油经济性的要求越来越高，目前30 MPG标准成为衡量各家公司车型竞争力度的标杆。——译者注

理第二个请求。控制器的责任是分解请求，并且以正确的顺序将适当的命令提供给适当的磁盘，之后还要在内存中将结果正确地装配起来。0级RAID的性能是杰出的而实现是简单明了的。

对于习惯于每次请求一个扇区的操作系统，0级RAID工作性能最为糟糕。虽然结果会是正确的，但是却不存在并行性，因此也就没有增进性能。这一结构的另一个劣势是其可靠性潜在地比SLED还要差。如果一个RAID由四块磁盘组成，每块磁盘的平均故障间隔时间是20 000小时，那么每隔5000小时就会有一个驱动器出现故障并且所有数据将完全丢失。与之相比，平均故障间隔时间为20 000小时的SLED的可靠性要高出四倍。由于在这一设计中未引入冗余，实际上它还不是真正的RAID。

下一个选择——1级RAID如图5-20b所示，这是一个真正的RAID。它复制了所有的磁盘，所以存在四个主磁盘和四个备份磁盘。在执行一次写操作时，每个条带都被写了两次。在执行一次读操作时，则可以使用其中的任意一个副本，从而将负荷分布在更多的驱动器上。因此，写性能并不比单个驱动器好，但是读性能能够比单个驱动器高出两倍。容错性是突出的：如果一个驱动器崩溃了，只要用副本来替代就可以了。恢复也十分简单，只要安装一个新驱动器并且将整个备份驱动器复制到其上就可以了。

0级RAID和1级RAID操作的是扇区条带，与此不同，2级RAID工作在字的基础上，甚至可能是字节的基础上。想象一下将单个虚拟磁盘的每个字节分割成4位的半字节对，然后对每个半字节加入一个汉明码从而形成7位的字，其中1、2、4位为奇偶校验位。进一步想象如图5-20c所示的7个驱动器在磁盘臂位置与旋转位置方面是同步的。那么，将7位汉明编码的字写到7个驱动器上，每个驱动器写一位，这样做是可行的。

Thinking Machine公司的CM-2计算机采用了这一方案，它采用32位数据字并加入6个奇偶校验位形成一个38位的汉明字，再加上一个额外的位用于汉明字的奇偶校验，并且将每个字分布在39个磁盘驱动器上。因为在一个扇区时间里可以写32个扇区的数据，所以总的吞吐量是巨大的。此外，一个驱动器的损坏不会引起问题，因为损坏一个驱动器等同于在每个39位字的读操作中损失一位，而这是汉明码可以轻松处理的事情。

不利的一面是，这一方案要求所有驱动器的旋转必须同步，并且只有在驱动器数量很充裕的情况下才有意义（即使对于32个数据驱动器和6个奇偶驱动器而言，也存在19%的开销）。这一方案还对控制器提出许多要求，因为它必须在每个位时间里求汉明校验和。

3级RAID是2级RAID的简化版本，如图5-20d所示。其中要为每个数据字计算一个奇偶校验位并且将其写入一个奇偶驱动器中。与2级RAID一样，各个驱动器必须精确地同步，因为每个数据字分布在多个驱动器上。

乍一想，似乎单个奇偶校验位只能检测错误，而不能纠正错误。对于随机的未知错误的情形，这样的看法是正确的。然而，对于驱动器崩溃这样的情形，由于坏位的位置是已知的，所以这样做完全能够纠正1位错误。如果一个驱动器崩溃了，控制器只需假装该驱动器的所有位为0，如果一个字有奇偶错误，那么来自废弃了的驱动器上的位原来一定是1，这样就纠正了错误。尽管2级RAID和3级RAID两者都提供了非常高的数据率，但是每秒钟它们能够处理的单独的I/O请求的数目并不比单个驱动器好。

4级RAID和5级RAID再次使用条带，而不是具有奇偶校验的单个字。如图5-20e所示，4级RAID与0级RAID相类似，但是它将条带对条带的奇偶条带写到一个额外的磁盘上。例如，如果每个条带k字节长，那么所有的条带进行异或操作，就得到一个k字节长的奇偶条带。如果一个驱动器崩溃了，则损失的字节可以通过读出整个驱动器组从奇偶驱动器重新计算出来。

这一设计对一个驱动器的损失提供了保护，但是对于微小的更新其性能很差。如果一个扇区被修改了，那么就必须读取所有的驱动器以便重新计算奇偶校验，然后还必须重写奇偶校验。作为另一选择，它也可以读取旧的用户数据和旧的奇偶校验数据，并且用它们重新计算新的奇偶校验。即使是对于这样的优化，微小的更新也还是需要两次读和两次写。

结果，奇偶驱动器的负担十分沉重，它可能会成为一个瓶颈。通过以循环方式在所有驱动器上均匀地分布奇偶校验位，5级RAID消除了这一瓶颈，如图5-20f所示。然而，如果一个驱动器发生崩溃，重新构造故障驱动器的内容是一个非常复杂的过程。

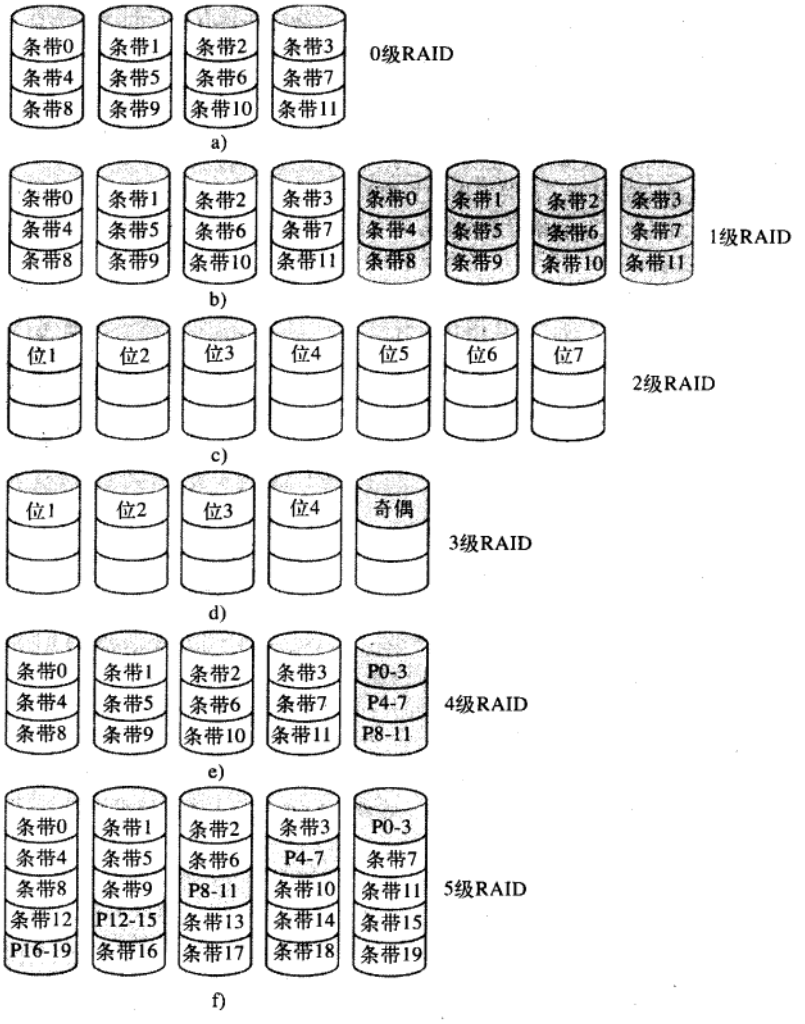


图5-20 0级RAID到5级RAID（备份驱动器及奇偶驱动器以阴影显示）

3. CD-ROM

最近几年，光盘（与磁盘相对应）开始流行。光盘比传统的磁盘具有更高的记录密度。光盘最初是为记录电视节目而开发的，但是作为计算机存储设备它们可以被赋予更为重要的用途。由于它们潜在的巨大容量，光盘一直是大量研究工作的主题，并且经历了令人难以置信的快速发展。

第一代光盘是荷兰的电子集团公司飞利浦为保存电影而发明的。它们的直径为30 cm并且以LaserVision的名字上市，但是它们没有流行起来（日本除外）。

1980年，飞利浦连同索尼开发了CD（Compact Disc，压缩光盘），它很快就取代了每分钟33 1/3转的乙烯树脂唱片来记录音乐（艺术鉴赏家除外，他们仍旧喜爱乙烯树脂唱片）。CD的准确技术细节以正式国际标准（IS 10149）的形式出版，由于其封面的颜色而通俗地被称为红皮书（Red Book）。（国际标准由国际标准化组织发布，国际标准化组织是诸如ANSI、DIN等国家标准团体的国际对等机构。每一个国际标准都有一个IS号码。）将光盘以及驱动器的规范作为国际标准出版，其目的在于让来自不同音乐出版商的CD和来自不同电子设备制造商的播放器能够一同工作。所有的CD都是直径120 mm，厚度1.2 mm，中间有一个15 mm的圆孔。音频CD是第一个成功的大众市场数字存储介质。它们被设想应该能够耐用100年。请在2080年进行核对，看一看第一批CD还能不能很好地工作。

一张CD的准备分成几个步骤，包括使用高功率的红外激光在具有涂层的玻璃母盘上烧出许多直径为0.8μm的小孔。从这张母盘可以制作出铸模，铸模在激光孔所在的位置具有突起。将熔化的聚碳酸酯树脂注入这一铸模，就可以形成具有与玻璃母盘相同小孔模式的一张CD。然后将一个非常薄的反射铝层沉积在聚碳酸酯上，再加上一层保护性的漆膜，最后加上一个标签。聚碳酸酯基片中的凹陷处称为凹痕（pit），凹痕之间未被烧的区域称为槽脊（land）。

在回放的时候，低功率的激光二极管发出波长为0.78μm的红外光，随着凹痕和槽脊的通过照射在其上。激光在聚碳酸酯一面，所以凹痕朝着激光的方向突出，就像是另一侧平坦表面上的突起一样。因为凹痕的高度是激光波长的四分之一，所以从凹痕反射回来的光线与从周围表面反射回来的光线在相位上相差半个波长。结果，两部分相消干涉，与从槽脊反射回的光线相比只返回很少的光线到播放器的光电探测器。这样播放器就可以区分凹痕和槽脊。尽管使用凹痕记录0并且使用槽脊记录1看起来非常简单，但是使用凹痕/槽脊或槽脊/凹痕的过渡来记录1而用这种过渡的缺失来记录0却更加可靠，所以采用这一方案。

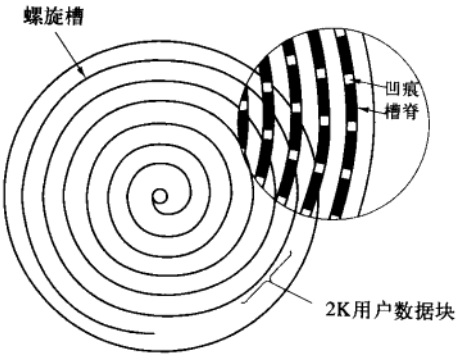


图5-21 压缩光盘或CD-ROM的记录结构

凹痕和槽脊写在一个连续螺旋中，该螺旋起源于接近中间圆孔的地方并且向边缘延伸出32 mm的距离。螺旋环绕着光盘旋转了22 188圈（大约每毫米600圈），如果展开的话，它将有5.6 km长。螺旋如图5-21所示。

为了以均匀的速度播放音乐，必须让凹痕和槽脊以恒定的线速度通过。因此，当CD的读出头从CD的内部向外部移动时，CD的旋转速度必须连续地降低。在内部，旋转速度是530rpm以便达到期望的每秒120 cm的流动速度；而在外部，旋转速度必须降到200rpm以便在激光头处得到相同的线速度。恒定线速度驱动器与磁盘驱动器存在相当大的区别，后者以恒定角速度操作，与磁头当前处于什么位置无关。此外，530rpm与大多数磁盘3600～7200rpm的旋转速度相比存在相当大的距离。

1984年，飞利浦和索尼认识到使用CD存放计算机数据的潜力，所以他们出版了黄皮书（Yellow Book），定义了现在称为CD-ROM（Compact Disc-Read Only Memory，压缩光盘-只读存储器）的光盘的确切标准。为了借助在当时已经十分牢固的音频CD市场，CD-ROM在物理尺寸上与音频CD相同，在机械上和光学上也与之兼容，并且使用相同的聚碳酸酯注模机器生产。这一决策的结果是，不但需要缓慢的可变速度的电机，而且在适度的销量下CD-ROM的制造成本将很好地控制在1美元以下。

黄皮书所定义的是计算机数据的格式化。它还改进了系统的纠错能力，这是一个必要的措施，因为尽管音乐爱好者并不介意在这里或那里丢失一位，但是计算机爱好者往往对此非常挑剔。CD-ROM的基本格式是每个字节以14位的符号进行编码。正如我们在前面看到的，14位足以对一个8位的字节进行汉明编码，并且剩下2位。实际上，CD-ROM使用的是功能更为强大的编码系统[⊖]。对于读操作而言，14到8映射是通过查找表由硬件实现的。

在下一个层次上，一组42个连续符号形成一个588位的帧（frame）。每一帧拥有192个数据位（24个字节），剩余的396位用于纠错和控制。在这396位中，252位是14位符号中的纠错位，而144位包含在8位符号的有效载荷中[⊜]。到目前为止，这一方案对于音频CD和CD-ROM是完全一致的。

黄皮书所增加的是将98帧编组为一个CD-ROM扇区（CD-ROM sector），如图5-22所示。每个CD-

⊖ 该编码系统称为EFM（Eight to Fourteen Modulation，8到14调制）编码，就是把一个8位的数据（即1个字节）用14位编码来表示。——译者注

⊜ 此处的描述不甚准确。在588位的一帧数据中，有24位同步信息（这24位同步位不经EFM编码）和33个数据字节（每个字节经过14位EFM编码）。在33个数据字节中，包含有效数据（或称有效载荷）24字节，其余9字节用于控制和校验。为了确保读出信号的可靠性，每个编码字之间插入3位结合位，在帧尾还有3位结合位，因此一帧的长度为24+33×14+34×3=588位。——译者注

ROM扇区以一个16字节的前导码开始，其中前12个字节为00FFFFFFFFFFFFFFFFFFFF00（十六进制），以便让播放器识别一个CD-ROM扇区的开始。接下来的3个字节包含扇区号，这是必需的，因为在具有单个数据螺旋的CD-ROM上寻道比在具有均匀同心磁道的磁盘上寻道要困难得多。为了进行寻道，驱动器中的软件要计算出一个近似的位置，将激光头移动到那里，然后开始在四周搜索一个前导码来看一看猜测的如何。前导码的最后一个字节是模式。

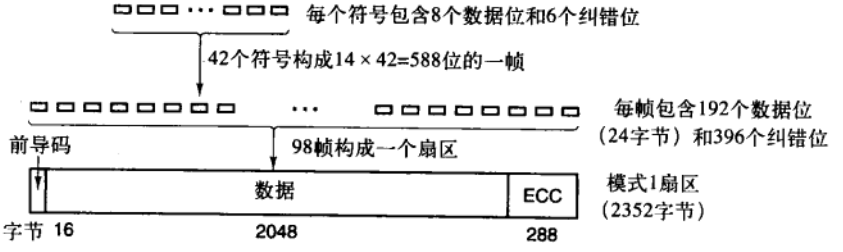


图5-22 CD-ROM上的逻辑数据布局

黄皮书定义了两种模式。模式1使用图5-22的布局，具有16字节的前导码、2048个数据字节和一个288字节的纠错码（横交叉Reed-Solomon码）。模式2将数据和ECC域合并成一个2336字节的数据域，用于不需要纠错（或者抽不出时间执行纠错）的应用，例如音频和视频。注意，为了提供优异的可靠性，在符号内部、帧内部和CD-ROM扇区内部使用了三种独立的纠错方案。单个位的错误在最低的层次上纠正，短暂的突发错误在帧的层次上纠正，任何残留的错误在扇区的层次上捕获。为这一可靠性付出的代价是花费98个588位的帧（7203字节）来容纳2048字节的有效载荷，效率只有28%。

单速CD-ROM驱动器以75扇区/秒的速度工作，提供的数据率在模式1下是153 600字节/秒，在模式2下是175 200字节/秒。双速驱动器快两倍，以此类推，直到最高的速度。因此，一个40倍速的驱动器能够以40×153 600字节/秒的速度传递数据，假设驱动器接口、总线以及操作系统都能够处理这样的数据率。一个标准的音频CD具有存放74分钟音乐的空间，如果将其用于在模式1下存放数据，提供的容量是681 984 000字节。这一数字通常被报告为650MB，这是因为1MB是2^{20}字节（1 048 576字节），而不是1 000 000字节。

注意，即使一个32倍速的CD-ROM驱动器（数据率为4 915 200字节/秒）也无法与速度为10MB/s的快速SCSI-2磁盘驱动器相配，尽管许多CD-ROM驱动器使用了SCSI接口（也存在IDE CD-ROM驱动器）。当你意识到寻道时间通常是几百毫秒时，就会清楚CD-ROM驱动器与磁盘驱动器在性能上不属于同样的范畴，尽管它们有非常大的容量。

1986年，飞利浦以绿皮书（Green Book）再度出击，补充了图形以及在相同的扇区中保存交错的音频、视频和数据的能力，这对于多媒体CD-ROM而言是十分必要的。

CD-ROM的最后一个难题是文件系统。为了使相同的CD-ROM能够在不同的计算机上使用，有关CD-ROM文件系统的协议是必要的。为了达成这一协议，许多计算机公司的代表相聚在加利福尼亚和内华达两州边界处Tahoe湖畔的High Sierra宾馆，设计了被他们称为High Sierra的文件系统，这一文件系统后来发展成为一个国际标准（IS 9660）。该文件系统有三个层次。第一层使用最多8个字符的文件名，可选地跟随最多3个字符的扩展名（MS-DOS的文件命名约定）。文件名只能够包含大写字母、数字和下划线。目录能够嵌套最多8层深度，但是目录名不能包含扩展名。第一层要求所有文件都是连续的，这对于只能写一次的介质来说并不是一个问题。符合IS 9660标准第一层的任何CD-ROM都可以使用MS-DOS、苹果计算机、UNIX计算机或者几乎任何其他计算机读出。CD-ROM出版商十分看重这一特性，视其为重大的有利因素。

IS 9660第二层允许文件名最多有32个字符，第三层允许文件是不连续的。Rock Ridge扩展允许非常长的文件名（针对UNIX）、UID、GID和符号连接，但是不符合第一层标准的CD-ROM将不能在所有计算机上可读。

对于出版各种游戏、电影、百科全书、地图集以及参考手册，CD-ROM已经变得非常流行。大多数商业软件现在也是通过CD-ROM发行的。巨大的容量和低廉的生产成本相结合，使得CD-ROM适合无数的应用。

4. 可刻录CD

起初，制造一片CD-ROM 母盘（或音频CD母盘，就此事而言）所需要的设备极其昂贵。但是按照计算机产业的惯例，没有什么东西能够长久地保持高价位。到20世纪90年代中期，尺寸不比CD播放器大的CD刻录机在大多数计算机商店中已经是可以买到的常见外部设备。这些设备仍然不同于磁盘，因为一旦写入，CD-ROM就不能被擦除了。然而，它们很快就找到了适当的位置，即作为大容量硬盘的备份介质，并且还可以让个人或刚起步的公司制造他们自己的小批量的CD-ROM，或者制作母盘以便递交给高产量的商业CD复制工厂。这些驱动器被称为是CD-R（CD-Recordable，可刻录CD）。

物理上，CD-R在开始的时候是像CD-ROM一样的120mm的聚碳酸酯空盘，不同的是CD-R包含一个0.6mm宽的凹槽来引导激光进行写操作。凹槽具有3mm的正弦振幅，频率精确地为22.05 kHz，以便提供连续的反馈，这样就可以正确地监视旋转速度并且在需要的时候对其进行调整。CD-R看上去就像是常规的CD-ROM，只是CD-R顶面是金色的而不是银色的。金色源于使用真金代替铝作为反射层。银色的CD在其上具有物理的凹陷，与此不同的是，在CD-R上，必须模拟凹痕和槽脊的不同反射率。这是通过在聚碳酸酯与反射金层之间添加一层染料而实现的，如图5-23所示。使用的染料有两种：绿色的花菁和淡橘黄色的酞菁。至于哪一种染料更好化学家们可能会无休止地争论下去。这些染料与摄影技术中使用的染料相类似，这就解释了为什么柯达和富士是主要的空白CD-R制造商。

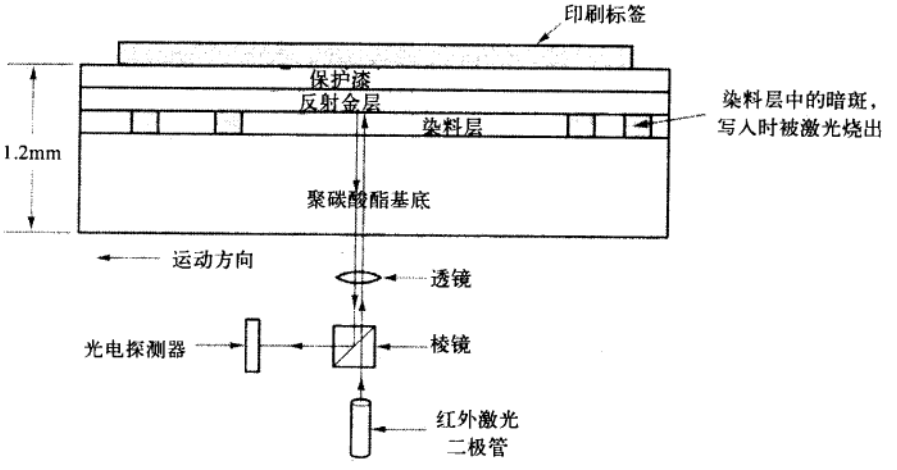


图5-23 CD-R盘和激光的横截面（未按比例画）。银色的CD-ROM具有类似的结构，只是不具有染料层并且以有凹痕的铝层代替金层

在初始状态下，染料层是透明的，能够让激光透过并且从金层反射回来。写入时，CD-R激光提升到高功率（8～16mW）。当光束遇到染料时，将其加热，从而破坏其化学结合力，这一分子结构的变化造成一个暗斑。当读回时（以0.5mW），光电探测器会识别出已经被烧过的染料处的暗斑与完好的透明区域之间的区别。这一区别被解释为凹痕与槽脊之间的差别，即使在常规的CD-ROM阅读器甚至在音频CD播放器上读回时，也是如此。

如果没有一本“有色的”书，就没有CD的新类型能够骄傲地昂起头，所以CD-R具有橘皮书（Orange Book），出版于1989年。这份文档定义了CD-R和一个新格式CD-ROM XA，它允许CD-R被逐渐增长地写入，今天几个扇区，明天几个扇区，下个月几个扇区。一次写入的一组连续的扇区称为一个CD-ROM光轨（CD-ROM track）。

CD-R的最初应用之一是柯达PhotoCD。在这一系统中，消费者将一卷已曝光的胶片和老的PhotoCD带给照片加工者，并且取回同一个PhotoCD，其中新的照片已经添加到老的照片之后。新的一批照片是通过扫描底片创建的，它们作为单独的CD-ROM光轨写在PhotoCD上。逐渐增长式写入是需要的，因为在这一产品引入的时候，CD-R空盘还过于昂贵，以至于负担不起为每个胶卷提供一张盘。

然而，逐渐增长式写入造成一个新的问题。在橘皮书之前，所有的CD-ROM在开始处有一个VTOC（Volume Table of Contents，卷目录）。这一方法对于逐渐增长式（也就是多光轨）写入是行不通的。橘皮书的解决方案是给每个CD-ROM光轨提供自己的VTOC，在VTOC中列出的文件可以包含某些或者所有来自先前光轨中的文件。当CD-R被插入到驱动器之后，操作系统从头到尾搜索所有的CD-ROM光轨以定位最近的VTOC，它提供了光盘的当前状态。通过在当前VTOC包含来自先前光轨中的某些而不是全部文件，可能会引起错觉，即文件已经被删除了。光轨可以被分组成段（session），这样就引出了多段（multisession）CD-ROM。标准的音频CD播放器不能处理多段CD，因为它们要求在开始处有一个VTOC。可是，某些计算机应用程序可以处理它们。

CD-R使得个人和公司轻松地复制CD-ROM（和音频CD）成为可能，只是通常会侵犯出版商的版权。人们设计了几种方案使这种盗版行为更加困难，并且使除了出版商的软件以外的任何软件都难于用来读取CD-ROM。方案之一是在CD-ROM上将所有文件的长度记录为几吉字节，从而挫败任何使用标准复制软件将文件复制到硬盘上的企图。实际的文件长度嵌入在出版商的软件中，或者隐藏（可能是加密的）在CD-ROM上意想不到的地方。另一种方案是在挑选出来的扇区中故意使用错误的ECC，期望CD复制软件将会“修正”这些错误，而应用程序软件则核对ECC本身，如果是正确的就拒绝工作。使用光轨间非标准的间隙和其他物理“瑕疵”也是可能的。

5. 可重写CD

尽管人们习惯于使用其他一次性写的介质，例如纸张和摄影胶片，但是却存在着对可重写CD-ROM的需求。目前可用的一个技术是CD-RW（CD-ReWritable，可重写CD），它使用与CD-ROM相同尺寸的介质。然而，CD-RW使用银、铟、锑和碲合金作为记录层，以取代花菁和酞菁染料。这一合金具有两个稳定的状态：结晶态和非结晶态，两种状态具有不同的反射率。

CD-RW驱动器使用具有三种不同功率的激光。在高功率下，激光将合金融化，将其从高反射率的结晶态转化为低反射率的非结晶态，代表一个凹痕。在中功率下，激光将合金融化并重构其自然结晶状态以便再次成为一个槽脊。在低功率下，材料的状态被感知（用于读取），但是不发生状态的转化。

CD-RW没有取代CD-R的原因是CD-RW空白盘比CD-R空白盘要昂贵得多。此外，对于涉及对硬盘进行备份的应用程序来说，实际情况就是一次性写入，CD-R不会被意外地擦除是一大好事。

6. DVD

基本CD/CD-ROM格式自1980年以来经受了考验。从那时起，技术在不断改进，所以更高容量的光盘现在在经济上是可行的，并且存在着对它们的巨大需求。好莱坞热切地希望用数字光盘来取代模拟录像磁带，因为光盘具有更高的容量，更低廉的制造成本，更长的使用时间，占用音像商店更少的货架空间，并且不必倒带。消费性电子公司正期待着一种新型的一鸣惊人的产品，而许多计算机公司则希望为他们的软件增添多媒体特性。

这三个极其富有并且势力强大的产业在技术与需求方面的结合引出了DVD，最初DVD是Digital Video Disk（数字视盘）的首字母缩写，但是现在官方的名称是Digital Versatile Disk（数字通用光盘）。DVD采用与CD同样的总体设计，使用120 mm的注模聚碳酸酯盘片，包含凹痕和槽脊，它们由激光二极管照明并且由光电探测器读取。新特性包括使用了：

1) 更小的凹痕（0.4μm，CD是0.8μm）。

2) 更密的螺旋（轨迹间距0.74μm，CD是1.6μm）。

3) 红色激光（波长0.65μm，CD是0.78μm）。

综合起来，这些改进将容量提高了7倍，达到4.7GB。一个1倍速的DVD驱动器以1.4 MB/s的速率运转（CD是150 KB/s）。但是，切换到红色激光意味着DVD播放器需要第二个激光器或者价格高昂的光学转换器才能够读取现有的CD和CD-ROM。随着激光器价格的下降，现在大多数驱动器都有两种激光器，所以它们能够读取两种类型的介质。

是不是4.7GB就足够了？也许是。采用MPEG-2压缩（在IS 13346中标准化），一块4.7GB的DVD盘能够保存133分钟高分辨率（720×480）的全屏幕、全运动视频，以及最多8种语言的音轨和最多32种语言的字幕。好莱坞曾经制作的全部电影中大约92%在133分钟以下。然而，某些应用（例如多媒体游戏或者

参考手册）可能需要更多的空间，并且好莱坞希望将多部电影放在同一张盘上，为此定义了四种格式：

1) 单面单层（4.7GB）。

2) 单面双层（8.5GB）。

3) 双面单层（9.4GB）。

4) 双面双层（17GB）。

为什么要如此多种格式？一句话：政治利益。飞利浦和索尼对于高容量的版本希望采用单面双层盘，而东芝和时代华纳则希望采用双面单层盘。飞利浦和索尼认为人们不会愿意将盘片翻面，而东芝和时代华纳则不相信将两层放在一面能够工作。妥协是支持全部组合，但是市场将决定哪些格式会生存下来。

双层技术在底部具有一个反射层，在上面加上一个半反射层。激光从一层还是从另一层反射回来取决于激光在何处汇聚。下面一层需要稍微大一些的凹痕和槽脊，以便可靠地读出，所以其容量比上面一层稍微小一些。

双面盘是通过采用两片0.6 mm的单面盘并且将它们背对背地粘合在一起做成的。为了使所有版本的厚度相同，单面盘包含一个0.6 mm的盘片，粘合在一片空白的基底上（或者也许在将来是粘合在一个包含133分钟广告的盘上，期望人们会好奇其中包含什么）。双面双层盘的结构如图5-24所示。

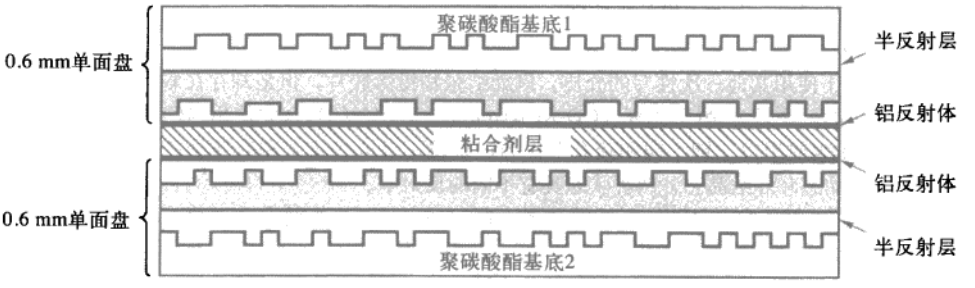


图5-24 双面双层DVD盘

DVD是由10家消费性电子公司的联盟在主要的好莱坞制片厂的紧密协作下设计的，其中7家是日本公司，而其中一些好莱坞制片厂也是由联盟中的日本电子公司所拥有。计算机与电信产业未被邀请参加这一野餐会，导致的结果是注意力集中在将DVD用于电影租赁与营业性放映上。例如，标准特性包括实时跳过色情场景（使父母得以将一部等级为NC17[⊖]的影片转变成对儿童安全的影片），包含六声道声音，并且支持摇摄及扫描。最后一个特性是允许动态地决定如何将电影（其宽高比为3:2）的左和右边缘修剪掉以便适合当前的电视机（其宽高比为4:3）。

另一个计算机业大概不会考虑的项目是在供应给美国的光盘与供应给欧洲的光盘以及适用于其他大陆的其他标准之间故意不兼容。因为新影片总是首先在美国发行，然后当视频产品在美国上市的时候再输出到欧洲，所以好莱坞需要这一“特性”。这一主意可以确保欧洲的音像商店不能过早地在美国买到视频产品，因而减少新电影在欧洲的票房收入。如果计算机产业是由好莱坞来运作的，那么就会在美国只能使用3.5英寸的软盘而在欧洲只能使用9厘米的软盘。

发明单面/双面和单层/双层DVD的那些人再一次陷入混战。由于产业界参与者政治上的争论，下一代DVD仍然缺乏单一的标准。一种新的设备是Blu-ray（蓝光光盘），它使用0.405(m（蓝色）激光将25 GB压入单层盘中，或者将50GB压入双层盘中。另一种设备是HD DVD[⊖]，它使用相同的蓝色激光，但是容量只有15 GB（单层）或者30 GB（双层）。这种格式之战将电影制片厂、计算机制造商和软件公司割裂开来。缺乏标准的结果是，这一代DVD推广得非常慢，因为消费者在等待着尘埃落定，看哪一个格式胜出。产业界这些愚蠢的行为让人想起George Santayana[⊜]的名言：“不能以史为鉴的人注定要重蹈覆辙”。

⊖ NC17代表No Children Under 17 Admitted，即17岁以下儿童不得观看。——译者注

⊖ HD代表High Density（高密度）。——译者注

⊜ George Santayana（乔治·桑塔亚纳，1863—1952），美国著名哲学家、美学家。——译者注

5.4.2 磁盘格式化

硬盘由一叠铝的、合金的或玻璃的盘片组成，直径为5.25英寸或3.5英寸（在笔记本电脑上甚至更小）。在每个盘片上沉积着薄薄的可磁化的金属氧化物。在制造出来之后，磁盘上不存在任何信息。

在磁盘能够使用之前，每个盘片必须经受由软件完成的*低级格式化*（low-level format）。该格式包含一系列同心的磁道，每个磁道包含若干数目的扇区，扇区间存在短的间隙。一个扇区的格式如图5-25所示。

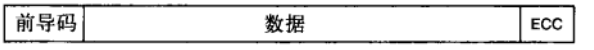


图5-25 一个磁盘扇区

前导码以一定的位模式开始，位模式使硬件得以识别扇区的开始。前导码还包含柱面与扇区号以及某些其他信息。数据部分的大小是由低级格式化程序决定的，大多数磁盘使用512字节的扇区。ECC域包含冗余信息，可以用来恢复读错误。该域的大小和内容随生产商的不同而不同，它取决于设计者为了更高的可靠性愿意放弃多少磁盘空间以及控制器能够处理的ECC编码有多复杂。16字节的ECC域并不是罕见的。此外，所有硬盘都分配有某些数目的备用扇区，用来取代具有制造瑕疵的扇区。

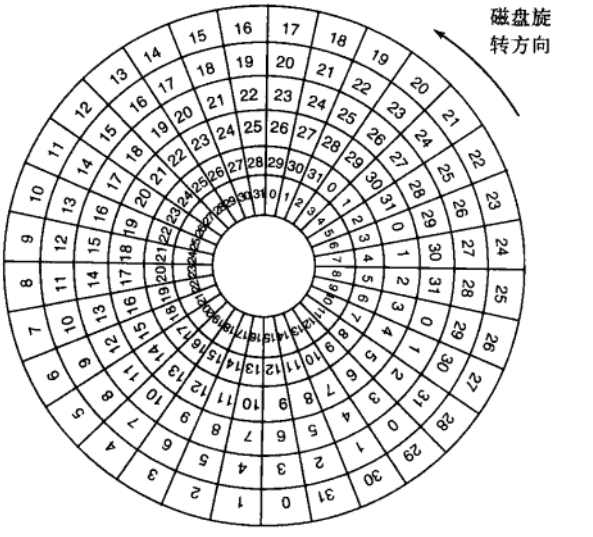


图5-26 柱面斜进示意图

在设置低级格式时，每个磁道上第0扇区的位置与前一个磁道存在偏移。这一偏移称为*柱面斜进*（cylinder skew），这样做是为了改进性能，想法是让磁盘在一次连续的操作中读取多个磁道而不丢失数据。观察图5-19a就可以明白问题的本质。假设一个读请求需要最内侧磁道上从第0扇区开始的18个扇区，磁盘旋转一周可以读取前16个扇区，但是为了得到第17个扇区，则需要一次寻道操作以便磁头向外移动一个磁道。到磁头移动了一个磁道时，第0扇区已经转过了磁头，所以需要旋转一整周才能等到它再次经过磁头。通过图5-26所示的将扇区偏移即可消除这一问题。

柱面斜进量取决于驱动器的几何规格。例如，一个10 000rpm的驱动器每6ms旋转一周，如果一个磁道包含300个扇区，那么每20μs就有一个新扇区在磁头下通过。如果磁道到磁道的寻道时间是800μs，那么在寻道期间将有40个扇区通过，所以柱面斜进应该是40个扇区而不是图5-26中的三个扇区。值得一提的是，像柱面斜进一样也存在着*磁头斜进*（head skew），但是磁头斜进不是非常大。

低级格式化的结果是磁盘容量减少，减少的量取决于前导码、扇区间间隙和ECC的大小以及保留的备用扇区的数目。通常格式化的容量比未格式化的容量低20%。备用扇区不计入格式化的容量，所以一种给定类型的所有磁盘在出厂时具有完全相同的容量，与它们实际具有多少坏扇区无关（如果坏扇区的数目超出了备用扇区的数目，则该驱动器是不合格的，不会出厂）。

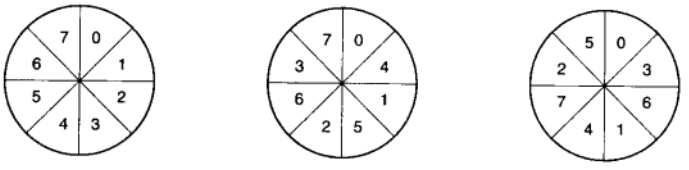
关于磁盘容量存在着相当大的混淆，这是因为某些制造商广告宣传的是未格式化的容量，从而使他们的驱动器看起来比实际的容量要大。例如，考虑一个未格式化容量为200×10^9字节的驱动器，它或许是作为200GB的磁盘销售的。然而，格式化之后，也许只有170×10^9字节可用于存放数据。使这一混淆进一步加剧的是操作系统可能将这一容量报告为158GB，而不是170GB，因为软件把1GB看作是2^{30}（1 073 741 824）字节，而不是10^9（1 000 000 000）字节。

在数据通信世界里，1Gbps意味着1 000 000 000位/秒，因为前缀G（吉）确实表示10^9（毕竟一千米是1000米，而不是1024米），所以使事情更加糟糕。只有在关于内存和磁盘的大小的情况下，kilo（千）、mega（兆）、giga（吉）和tera（太）才分别表示2^{10}、2^{20}、2^{30}和2^{40}。

格式化还对性能产生影响。如果一个10 000RPM的磁盘每个磁道有300个扇区，每个扇区512字节，那么用6ms可以读出一个磁道上的153 600字节，使数据率为25 600 000字节/秒或24.4 MB/s。不论引入什么种类的接口，都不可能比这个速度更快，即便是80 MB/s或160 MB/s的SCSI接口也不行。

实际上，以这一速率连续地读磁盘要求控制器中有一个大容量的缓冲区。例如，考虑一个控制器，它具有一个扇区的缓冲区，该控制器接到一条命令要读两个连续的扇区。当从磁盘上读出第一个扇区并做了ECC计算之后，数据必须传送到主存中。就在传送正在进行时，下一个扇区将从磁头下通过。当完成了向主存的复制时，控制器将不得不等待几乎一整周的旋转时间才能等到第二个扇区再次回来。

通过在格式化磁盘时以交错方式对扇区进行编号可以消除这一问题。在图5-27a中，我们看到的是通常的编号模式（此处忽略柱面斜进）。在图5-27b中，我们看到的是*单交错*（single interleaving），它可以在连续的扇区之间给控制器以喘息的空间以便将缓冲区复制到主存。



a) b) c)

图5-27 a) 无交错；b) 单交错；c) 双交错

如果复制过程非常慢，可能需要如图5-27c中的*双交错*（double interleaving）。如果控制器拥有的缓冲区只有一个扇区，那么从缓冲区到主存的复制无论是由控制器完成还是由主CPU或着DMA芯片完成都无关紧要，都要花费某些时间。为了避免需要交错，控制器应该能够对整个磁道进行缓存，许多现代控制器都能够这样做。

在低级格式化完成之后，要对磁盘进行分区。在逻辑上，每个分区就像是一个独立的磁盘。分区对于多个操作系统共存是必需的。此外，在某些情况下，分区可以用来进行交换。在Pentium和大多数其他计算机上，0扇区包含*主引导记录*（master boot record），它包含某些引导代码和末尾的分区表。分区表给出了每个分区的起始扇区和大小。在Pentium上，分区表具有四个分区的空间。如果这四个分区都用于Windows，那么它们将被称为C:、D:、E:和F:，并且作为单独的驱动器对待。如果它们中有三个用于Windows一个用于UNIX，那么Windows会将它的分区称为C:、D:和E:，然后第一个CD-ROM是F:。为了能够从硬盘引导，在分区表中必须有一个分区被标记为活动的。

在准备一块磁盘以便于使用的最后一步是对每一个分区分别执行一次*高级格式化*（high-level format）。这一操作要设置一个引导块、空闲存储管理（空闲列表或位图）、根目录和一个空文件系统。这一操作还要将一个代码设置在分区表项中，以表明在分区中使用的是哪个文件系统，因为许多操作系统支持多个兼容的文件系统（由于历史原因）。这时，系统就可以引导了。

当电源打开时，BIOS最先运行，它读入主引导记录并跳转到主引导记录。然后这一引导程序进行检查以了解哪个分区是活动的。引导扇区包含一个小的程序，它一般会装入一个较大的引导程序装载器，该引导程序装载器将搜索文件系统以找到操作系统内核。该程序被装入内存并执行。

5.4.3 磁盘臂调度算法

本小节我们将一般地讨论与磁盘驱动程序有关的几个问题。首先，考虑读或者写一个磁盘块需要多长时间。这个时间由以下三个因素决定：

1) 寻道时间（将磁盘臂移动到适当的柱面上所需的时间）。

2) 旋转延迟（等待适当扇区旋转到磁头下所需的时间）。

3) 实际数据传输时间。

对大多数磁盘而言，寻道时间与另外两个时间相比占主导地位，所以减少平均寻道时间可以充分地改善系统性能。

如果磁盘驱动程序每次接收一个请求并按照接收顺序完成请求，即*先来先服务*（First-Come，First-

Served，FCFS），则很难优化寻道时间。然而，当磁盘负载很重时，可以采用其他策略。很有可能当磁盘臂为一个请求寻道时，其他进程会产生其他磁盘请求。许多磁盘驱动程序都维护着一张表，该表按柱面号索引，每一柱面的未完成的请求组成一个链表，链表头存放在表的相应表目中。

给定这种数据结构，我们可以改进先来先服务调度算法。为了说明如何实现，考虑一个具有40个柱面的假想的磁盘。假设读柱面11上一个数据块的请求到达，当对柱面11的寻道正在进行时，又按顺序到达了对柱面1、36、16、34、9和12的请求，则让它们进入未完成的请求表，每一个柱面对应一个单独的链表。图5-28显示了这些请求。

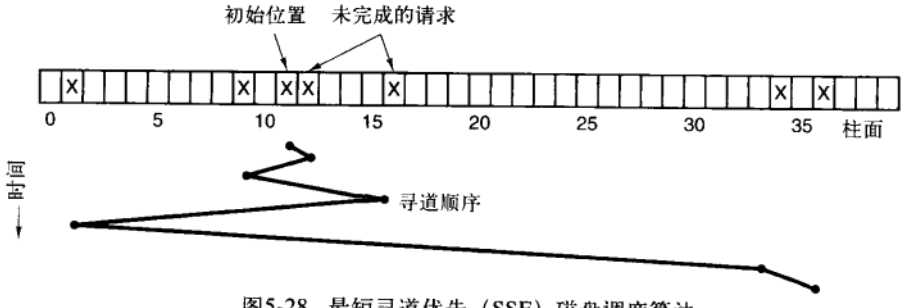


图5-28 最短寻道优先（SSF）磁盘调度算法

当前请求（请求柱面11）结束后，磁盘驱动程序要选择下一次处理哪一个请求。若使用FCFS算法，则首先选择柱面1，然后是36，以此类推。这个算法要求磁盘臂分别移动10、35、20、18、25和3个柱面，总共需要移动111个柱面。

另一种方法是下一次总是处理与磁头距离最近的请求以使寻道时间最小化。对于图5-28中给出的请求，选择请求的顺序如图5-28中下方的折线所示，依次为12、9、16、1、34和36。按照这个顺序，磁盘臂分别需要移动1、3、7、15、33和2个柱面，总共需要移动61个柱面。这个算法即*最短寻道优先*（Shortest Seek First，SSF），与FCFS算法相比，该算法的磁盘臂移动几乎减少了一半。

但是，SSF算法存在一个问题。假设当图5-28所示的请求正在处理时，不断地有其他请求到达。例如，磁盘臂移到柱面16以后，到达一个对柱面8的新请求，那么它的优先级将比柱面1要高。如果接着又到达了一个对柱面13的请求，磁盘臂将移到柱面13而不是柱面1。如果磁盘负载很重，那么大部分时间磁盘臂将停留在磁盘的中部区域，而两端极端区域的请求将不得不等待，直到负载中的统计波动使得中部区域没有请求为止。远离中部区域的请求得到的服务很差。因此获得最小响应时间的目标和公平性之间存在着冲突。

高层建筑也要进行这种权衡处理，高层建筑中的电梯调度问题和磁盘臂调度很相似。电梯请求不断地到来，随机地要求电梯到各个楼层（柱面）。控制电梯的计算机能够很容易地跟踪顾客按下请求按钮的顺序，并使用FCFS或者SSF为他们提供服务。

然而，大多数电梯使用一种不同的算法来协调效率和公平性这两个相互冲突的目标。电梯保持按一个方向移动，直到在那个方向上没有请求为止，然后改变方向。这个算法在磁盘世界和电梯世界都被称为*电梯算法*（elevator algorithm），它需要软件维护一个二进制位，即当前方向位：UP（向上）或是DOWN（向下）。当一个请求处理完之后，磁盘或电梯的驱动程序检查该位，如果是UP，磁盘臂或电梯舱移至下一个更高的未完成的请求。如果更高的位置没有未完成的请求，则方向位取反。当方向位设置为DOWN时，同时存在一个低位置的请求，则移向该位置。

图5-29显示了使用与图5-28相同的7个请求的电梯算法的情况。假设方向位初始为UP，则各柱面获得服务的顺序是12、16、34、36、9和1，磁盘臂分别移动1、4、18、2、27和8个柱面，总共移动60个柱面。在本例中，电梯算法比SSF还要稍微好一点，尽管通常它不如SSF。电梯算法的一个优良特性是对任意的一组给定请求，磁盘臂移动总次数的上界是固定的：正好是柱面数的两倍。

对这个算法稍加改进可以在响应时间上具有更小的变异（Teory，1972），方法是总是按相同的方向进行扫描。当处理完最高编号柱面上未完成的请求之后，磁盘臂移动到具有未完成的请求的最低编号的

柱面，然后继续沿向上的方向移动。实际上，这相当于将最低编号的柱面看作是最高编号的柱面之上的相邻柱面。

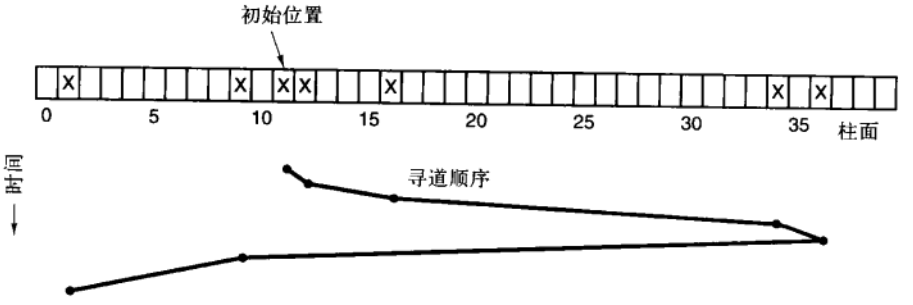


图5-29 调度磁盘请求的电梯算法

某些磁盘控制器提供了一种方法供软件检查磁头下方的当前扇区号。对于这种磁盘控制器，还可以进行另一种优化。如果针对同一柱面有两个或多个请求正等待处理，驱动程序可以发出请求读写下一次要通过磁头的扇区。注意，当一个柱面有多条磁道时，相继的请求可能针对不同的磁道，故没有任何代价。因为选择磁头既不需要移动磁盘臂也没有旋转延迟，所以控制器几乎可以立即选择任意磁头。

如果磁盘具有寻道时间比旋转延迟快很多的特性，那么应该使用不同的优化策略。未完成的请求应该按扇区号排序，并且当下一个扇区就要通过磁头的时候，磁盘臂应该飞快地移动到正确的磁道上对其进行读或者写。

对于现代硬盘，寻道和旋转延迟是如此影响性能，所以一次只读取一个或两个扇区的效率是非常低下的。由于这个原因，许多磁盘控制器总是读出多个扇区并对其进行高速缓存，即使只请求一个扇区时也是如此。典型地，读一个扇区的任何请求将导致该扇区和当前磁道的多个或者所有剩余的扇区被读出，读出的扇区数取决于控制器的高速缓存中有多少可用的空间。例如，在图5-18所描述的磁盘中有4MB的高速缓存。高速缓存的使用是由控制器动态地决定的。在最简单的模式下，高速缓存被分成两个区段，一个用于读，一个用于写。如果后来的读操作可以用控制器的高速缓存来满足，那么就可以立即返回被请求的数据。

值得注意的是，磁盘控制器的高速缓存完全独立于操作系统的高速缓存。控制器的高速缓存通常保存还没有实际被请求的块，但是这对于读操作是很便利的，因为它们只是作为某些其他读操作的附带效应而恰巧要在磁头下通过。与之相反，操作系统所维护的任何高速缓存由显式地读出的块组成，并且操作系统认为它们在较近的将来可能再次需要（例如，保存目录块的一个磁盘块）。

当同一个控制器上有多个驱动器时，操作系统应该为每个驱动器都单独地维护一个未完成的请求表。一旦任何一个驱动器空闲下来，就应该发出一个寻道请求将磁盘臂移到下一个将被请求的柱面处（假设控制器允许重叠寻道）。当前传输结束时，将检查是否有驱动器的磁盘臂位于正确的柱面上。如果存在一个或多个这样的驱动器，则在磁盘臂已经位于正确柱面处的驱动器上开始下一次传输。如果没有驱动器的磁盘臂处于正确的位置，则驱动程序在刚刚完成传输的驱动器上发出一个新的寻道命令并且等待，直到下一次中断到来时检查哪一个磁盘臂首先到达了目标位置。

上面所有的磁盘调度算法都是默认地假设实际磁盘的几何规格与虚拟几何规格相同，认识到这一点十分重要。如果不是这样，那么调度磁盘请求就毫无意义，因为操作系统实际上不能断定柱面40与柱面200哪一个与柱面39更接近。另一方面，如果磁盘控制器能够接收多个未完成的请求，它就可以在内部使用这些调度算法。在这样的情况下，算法仍然是有效的，但是低了一个层次，局限在控制器内部。

5.4.4 错误处理

磁盘制造商通过不断地加大线性位密度而持续地推进技术的极限。在一块5.25英寸的磁盘上，处于中间位置的一个磁道大约有300mm的周长。如果该磁道存放300个512字节的扇区，考虑到由于前导码、ECC和扇区间隙而损失了部分空间这样的实际情况，线性记录密度大约是5000b/mm。记录5000b/mm需

要极其均匀的基片和非常精细的氧化物涂层。但是，按照这样的规范制造磁盘而没有瑕疵是不可能的。一旦制造技术改进到一种程度，即在那样的密度下能够无瑕疵地操作，磁盘设计者就会转到更高的密度以增加容量。这样做可能会再次引入瑕疵。

制造时的瑕疵会引入坏扇区，也就是说，扇区不能正确地读回刚刚写到其上的值。如果瑕疵非常小，比如说只有几位，那么使用坏扇区并且每次只是让ECC校正错误是可能的。如果瑕疵较大，那么错误就不可能被掩盖。

对于坏块存在两种一般的处理方法：在控制器中对它们进行处理或者在操作系统中对它们进行处理。在前一种方法中，磁盘在从工厂出厂之前要进行测试，并且将一个坏扇区列表写在磁盘上。对于每一个坏扇区，用一个备用扇区替换它。

有两种方法进行这样的替换。在图5-30a中，我们看到单个磁盘磁道，它具有30个数据扇区和两个备用扇区。扇区7是有瑕疵的。控制器所能够做的事情是将备用扇区之一重映射为扇区7，如图5-30b所示。另一种方法是将所有扇区向上移动一个扇区，如图5-30c所示。在这两种情况下，控制器都必须知道哪个扇区是哪个扇区。它可以通过内部的表来跟踪这一信息（每个磁道一张表），或者通过重写前导码来给出重映射的扇区号。如果是重写前导码，那么图5-30c的方法就要做更多的工作（因为23个前导码必须重写），但是最终会提供更好的性能，因为整个磁道仍然可以在旋转一周中读出。

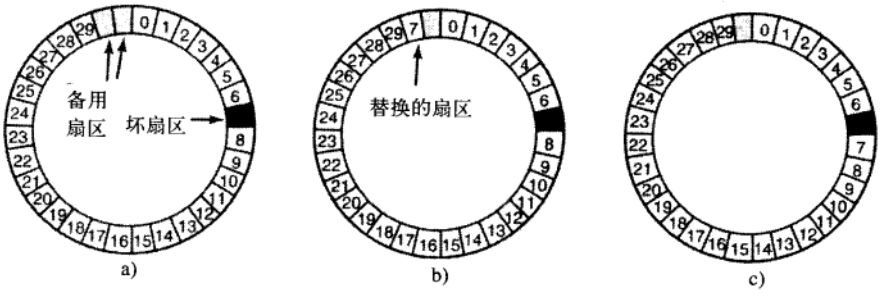


图5-30 a) 具有一个坏扇区的磁盘磁道；b) 用备用扇区替换坏扇区；c) 移动所有扇区以回避坏扇区

驱动器安装之后在正常工作期间也会出现错误。在遇到ECC不能处理的错误时，第一道防线只是试图再次读。某些读错误是瞬时性的，也就是说是由磁头下的灰尘导致的，在第二次尝试时错误就消失了。如果控制器注意到它在某个扇区遇到重复性的错误，那么可以在该扇区完全死掉之前切换到一个备用扇区。这样就不会丢失数据并且操作系统和用户甚至都不会注意到这一问题。通常使用的是图5-30b的方法，因为其他扇区此刻可能包含数据。而使用图5-30c的方法则不但要重写前导码，还要复制所有的数据。

前面我们曾说过存在两种一般的处理错误的方法：在控制器中或者在操作系统中处理错误。如果控制器不具有像我们已经讨论过的那样透明地重映射扇区的能力，那么操作系统必须在软件中做同样的事情。这意味着操作系统必须首先获得一个坏扇区列表，或者是通过从磁盘中读出该列表，或者只是由它自己测试整个磁盘。一旦操作系统知道哪些扇区是坏的，它就可以建立重映射表。如果操作系统想使用图5-30c的方法，它就必须将扇区7到扇区29中的数据向上移动一个扇区。

如果由操作系统处理重映射，那么它必须确保坏扇区不出现在任何文件中，并且不出现在空闲列表或位图中。做到这一点的一种方法是创建一个包含所有坏扇区的秘密的文件。如果该文件不被加入文件系统，用户就不会意外地读到它（或者更糟糕地，释放它）。

然而，还存在另一个问题：备份。如果磁盘是一个文件一个文件地做备份，那么非常重要的是备份实用程序不去尝试复制坏块文件。为了防止发生这样的事情，操作系统必须很好地隐藏坏块文件，以至于备份实用程序也不能发现它。如果磁盘是一个扇区一个扇区地做备份而不是一个文件一个文件地做备份，那么在备份期间防止读错误是十分困难的，如果不是不可能的话。惟一的希望是备份程序具有足够的智能，在读失败10次后放弃并且继续下一个扇区。

坏扇区不是惟一的错误来源，也可能发生磁盘臂中的机械故障引起的寻道错误。控制器内部跟踪着

磁盘臂的位置，为了执行寻道，它发出一系列脉冲给磁盘臂电机，每个柱面一个脉冲，这样将磁盘臂移到新的柱面。当磁盘臂移到其目标位置时，控制器从下一个扇区的前导码中读出实际的柱面号。如果磁盘臂在错误的位置上，则发生寻道错误。

大多数硬盘控制器可以自动纠正寻道错误，但是大多数软盘控制器（包括Pentium的）只是设置一个错误标志位而把余下的工作留给驱动程序。驱动程序对这一错误的处理办法是发出一个recalibrate（重新校准）命令，让磁盘臂尽可能地向最外面移动，并将控制器内部的当前柱面重置为0。通常这样就可以解决问题了。如果还不行，则只好修理驱动器。

正如我们已经看到的，控制器实际是一个专用的小计算机，它有软件、变量、缓冲区，偶尔还出现故障。有时一个不寻常的事件序列，例如一个驱动器发生中断的同时另一个驱动器发出recalibrate命令，就可能引发一个故障，导致控制器陷入一个循环或失去对正在做的工作的跟踪。控制器的设计者通常考虑到最坏的情形，在芯片上提供了一个引脚，当该引脚被置起时，迫使控制器忘记它正在做的任何事情并且将其自身复位。如果其他方法都失败了，磁盘驱动程序可以设置一个控制位以触发该信号，将控制器复位。如果还不成功，驱动程序所能做的就是打印一条消息并且放弃。

重新校准一块磁盘会发出古怪的噪音，但是正常工作时并不让人烦扰。然而，存在这样一种情形，对于具有实时约束的系统而言重新校准是一个严重的问题。当从硬盘播放视频时，或者当将文件从硬盘烧录到CD-ROM上时，来自硬盘的位流以均匀的速率到达是必需的。在这样的情况下，重新校准会在位流中插入间隙，因此是不可接受的。称为AV盘（Audio Visual disk，音视盘）的特殊驱动器永远不会重新校准，因而可用于这样的应用。

5.4.5 稳定存储器

正如我们已经看到的，磁盘有时会出现错误。好扇区可能突然变成坏扇区，整个驱动器也可能出乎意料地死掉。RAID可以对几个扇区出错或者整个驱动器崩溃提供保护。然而，RAID首先不能对将坏数据写下的写错误提供保护，并且也不能对写操作期间的崩溃提供保护，这样就会破坏原始数据而不能以更新的数据替换它们。

对于某些应用而言，决不丢失或破坏数据是绝对必要的，即使面临磁盘和CPU错误也是如此。理想的情况是，磁盘应该始终没有错误地工作。但是，这是做不到的。所能够做到的是，一个磁盘子系统具有如下特性：当一个写命令发给它时，磁盘要么正确地写数据，要么什么也不做，让现有的数据完整无缺地留下。这样的系统称为*稳定存储器*（stable storage），并且是在软件中实现的（Lampson和Sturgis，1979）。目标是不惜一切代价保持磁盘的一致性。下面我们将描述这种最初思想的一个微小的变体。

在描述算法之前，重要的是对于可能发生的错误有一个清晰的模型。该模型假设在磁盘写一个块（一个或多个扇区）时，写操作要么是正确的，要么是错误的，并且该错误可以在随后的读操作中通过检查ECC域的值检测出来。原则上，保证错误检测是根本不可能的，这是因为，假如使用一个16字节的ECC域保护一个512字节的扇区，那么存在着2^{4096}个数据值而仅有2^{144}个ECC值。因此，如果一个块在写操作期间出现错误但是ECC没有出错，那么存在着几亿亿个错误的组合可以产生相同的ECC。如果某些这样的错误出现，则错误不会被检测到。大体上，随机数据具有正确的16字节ECC的概率大约是2^{-144}。该概率值足够小以至于我们可以视其为零，尽管它实际上并不为零。

该模型还假设一个被正确写入的扇区可能会自发地变坏并且变得不可读。然而，该假设是：这样的事件非常少见，以至于在合理的时间间隔内（例如1天）让相同的扇区在第二个（独立的）驱动器上变坏的概率小到可以忽略的程度。

该模型还假设CPU可能出故障，在这样的情况下只能停机。在出现故障的时刻任何处于进行中的磁盘写操作也会停止，导致不正确的数据写在一个扇区中并且后来可能会检测到不正确的ECC。在所有这些情况下，稳定存储器就写操作而言可以提供100%的可靠性，要么就正确地工作，要么就让旧的数据原封不动。当然，它不能对物理灾难提供保护，例如，发生地震，计算机跌落100m掉入一个裂缝并且陷入沸腾的岩浆池中，在这样的情况下用软件将其恢复是勉为其难的。

稳定存储器使用一对完全相同的磁盘，对应的块一同工作以形成一个无差错的块。当不存在错误时，在两个驱动器上对应的块是相同的，读取任意一个都可以得到相同的结果。为了达到这一目的，定义了

下述三种操作：

1) 稳定写（stable write）。稳定写首先将块写到驱动器1上，然后将其读回以校验写的是正确的。如果写的不正确，那么就再次做写和重读操作，一直到n次，直到正常为止。经过n次连续的失败之后，就将该块重映射到一个备用块上，并且重复写和重读操作直到成功为止，无论要尝试多少个备用块。在对驱动器1的写成功之后，对驱动器2上对应的块进行写和重读，如果需要的话就重复这样的操作，直到最后成功为止。如果不存在CPU崩溃，那么当稳定写完成后，块就正确地被写到两个驱动器上，并且在两个驱动器上得到校验。

2) 稳定读（stable read）。稳定读首先从驱动器1上读取块。如果这一操作产生错误的ECC，则再次尝试读操作，一直到n次。如果所有这些操作都给出错误的ECC，则从驱动器2上读取对应的数据块。给定一个成功的稳定写为数据块留下两个可靠的副本这样的事实，并且我们假设在合理的时间间隔内相同的块在两个驱动器上自发地变坏的概率可以忽略不计，那么稳定读就总是成功的。

3) 崩溃恢复（crash recovery）。崩溃之后，恢复程序扫描两个磁盘，比较对应的块。如果一对块都是好的并且是相同的，就什么都不做。如果其中一个具有ECC错误，那么坏块就用对应的好块来覆盖。如果一对块都是好的但是不相同，那么就将驱动器1上的块写到驱动器2上。

如果不存在CPU崩溃，那么这一方法总是可行的，因为稳定写总是对每个块写下两个有效的副本，并且假设自发的错误决不会在相同的时刻发生在两个对应的块上。如果在稳定写期间出现CPU崩溃会怎么样？这就取决于崩溃发生的精确时间。有5种可能性，如图5-31所示。

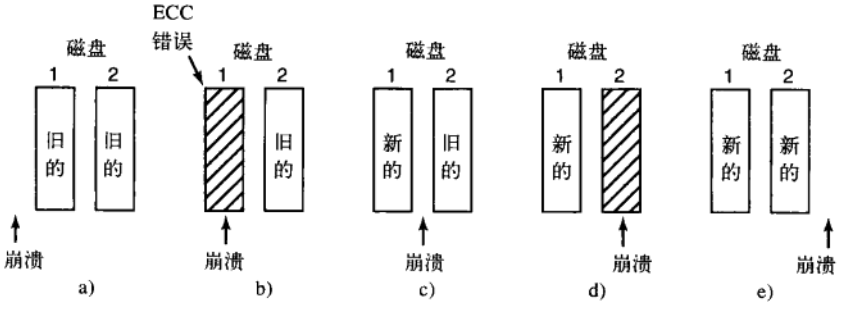


图5-31 崩溃对于稳定写的影响的分析

在图5-31a中，CPU崩溃发生在写块的两个副本之前。在恢复的时候，什么都不用修改而旧的值将继续存在，这是允许的。

在图5-31b中，CPU崩溃发生在写驱动器1期间，破坏了该块的内容。然而恢复程序能够检测出这一错误，并且从驱动器2恢复驱动器1上的块。因此，这一崩溃的影响被消除并且旧的状态完全被恢复。

在图5-31c中，CPU崩溃发生在写完驱动器1之后但是还没有写驱动器2之前。此时已经过了无法复原的时刻：恢复程序将块从驱动器1复制到驱动器2上。写是成功的。

图5-31d与图5-31b相类似：在恢复期间用好的块覆盖坏的块。不同的是，两个块的最终取值都是新的。

最后，在图5-31e中，恢复程序看到两个块是相同的，所以什么都不用修改并且在此处写也是成功的。

对于这一模式进行各种各样的优化和改进都是可能的。首先，在崩溃之后对所有的块两个两个地进行比较是可行的，但是代价高昂。一个巨大的改进是在稳定写期间跟踪被写的是哪个块，这样在恢复的时候必须被检验的块就只有一个。某些计算机拥有少量的非易失性RAM（nonvolatile RAM），它是一个特殊的CMOS存储器，由锂电池供电。这样的电池能够维持很多年，甚至有可能是计算机的整个生命周期。与主存不同（它在崩溃之后就丢失了），非易失性RAM在崩溃之后并不丢失。每天的时间通常就保存在这里（并且通过一个特殊的电路进行增值），这就是为什么计算机即使在拔掉电源之后仍然知道是什么时间。

假设非易失性RAM的几个字节可供操作系统使用，稳定写就可以在开始写之前将准备要更新的块的编号放到非易失性RAM里。在成功地完成稳定写之后，在非易失性RAM中的块编号用一个无效的块编号（例如−1）覆盖掉。在这些情形下，崩溃之后恢复程序可以检验非易失性RAM以了解在崩溃期间

是否有一个稳定写正在进行中，如果是的话，还可以了解在崩溃发生的时候被写的是哪一个块。然后，可以对块的两个副本进行正确性和一致性检验。

如果没有非易失性RAM可用，可以对它模拟如下。在稳定写开始时，用将要被稳定写的块的编号覆盖驱动器1上的一个固定的块，然后读回该块以对其进行校验。在使得该块正确之后，对驱动器2上对应的块进行写和校验。当稳定写正确地完成时，用一个无效的块编号覆盖两个块并进行校验。这样一来，崩溃之后就很容易确定在崩溃期间是否有一个稳定写正在进行中。当然，这一技术为了写一个稳定的块需要8次额外的磁盘操作，所以应该极少量地应用该技术。

还有最后一点值得讨论。我们假设每天每一对块只发生一个好块自发损坏成为坏块。如果经过足够长的时间，另一个块也可能变坏。因此，为了修复任何损害每天必须对两块磁盘进行一次完整的扫描。这样，每天早晨两块磁盘总是一模一样的。即便在一个时期内一对中的两个块都坏了，所有的错误也都能正确地修复。

5.5 时钟

时钟（clock）又称为定时器（timer），由于各种各样的原因决定了它对于任何多道程序设计系统的操作都是至关重要的。时钟负责维护时间，并且防止一个进程垄断CPU，此外还有其他的功能。时钟软件可以采用设备驱动程序的形式，尽管时钟既不像磁盘那样是一个块设备，也不像鼠标那样是一个字符设备。我们对时钟的研究将遵循与前面几节相同的模式：首先考虑时钟硬件，然后考虑时钟软件。

5.5.1 时钟硬件

在计算机里通常使用两种类型的时钟，这两种类型的时钟与人们使用的钟表和手表有相当大的差异。比较简单的时钟被连接到110V或220V的电源线上，这样每个电压周期产生一个中断，频率是50Hz或60Hz。这些时钟过去曾经占据统治地位，但是如今却非常罕见。

另一种类型的时钟由三个部件构成：晶体振荡器、计数器和存储寄存器，如图5-32所示。当把一块石英晶体适当地切割并且安装在一定的压力之下时，它就可以产生非常精确的周期性信号，典型的频率范围是几百兆赫兹，具体的频率值与所选的晶体有关。使用电子器件可以将这一基础信号乘以一个小的整数来获得高达1000MHz甚至更高的频率。在任何一台计算机里通常都可以找到至少一个这样的电路，它给计算机的各种电路提供同步信号。该信号被送到计数器，使其递减计数至0。当计数器变为0时，产生一个CPU中断。

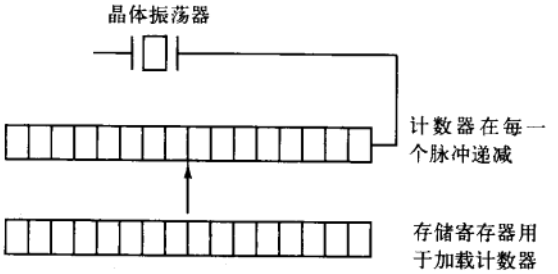


图5-32 可编程时钟

可编程时钟通常具有几种操作模式。在*一次完成模式*（one-shot mode）下，当时钟启动时，它把存储寄存器的值复制到计数器中，然后，来自晶体的每一个脉冲使计数器减1。当计数器变为0时，产生一个中断，并停止工作，直到软件再一次显式地启动它。在*方波模式*（square-wave mode）下，当计数器变为0并且产生中断之后，存储寄存器的值自动复制到计数器中，并且整个过程无限期地再次重复下去。这些周期性的中断称为*时钟滴答*（clock tick）。

可编程时钟的优点是其中断频率可以由软件控制。如果采用500MHz的晶体，那么计数器将每隔2ns脉动一次。对于（无符号）32位寄存器，中断可以被编程为从2ns时间间隔发生一次到8.6s时间间隔发生一次。可编程时钟芯片通常包含两个或三个独立的可编程时钟，并且还具有许多其他选项（例如，用正计时代替倒计时、屏蔽中断等）。

为了防止计算机的电源被切断时丢失当前时间，大多数计算机具有一个由电池供电的备份时钟，它是由在数字手表中使用的那种类型的低功耗电路实现的。电池时钟可以在系统启动的时候读出。如果不存在备份时钟，软件可能会向用户询问当前日期和时间。对于一个连入网络的系统而言还有一种从远程主机获取当前时间的标准方法。无论是哪种情况，当前时间都要像UNIX所做的那样转换成自1970年1月

1日上午12时UTC（Universal Time Coordinated，协调世界时，以前称为格林威治平均时）以来的时钟滴答数，或者转换成自某个其他标准时间以来的时钟滴答数。Windows的时间原点是1980年1月1日。每一次时钟滴答都使实际时间增加一个计数。通常会提供实用程序来手工设置系统时钟和备份时钟，并且使两个时钟保持同步。

5.5.2 时钟软件

时钟硬件所做的全部工作是根据已知的时间间隔产生中断。涉及时间的其他所有工作都必须由软件——时钟驱动程序完成。时钟驱动程序的确切任务因操作系统而异，但通常包括下面的大多数任务：

1) 维护日时间。

2) 防止进程超时运行。

3) 对CPU的使用情况记账。

4) 处理用户进程提出的alarm系统调用。

5) 为系统本身的各个部分提供监视定时器。

6) 完成概要剖析、监视和统计信息收集。

时钟的第一个功能是维持正确的日时间，也称为实际时间（real time），这并不难实现，只需要如前面提到的那样在每个时钟滴答将计数器加1即可。惟一要当心的事情是日时间计数器的位数，对于一个频率为60Hz的时钟来说，32位的计数器仅仅超过2年就会溢出。很显然，系统不可能在32位中按照自1970年1月1日以来的时钟滴答数来保存实际时间。

可以采取三种方法来解决这一问题。第一种方法是使用一个64位的计数器，但这样做使维护计数器的代价很高，因为1秒内需要做很多次维护计数器的工作。第二种方法是以秒为单位维护日时间，而不是以时钟滴答为单位，该方法使用一个辅助计数器来对时钟滴答计数，直到累计完整的一秒。因为2^{32}秒超过了136年，所以该方法可以工作到22世纪。

第三种方法是对时钟滴答计数，但是这一计数工作是相对于系统引导的时间，而不是相对于一个固定的外部时间。当读入备份时钟或者用户输入实际时间时，系统引导时间就从当前日时间开始计算，并且以任何方便的形式存放在内存中。以后，当请求日时间时，存储的日时间值加到计数器上就可以得到当前的日时间。所有这三种方法如图5-33所示。

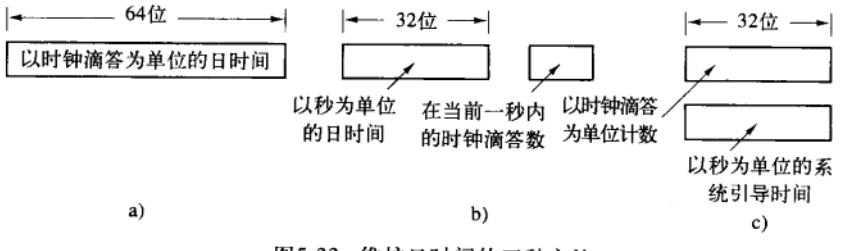


图5-33 维护日时间的三种方法

时钟的第二个功能是防止进程超时运行。每当启动一个进程时，调度程序就将一个计数器初始化为以时钟滴答为单位的该进程时间片的取值。每次时钟中断时，时钟驱动程序将时间片计数器减1。当计数器变为0时，时钟驱动程序调用调度程序以激活另一个进程。

时钟的第三个功能是CPU记账。最精确的记账方法是，每当一个进程启动时，便启动一个不同于主系统定时器的辅助定时器。当进程终止时，读出这个定时器的值就可以知道该进程运行了多长时间。为了正确地记账，当中断发生时应该将辅助定时器保存起来，中断结束后再将其恢复。

一个不太精确但更加简单的记账方法是在一个全局变量中维护一个指针，该指针指向进程表中当前运行的进程的表项。在每一个时钟滴答，使当前进程的表项中的一个域加1。通过这一方法，每个时钟滴答由在该滴答时刻运行的进程“付费”。这一策略的一个小问题是：如果在一个进程运行过程中多次发生中断，即使该进程没有做多少工作，它仍然要为整个滴答付费。由于在中断期间恰当地对CPU进行记账的方法代价过于昂贵，因此很少使用。

在许多系统中，进程可以请求操作系统在一定的时间间隔之后向它报警。警报通常是信号、中断、消息或者类似的东西。需要这类报警的一个应用是网络，当一个数据包在一定时间间隔之内没有被确认时，该数据包必须重发。另一个应用是计算机辅助教学，如果学生在一定时间内没有响应，就告诉他答案。

如果时钟驱动程序拥有足够的时钟，它就可以为每个请求设置一个单独的时钟。如果不是这样的情况，它就必须用一个物理时钟来模拟多个虚拟时钟。一种方法是维护一张表，将所有未完成的定时器的信号时刻记入表中，还要维护一个变量给出下一个信号的时刻。每当日时间更新时，时钟驱动程序进行检查以了解最近的信号是否已经发生。如果是的话，则在表中搜索下一个要发生的信号的时刻。

如果预期有许多信号，那么通过在一个链表中把所有未完成的时钟请求按时间排序链接在一起，这样来模拟多个时钟则更为有效，如图5-34所示。链表中的每个表项指出在前一个信号之后等待多少时钟滴答引发下一个信号。在本例中，等待处理的信号对应的时钟滴答分别是4203、4207、4213、4215和4216。

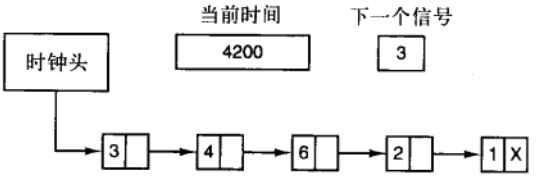


图5-34 用单个时钟模拟多个定时器

在图5-34中，经过3个时钟滴答发生下一个中断。每一次滴答时，下一个信号减1，当它变为0时，就引发与链表中第一个表项相对应的信号，并将这一表项从链表中删除，然后将下一个信号设置为现在处于链表头的表项的取值，在本例中是4。

注意在时钟中断期间，时钟驱动程序要做几件事情——将实际时间增1，将时间片减1并检查它是否为0，对CPU记账，以及将报警计数器减1。然而，因为这些操作在每一秒之中要重复许多次，所以每个操作都必须仔细地安排以加快速度。

操作系统的组成部分也需要设置定时器，这些定时器被称为监视定时器（watchdog timer）[⊖]。例如，为了避免磨损介质和磁头，软盘在不使用时是不旋转的。当数据需要从软盘读出时，电机必须首先启动。只有当软盘以全速旋转时，I/O才可以开始。当一个进程试图从一个空闲的软盘读取数据时，软盘驱动程序启动电机然后设置一个监视定时器以便在足够长的时间间隔之后引发一个中断（因为不存在来自软盘本身的达到速度的中断）。

时钟驱动程序用来处理监视定时器的机制和用于用户信号的机制是相同的。惟一的区别是当一个定时器时间到时，时钟驱动程序将调用一个由调用者提供的过程，而不是引发一个信号。这个过程是调用者代码的一部分。被调用的过程可以做任何需要做的工作，甚至可以引发一个中断，但是在内核之中中断通常是不方便的并且信号也不存在。这就是为什么要提供监视定时器机制。值得注意的是，只有当时钟驱动程序与被调用的过程处于相同的地址空间时，监视定时器机制才起作用。

时钟最后要做的事情是剖析（profiling）。某些操作系统提供了一种机制，通过该机制用户程序可以让系统构造它的程序计数器的一个直方图，这样它就可以了解时间花在了什么地方。当剖析是可能的事情时，在每一时钟滴答驱动程序都要检查当前进程是否正在被进行剖析，如果是，则计算对应于当前程序计数器的区间（bin）[⊜]号（一段地址范围），然后将该区间的值加1。这一机制也可用来对系统本身进行剖析。

5.5.3 软定时器

大多数计算机拥有辅助可编程时钟，可以设置它以程序需要的任何速率引发定时器中断。该定时器是主系统定时器以外的，而主系统定时器的功能已经在上面讲述了。只要中断频率比较低，将这个辅助定时器用于应用程序特定的目的就不存在任何问题。但是当应用程序特定的定时器的频率非常高时，麻烦就来了。下面我们将简要描述一个基于软件的定时器模式，它在许多情况下性能良好，甚至在相当高的频率

⊖ watchdog timer也经常译为看门狗定时器。——译者注

⊜ 直方图（histogram）用于描述随机变量取值分布的情况，虽然在中文术语中有一个“图”字，但并不是必须用图形来表示。它将随机变量（对于本例而言是程序计数器的取值）的值空间（对于本例而言是进程的地址空间）划分成若干个小的区间，每个小区间就是一个bin。通过计算随机变量的取值落在每个小区间内的次数就可以得到直方图。如果用图形表示直方图的话则表现为一系列高度不等的柱状图形。——译者注

下也是如此。这一思想起因于（Aron和Druschel，1999）。关于更详细的细节，请参阅他们的论文。

一般而言，有两种方法管理I/O：中断和轮询。中断具有较低的等待时间，也就是说，它们在事件本身之后立即发生，具有很少的延迟或者没有延迟。另一方面，对于现代CPU而言，由于需要上下文切换以及对于流水线、TLB和高速缓存的影响，中断具有相当大的开销。

替代中断的是让应用程序对它本身期待的事件进行轮询。这样做避免了中断，但是可能存在相当长的等待时间，因为一个事件可能正好发生在一次轮询之后，在这种情况下它就要等待几乎整个轮询间隔。平均而言，等待时间是轮询间隔的一半。

对于某些应用而言，中断的开销和轮询的等待时间都是不能接受的。例如，考虑一个高性能的网络，如千兆位以太网。该网络能够每12μs接收或者发送一个全长的数据包。为了以优化的输出性能运行，每隔12μs就应该发出一个数据包。

达到这一速率的一种方法是当一个数据包传输完成时引发一个中断，或者将辅助定时器设置为每12μs中断一次。问题是在一个300 MHz的Pentium II计算机上该中断经实测要花费4.45μs的时间（Aron和Druschel，1999）。这样的开销比20世纪70年代的计算机好不了多少。例如，在大多数小型机上，一个中断要占用4个总线周期：将程序计数器和PSW压入堆栈并且加载一个新的程序计数器和PSW。现如今涉及流水线、MMU、TLB和高速缓存，更是增加了大量的开销。这些影响可能在时间上使情况变得更坏而不是变得更好，因此抵消了更快的时钟速率。

软定时器（soft timer）避免了中断。无论何时当内核因某种其他原因在运行时，在它返回到用户态之前，它都要检查实时时钟以了解软定时器是否到期。如果这个定时器已经到期，则执行被调度的事件（例如，传送数据包或者检查到来的数据包），而无需切换到内核态，因为系统已经在内核态。在完成工作之后，软定时器被复位以便再次闹响。要做的全部工作是将当前时钟值复制给定时器并且将超时间隔加上。

软定时器随着因为其他原因进入内核的频率而脉动。这些原因包括：

1) 系统调用。

2) TLB未命中。

3) 页面故障。

4) I/O中断。

5) CPU变成空闲。

为了了解这些事件发生得有多频繁，Aron和Druschel对于几种CPU负载进行了测量，包括全负载Web服务器、具有计算约束后台作业的Web服务器、从因特网上播放实时音频以及重编译UNIX内核。进入内核的平均进入率在2μs到1μs之间变化，其中大约一半是系统调用。因此，对于一阶近似，让一个软定时器每隔2μs闹响一次是可行的，虽然这样做偶尔会错过最终时限。对于发送数据包或者轮询到来的数据包这样的应用而言，有时可能晚10μs比让中断消耗35%的CPU时间要好。

当然，可能有一段时间不存在系统调用、TLB未命中或页面故障，在这些情况下，没有软定时器会闹响。为了在这些时间间隔上设置一个最大值，可以将辅助硬件定时器设置为每隔一定时间（例如1ms）闹响一次。如果应用程序对于偶然的时间间隔能够忍受每秒只有1000个数据包，那么软定时器和低频硬件定时器的组合可能比纯粹的中断驱动I/O或者纯粹的轮询要好。

5.6 用户界面：键盘、鼠标和监视器

每台通用计算机都配有一个键盘和一个监视器（并且通常还有一只鼠标），使人们可以与之交互。尽管键盘和监视器在技术上是独立的设备，但是它们紧密地一同工作。在大型机上，通常存在许多远程用户，每个用户拥有一个设备，该设备包括一个键盘和一个连在一起的显示器作为一个单位。这些设备在历史上被称为*终端*（terminal）。人们通常继续使用该术语，即便是讨论个人计算机时（主要是因为缺乏更好的术语）。

5.6.1 输入软件

用户输入主要来自键盘和鼠标，所以我们要了解它们。在个人计算机上，键盘包含一个嵌入式微处理器，该微处理器通过一个特殊的串行端口与主板上的控制芯片通信（尽管键盘越来越多地连接到USB端口上）。每当一个键被按下的时候都会产生一个中断，并且每当一个键被释放的时候还会产生第二个

中断。在发生每个这样的键盘中断时，键盘驱动程序都要从与键盘相关联的I/O端口提取信息，以了解发生了什么事情。其他的一切事情都是在软件中发生的，在相当大的程度上独立于硬件。

当想象往shell窗口（命令行界面）键入命令时，可以更好地理解本小节余下的大部分内容。这是程序员通常的工作方式。我们将在下面讨论图形界面。

1. 键盘软件

I/O端口中的数字是键编号，称为*扫描码*（scan code），而不是ASCII码。键盘所拥有的键不超过128个，所以只需7个位表示键编号。当键按下时，第8位设置为0，当键释放时，第8位设置为1。跟踪每个键的状态（按下或弹起）是驱动程序的任务。

例如，当A键被按下时，扫描码（30）被写入一个I/O寄存器。驱动程序应该负责确定键入的是小写字母、大写字母、CTRL-A、ALT-A、CTRL-ALT-A还是某些其他组合。由于驱动程序可以断定哪些键已经按下但是还没有被释放（例如SHIFT），所以它拥有足够多的信息来做这一工作。

例如，击键序列

按下SHIFT，按下A，释放A，释放SHIFT

指示的是大写字母A。然而击键序列

按下SHIFT，按下A，释放SHIFT，释放A

指示的也是大写字母A。尽管该键盘接口将所有的负担都加在软件上，但是却极其灵活。例如，用户程序可能对刚刚键入的一个数字是来自顶端的一排键还是来自边上的数字键盘感兴趣。原则上，驱动程序能够提供这一信息。

键盘驱动程序可以采纳两种可能的处理方法。在第一种处理方法中，驱动程序的工作只是接收输入并且不加修改地向上层传送。这样，从键盘读数据的程序得到的是ASCII码的原始序列。（向用户程序提供扫描码过于原始，并且高度地依赖于机器。）

这种处理方法非常适合于像emacs那样的复杂屏幕编辑器的需要，它允许用户对任意字符或字符序列施加任意的动作。然而，这意味着如果用户键入的是dste而不是date，为了修改错误而键入三个退格键和ate，然后是一个回车键，那么提供给用户程序的是键入的全部11个ASCII码，如下所示：

dste← ← ←ate CR

并非所有的程序都想要这么多的细节，它们常常只想要校正后的输入，而不是如何产生它的准确的序列。这一认识导致了第二种处理方法：键盘驱动程序处理全部行内编辑，并且只将校正后的行传送给用户程序。第一种处理方法是面向字符的；第二种处理方法是面向行的。最初它们分别被称为*原始模式*（raw mode）和*加工模式*（cooked mode）。POSIX标准使用稍欠生动的*术语规范模式*（canonical mode）来描述面向行的模式。*非规范模式*（noncanonical mode）与原始模式是等价的，尽管终端行为的许多细节可能被修改了。POSIX兼容的系统提供了若干库函数，支持选择这两种模式中的一种并且修改许多参数。

如果键盘处于规范（加工）模式，则字符必须存储起来直到积累完整的一行，因为用户随后可能决定删除一行中的一部分。即使键盘处于原始模式，程序也可能尚未请求输入，所以字符也必须缓冲起来以便允许用户提前键入。可以使用专用的缓冲区，或者缓冲区也可以从池中分配。前者对提前键入提出了固定的限制，后者则没有。当用户在shell窗口（Windows的命令行窗口）中击键并且刚刚发出一条尚未完成的命令（例如编译）时，将引起尖锐的问题。后继键入的字符必须被缓冲，因为shell还没有准备好读它们。那些不允许用户提前键入的系统设计者应该被涂柏油、粘羽毛[⊖]，或者更加严重的惩罚是，强迫他们使用他们自己设计的系统。

虽然键盘与监视器在逻辑上是两个独立的设备，但是很多用户已经习惯于看到他们刚刚键入的字符出现在屏幕上。这个过程叫做*回显*（echoing）。

当用户正在击键的时候程序可能正在写屏幕，这一事实使回显变得错综复杂（请再一次想象在shell

⊖ 原文为be tarred and feathered，是英国古代的一种酷刑。受刑人全身涂上灼热的柏油（tarred），然后将其身上粘满羽毛（feathered）。这样，羽毛当然很难脱下，要脱下也难免皮肉之伤。be tarred and feathered现用于比喻受到严厉惩罚。——译者注

窗口中击键)。最起码，键盘驱动程序必须解决在什么地方放置新键入的字符而不被程序的输出所覆盖。

当超过80个字符必须在具有80字符行（或某个其他数字）的窗口中显示时，也使回显变得错综复杂。根据应用程序，折行到下一行可能是适宜的。某些驱动程序只是通过丢弃超出80列的所有字符而将每行截断到80个字符。

另一个问题是制表符的处理。通常由驱动程序来计算光标当前定位在什么位置，它既要考虑程序的输出又要考虑回显的输出，并且要计算要回显的正确的空格个数。

现在我们讨论设备等效性问题。逻辑上，在一个文本行的结尾，人们需要一个回车和一个换行，回车使光标移回到第一列，换行使光标前进到下一行。要求用户在每一行的结尾键入回车和换行是不受欢迎的。这就要求驱动程序将输入转化成操作系统使用的格式。在UNIX中，ENTER键被转换成一个换行用于内部存储；而在Windows中，它被转换成一个回车跟随一个换行。

如果标准形式只是存储一个换行（UNIX约定），那么回车（由Enter键造成）应该转换为换行。如果内部格式是存储两者（Windows约定），那么驱动程序应该在得到回车时生成一个换行并且在得到换行时生成一个回车。不管是什么内部约定，监视器可能要求换行和回车两者都回显，以便正确地更新屏幕。在诸如大型计算机这样的多用户系统上，不同的用户可能拥有不同类型的终端连接到大型计算机上，这就要求键盘驱动程序将所有不同的回车/换行组合转换成内部系统标准并且安排好正确地实现回显。

字 符	POSIX名	注 释
CTRL-H	ERASE	退格一个字符
CTRL-U	KILL	擦除正在键入的整行
CTRL-V	LNEXT	按字面意义解释下一个字符
CTRL-S	STOP	停止输出
CTRL-Q	START	开始输出
DEL	INTR	中断进程（SIGINT）
CTRL-\	QUIT	强制核心转储（SIGQUIT）
CTRL-D	EOF	文件结尾
CTRL-M	CR	回车（不可修改的）
CTRL-J	NL	换行（不可修改的）

图5-35 在规范模式下特殊处理的字符

在规范模式下操作时，许多输入字符具有特殊的含义。图5-35显示出了POSIX要求的所有特殊字符。默认的是所有控制字符，这些控制字符应该不与程序所使用的文本输入或代码相冲突，但是除了最后两个以外所有字符都可以在程序的控制下修改。

ERASE字符允许用户删除刚刚键入的字符。它通常是退格符（CTRL+H）。它并不添加到字符队列中，而是从队列中删除前一个字符。它应该被回显为三个字符的序列，即退格符、空格和退格符，以便从屏幕上删除前一个字符。如果前一个字符是制表符，那么删除它取决于当它被键入的时候是如何处理的。如果制表符直接展开成空格，那么就需要某些额外的信息来决定后退多远。如果制表符本身被存放在输入队列中，那么就可以将其删除并且重新输出整行。在大多数系统中，退格只删除当前行上的字符，不会删除回车并且后退到前一行。

当用户注意到正在键入的一行的开头有一个错误时，擦除一整行并且从头再来常常比较方便。KILL字符擦除一整行。大多数系统使被擦除的行从屏幕上消失，但是也有少数古老的系统回显该行并且加上一个回车和换行，因为有些用户喜欢看到旧的一行。因此，如何回显KILL是个人喜好问题。与ERASE一样，KILL通常也不可能从当前行进一步回退。当一个字符块被删除时，如果使用了缓冲，那么烦劳驱动程序将缓冲区退还给缓冲池可能值得做也可能不值得做。

有时ERASE或KILL字符必须作为普通的数据键入。LNEXT字符用作一个转义字符(escape character)。在UNIX中，CTRL＋V是默认的转义字符。例如，更加古老的UNIX系统常常使用@作为KILL字符，但是因特网邮件系统使用linda@cs.washington.edu形式的地址。有的人觉得老式的约定更加舒服从而将KILL重定义为@，但是之后又需要按字面意义键入一个@符号到电子邮件地址中。这可以通过键入CTRL+V @来实现。CTRL+V本身可以通过键入CTRL+V CTRL+V而按字面意义键入。看到一个CTRL+V之后，驱动程序设置一个标志，表示下一字符免除特殊处理。LNEXT字符本身并不进入字符队列。

为了让用户阻止屏幕图像滚动出视线，提供了控制码以便冻结屏幕并且之后重新开始滚动。在UNIX系统中，这些控制码分别是STOP（CTRL+S）和START（CTRL+Q）。它们并不被存储，只是用来设置或清除键盘数据结构中的一个标志。每当试图输出时，就检查这个标志。如果标志已设置，则不

输出。通常，回显也随程序输出一起被抑制。

杀死一个正在被调试的失控程序经常是有必要的，INTR（DEL）和QUIT（CTRL+\）字符可以用于这一目的。在UNIX中，DEL将SIGINT信号发送到从该键盘启动的所有进程。实现DEL是相当需要技巧的，因为UNIX从一开始就被设计成在同一时刻处理多个用户。因此，在一般情况下，可能存在多个进程代表多个用户在运行，而DEL键必须只能向用户自己的进程发信号。困难之处在于从驱动程序获得信息送给系统处理信号的那部分，后者毕竟还没有请求这个信息。

CTRL+\与DEL相类似，只是它发送的是SIGQUIT信号，如果这个信号没有被捕捉到或被忽略，则强迫进行核心转储。当敲击这些键中的任意一个键时，驱动程序应该回显一个回车和换行并且为了全新的开始而放弃累积的全部输入。INTR的默认值经常是CTRL+C而不是DEL，因为许多程序针对编辑操作可互换地使用DEL与退格符。

另一个特殊字符是EOF（CTRL+D）。在UNIX中，它使任何一个针对该终端的未完成的读请求以缓冲区中可用的任何字符来满足，即使缓冲区是空的。在一行的开头键入CTRL+D将使得程序读到0个字节，按惯例该字符被解释为文件结尾，并且使大多数程序按照它们在处理输入文件时遇到文件结尾的同样方法对其进行处理。

2. 鼠标软件

大多数PC机具有一个鼠标，或者具有一个跟踪球，跟踪球不过是躺在其背部上的鼠标。一种常见类型的鼠标在内部具有一个橡皮球，该橡皮球通过鼠标底部的一个圆洞突出，当鼠标在一个粗糙表面上移动时橡皮球会随着旋转。当橡皮球旋转时，它与放置在相互垂直的滚轴上的两个橡皮滚筒相摩擦。东西方向的运动导致平行于y轴的滚轴旋转，南北方向的运动导致平行于x轴的滚轴旋转。

另一种流行的鼠标类型是光学鼠标，它在其底部装备有一个或多个发光二极管和光电探测器。早期的光学鼠标必须在特殊的鼠标垫上操作，鼠标垫上刻有矩形的网格，这样鼠标能够计数穿过的线数。现代光学鼠标在其中有图像处理芯片并且获取处于它们下方的连续的低分辨率照片，寻找从图像到图像的变化。

当鼠标在随便哪个方向移动了一个确定的最小距离，或者按钮被按下或释放时，都会有一条消息发送给计算机。最小距离大约是0.1mm（尽管它可以在软件中设置）。有些人将这一单位称为一个鼠标步（mickey）。鼠标可能具有一个、两个或者三个按钮，这取决于设计者对于用户跟踪多个按钮的智力的估计。某些鼠标具有滚轮，可以将额外的数据发送回计算机。无线鼠标与有线鼠标相同，区别是无线鼠标使用低功率无线电，例如使用蓝牙（Bluetooth）标准将数据发送回计算机，而有线鼠标是通过导线将数据发送回计算机。

发送到计算机的消息包含三个项目：Δx、Δy、按钮，即自上一次消息之后x位置的变化、自上一次消息之后y位置的变化、按钮的状态。消息的格式取决于系统和鼠标所具有的按钮的数目。通常，消息占3字节。大多数鼠标返回报告最多每秒40次，所以鼠标自上一次报告之后可能移动了多个鼠标步。

注意，鼠标仅仅指出位置上的变化，而不是绝对位置本身。如果轻轻地拿起鼠标并且轻轻地放下而不导致橡皮球旋转，那么就不会有消息发出。

某些GUI区分单击与双击鼠标按钮。如果两次点击在空间上（鼠标步）足够接近，并且在时间上（毫秒）也足够接近，那么就会发出双击信号。最大的“足够接近”是软件的事情，并且这两个参数通常是用户可设置的。

5.6.2 输出软件

下面我们考虑输出软件。首先我们将讨论到文本窗口的简单输出，这是程序员通常喜欢使用的方式。然后，我们将考虑图形用户界面，这是其他用户经常喜欢使用的。

1. 文本窗口

当输出是连续的单一字体、大小和颜色的形式时，输出比输入简单。大体上，程序将字符发送到当前窗口，而字符在那里显示出来。通常，一个字符块或者一行是在一个系统调用中被写到窗口上的。

屏幕编辑器和许多其他复杂的程序需要能够以更加复杂的方式更新屏幕，例如在屏幕的中间替换一行。为满足这样的需要，大多数输出驱动程序支持一系列命令来移动光标，在光标处插入或者删除字符或行。这些命令常常被称为转义序列（escape sequence）。在25行80列ASCII哑终端的全盛期，有数百种终

端类型，每一种都有自己的转义序列。因而，编写在一种以上的终端类型上工作的软件是十分困难的。

一种解决方案是称为termcap的终端数据库，它是在伯克利UNIX中引入的。该软件包定义了许多基本动作，例如将光标移动到（行，列）。为了将光标移动到一个特殊的位置，软件（如一个编辑器）使用一个一般的转义序列，然后该转义序列被转换成将要被执行写操作的终端的实际转义序列。以这种方式，该编辑器就可以工作在任何具有termcap数据库入口的终端上。许多UNIX软件仍然以这种方式工作，即使在个人计算机上。

逐渐地，业界看到了转义序列标准化的需要，所以就开发了一个ANSI标准。图5-36所示为一些该标准的取值。

下面考虑文本编辑器怎样使用这些转义序列。假设用户键入了一条命令指示编辑器完全删除第3行，然后封闭第2行和第4行之间的间隙。编辑器可以通过串行线向终端发送如下的转义序列：

ESC [3;1 H ESC [0 K ESC [1 M

（其中在上面使用的空格只是为了分开符号，它们并不传送）。这一序列将光标移动到第3行的开头，擦除整个一行，然后删除现在的空行，使从第4行开始的所有行向上移动一行。现在，第4行变成了第3行，第5行变成了第4行，以此类推。类似的转义序列可以用来在显示器的中间添加文本。字和字符可以以类似的方式添加或删除。

转义序列	含　义
ESC [*n*A	向上移动*n*行
ESC [*n*B	向下移动*n*行
ESC [*n*C	向右移动*n*个间隔
ESC [*n*D	向左移动*n*个间隔
ESC [*m*;*n*H	将光标移动到（*m*,*n*）
ESC [*s*J	从光标清除屏幕（0到结尾、1从开始、2两者）
ESC [*s*K	从光标清除行（0到结尾、1从开始、2两者）
ESC [*n*L	在光标处插入*n*行
ESC [*n*M	在光标处删除*n*行
ESC [*n*P	在光标处删除*n*个字符
ESC [*n*@	在光标处插入*n*个字符
ESC [*n*m	允许再现*n*（0=常规、4=粗体、5=闪烁、7=反白）
ESC M	如果光标在顶行上则向后滚动屏幕

图5-36　终端驱动程序在输出时接受的ANSI转义序列。ESC表示ASCII转义字符（0x1B），*n*、*m*和*s*是可选的数值参数

2. X窗口系统

几乎所有UNIX系统的用户界面都以X窗口系统（X Window System）为基础，X窗口系统经常仅称为X，它是作为Athena计划[⊖]的一部分于20世纪80年代在MIT开发的。X窗口系统具有非常好的可移植性，并且完全运行在用户空间中。人们最初打算将其用于将大量的远程用户终端与中央计算服务器相连接，所以它在逻辑上分成客户软件和主机软件，这样就有可能运行在不同的计算机上。在现代个人计算机上，两部分可以运行在相同的机器上。在Linux系统上，流行的Gnome和KDE桌面环境就运行在X之上。

当X在一台机器上运行时，从键盘或鼠标采集输入并且将输出写到屏幕上的软件称为X服务器（X server）。它必须跟踪当前选择了哪个窗口（鼠标指针所在处），这样它就知道将新的键盘输入发送给哪个客户。它与称为X客户（X client）的运行中的程序进行通信（可能通过网络）。它将键盘与鼠标输入发送给X客户，并且从X客户接收显示命令。

⊖　Athena（雅典娜）指麻省理工学院（MIT）校园范围内基于UNIX的计算环境。——译者注

X服务器总是位于用户的计算机内部，而X客户有可能在远方的远程计算服务器上，这看起来也许有些不可思议，但是X服务器的主要工作是在屏幕上显示位，所以让它靠近用户是有道理的。从程序的观点来看，它是一个客户，吩咐服务器做事情，例如显示文本和几何图形。服务器（在本地PC中）只是做客户吩咐它做的事情，就像所有服务器所做的那样。

对于X客户和X服务器在不同机器上的情形，客户与服务器的布置如图5-37所示。但是当在单一的机器上运行Gnome或者KDE时，客户只是使用X库与相同机器上的X服务器进行会话的某些应用程序（但是通过套接字使用TCP连接，与远程情形中所做的工作相同）。

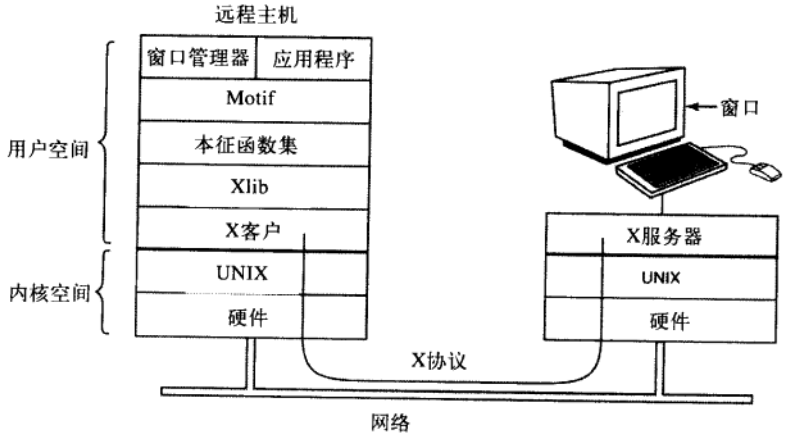


图5-37 MIT X窗口系统中的客户与服务器

在单机上或者通过网络在UNIX（或其他操作系统）之上运行X窗口系统都是可行的，其原因在于X实际上定义的是X客户与X服务器之间的X协议，如图5-37所示。客户与服务器是在同一台机器上，还是通过一个局域网隔开了100m，或者是相距几千公里并且通过Internet相连接都无关紧要。在所有这些情况下，协议与系统操作都是完全相同的。

X只是一个窗口系统，它不是一个完全的GUI。为了获得完全的GUI，要在其上运行其他软件层。一层是Xlib，它是一组库过程，用于访问X的功能。这些过程形成了X窗口系统的基础，我们将在下面对其进行分析，但是这些过程过于原始了，以至于大多数用户程序不能直接访问它们。例如，每次鼠标点击是单独报告的，所以确定两次点击实际上形成了双击必须在Xlib之上处理。

为了使得对X的编程更加容易，作为X的一部分提供了一个工具包，组成了Intrinsics（本征函数集）。这一层管理按钮、滚动条以及其他称为窗口小部件（widget）的GUI元素。为了产生真正的GUI界面，具有一致的外观与感觉，还需要另外一层软件（或者几层软件）。一个例子是Motif，如图5-37所示，它是Solaris和其他商业UNIX系统上使用的公共桌面环境（Common Desktop Environment）的基础。大多数应用程序利用的是对Motif的调用，而不是对Xlib的调用。Gnome和KDE具有与图5-37相类似的结构，只是库有所不同。Gnome使用GTK+库，KDE使用Qt库。拥有两个GUI是否比一个好是有争议的。

此外，值得注意的是窗口管理并不是X本身的组成部分。将其遗漏的决策完全是故意的。一个单独的客户进程，称为窗口管理器（window manager），控制着屏幕上窗口的创建、删除以及移动。为了管理窗口，窗口管理器要发送命令到X服务器告诉它做什么。窗口管理器经常运行在与X客户相同的机器上，但是理论上它可以运行在任何地方。

这一模块化设计，包括若干层和多个程序，使得X高度可移植和高度灵活。它已经被移植到UNIX的大多数版本上，包括Solaris、BSD的所有派生版本、AIX、Linux等，这就使得对于应用程序开发人员来说在多种平台上拥有标准的用户界面成为可能。它还被移植到其他操作系统上。相反，在Windows中，窗口与GUI系统在GDI中混合在一起并且处于内核之中，这使得它们维护起来十分困难，并且当然是不可移植的。

现在让我们像是从Xlib层观察那样来简略地看一看X。当一个X程序启动时，它打开一个到一个或

多个X服务器（我们称它们为工作站）的连接，即使它们可能与X程序在同一台机器上。在消息丢失与重复由网络软件来处理的意义上，X认为这一连接是可靠的，并且它不用担心通信错误。通常在服务器与客户之间使用的是TCP/IP。

四种类型的消息通过连接传递：

1) 从程序到工作站的绘图命令。

2) 工作站对程序请求的应答。

3) 键盘、鼠标以及其他事件的通告。

4) 错误消息。

从程序到工作站的大多数绘图命令是作为单向消息发送的，不期望应答。这样设计的原因是当客户与服务器进程在不同的机器上时，命令到达服务器并且执行要花费相当长的时间周期。在这一时间内阻塞应用程序将不必要地降低其执行速度。另一方面，当程序需要来自工作站的信息时，它只好等待直到应答返回。

与Windows类似，X是高度事件驱动的。事件从工作站流向程序，通常是为响应人的某些行动，例如键盘敲击、鼠标移动或者一个窗口被显现。每个事件消息32个字节，第一个字节给出事件类型，下面的31个字节提供附加的信息。存在许多种类的事件，但是发送给一个程序的只有那些它宣称愿意处理的事件。例如，如果一个程序不想得知键释放的消息，那么键释放的任何事件都不会发送给它。与在Windows中一样，事件是排成队列的，程序从队列中读取事件。然而，与Windows不同的是，操作系统绝对不会主动调用在应用程序之内的过程，它甚至不知道哪个过程处理哪个事件。

X中的一个关键概念是资源（resource）。资源是一个保存一定信息的数据结构。应用程序在工作站上创建资源。在工作站上，资源可以在多个进程之间共享。资源的存活期往往很短，并且当工作站重新启动后资源不会继续存在。典型的资源包括窗口、字体、颜色映射（调色板）、像素映射（位图）、光标以及图形上下文。图形上下文用于将属性与窗口关联起来，在概念上与Windows的设备上下文相类似。

X程序的一个粗略的、不完全的框架如图5-38所示。它以包含某些必需的头文件开始，之后声明某些变量。然后，它与X服务器连接，X服务器是作为XOpenDisplay的参数设定的。接着，它分配一个窗

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                              /* 服务器标识符 */
    Window win;                                /* 窗口标识符 */
    GC gc;                                     /* 图形上下文标识符 */
    XEvent event;                              /* 用于存储一个事件 */
    int running = 1;

    disp = XOpenDisplay("display_name");       /* 连接到X服务器 */
    win = XCreateSimpleWindow(disp, ... );     /* 为新窗口分配内存 */
    XSetStandardProperties(disp, ...);         /* 向窗口管理器宣布窗口 */
    gc = XCreateGC(disp, win, 0, 0);           /* 创建图形上下文 */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);                     /* 显示窗口；发送Expose事件 */

    while (running) {
        XNextEvent(disp, &event);              /* 获得下一个事件 */
        switch (event.type) {
            case Expose:        ...;  break;   /* 重绘窗口 */
            case ButtonPress:   ...;  break;   /* 处理鼠标点击 */
            case Keypress:      ...;  break;   /* 处理键盘输入 */
        }
    }

    XFreeGC(disp, gc);                         /* 释放图形上下文 */
    XDestroyWindow(disp, win);                 /* 回收窗口的内存空间 */
    XCloseDisplay(disp);                       /* 拆卸网络连接 */
}
```

图5-38 X窗口应用程序的框架

口资源并且将指向该窗口资源的句柄存放在win中。实际上，一些初始化应该出现在这里，在初始化之后X程序通知窗口管理器新窗口的存在，因而窗口管理器能够管理它。

对XCreateGC的调用创建一个图形上下文，窗口的属性就存放在图形上下文中。在一个更加复杂的程序中，窗口的属性应该在这里被初始化。下一条语句对XSelectInput的调用通知X服务器程序准备处理哪些事件，在本例中，程序对鼠标点击、键盘敲击以及窗口被显现感兴趣。实际上，一个真正的程序还会对其他事件感兴趣。最后，对XMapRaised的调用将新窗口作为最顶层的窗口映射到屏幕上。此时，窗口在屏幕上成为可见的。

主循环由两条语句构成，并且在逻辑上比Windows中对应的循环要简单得多。此处，第一条语句获得一个事件，第二条语句对事件类型进行分派从而进行处理。当某个事件表明程序已经结束的时候，running被设置为0，循环结束。在退出之前，程序释放了图形上下文、窗口和连接。

值得一提的是，并非每个人都喜欢GUI。许多程序员更喜欢上面5.6.2节讨论的那种传统的面向命令行的界面。X通过一个称为xterm的客户程序解决了这一问题。该程序仿真了一台古老的VT102智能终端，完全具有所有的转义序列。因此，编辑器（例如vi和emacs）以及其他使用termcap的软件无需修改就可以在这些窗口中工作。

3. 图形用户界面

大多数个人计算机提供了GUI（Graphical User Interface，图形用户界面）。首字母缩写词GUI的发音是“gooey”。

GUI是由斯坦福研究院的Douglas Engelbart和他的研究小组发明的。之后GUI被Xerox PARC的研究人员摹仿。在一个风和日丽的日子，Apple公司的共同创立者Steve Jobs参观了PARC，并且在一台Xerox计算机上见到了GUI。这使他产生了开发一种新型计算机的想法，这种新型计算机就是Apple Lisa。Lisa因为太过昂贵因而在商业上是失败的，但是它的后继者Macintosh获得了巨大的成功。

当Microsoft得到Macintosh的原型从而能够在其上开发Microsoft Office时，Microsoft请求Apple发放界面许可给所有新来者，这样Macintosh就能够成为新的业界标准。（Microsoft从Office获得了比MS-DOS多得多的收入，所以它愿意放弃MS-DOS以获得更好的平台用于Office。）Apple负责Macintosh的主管Jean-Louis Gassée拒绝了Microsoft的请求，并且Steve Jobs已经离开了Apple而不能否决他。最终，Microsoft得到了界面要素的许可证，这形成了Windows的基础。当Microsoft开始追上Apple时，Apple提起了对Microsoft的诉讼，声称Microsoft超出了许可证的界限，但是法官并不认可，并且Windows继续追赶并超过了Macintosh。如果Gassée同意Apple内部许多人的看法（他们也希望将Macintosh软件许可给任何人），那么Apple或许会因为许可费而变得无限富有，并且现在就不会存在Windows了。

GUI具有用字符WIMP表示的四个基本要素，这些字母分别代表窗口（Window）、图标（Icon）、菜单（Menu）和定点设备（Pointing device）。窗口是一个矩形块状的屏幕区域，用来运行程序。图标是小符号，可以在其上点击导致某个动作发生。菜单是动作列表，人们可以从中进行选择。最后，定点设备是鼠标、跟踪球或者其他硬件设备，用来在屏幕上移动光标以便选择项目。

GUI软件可以在用户级代码中实现（如UNIX系统所做的那样），也可以在操作系统中实现（如Windows的情况）。

GUI系统的输入仍然使用键盘和鼠标，但是输出几乎总是送往特殊的硬件电路板，称为图形适配器（graphics adapter）。图形适配器包含特殊的内存，称为视频RAM（video RAM），它保存出现在屏幕上的图像。高端的图形适配器通常具有强大的32位或64位CPU和多达1GB自己的RAM，独立于计算机的主存。

每个图形适配器支持几种屏幕尺寸。常见的尺寸是1024×768、1280×960、1600×1200和1920×1200。除了1920×1200以外，所有这些尺寸的宽高比都是4:3，符合NTSC和PAL电视机的屏幕宽高比，因此可以在用于电视机的相同的监视器上产生正方形的像素。1920×1200尺寸意在用于宽屏监视器，它的宽高比与这一分辨率相匹配。在最高的分辨率下，每个像素具有24位的彩色显示，只是保存图像就需要大约6.5MB的RAM，所以，拥有256MB或更多的RAM，图形适配器就能够一次保存许多图像。如果整个屏幕每秒刷新75次，那么视频RAM必须能够连续地以每秒489MB的速率发送数据。

GUI的输出软件是一个巨大的主题。单是关于Windows GUI就写下了许多1500多页的书（例如

Petzold，1999；Simon，1997；Rector 和Newcomer，1997）。显然，在这一小节中，我们只可能浅尝其表面并且介绍少许基本的概念。为了使讨论具体化，我们将描述Win32 API，它被Windows的所有32位版本所支特。在一般意义上，其他GUI的输出软件大体上是相似的，但是细节迥然不同。

屏幕上的基本项目是一个矩形区域，称为窗口（window）。窗口的位置和大小通过给定两个斜对角的坐标（以像素为单位）惟一地决定。窗口可以包含一个标题条、一个菜单条、一个工具条、一个垂直滚动条和一个水平滚动条。典型的窗口如图5-39所示。注意，Windows的坐标系将原点置于左上角并且y向下增长，这不同于数学中使用的笛卡儿坐标。

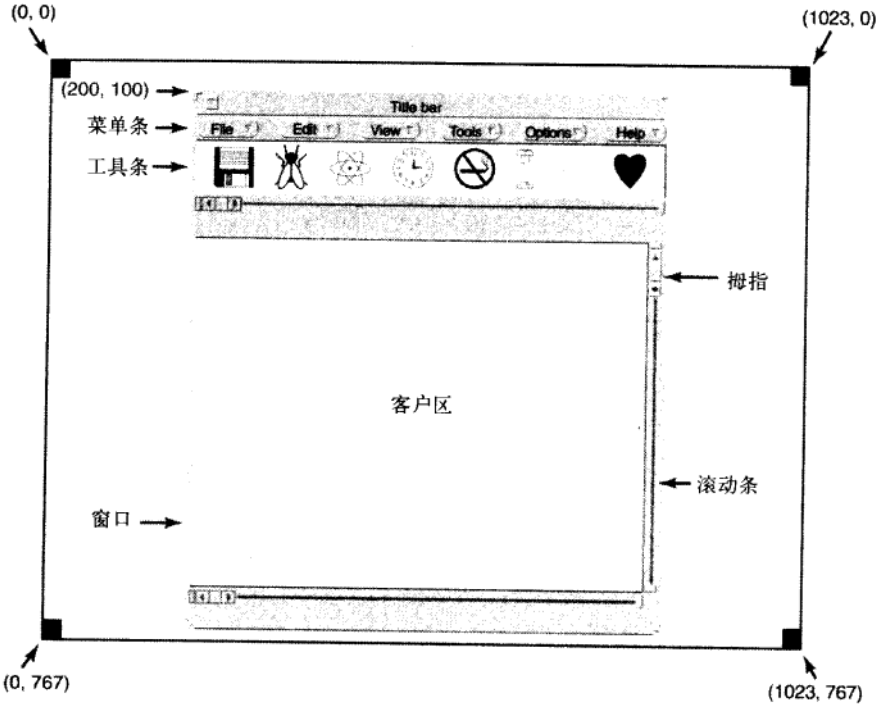


图5-39 XGA显示器上位于（200, 100）处的一个窗口样例

当窗口被创建时，有一些参数可以设定窗口是否可以被用户移动，是否可以被用户调整大小，或者是否可以被用户滚动（通过拖动滚动条上的拇指）。大多数程序产生的主窗口可以被移动、调整大小和滚动，这对于Windows程序的编写方式具有重大的意义。特别地，程序必须被告知关于其窗口大小的改变，并且必须准备在任何时刻重画其窗口的内容，即使在程序最不期望的时候。

因此，Windows程序是面向消息的。涉及键盘和鼠标的用户操作被Windows所捕获，并且转换成消息，送到正在被访问的窗口所属于的程序。每个程序都有一个消息队列，与程序的所有窗口相关的消息都被发送到该队列中。程序的主循环包括提取下一条消息，并且通过调用针对该消息类型的内部过程对其进行处理。在某些情况下，Windows本身可以绕过消息队列而直接调用这些过程。这一模型与UNIX的过程化代码模型完全不同，UNIX模型是提请系统调用与操作系统相互作用的。然而，X是面向事件的。

为了使这一编程模型更加清晰，请考虑图5-40的例子。在这里我们看到的是Windows主程序的框架，它并不完整并且没有做错误检查，但是对于我们的意图而言它显示了足够的细节。程序的开头包含一个头文件windows.h，它包含许多宏、数据类型、常数、函数原型，以及Windows程序所需要的其他信息。

主程序以一个声明开始，该声明给出了它的名字和参数。WINAPI宏是一条给编译器的指令，让编译器使用一定的参数传递约定并且不需要我们进一步关心。第一个参数h是一个实例句柄，用来向系统的其他部分标识程序。在某种程度上，Win32是面向对象的，这意味着系统包含对象（例如程序、文件

和窗口）。对象具有状态和相关的代码，而相关的代码称为*方法*（method），它对于状态进行操作。对象是使用句柄来引用的，在该示例中，h标识的是程序。第二个参数只是为了向后兼容才出现的，它已不再使用。第三个参数szCmd是一个以零终止的字符串，包含启动该程序的命令行，即使程序不是从命令行启动的。第四个参数iCmdShow表明程序的初始窗口应该占据整个屏幕，占据屏幕的一部分，还是一点也不占据屏幕（只是任务条）。

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass;              /* 本窗口的类对象 */
    MSG msg;                        /* 进入的消息存放在这里 */
    HWND hwnd;                      /* 窗口对象的句柄（指针） */

    /* 初始化wndclass*/
    wndclass.lpfnWndProc = WndProc;          /* 指示调用哪个过程 */
    wndclass.lpszClassName = "Program name"; /* 标题条的文本 */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);  /* 装载程序图标 */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);    /* 装载鼠标光标 */

    RegisterClass(&wndclass);       /* 向Windows注册wndclass */
    hwnd = CreateWindow ( ... )     /* 为窗口分配存储 */
    ShowWindow(hwnd, iCmdShow); /* 在屏幕上显示窗口 */
    UpdateWindow(hwnd);             /* 指示窗口绘制自身 */

    while (GetMessage(&msg, NULL, 0, 0)) {   /* 从队列中获取消息 */
        TranslateMessage(&msg);   /* 转换消息 */
        DispatchMessage(&msg);    /* 将msg发送给适当的过程 */
    }
    return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
   /* 这里是声明 */

    switch (message) {
        case WM_CREATE:    ... ;   return ... ;   /* 创建窗口 */
        case WM_PAINT:     ... ;   return ... ;   /* 重绘窗口的内容 */
        case WM_DESTROY:   ... ;   return ... ;   /* 销毁窗口 */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam));/* 默认 */
}
```

图5-40 Windows主程序的框架

该声明说明了一个广泛采用的Microsoft约定，称为*匈牙利记号*（Hungarian notation）。该名称是一个涉及波兰记号的双关语，波兰记号是波兰逻辑学家J. Lukasiewicz发明的后缀系统，用于不使用优先级和括号表示代数公式。匈牙利记号是Microsoft的一名匈牙利程序员Charles Simonyi发明的，它使用标识符的前几个字符来指定类型。允许的字母和类型包括c（character，字符）、w（word，字，现在意指无符号16位整数）、i（integer，32位有符号整数）、l（long，也是一个32位有符号整数）、s（string，字符串）、sz（string terminated by a zero byte，以零字节终止的字符串）、p（pointer，指针）、fn（function，函数）和h（handle，句柄）。因此，举例来说，szCmd是一个以零终止的字符串并且iCmdShow是一个整数。许多程序员认为在变量名中像这样对类型进行编码没有什么价值，并且使Windows代码异常地难于阅读。在UNIX中就没有类似这样的约定。

每个窗口必须具有一个相关联的类对象定义其属性，在图5-40中，类对象是wndclass。对象类型WNDCLASS具有10个字段，其中4个字段在图5-40中被初始化，在一个以实际的程序中，其他6个字段也要被初始化。最重要的字段是lpfnWndProc，它是一个指向函数的长（即32位）指针，该函数处理引向该窗口的消息。此处被初始化的其他字段指出在标题条中使用哪个名字和图标，以及对于鼠标光标使

用哪个符号。

在wndclass被初始化之后，RegisterClass被调用，将其发送给Windows。特别地，在该调用之后Windows就会知道当各种事件发生时要调用哪个过程。下一个调用CreateWindow为窗口的数据结构分配内存并且返回一个句柄以便以后引用它。然后，程序做了另外两个调用，将窗口轮廓置于屏幕之上，并且最终完全地填充窗口。

此刻我们到达了程序的主循环，它包括获取消息，对消息做一定的转换，然后将其传回Windows以便让Windows调用WndProc来处理它。要回答这一完整的机制是否能够得到化简的问题，答案是肯定的，但是这样做是由于历史的缘故，并且我们现在坚持这样做。

主循环之后是过程WndProc，它处理发送给窗口的各种消息。此处CALLBACK的使用与上面的WINAPI相类似，为参数指明要使用的调用序列。第一个参数是要使用的窗口的句柄。第二个参数是消息类型。第三和第四个参数可以用来在需要的时候提供附加的信息。

消息类型WM_CREATE和WM_DESTROY分别在程序的开始和结束时发送。它们给程序机会为数据结构分配内存，并且将其返回。

第三个消息类型WM_PAINT是一条指令，让程序填充窗口。它不仅当窗口第一次绘制时被调用，而且在程序执行期间也经常被调用。与基于文本的系统相反，在Windows中程序不能够假定它在屏幕上画的东西将一直保持在那里直到将其删除。其他窗口可能会被拖拉到该窗口的上面，菜单可能会在窗口上被拉下，对话框和工具提示可能会覆盖窗口的某一部分，如此等等。当这些项目被移开后，窗口必须重绘。Windows告知一个程序重绘窗口的方法是发送WM_PAINT消息。作为一种友好的姿态，它还会提供窗口的哪一部分曾经被覆盖的信息，这样程序就更加容易重新生成窗口的那一部分而不必重绘整个窗口。

Windows有两种方法可以让一个程序做某些事情。一种方法是投递一条消息到其消息队列。这种方法用于键盘输入、鼠标输入以及定时器到时。另一种方法是发送一条消息到窗口，从而使Windows直接调用WndProc本身。这一方法用于所有其他事件。由于当一条消息完全被处理后Windows会得到通报，这样Windows就能够避免在前一个调用完成前产生新的调用，由此可以避免竞争条件。

还有许多其他消息类型。当一个不期望的消息到达时为了避免异常行为，最好在WndProc的结尾处调用DefWindowProc，让默认处理过程处理其他情形。

总之，Windows程序通常创建一个或多个窗口，每个窗口具有一个类对象。与每个程序相关联的是一个消息队列和一组处理过程。最终，程序的行为由到来的事件驱动，这些事件由处理过程来处理。与UNIX采用的过程化观点相比，这是一个完全不同的世界观模型。

对屏幕的实际绘图是由包含几百个过程的程序包处理的，这些过程捆在一起形成了GDI（Graphics Device Interface，图形设备接口）。它能够处理文本和各种类型的图形，并且被设计成与平台和设备无关的。在一个程序可以在窗口中绘图（即绘画）之前，它需要获取一个设备上下文（device context）：设备上下文是一个内部数据结构，包含窗口的属性，诸如当前字体、文本颜色、背景颜色等。大多数GDI调用使用设备上下文，不管是为了绘图，还是为了获取或设置属性。

有许许多多的方法可用来获取设备上下文。下面是一个获取并使用设备上下文的简单例子：

```
hdc=GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

第一条语句获取一个设备上下文的句柄hdc。第二条语句使用设备上下文在屏幕上写一行文本，该语句设定了字符串开始处的（x, y）坐标、一个指向字符串本身的指针以及字符串的长度。第三个调用释放设备上下文，表明程序在当时已通过了绘图操作。注意，hdc的使用方式与UNIX的文件描述符相类似。还需要注意的是，ReleaseDC包含冗余的信息（使用hdc就可以惟一地指定一个窗口）。使用不具有实际价值的冗余信息在Windows中是很常见的。

另一个有趣的注意事项是，当hdc以这样的方式被获取时，程序只能够写窗口的客户区，而不能写标题条和窗口的其他部分。在内部，在设备上下文的数据结构中，维护着一个修剪区域。在修剪区域之外的任何绘图操作都将被忽略。然而，存在着另一种获取设备上下文的方法GetWindowDC，它将修剪

区域设置为整个窗口。其余的调用以其他的方法限定修剪区域。拥有多种调用做几乎相同的事情是Windows的另一个特性。

GDI的完全论述超出了这里讨论的范围。对于感兴趣的读者，上面引用的参考文献提供了补充的信息。然而，关于GDI可能还值得再说几句话，因为GDI是如此之重要。GDI具有各种各样的过程调用以获取和释放设备上下文，获取关于设备上下文的信息，获取和设置设备上下文的属性（例如背景颜色），使用GDI对象（例如画笔、画刷和字体，其中每个对象都有自己的属性）。最后，当然存在许多实际在屏幕上绘图的GDI调用。

绘图过程分成四种类型：绘制直线和曲线、绘制填充区域、管理位图以及显示文本。我们在上面看到了绘制文本的例子，所以让我们快速地看看其他类型之一。调用

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

将绘制一个填充的矩形，它的左上角和右下角分别是（xleft, ytop）和（xright, ybottom）。例如，

```
Rectangle(hdc,2,1,6,4);
```

将绘制一个如图5-41所示的矩形。线宽和颜色以及填充颜色取自设备上下文。其他的GDI调用在形式上是类似的。

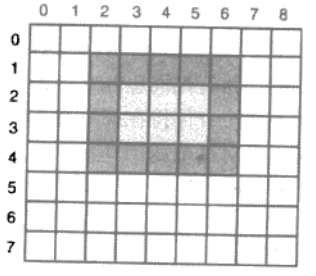


图5-41 使用Rectangle绘制矩形的例子。每个方框代表一个像素

4. 位图

GDI过程是矢量图形学的实例。它们用于在屏幕上放置几何图形和文本。它们能够十分容易地缩放到较大和较小的屏幕（如果屏幕上的像素数是相同的）。它们还是相对设备无关的。一组对GDI过程的调用可以聚集在一个文件中，描述一个复杂的图画。这样的文件称为Windows元文件（metafile），广泛地用于从一个Windows程序到另一个Windows程序传送图画。这样的文件具有扩展名.wmf。

许多Windows程序允许用户复制图画（或一部分）并且放在Windows的剪贴板上，然后用户可以转入另一个程序并且粘贴剪贴板的内容到另一个文档中。做这件事的一种方法是由第一个程序将图画表示为Windows元文件并且将其以.wmf格式放在剪贴板上。此外，还有其他的方法做这件事。

并不是计算机处理的所有图像都能够使用矢量图形学来生成。例如，照片和视频就不使用矢量图形学。反之，这些项目可以通过在图像上覆盖一层网格扫描输入。每一个网格方块的平均红、绿、蓝取值被采样并且保存为一个像素的值。这样的文件称为位图（bitmap）。Windows中有大量的工具用于处理位图。

位图的另一个用途是用于文本。在某种字体中表示一个特殊字符的一种方法是将其表示为小的位图。于是往屏幕上添加文本就变成移动位图的事情。

使用位图的一种一般方法是通过调用BitBlt过程，该过程调用如下：

```
BitBlt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

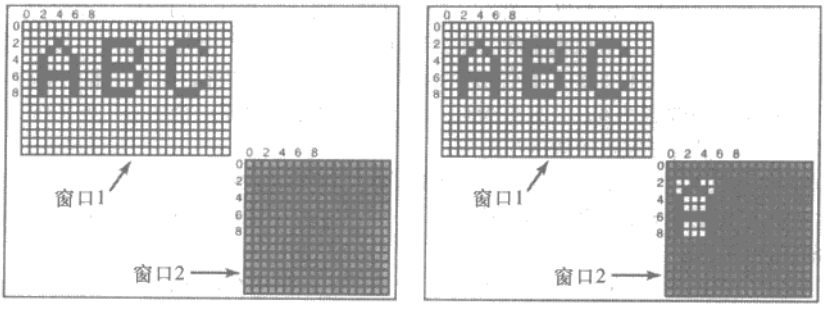
在其最简单的形式中，该过程从一个窗口中的一个矩形复制位图到另一个窗口（或同一个窗口）的一个矩形中。前三个参数设定目标窗口和位置，然后是宽度和高度，接下来是源窗口和位置。注意，每个窗口都有其自己的坐标系，（0，0）在窗口的左上角处。最后一个参数将在下面描述。

```
BitBlt(hdc2, 1, 2, 5, 7, hdcl, 2, 2, SRCCOPY);
```

的效果如图5-42所示。注意字母A的整个5×7区域被复制了，包括背景颜色。

除了复制位图外，BitBlt还可以做很多事情。最后一个参数提供了执行布尔运算的可能，从而可以将源位图与目标位图合并在一起。例如，源位图可以与目标位图执行或运算，从而融入目标位图；源位图还可以与目标位图执行异或运算，该运算保持了源位图和目标位图的特征。

位图具有的一个问题是它们不能缩放。8×12方框内的一个字符在640×480的显示器上看起来是适度的。然而，如果该位图以每英寸1200点复制到10 200位×13 200位的打印页面上，那么字符宽度（8像素）为8/1200英寸或0.17mm。此外，在具有不同彩色属性的设备之间进行复制，或者在单色设备与彩色设备之间进行复制效果并不理想。



a) b)

图5-42 使用BitBlt复制位图：a) 复制前；b) 复制后

由于这样的缘故，Windows还支持一个称为DIB（Device Independent Bitmap，设备无关的位图）的数据结构。采用这种格式的文件使用扩展名.bmp。这些文件在像素之前具有文件与信息头以及一个颜色表，这样的信息使得在不同的设备之间移动位图十分容易。

5. 字体

在Windows 3.1版之前的版本中，字符表示为位图，并且使用BitBlt复制到屏幕上或者打印机上。这样做的问题是，正如我们刚刚看到的，在屏幕上有意义的位图对于打印机来说太小了。此外，对于每一尺寸的每个字符，需要不同的位图。换句话说，给定字符A的10点阵字型的位图，没有办法计算它的12点阵字型。因为每种字体的每一个字符可能都需要从4点到120点范围内的各种尺寸，所以需要的位图的数目是巨大的。整个系统对于文本来说简直是太笨重了。



图5-43 不同点阵尺寸的字符轮廓的一些例子

该问题的解决办法是TrueType字体的引入，TrueType字体不是位图而是字符的轮廓。每个TrueType字符是通过围绕其周界的一系列点来定义的，所有的点都是相对于（0，0）原点。使用这一系统，放大或者缩小字符是十分容易的，必须要做的全部事情只是将每个坐标乘以相同的比例因子。采用这种方法，TrueType字符可以放大或者缩小到任意的点阵尺寸，甚至是分数点阵尺寸。一旦给定了适当的尺寸，各个点可以使用幼儿园教的著名的逐点连算法连接起来（注意现代幼儿园为了更加光滑的结果而使用曲线尺）。轮廓完成之后，就可以填充字符了。图5-43给出了某些字符缩放到三种不同点阵尺寸的一个例子。

一旦填充的字符在数学形式上是可用的，就可以对它进行栅格化，也就是说，以任何期望的分辨率将其转换成位图。通过首先缩放然后栅格化，我们可以肯定显示在屏幕上的字符与出现在打印机上的字符将是尽可能接近的，差别只在于量化误差。为了进一步改进质量，可以在每个字符中嵌入表明如何进行栅格化的线索。例如，字母T顶端的两个衬线应该是完全相同的，否则由于舍入误差可能就不是这样的情况了。

5.7 瘦客户机

多年来，主流计算范式一直在中心化计算和分散化计算之间振荡。最早的计算机（例如ENIAC）虽然是庞然大物，但实际上是个人计算机，因为一次只有一个人能够使用它。然后出现的是分时系统，在分时系统中许多远程用户在简单的终端上共享一个大型的中心计算机。接下来是PC时代，在这一阶段用户再次拥有他们自己的个人计算机。

虽然分散化的PC模型具有长处，但是它也有着某些严重的不利之处，人们刚刚开始认真思考这些

不利之处。或许最大的问题是，每台PC机都有一个大容量的硬盘以及复杂的软件必须维护。例如，当操作系统的一个新版本发布时，必须做大量的工作分别在每台机器上进行升级。在大多数公司中，做这类软件维护的劳动力成本大大高于实际的硬件与软件成本。对于家庭用户而言，在技术上劳动力是免费的，但是很少有人能够正确地做这件事，并且更少有人乐于做这件事。对于一个中心化的系统，只有一台或几台机器必须升级，并且有专家班子做这些工作。

一个相关的问题是，用户应该定期地备份他们的几吉字节的文件系统，但是很少有用户这样做。当灾难袭来时，相随的将是仰天长叹和捶胸顿足。对于一个中心化的系统，自动化的磁带机器人在每天夜里都可以做备份。

中心化系统的另一个长处是资源共享更加容易。一个系统具有256个远程用户，每个用户拥有256MB RAM，在大多数时间这个系统的这些RAM大多是空闲的，然而某些用户临时需要大量的RAM但是却得不到，因为RAM在别人的PC上。对于一个具有64GB RAM的中心化系统，这样的事情决不会发生。同样的论据对于磁盘空间和其他资源也是有效的。

最后，我们将开始考察从以PC为中心的计算到以Web为中心的计算的转移。一个领域是电子邮件，在该领域中这种转移是长远的。人们过去获取投送到他们家庭计算机上的电子邮件，并且在家庭计算机上阅读。今天，许多人登录到Gmail、Hotmail或者Yahoo上，并且在那里阅读他们的邮件。下一步人们会登录到其他网站中，进行字处理、建立电子数据表以及做其他过去需要PC软件才能做的事情。最后甚至有可能人们在自己的PC上运行的惟一软件是一个Web浏览器，或许甚至没有软件。

一个合理的结论大概是：大多数用户想要高性能的交互式计算，但是实在不想管理一台计算机。这一结论导致研究人员重新研究了分时系统使用的哑终端（现在文雅地称为*瘦客户机*（thin client)），它们符合现代终端的期望。X是这一方向的一个步骤并且专用的X终端一度十分流行，但是它们现在已经失宠，因为它们的价格与PC相仿，能做的事情更少，并且仍然需要某些软件维护。圣杯（holy grail）应该是一个高性能的交互式计算系统，在该系统中用户的机器根本就没有软件。十分有趣的是，这一目标是可以达到的。下面我们将描述一个这样的瘦客户机系统，称为THINC，它是由哥伦比亚大学的研究人员开发的（Baratto等人，2005；Kim 等人，2006；Lai和Nieh，2006）。

此处的基本思想是从客户机剥离一切智能和软件，只是将其用作一台显示器，使所有计算（包括建立待显示的位图）都在服务器端完成。客户机和服务器之间的协议只是通知显示器如何更新视频RAM，再无其他。两端之间的协议中使用了五条命令，它们列在图5-44中。

命令	描述
Raw	在给定的位置显示原始像素数据
Copy	复制帧缓冲器区域到指定的坐标
Sfill	以给定的像素颜色值填充一个区域
Pfill	以给定的像素模式填充一个区域
Bitmap	使用位图图像填充一个区域

图5-44 THINC协议显示命令

现在我们将考察这些命令。Raw用于传输像素数据并且将它们逐字地显示在屏幕上。原则上，这是惟一需要的命令。其他命令只是为了优化。

Copy指示显示器从其视频RAM的一个部分移动数据到另一个部分。这对于滚卷屏幕而不必重新传输所有数据是有用的。

Sfill以单一的像素值填充屏幕的一个区域。许多屏幕具有某种颜色的一致的背景，该命令用于首先生成背景，然后可以绘制文本、图标和其他项目。

Pfill在某个区域上复制一个模式。它还可以用于背景，但是某些背景比单一颜色要复杂一些，在这种情况下，该命令可以完成工作。

最后，Bitmap也是用于绘制区域，但是具有前景色和背景色。总而言之，这些是非常简单的命令，在客户端需要非常少的软件。所有建立位图填充屏幕的复杂操作都是在服务器上完成的。为了改进效率，多条命令可以聚集成单一的数据包，通过网络从服务器传送到客户机。

在服务器端，图形程序使用高级命令以绘制屏幕。这些命令被THINC软件截获，并且翻译成可以发送到客户机的命令。命令可能要重排序以改进效率。

论文通过在距客户机10～10 000km距离的服务器上运行众多的常用应用程序，给出了大量的性能

测量。一般而言，性能超过了其他广域网系统，即使对于实时视频也是如此。关于更多的信息，请读者参阅论文。

5.8 电源管理

第一代通用电子计算机ENIAC具有180 00个电子管并且消耗140 000瓦的电力。结果，它迅速积累起非同一般的电费账单。晶体管发明后，电力的使用量戏剧性地下降，并且计算机行业失去了在电力需求方面的兴趣。然而，如今电源管理由于若干原因又像过去一样成为焦点，并且操作系统在这里扮演着重要的角色。

我们从桌面PC开始讨论。桌面PC通常具有200瓦的电源（其效率一般是85%，15%进来的能量损失为热量）。如果全世界1亿台这样的机器同时开机，合起来它们要用掉20 000兆瓦的电力。这是20座中等规模的核电站的总产出。如果电力需求能够削减一半，我们就可以削减10座核电站。从环保的角度看，削减10座核电站（或等价数目的矿物燃料电站）是一个巨大的胜利，非常值得追求。

另一个要着重考虑电源的场合是电池供电的计算机，包括笔记本电脑、掌上机以及Web便笺簿等。问题的核心是电池不能保存足够的电荷以持续非常长的时间，至多也就是几个小时。此外，尽管电池公司、计算机公司和消费性电子产品公司进行了巨大的研究努力，但进展仍然缓慢。对于一个已经习惯于每18个月性能翻一番（摩尔定律）的产业来说，毫无进展就像是违背了物理定律，但这就是现状。因此，使计算机使用较少的能量因而现有的电池能够持续更长的时间就高悬在每个人的议事日程之上。操作系统在这里扮演着主要的角色，我们将在下面看到这一点。

在最低的层次，硬件厂商试图使他们的电子装置具有更高的能量效率。使用的技术包括减少晶体管的尺寸、利用动态电压调节、使用低摆幅并隔热的总线以及类似的技术。这些内容超出了本书的范围，感兴趣的读者可以在Venkatachalam和Franz（2005）的论文中找到很好的综述。

存在两种减少能量消耗的一般方法。第一种方法是当计算机的某些部件（主要是I/O设备）不用的时候由操作系统关闭它们，因为关闭的设备使用的能量很少或者不使用能量。第二种方法是应用程序使用较少的能量，这样为了延长电池时间可能会降低用户体验的质量。我们将依次看一看这些方法，但是首先就电源使用方面谈一谈硬件设计。

5.8.1 硬件问题

电池一般分为两种类型：一次性使用的和可再充电的。一次性使用的电池（AAA、AA与D电池）可以用来运转掌上设备，但是没有足够的能量为具有大面积发光屏幕的笔记本电脑供电。相反，可再充电的电池能够存储足够的能量为笔记本电脑供电几个小时。在可再充电的电池中，镍镉电池曾经占据主导地位，但是它们后来让位给了镍氢电池，镍氢电池持续的时间更长并且当它们最后被抛弃时不如镍镉电池污染环境那么严重。锂电池更好一些，并且不需要首先完全耗尽就可以再充电，但是它们的容量同样非常有限。

大多数计算机厂商对于电池节约采取的一般措施是将CPU、内存以及I/O设备设计成具有多种状态：工作、睡眠、休眠和关闭。要使用设备，它必须处于工作状态。当设备在短时间内暂时不使用时，可以将其置于睡眠状态，这样可以减少能量消耗。当设备在一个较长的时间间隔内不使用时，可以将其置于休眠状态，这样可以进一步减少能量消耗。这里的权衡是，使一个设备脱离休眠状态常常比使一个设备脱离睡眠状态花费更多的时间和能量。最后，当一个设备关闭时，它什么事情也不做并且也不消耗电能。并非所有的设备都具有这些状态，但是当它们具有这些状态时，应该由操作系统在正确的时机管理状态的变迁。

某些计算机具有两个甚至三个电源按钮。这些按钮之一可以将整个计算机置于睡眠状态，通过键入一个字符或者移动鼠标，能够从该状态快速地唤醒计算机。另一个按钮可以将计算机置于休眠状态，从该状态唤醒计算机花费的时间要长得多。在这两种情况下，这些按钮通常除了发送一个信号给操作系统外什么也不做，剩下的事情由操作系统在软件中处理。在某些国家，依照法律，电气设备必须具有一个机械的电源开关，出于安全性考虑，该开关可以切断电路并且从设备撤去电能。为了遵守这一法律，可能需要另一个开关。

电源管理提出了操作系统必须处理的若干问题，其中许多问题涉及资源休眠——选择性地、临时性地关闭设备，或者至少当它们空闲时减少它们的功率消耗。必须回答的问题包括：哪些设备能够被控制？它们是工作的还是关闭的，或者它们具有中间状态吗？在低功耗状态下节省了多少电能？重启设备消耗能量吗？当进入低功耗状态时是不是必须保存某些上下文？返回到全功耗状态要花费多长时间？当然，对这些问题的回答是随设备而变化的，所以操作系统必须能够处理一个可能性的范围。

许多研究人员研究了笔记本电脑以了解电能的去向。Li等人（1994）测量了各种各样的工作负荷，得出的结论如图5-45所示。Lorch和Smith（1998）在其他机器上进行了测量，得出的结论如图5-45所示。Weiser等人（1994）也进行了测量，但是没有发表数值结果。这些结论清楚地说明能量吸收的前三名依次是显示器、硬盘和CPU。可能因为测量的不同品牌的计算机确实具有不同的能量需求，这些数字并不紧密地吻合，但是很显然，显示器、硬盘和CPU是节约能量的目标。

设备	Li等人（1994）	Lorch和Smith（1998）
显示器	68%	39%
CPU	12%	18%
硬盘	20%	12%
调制解调器		6%
声卡		2%
内存	0.5%	1%
其他		22%

图5-45 笔记本电脑各部件的功率消耗

5.8.2 操作系统问题

操作系统在能量管理上扮演着一个重要的角色，它控制着所有的设备，所以它必须决定关闭什么设备以及何时关闭。如果它关闭了一个设备并且该设备很快再次被用户需要，可能在设备重启时存在恼人的延迟。另一方面，如果它等待了太长的时间才关闭设备，能量就白白地浪费了。

这里的技巧是找到算法和试探法，让操作系统对关于关闭什么设备以及何时关闭能够作出良好的决策。问题是“良好”是高度主观的。一个用户可能觉得在30s未使用计算机之后计算机要花费2s的时间响应击键是可以接受的。另一个用户在相同的条件下可能会发出一连串的诅咒。

1. 显示器

现在我们来看一看能量预算的几大消耗者，考虑一下对于它们能够做些什么。在每个人的能量预算中最大的项目是显示器。为了获得明亮而清晰的图像，屏幕必须是背光照明的，这样会消耗大量的能量。许多操作系统试图通过当几分钟的时间没有活动时关闭显示器而节省能量。通常用户可以决定关闭的时间间隔，因此将屏幕频繁地熄灭和很快用光电池之间的折中推回给用户（用户可能实际上并不希望这样）。关闭显示器是一个睡眠状态，因为当任意键被敲击或者定点设备移动时，它能够（从视频RAM）即时地再生。

Flinn和Satyanarayanan（2004）提出了一种可能的改进。他们建议让显示器由若干数目的区域组成，这些区域能够独立地开启和关闭。在图5-46中，我们描述了16个区域，使用虚线分开它们。当光标在窗口2中的时候，如图5-46a所示，只有右下角的4个区域必须点亮。其他12个区域可以是黑暗的，节省了3/4的屏幕功耗。

当用户移动鼠标到窗口1时，窗口2的区域可以变暗并且窗口1后面的区域可以开启。然而，因为窗口1横跨9个区域，所以需要更多的电能。如果窗口管理器能够感知正在发生的事情，它可以通过一种对齐区域的动作自动地移动窗口1以适合4个区域，如图5-46b所示。为了达到这一从9/16全功率到4/16全功率的缩减，窗口管理器必须理解电源管理或者能够从系统的某些其他做这些工作的部分接收指令。更加复杂的是能够部分地照亮不完全充满的窗口（例如，包含文本短线的窗口可以在右手边保持黑暗）。

2. 硬盘

另一个主要的祸首是硬盘，它消耗大量的能量以保持高速旋转，即使不存在存取操作。许多计算机，特别是笔记本电脑，在几秒钟或者几分钟不活动之后将停止磁盘旋转。当下一次需要磁盘的时候，磁盘将再次开始旋转。不幸的是，一个停止的磁盘是休眠而不是睡眠，因为要花费相当多的时间将磁盘再次旋转起来，导致用户感到明显的延迟。

此外，重新启动磁盘将消耗相当多额外的能量。因此，每个磁盘都有一个特征时间T_d为它的盈亏平衡点，T_d通常在5～15s的范围之间。假设下一次磁盘存取预计在未来的某个时间t到来。如果$t<T_d$，那么保持磁盘旋转比将其停止然后很快再将其开启要消耗更少的能量。如果$t>T_d$，那么使得磁盘停止而后在

较长时间后再次启动磁盘是十分值得的。如果可以做出良好的预测（例如基于过去的存取模式），那么操作系统就能够做出良好的关闭预测并且节省能量。实际上，大多数操作系统是保守的，往往是在几分钟不活动之后才停止磁盘。

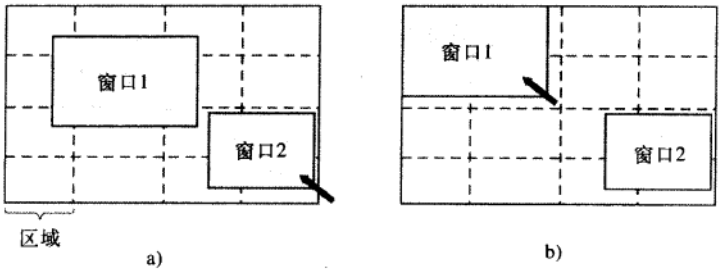


图5-46 针对背光照明的显示器使用区域：a) 当窗口2被选中时，该窗口不移动；b) 当窗口1被选中时，该窗口移动以减少照明的区域的数目

节省磁盘能量的另一种方法是在RAM中拥有一个大容量的磁盘高速缓存。如果所需要的数据块在高速缓存中，空闲的磁盘就不必为满足读操作而重新启动。类似地，如果对磁盘的写操作能够在高速缓存中缓冲，一个停止的磁盘就不必只为了处理写操作而重新启动。磁盘可以保持关闭状态直到高速缓存填满或者读缺失发生。

避免不必要的磁盘启动的另一种方法是：操作系统通过发送消息或信号保持将磁盘的状态通知给正在运行的程序。某些程序具有可以自由决定的写操作，这样的写操作可以被略过或者推迟。例如，一个字处理程序可能被设置成每隔几分钟将正在编辑的文件写入磁盘。如果字处理程序知道当它在正常情况下应该将文件写到磁盘的时刻磁盘是关闭的，它就可以将本次写操作推迟直到下一次磁盘开启时，或者直到某个附加的时间逝去。

3. CPU

CPU也能够被管理以节省能量。笔记本电脑的CPU能够用软件置为睡眠状态，将电能的使用减少到几乎为零。在这一状态下CPU惟一能做的事情是当中断发生时醒来。因此，只要CPU变为空闲，无论是因为等待I/O还是因为没有工作要做，它都可以进入睡眠状态。

在许多计算机上，在CPU电压、时钟周期和电能消耗之间存在着关系。CPU电压可以用软件降低，这样可以节省能量但是也会（近似线性地）降低时钟速度。由于电能消耗与电压的平方成正比，将电压降低一半会使CPU的速度减慢一半，而电能消耗降低到只有1/4。

对于具有明确的最终时限的程序而言，这一特性可以得到利用，例如多媒体观察器必须每40ms解压缩并显示一帧，但是如果它做得太快它就会变得空闲。假设CPU全速运行40ms消耗了x焦耳能量，那么半速运行则消耗$x/4$焦耳的能量。如果多媒体观察器能够在20ms内解压缩并显示一帧，那么操作系统能够以全功率运行20ms，然后关闭20ms，总的能量消耗是$x/2$焦耳。作为替代，它能够以半功率运行并且恰好满足最终时限，但是能量消耗是$x/4$焦耳。以全速和全功率运行某个时间间隔与以半速和四分之一功率运行两倍长时间的比较如图5-47所示。在这两种情况下做了相同的工作，但是在图5-47b中只消耗了一半的能量。

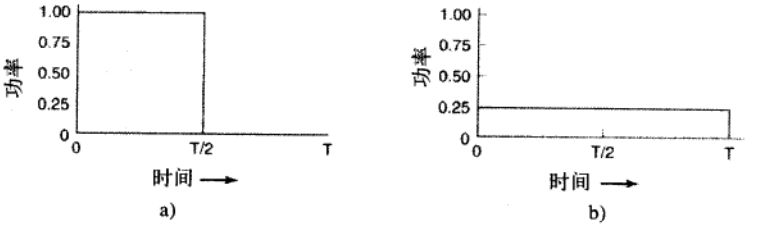


图5-47 a) 以全时钟速度运行；b) 电压减半使时钟速度削减一半并且功率削减到1/4

类似地，如果用户以每秒1个字符的速度键入字符，但是处理字符所需的工作要花费100ms的时间，操作系统最好检测出长时间的空闲周期并且将CPU放慢10倍。简而言之，慢速运行比快速运行具有更高的能量效率。

4. 内存

对于内存，存在两种可能的选择来节省能量。首先，可以刷新然后关闭高速缓存。高速缓存总是能够从内存重新加载而不损失信息。重新加载可以动态并且快速地完成，所以关闭高速缓存是进入睡眠状态。

更加极端的选择是将主存的内容写到磁盘上，然后关闭主存本身。这种方法是休眠，因为实际上所有到内存的电能都被切断了，其代价是相当长的重新加载时间，尤其是如果磁盘也被关闭了的话。当内存被切断时，CPU或者也被关闭，或者必须自ROM执行。如果CPU被关闭，将其唤醒的中断必须促使它跳转到ROM中的代码，从而能够重新加载内存并且使用内存。尽管存在所有这些开销，将内存关闭较长的时间周期（例如几个小时）也许是值得的。与常常要花费一分钟或者更长时间从磁盘重新启动操作系统相比，在几秒钟之内重新启动内存想来更加受欢迎。

5. 无线通信

越来越多的便携式计算机拥有到外部世界（例如Internet）的无线连接。无线通信必需的无线电发送器和接收器是头等的电能贪吃者。特别是，如果无线电接收器为了侦听到来的电子邮件而始终开着，电池可能很快耗干。另一方面，如果无线电设备在1分钟空闲之后关闭，那么就可能会错过到来的消息，这显然是不受欢迎的。

针对这一问题，Kravets和Krishnan（1998）提出了一种有效的解决方案。他们的解决方案的核心利用了这样的事实，即移动的计算机是与固定的基站通信，而固定基站具有大容量的内存与磁盘并且没有电源限制。他们的解决方案是当移动计算机将要关闭无线电设备时，让移动计算机发送一条消息到基站。从那时起，基站在其磁盘上缓冲到来的消息。当移动计算机再次打开无线电设备时，它会通知基站。此刻，所有积累的消息可以发送给移动计算机。

当无线电设备关闭时，生成的外发的消息可以在移动计算机上缓冲。如果缓冲区有填满的危险，可以将无线电设备打开并且将排队的消息发送到基站。

应该在何时将无线电设备关闭?一种可能是让用户或应用程序来决定。另一种方法是在若干秒的空闲时间之后将其关闭。应该在何时将无线电设备再次打开?用户或应用程序可以再一次做出决定，或者可以周期性地将其打开以检查到来的消息并且发送所有排队的消息。当然，当输出缓冲区接近填满时也应该将其打开。各种各样的其他休眠方法也是可能的。

6. 热量管理

一个有一点不同但是仍然与能量相关的问题是热量管理。现代CPU由于高速度而会变得非常热。桌面计算机通常拥有一个内部电风扇将热空气吹出机箱。由于对于桌面计算机来说减少功率消耗通常并不是一个重要的问题，所以风扇通常是始终开着的。

对于笔记本电脑，情况是不同的。操作系统必须连续地监视温度，当温度接近最大可允许温度时，操作系统可以选择打开风扇，这样会发出噪音并且消耗电能。作为替代，它也可以借助于降低屏幕背光、放慢CPU速度、更为激进地关闭磁盘等来降低功率消耗。

来自用户的某些输入也许是颇有价值的指导。例如，用户可以预先设定风扇的噪音是令人不快的，因而操作系统将选择降低功率消耗。

7. 电池管理

在过去，电池仅仅提供电流直到其耗干，在耗干时电池就不会再有电了。现在笔记本电脑使用的是智能电池，它可以与操作系统通信。在请求时，它可以报告其状况，例如最大电压、当前电压、最大负荷、当前负荷、最大消耗速率、当前消耗速率等。大多数笔记本电脑拥有能够查询与显示这些参数的程序。在操作系统的控制下，还可以命令智能电池改变各种工作参数。

某些笔记本电脑拥有多块电池。当操作系统检测到一块电池将要用完时，它必须适度地安排转换到下一块电池，在转换期间不能导致任何故障。当最后一块电池濒临耗尽时，操作系统要负责向用户发出警告然后促成有序的关机，例如，确保文件系统不被破坏。

8. 驱动程序接口

Windows系统拥有一个进行电源管理的精巧的机制，称为ACPI（Advanced Configuration and Power Interface，高级配置与电源接口）。操作系统可以向任何符合标准的驱动程序发出命令，要求它报告其设备的性能以及它们当前的状态。当与即插即用相结合时，该特性尤其重要，因为在系统刚刚引导之后，操作系统甚至还不知道存在什么设备，更不用说它们关于能量消耗或电源管理的属性了。

ACPI还可以发送命令给驱动程序，命令它们削减其功耗水平（当然要基于早先获悉的设备性能）。还存在某些其他方式的通信。特别地，当一个设备（例如键盘或鼠标）在经历了一个时期的空闲之后检测到活动时，这是一个信号让系统返回到（接近）正常运转。

5.8.3 应用程序问题

到目前为止，我们了解了操作系统能够降低各种类型的设备的能量使用量的方法。但是，还存在着另一种方法：指示程序使用较少的能量，即使这意味着提供低劣的用户体验（低劣的体验也比电池耗干并且屏幕熄灭时没有体验要好）。一般情况下，当电池的电荷低于某个阈值时传递这样的信息，然后由应用程序负责在退化性能以延长电池寿命与维持性能并且冒着用光电池的危险之间作出决定。

这里出现的一个问题是程序怎样退化其性能以节省能量？Flinn和Satyanarayanan（2004）研究了这一问题，他们提供了退化的性能怎样能够节省能量的4个例子。我们现在就看一看这些例子。

在他们的研究中，信息以各种形式呈现给用户。当退化不存在时，呈现的是最优可能的信息。当退化存在时，呈现给用户的信息的保真度（准确度）比它能够达到的保真度要差。我们很快就会看到这样的例子。

为了测量能量使用量，Flinn和Satyanarayanan发明了一个称为PowerScope的软件工具。PowerScope所做的事情是提供一个程序的电能使用量的概要剖析。为了使用PowerScope，计算机必须通过一个软件控制的数字万用表接通一个外部电源。使用万用表，软件可以读出从电源流进的电流的毫安数，并且因此确定计算机正在消耗的瞬时功率。PowerScope所做的工作是周期性地采样程序计数器和电能使用量并且将这些数据写到一个文件中。当程序终止后，对文件进行分析就可以给出每个过程的能量使用量。这些测量形成了他们的观察结果的基础。他们还利用硬件能量节约测量并且形成了基准线，对照该基准线测量了退化的性能。

测量的第一个程序是一个视频播放器。在未退化模式下，播放器以全分辨率和彩色方式每秒播放30帧。一种退化形式是舍弃彩色信息并且以黑白方式显示视频。另一种退化形式是降低帧速率，这会导致闪烁并且使电影呈现抖动的质量。还有一种退化形式是在两个方向上减少像素数目，或者是通过降低空间分辨率，或者是使显示的图像更小。对这种类型的测量表明节省了大约30%的能量。

第二个程序是一个语音识别器，它对麦克风进行采样以构造波形。该波形可以在笔记本电脑上进行分析，也可以通过无线链路发送到固定计算机上进行分析，这样做节省了CPU消耗的能量但是会为无线电设备而消耗能量。通过使用比较小的词汇量和比较简单的声学模型可以实现退化，这样做的收益大约是35%。

第三个例子是一个通过无线链路获取地图的地图观察器。退化在于或者将地图修剪到比较小的尺度，或者告诉远程服务器省略比较小的道路，从而需要比较少的位来传输。这样获得的收益大约也是35%。

第四个实验是传送JPEG图像到一个Web浏览器。JPEG标准允许各种算法，在图像质量与文件大小之间进行中。这里的收益平均只有9%。总而言之，实验表明通过接受一些质量退化，用户能够在一个给定的电池上运行更长的时间。

5.9 有关输入/输出的研究

关于输入/输出有大量的研究，但是大多数研究集中在特别的设备上，而不是一般性的I/O。研究的目标常常是想方设法改进性能。

磁盘系统是一个恰当的事例。磁盘臂调度算法曾经是一个流行的研究领域（Bachmat和Braverman，2006；Zarandioon和Thomasim，2006），磁盘阵列也是如此（Arnan等人，2007）。优化完整的I/O路径也引起了人们的兴趣（Riska等人，2007）。还有关于磁盘工作量特性的研究（Riska和Riedel，2006）。一

个新的与磁盘相关的研究领域是高性能闪存盘（Birrell等人，2007；Chang，2007）。设备驱动程序也得到某些必要的关注（Ball等人，2006；Ganapathy等人，2007；Padioleau等人，2006）。

另一种新的存储技术是MEMS（Micro-Electrical-Mechanical System，微电子机械系统），它潜在地可以取代磁盘，或者至少是磁盘的补充（Rangaswami等人，2007；Yu等人，2007）。另一个新兴的研究领域是如何在磁盘控制器内部最好地利用CPU，例如，为了改进性能（Gurumurthi，2007）或者是为了检测病毒（Paul等人，2005）。

稍稍让人吃惊的是，身份低下的时钟仍然是研究的主题。为了提供更好的分辨率，某些操作系统在1000Hz的时钟下运行，这会导致相当大的开销。摆脱这一开销正是新兴的研究课题（Etsion等人，2003；Tsafir等人，2005）。

瘦客户机也是相当引人注目的研究主题（Kissler和Hoyt，2005；Ritschard，2006；Schwartz和Guerrazzi，2005）。

考虑到研究笔记本电脑的为数众多的计算机科学家，并且考虑到大多数笔记本电脑微不足道的电池寿命，看到在利用软件技术减少电能消耗方面有巨大的研究兴趣就不足为奇了。研究中的特别主题包括：编写应用程序代码以最大化磁盘空闲时间（Son等人，2006），当使用比较少时让磁盘降低转速（Gurumurthi等人，2003），使用程序模型预测无线网卡何时可以关闭（Hom和Kremer，2003），节省VoIP的电能（Gleeson等人，2006），调查安全性的能量代价（Aaraj等人，2007），以能源效率高的方式执行多媒体调度（Yuan和Nahrstedt，2006），以及让内置的摄像机检测是否有人在看显示器并且在没有人看的时候将其关闭（Dalton和Ellis，2003）。在低端，另一个热门话题是传感器网络中能源的使用（Min等人，2007；Wang和Xiao，2006）。而在高端，在大型服务器园区中节省能源也引起人们的关注（Fan等人，2007；Tolentino等人，2007）。

5.10 小结

输入/输出是一个经常被忽略但是十分重要的话题。任何一个操作系统都有大量的组分与I/O有关。I/O可以用三种方式来实现。第一是程序控制I/O，在这种方式下主CPU输入或输出每个字节或字并且闲置在一个密封的循环中等待，直到它能够获得或者发送下一个字节或字。第二是中断驱动的I/O，在这种方式下CPU针对一个字节或字开始I/O传送并且离开去做别的事情，直到一个中断到来发出信号通知I/O完成。第三是DMA，在这种方式下有一个单独的芯片管理着一个数据块的完整传送过程，只有当整个数据块完成传送时才引发一个中断。

I/O可以组织成4个层次：中断服务程序、设备驱动程序、与设备无关的I/O软件和运行在用户空间的I/O库与假脱机程序。设备驱动程序处理运行设备的细节并且向操作系统的其余部分提供统一的接口。与设备无关的I/O软件做类似缓冲与错误报告这样的事情。

盘具有多种类型，包括磁盘、RAID和各类光盘。磁盘臂调度算法经常用来改进磁盘性能，但是虚拟几何规格的出现使事情变得十分复杂。通过将两块磁盘组成一对，可以构造稳定的存储介质，具有某些有用的性质。

时钟可以用于跟踪实际时间，限制进程可以运行多长时间，处理监视定时器，以及进行记账。

面向字符的终端具有多种多样的问题，这些问题涉及特殊的字符如何输入以及特殊的转义序列如何输出。输入可以采用原始模式或加工模式，取决于程序对于输入需要有多少控制。针对输出的转义序列控制着光标的移动并且允许在屏幕上插入和删除文本。

大多数UNIX系统使用X窗口系统作为用户界面的基础。它包含与特殊的库相绑定并发出绘图命令的程序，以及在显示器上执行绘图的服务器。

许多个人计算机使用GUI作为它们的输出。GUI基于WIMP范式：窗口、图标、菜单和定点设备。基于GUI的程序一般是事件驱动的，当键盘事件、鼠标事件和其他事件发生时立刻会被发送给程序以便处理。在UNIX系统中，GUI几乎总是运行在X之上。

瘦客户机与标准PC相比具有某些优势，对用户而言，值得注意的是简单性并且需要较少维护。对THINC瘦客户机进行的实验表明，以五条简单的原语就能制造出具有良好性能的客户机，即使对于视频也是如此。

最后，电源管理对于笔记本电脑来说是一个主要的问题，因为电池寿命是有限的。操作系统可以采用各种技术来减少功率消耗。通过牺牲某些质量以换取更长的电池寿命，应用程序也可以做出贡献。

习题

1. 芯片技术的进展已经使得将整个控制器包括所有总线访问逻辑放在一个便宜的芯片上成为可能。这对于图1-5的模型具有什么影响?
2. 已知图5-1列出的速度，是否可能以全速从一台扫描仪扫描文档并且通过802.11g网络对其进行传输?请解释你的答案。
3. 图5-3b显示了即使在存在单独的总线用于内存和用于I/O设备的情况下使用内存映射I/O的一种方法，也就是说，首先尝试内存总线，如果失败则尝试I/O总线。一名聪明的计算机科学专业的学生想出了一个改进办法：并行地尝试两个总线，以加快访问I/O设备的过程。你认为这个想法如何?
4. 假设一个系统使用DMA将数据从磁盘控制器传送到内存。进一步假设平均花费$t_1$ns获得总线，并且花费$t_2$ns在总线上传送一个字（$t_1>>t_2$）。在CPU对DMA控制器进行编程之后，如果（a）采用一次一字模式，（b）采用突发模式，从磁盘控制器到内存传送1000个字需要多少时间?假设向磁盘控制器发送命令需要获取总线以传输一个字，并且应答传输也需要获取总线以传输一个字。
5. 假设一台计算机能够在10ns内读或者写一个内存字，并且假设当中断发生时，所有32位寄存器连同程序计数器和PSW被压入堆栈。该计算机每秒能够处理的中断的最大数目是多少?
6. CPU体系结构设计师知道操作系统编写者痛恨不精确的中断。取悦于OS人群的一种方法是当得到一个中断信号通知时，让CPU停止发射指令，但是允许当前正在执行的指令完成，然后强制中断。这一方案是否有缺点？请解释你的答案。
7. 在图5-9b中，中断直到下一个字符输出到打印机之后才得到应答。中断在中断服务程序开始时立刻得到应答是否同样可行?如果是，请给出像本书中那样在中断服务程序结束时应答中断的一个理由。如果不是，为什么?
8. 一台计算机具有如图1-6a所示的三阶段流水线。在每一个时钟周期，一条新的指令从PC所指向的地址处的内存中取出并放入流水线，同时PC值增加。每条指令恰好占据一个内存字。已经在流水线中的指令每个时钟周期前进一个阶段。当中断发生时，当前PC压入堆栈，并且将PC设置为中断处理程序的地址。然后，流水线右移一个阶段并且中断处理程序的第一条指令被取入流水线。该机器具有精确的中断吗?请解释你的答案。
9. 一个典型的文本打印页面包含50行，每行80个字符。设想某一台打印机每分钟可以打印6个页面，并且将字符写到打印机输出寄存器的时间很短以至于可以忽略。如果打印每一个字符要请求一次中断，而进行中断服务要花费总计50μs的时间，那么使用中断驱动的I/O来运行该打印机有没有意义?
10. 请解释OS如何帮助安装新的驱动程序而无须重新编译OS。
11. 以下各项工作是在四个I/O软件层的哪一层完成的?
    a) 为一个磁盘读操作计算磁道、扇区、磁头。
    b) 向设备寄存器写命令。
    c) 检查用户是否允许使用设备。
    d) 将二进制整数转换成ASCII码以便打印。
12. 一个局域网以如下方式使用：用户发出一个系统调用，请求将数据包写到网上，然后操作系统将数据复制到一个内核缓冲区中，再将数据复制到网络控制器接口板上。当所有数据都安全地存放在控制器中时，再将它们通过网络以10Mb/s的速率发送。在每一位被发送后，接收的网络控制器以每微秒一位的速率保存它们。当最后一位到达时，目标CPU被中断，内核将新到达的数据包复制到内核缓冲区中进行检查。一旦判明该数据包是发送给哪个用户的，内核就将数据复制到该用户空间。如果我们假设每一个中断及其相关的处理过程花费1ms时间，数据包为1024字节（忽略包头），并且复制一个字节花费1μs时间，那么将数据从一个进程转储到另一个进程的最大速率是多少?假设发送进程被阻塞直到接收端结束工作并且返回一个应答。为简单起见，假设获得返回应答的时间非常短，可以忽略不计。
13. 为什么打印机的输出文件在打印前通常都假脱机输出在磁盘上?
14. 3级RAID只使用一个奇偶驱动器就能够纠正一位错误。那么2级RAID的意义是什么?毕竟2级

RAID也只能纠正一位错误而且需要更多的驱动器。

15. 如果两个或更多的驱动器在很短的时间内崩溃，那么RAID就可能失效。假设在给定的一小时内一个驱动器崩溃的概率是p，那么在给定的一小时内具有k个驱动器的RAID失效的概率是多少?
16. 从读性能、写性能、空间开销以及可靠性方面对0级RAID到5级RAID进行比较。
17. 为什么光存储设备天生地比磁存储设备具有更高的数据密度?注意：本题需要某些高中物理以及磁场是如何产生的知识。
18. 光盘和磁盘的优点和缺点各是什么?
19. 如果一个磁盘控制器没有内部缓冲，一旦从磁盘上接收到字节就将它们写到内存中，那么交错编号还有用吗?请讨论。
20. 如果一个磁盘是双交错编号的，那么该磁盘是否还需要柱面斜进以避免在进行磁道到磁道的寻道时错过数据？请讨论你的答案。
21. 考虑一个包含16个磁头和400个柱面的磁盘。该磁盘分成4个100柱面的区域，不同的区域分别包含160个、200个、240个和280个扇区。假设每个扇区包含512字节，相邻柱面间的平均寻道时间为1ms，并且磁盘转速为7200rpm。计算a) 磁盘容量、b) 最优磁道斜进以及c) 最大数据传输率。
22. 一个磁盘制造商拥有两种5.25英寸的磁盘，每种磁盘都具有10 000个柱面。新磁盘的线性记录密度是老磁盘的两倍。在较新的驱动器上哪个磁盘的特性更好，哪个无变化?
23. 一个计算机制造商决定重新设计Pentium硬盘的分区表以提供四个以上的分区。这一变化有什么后果?
24. 磁盘请求以柱面10、22、20、2、40、6和38的次序进入磁盘驱动器。寻道时每个柱面移动需要6ms，以下各算法所需的寻道时间是多少?
    a) 先来先服务。
    b) 最近柱面优先。
    c) 电梯算法（初始向上移动）。
    在各情形下，假设磁臂起始于柱面20。
25. 调度磁盘请求的电梯算法的一个微小更改是总是沿相同的方向扫描。在什么方面这一更改的算法优于电梯算法?
26. 在讨论使用非易失性RAM的稳定的存储器时，掩饰了如下要点。如果稳定写完成但是在操作系统能够将无效的块编号写入非易失性RAM之前发生了崩溃，那么会有什么结果?这一竞争条件会毁灭稳定的存储器的抽象概念吗?请解释你的答案。
27. 在关于稳定的存储器的讨论中，证明了如果在写过程中发生了CPU崩溃，磁盘可以恢复到一个一致的状态（写操作或者已完成，或者完全没有发生）。如果在恢复的过程中CPU再次崩溃，这一特性是否还保持？请解释你的答案。
28. 某计算机上的时钟中断处理程序每一时钟滴答需要2ms（包括进程切换的开销），时钟以60Hz的频率运行，那么CPU用于时钟处理的时间比例是多少?
29. 一台计算机以方波模式使用一个可编程时钟。如果使用500MHz的晶体，为了达到如下时钟分辨率，存储寄存器的值应该是多少?
    a) 1ms（每毫秒一个时钟滴答）。
    b) 100μs。
30. 一个系统通过将所有未决的时钟请求链接在一起而模拟多个时钟，如图5-34所示。假设当前时刻是5000，并且存在针对时刻5008、5012、5015、5029和5037的未决的时钟请求。请指出在时刻5000、5005和5013时时钟头、当前时刻以及下一信号的值。请指出在时刻23时时钟头、当前时刻以及下一信号的值。
31. 许多UNIX版本使用一个32位无符号整数作为从时间原点计算的秒数来跟踪时间。这些系统什么时候会溢出（年与月）？你盼望这样的事情实际发生吗?
32. 一个位图模式的终端包含1280×960个像素。为了滚动一个窗口，CPU（或者控制器）必须向上移动所有的文本行，这是通过将文本行的所有位从视频RAM的一部分复制到另一部分实现的。如果一个特殊的窗口高60行宽80个字符（总共4800个字符），每个字符框宽8个像素高16像素，那么以每个字节50ns的复制速率滚动整个窗口需要多长时间?如果所有的行都是80个字符长，那么终端的等价波特率是多少?将一个字符显示在屏幕上需要5μs，每秒能够显示多少行?
33. 接收到一个DEL（SIGINT）字符之后，显示驱动程序将丢弃当前排队等候显示的所有输出。为什么?
34. 在最初IBM PC的彩色显示器上，在除了CRT电子束垂直回扫期间以外的任何时间向视频RAM中写数据都会导致屏幕上出现难看的斑点。一个屏幕映像为25×80个字符，每个字符

占据8×8像素的方框。每行640像素在电子束的一次水平扫描中绘出，需要花费6μs，包括水平回扫。屏幕每秒钟刷新60次，每次刷新均需要一个垂直回扫期以便使电子束回到屏幕顶端。在这一过程中可供写视频RAM的时间比例是多少？

35. 计算机系统的设计人员期望鼠标移动的最大速率为20cm/s。如果一个鼠标步是0.1mm，并且每个鼠标消息3个字节，假设每个鼠标步都是单独报告的，那么鼠标的最大数据传输率是多少？
36. 基本的加性颜色是红色、绿色和蓝色，这意味着任何颜色都可以通过这些颜色的线性叠加而构造出来。某人拥有一张不能使用全24位颜色表示的彩色照片，这可能吗？
37. 将字符放置在位图模式的屏幕上，一种方法是使用BitBlt从一个字体表复制位图。假设一种特殊的字体使用16×24像素的字符，并且采用RGB真彩色。
    (a) 每个字符占用多少字体表空间？
    (b) 如果复制一个字节花费100ns(包括系统开销)，那么到屏幕的输出率是每秒多少个字符？
38. 假设复制一个字节花费10ns，那么对于80字符×25行文本模式的内存映射的屏幕，完全重写屏幕要花费多长时间？采用24位彩色的1024×768像素的图形屏幕情况怎样？
39. 在图5-40中存在一个窗口类需要调用RegisterClass进行注册，在图5-38中对应的X窗口代码中，并不存在这样的调用或与此相似的任何调用。为什么？
40. 在课文中我们给出了一个如何在屏幕上画一个矩形的例子，即使用Windows GDI：

    Rectangle(hdc, xleft, ytop, xright, ybottom);

    是否存在对于第一个参数（hdc）的实际需要？如果存在，是什么？毕竟，矩形的坐标作为参数而显式地指明了。
41. 一台THINC终端用于显示一个网页，该网页包含一个动画卡通，卡通大小为400×160像素，以每秒10帧的速度播放。显示该卡通会消耗100Mbps决速以太网带宽多大的部分？
42. 在一次测试中，THINC系统被观测到对于1Mbps的网络工作良好。在多用户的情形中会有问题吗？提示：考虑大量的用户在观看时间表排好的TV节目，并且相同数目的用户在浏览万维网。
43. 如果一个CPU的最大电压V被削减到V/n，那么它的功率消耗将下降到其原始值的$1/n^2$，并且它的时钟速度下降到其原始值的$1/n$。假设一个用户以每秒1个字符的速度键入字符，处理每个字符所需要的CPU时间是100ms，n的最优值是多少？与不削减电压相比，以百分比表示相应的能量节约了多少？假设空闲的CPU完全不消耗能量。
44. 一台笔记本电脑被设置成最大地利用功率节省特性，包括在一段时间不活动之后关闭显示器和硬盘。一个用户有时在文本模式下运行UNIX程序，而在其他时间使用X窗口系统。她惊讶地发现当她使用仅限文本模式的程序时，电池的寿命相当长。为什么？
45. 编写一个程序模拟稳定的存储器，在你的磁盘上使用两个大型的固定长度的文件来模拟两块磁盘。
46. 编写一个程序实现三个磁盘臂调度算法。编写一个驱动程序随机生成一个柱面号序列（0～999），针对该序列运行三个算法并且打印出在三个算法中磁盘臂需要来回移动的总距离（柱面数）。
47. 编写一个程序使用单一的时钟实现多个定时器。该程序的输入包含四种命令（S <int>, T, E <int>, P）的序列：S <int>设置当前时刻为<int>；T是一个时钟滴答；E <int>调度一个信号在<int>时刻发生；P打印出当前时刻、下一信号和时钟头的值。当唤起一个信号时，你的程序还应该打印出一条语句。

第6章 死 锁

在计算机系统中有很多独占性的资源，在任一时刻它们都只能被一个进程使用。常见的有打印机、磁带以及系统内部表中的表项。打印机同时让两个进程打印将造成混乱的打印结果；两个进程同时使用同一文件系统表中的表项会引起文件系统的瘫痪。正因为如此，操作系统都具有授权一个进程（临时）排他地访问某一种资源的能力。

在很多应用中，需要一个进程排他性地访问若干种资源而不是一种。例如，有两个进程准备分别将扫描的文档记录到CD上。进程A请求使用扫描仪，并被授权使用。但进程B首先请求CD刻录机，也被授权使用。现在，A请求使用CD刻录机，但该请求在B释放CD刻录机前会被拒绝。但是，进程B非但不放弃CD刻录机，而且去请求扫描仪。这时，两个进程都被阻塞，并且一直处于这样的状态。这种状况就是*死锁*（deadlock）。

死锁也可能发生在机器之间。例如，许多办公室中都用计算机连成局域网，扫描仪、CD刻录机、打印机和磁带机等设备也连接到局域网上，成为共享资源，供局域网中任何机器上的人和用户使用。如果这些设备可以远程保留给某一个用户（比如，在用户家里的机器使用这些设备），那么，也会发生上面描述的死锁现象。更复杂的情形会引起三个、四个或更多设备和用户发生死锁。

除了请求独占性的I/O设备之外，别的情况也有可能引起死锁。例如，在一个数据库系统中，为了避免竞争，可对若干记录加锁。如果进程A对记录R1加了锁，进程B对记录R2加了锁，接着，这两个进程又试图各自把对方的记录也加锁，这时也会产生死锁。所以，软硬件资源都有可能出现死锁。

在本章里，我们准备考察几类死锁，了解它们是如何出现的，学习防止或者避免死锁的办法。尽管我们所讨论的是操作系统环境下出现的死锁问题，但是在数据库系统和许多计算机应用环境中都可能产生死锁，所以我们所介绍的内容实际上可以应用到包含多个进程的系统中。有很多有关死锁的著作，《Operating Systems Review》中列出了两本参考书（Newton, 1979; Zobel, 1983），有兴趣的读者可以参考这两本书。死锁方面的大多数研究工作在1980年以前就完成了，尽管所列的参考文献有些老，但是这些内容依然是很有用的。

6.1 资源

大部分死锁都和资源相关，所以我们首先来看看资源是什么。在进程对设备、文件等取得了排他性访问权时，有可能会出现死锁。为了尽可能使关于死锁的讨论通用，我们把这类需要排他性使用的对象称为*资源*（resource）。资源可以是硬件设备（如磁带机）或是一组信息（如数据库中一个加锁的记录）。通常在计算机中有多种（可获取的）资源。一些类型的资源会有若干个相同的实例，如三台磁带机。当某一资源有若干实例时，其中任何一个都可以用来满足对资源的请求。简单来说，资源就是随着时间的推移，必须能获得、使用以及释放的任何东西。

6.1.1 可抢占资源和不可抢占资源

资源分为两类：可抢占的和不可抢占的。*可抢占资源*（preemptable resource）可以从拥有它的进程中抢占而不会产生任何副作用，存储器就是一类可抢占的资源。例如，一个系统拥有256MB的用户内存和一台打印机。如果有两个256MB内存的进程都想进行打印，进程A请求并获得了打印机，然后开始计算要打印的值。在它没有完成计算任务之前，它的时间片就已经用完并被换出。

然后，进程B开始运行并请求打印机，但是没有成功。这时有潜在的死锁危险。由于进程A拥有打印机，而进程B占有了内存，两个进程都缺少另外一个进程拥有的资源，所以任何一个都不能继续执行。不过，幸运的是通过把进程B换出内存、把进程A换入内存就可以实现抢占进程B的内存。这样，进程A继续运行并执行打印任务，然后释放打印机。在这个过程中不会产生死锁。

相反，不可抢占资源（nonpreemptable resource）是指在不引起相关的计算失败的情况下，无法把它从占有它的进程处抢占过来。如果一个进程已开始刻盘，突然将CD刻录机分配给另一个进程，那么将划坏CD盘。在任何时刻CD刻录机都是不可抢占的。

总的来说，死锁和不可抢占资源有关，有关可抢占资源的潜在死锁通常可以通过在进程之间重新分配资源而化解。所以，我们的重点放在不可抢占资源上。

使用一个资源所需要的事件顺序可以用抽象的形式表示如下：

1) 请求资源。

2) 使用资源。

3) 释放资源。

若请求时资源不可用，则请求进程被迫等待。在一些操作系统中，资源请求失败时进程会自动被阻塞，在资源可用时再唤醒它。在其他的系统中，资源请求失败会返回一个错误代码，请求的进程会等待一段时间，然后重试。

当一个进程请求资源失败时，它通常会处于这样一个小循环中：请求资源，休眠，再请求。这个进程虽然没有被阻塞，但是从各角度来说，它不能做任何有价值的工作，实际和阻塞状态一样。在后面的讨论中，我们假设：如果某个进程请求资源失败，那么它就进入休眠状态。

请求资源的过程是非常依赖于系统的。在某些系统中，提供了request系统调用，用于允许进程资源请求。在另一些系统中，操作系统只知道资源是一些特殊文件，在任何时刻它们最多只能被一个进程打开。一般情况下，这些特殊文件用open调用打开。如果这些文件正在被使用，那么，发出open调用的进程会被阻塞，一直到文件的当前使用者关闭该文件为止。

6.1.2 资源获取

对于数据库系统中的记录这类资源，应该由用户进程来管理其使用。一种允许用户管理资源的可能方法是为每一个资源配置一个信号量。这些信号量都被初始化为1。互斥信号量也能起到相同的作用。上述的三个步骤可以实现为信号量的down操作来获取资源，使用资源，最后使用up操作来释放资源。这三个步骤如图6-1a所示。

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
      down(&resource_1);
      use_resource_1( );
      up(&resource_1);
}
```

a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}
```

b)

图6-1 使用信号量保护资源：a)一个资源；b)两个资源

有时候，进程需要两个或更多的资源，它们可以顺序获得，如图6-1b所示。如果需要两个以上的资源，通常都是连续获取。

到目前为止，进程的执行不会出现问题。在只有一个进程参与时，所有的工作都可以很好地完成。当然，如果只有一个进程，就没有必要这么慎重地获取资源，因为不存在资源竞争。

现在考虑两个进程（A和B）以及两个资源的情况。图6-2描述了两种不同的方式。在图6-2a中，两个进程以相同的次序请求资源；在图6-2b中，它们以不同的次序请求资源。这种不同看似微不足道，实则不然。

在图6-2a中，其中一个进程先于另一个进程获取资源。这个进程能够成功地获取第二个资源并完成它的任务。如果另一个进程想在第一个资源被释放之前获取该资源，那么它会由于资源加锁而被阻塞，直到该资源可用为止。

图6-2b的情况就不同了。可能其中一个进程获取了两个资源并有效地阻塞了另外一个进程，直到它使用完这两个资源为止。但是，也有可能进程A获取了资源1，进程B获取了资源2，每个进程如果都想请求另一个资源就会被阻塞，那么，每个进程都无法继续运行。这种情况就是死锁。

这里我们可以看到一个编码风格上的细微差别（哪一个资源先获取）造成了可以执行的程序和不能执行而且无法检测错误的程序之间的差别。因为死锁是非常容易发生的，所以有很多人研究如何处理这种情况。这一章就会详细讨论死锁问题，并给出一些对策。

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

b)

图6-2 a) 无死锁的编码；b) 有可能出现死锁的编码

6.2 死锁概述

死锁的规范定义如下：

如果一个进程集合中的每个进程都在等待只能由该进程集合中的其他进程才能引发的事件，那么，该进程集合就是死锁的。

由于所有的进程都在等待，所以没有一个进程能引发可以唤醒该进程集合中的其他进程的事件，这样，所有的进程都只好无限期等待下去。在这一模型中，我们假设进程只含有一个线程，并且被阻塞的进程无法由中断唤醒。无中断条件使死锁的进程不能被时钟中断等唤醒，从而不能引发释放该集合中的其他进程的事件。

在大多数情况下，每个进程所等待的事件是释放该进程集合中其他进程所占有的资源。换言之，这个死锁进程集合中的每一个进程都在等待另一个死锁的进程已经占有的资源。但是由于所有进程都不能运行，它们中的任何一个都无法释放资源，所以没有一个进程可以被唤醒。进程的数量以及占有或者请求的资源数量和种类都是无关紧要的，而且无论资源是何种类型（软件或者硬件）都会发生这种结果。这种死锁称为*资源死锁*（resource deadlock）。这是最常见的类型，但并不是惟一的类型。本节我们会详细介绍一下资源死锁，在本章末再概述其他类型的死锁。

6.2.1 资源死锁的条件

Coffman等人（1971）总结了发生（资源）死锁的四个必要条件：

1) 互斥条件。每个资源要么已经分配给了一个进程，要么就是可用的。

2) 占有和等待条件。已经得到了某个资源的进程可以再请求新的资源。

3) 不可抢占条件。已经分配给一个进程的资源不能强制性地被抢占，它只能被占有它的进程显式地释放。

4) 环路等待条件。死锁发生时，系统中一定有由两个或两个以上的进程组成的一条环路，该环路中的每个进程都在等待着下一个进程所占有的资源。

死锁发生时，以上四个条件一定是同时满足的。如果其中任何一个条件不成立，死锁就不会发生。

值得注意的是，每一个条件都与系统的一种可选策略相关。一种资源能否同时分配给不同的进程？一个进程能否在占有一个资源的同时请求另一个资源？资源能否被抢占？循环等待环路是否存在？我们在后面会看到怎样通过破坏上述条件来预防死锁。

6.2.2 死锁建模

Holt （1972）指出如何用有向图建立上述四个条件的模型。在有向图中有两类节点：用圆形表示的进程，用方形表示的资源。从资源节点到进程节点的有向边代表该资源已被请求、授权并被进程占用。

在图6-3a中，当前资源R正被进程A占用。

由进程节点到资源节点的有向边表明当前进程正在请求该资源，并且该进程已被阻塞，处于等待该资源的状态。在图6-3b中，进程B正等待着资源S。图6-3c说明进入了死锁状态：进程C等待着资源T，资源T被进程D占用着，进程D又等待着由进程C占用着的资源U。这样两个进程都得等待下去。图中的环表示与这些进程和资源有关的死锁。在本例中，环是C-T-D-U-C。

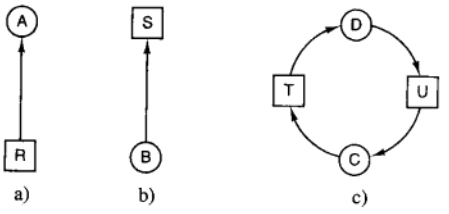


图6-3　资源分配图：a) 占有一个资源；b) 请求一个资源；c) 死锁

我们再看看使用资源分配图的方法。假设有三个进程（A，B，C）及三个资源（R，S，T）。三个进程对资源的请求和释放如图6-4a～图6-4c所示。操作系统可以随时选择任一非阻塞进程运行，所以它可选择A运行一直到A完成其所有工作，接着运行B，最后运行C。

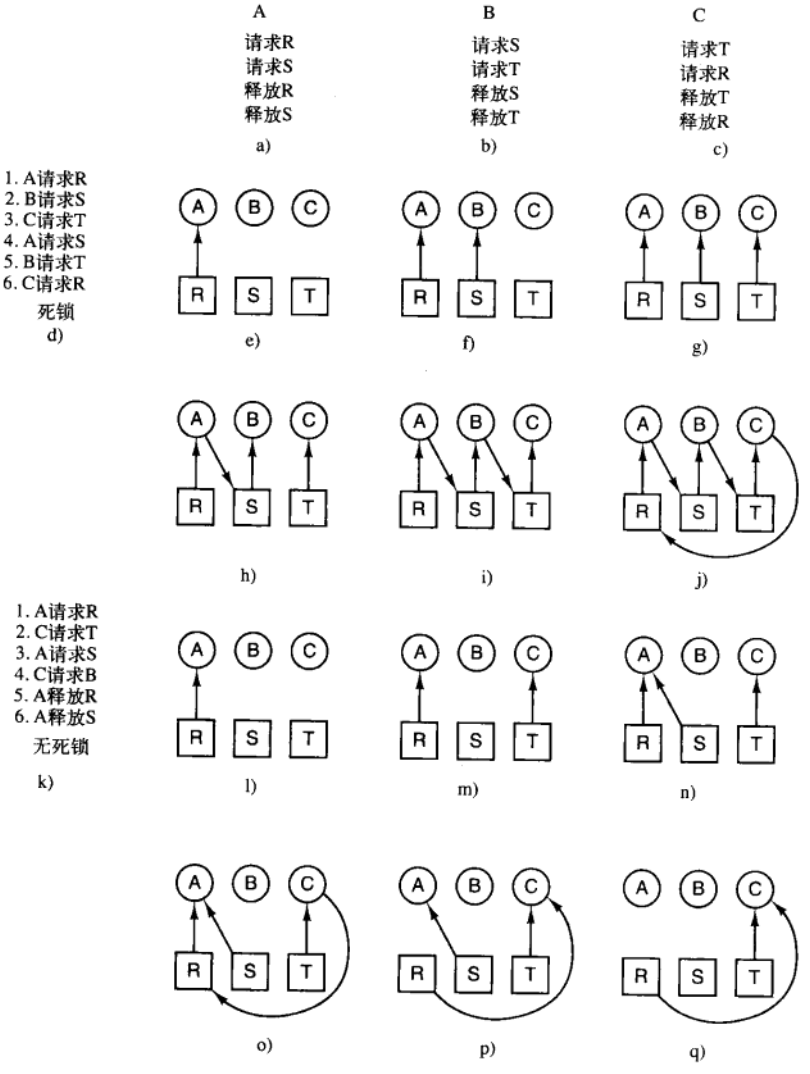


图6-4　一个死锁是如何产生以及如何避免的例子

上述的执行次序不会引起死锁（因为没有资源的竞争），但程序也没有任何并行性。进程在执行过程中，不仅要请求和释放资源，还要做计算或者输入/输出工作。如果进程是串行运行，不会出现当一个进程等待I/O时让另一个进程占用CPU进行计算的情形。因此，严格的串行操作有可能不是最优的。不过，如果所有的进程都不执行I/O操作，那么最短作业优先调度会比轮转调度优越，所以在这种情况下，串行运行有可能是最优的。

如果假设进程操作包含I/O和计算，那么轮转法是一种合适的调度算法。对资源请求的次序可能会如图6-4d所示。假如按这个次序执行，图6-4e～图6-4j是相应的资源分配图。在出现请求4后，如图6-4h所示，进程A被阻塞等待S，后续两步中的B和C也会被阻塞，结果如图6-4j所示，产生环路并导致死锁。

不过正如前面所讨论的，并没有规定操作系统要按照某一特定的次序来运行这些进程。特别地，对于一个有可能引起死锁的资源请求，操作系统可以干脆不批准请求，并把该进程挂起（即不参与调度）一直到处于安全状态为止。在图6-4中，假设操作系统知道有引起死锁的可能，那么它可以不把资源S分配给B，这样B被挂起。假如只运行进程A和C，那么资源请求和释放的过程会如图6-4k所示，而不是如图6-4d所示。这一过程的资源分配图在图6-4l～图6-4q中给出，其中没有死锁产生。

在第q步执行完后，就可以把资源S分配给B了，因为A已经完成，而且C获得了它所需要的所有资源。尽管B会因为请求T而等待，但是不会引起死锁，B只需要等待C结束。

在本章后面我们将考察一个具体的算法，用以做出不会引起死锁的资源分配决策。在这里需要说明的是，资源分配图可以用作一种分析工具，考察对一给定的请求/释放的序列是否会引起死锁。只需要按照请求和释放的次序一步步进行，每一步之后都检查图中是否包括了环路。如果有环路，那么就有死锁；反之，则没有死锁。在我们的例子中，虽然只和同一类资源有关，而且只包含一个实例，但是上面的原理完全可以推广到有多种资源并含有若干个实例的情况中去（Holt, 1972）。

总而言之，有四种处理死锁的策略：

1) 忽略该问题。也许如果你忽略它，它也会忽略你。

2) 检测死锁并恢复。让死锁发生，检测它们是否发生，一旦发生死锁，采取行动解决问题。

3) 仔细对资源进行分配，动态地避免死锁。

4) 通过破坏引起死锁的四个必要条件之一，防止死锁的产生。

下面四节将分别讨论这四种方法。

6.3 鸵鸟算法

最简单的解决方法是鸵鸟算法：把头埋到沙子里，假装根本没有问题发生[⊖]。每个人对该方法的看法都不相同。数学家认为这种方法根本不能接受，不论代价有多大，都要彻底防止死锁的产生；工程师们想要了解死锁发生的频度、系统因各种原因崩溃的发生次数以及死锁的严重性。如果死锁平均每5年发生一次，而每个月系统都会因硬件故障、编译器错误或者操作系统故障而崩溃一次，那么大多数的工程师不会以性能损失和可用性的代价去防止死锁。

为了能够让这一对比更具体，考虑如下情况的一个操作系统：当一个open系统调用因物理设备（例如CD-ROM驱动程序或者打印机）忙而不能得到响应的时候，操作系统会阻塞调用该系统调用的进程。通常是由设备驱动来决定在这种情况下应该采取何种措施。显然，阻塞或者返回一个错误代码是两种选择。如果一个进程成功地打开了CD-ROM驱动器，而另一个进程成功地打开了打印机，这时每个进程都会试图去打开另外一个设备，然后系统会阻塞这种尝试，从而发生死锁。现有系统很少能够检测到这种死锁。

6.4 死锁检测和死锁恢复

第二种技术是死锁检测和恢复。在使用这种技术时，系统并不试图阻止死锁的产生，而是允许死锁发生，当检测到死锁发生后，采取措施进行恢复。本节我们将考察检测死锁的几种方法以及恢复死锁的几种方法。

⊖ 这一民间传说毫无道理。鸵鸟每小时跑60公里，为了得到一顿丰盛的晚餐，它一脚的力量足以踢死一头狮子。

6.4.1 每种类型一个资源的死锁检测

我们从最简单的例子开始，即每种类型只有一个资源。这样的系统可能有一台扫描仪、一台CD刻录机、一台绘图仪和一台磁带机，但每种类型的资源都不超过一个，即排除了同时有两台打印机的情况。稍后我们将用另一种方法来解决两台打印机的情况。

可以对这样的系统构造一张资源分配图，如图6-3所示。如果这张图包含了一个或一个以上的环，那么死锁就存在。在此环中的任何一个进程都是死锁进程。如果没有这样的环，系统就没有发生死锁。

我们讨论一下更复杂的情况，假设一个系统包括A到G共7个进程，R到W共6种资源。资源的占有情况和进程对资源的请求情况如下：

1) A进程持有R资源，且需要S资源。

2) B进程不持有任何资源，但需要T资源。

3) C进程不持有任何资源，但需要S资源。

4) D进程持有U资源，且需要S资源和T资源。

5) E进程持有T资源，且需要V资源。

6) F进程持有W资源，且需要S资源。

7) G进程持有V资源，且需要U资源。

问题是："系统是否存在死锁？如果存在的话，死锁涉及了哪些进程？"

要回答这一问题，我们可以构造一张资源分配图，如图6-5a所示。可以直接观察到这张图中包含了一个环，如图6-5b所示。在这个环中，我们可以看出进程D、E、G已经死锁。进程A、C、F没有死锁，这是因为可把S资源分配给它们中的任一个，而且它们中的任一进程完成后都能释放S，于是其他两个进程可依次执行，直至执行完毕。（请注意，为了让这个例子更有趣，我们允许进程D每次请求两个资源。）

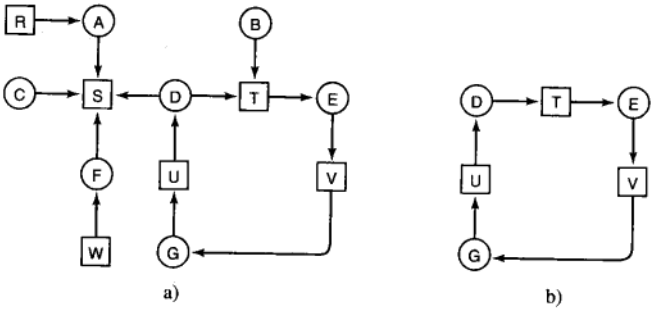


图6-5　a) 资源分配图；b) 从a中抽取的环

虽然通过观察一张简单的图就能够很容易地找出死锁进程，但为了实用，我们仍然需要一个正规的算法来检测死锁。众所周知，有很多检测有向图环路的方法。下面将给出一个简单的算法，这种算法对有向图进行检测，并在发现图中有环路存在或无环路时结束。这一算法使用了数据结构L，L代表一些节点的集合。在这一算法中，对已经检查过的弧（有向边）进行标记，以免重复检查。

通过执行下列步骤完成上述算法：

1) 对图中的每一个节点N，将N作为起始点执行下面5个步骤。

2) 将L初始化为空表，并清除所有的有向边标记。

3) 将当前节点添加到L的尾部，并检测该节点是否在L中已出现过两次。如果是，那么该图包含了一个环（已列在L中），算法结束。

4) 从给定的节点开始，检测是否存在没有标记的从该节点出发的弧（有向边）。如果存在的话，做第5步；如果不存在，跳到第6步。

5) 随机选取一条没有标记的从该节点出发的弧（有向边），标记它。然后顺着这条弧线找到新的当前节点，返回到第3步。

6) 如果这一节点是起始节点，那么表明该图不存在任何环，算法结束。否则意味着我们走进了死胡

同，所以需要移走该节点，返回到前一个节点，即当前节点前面的一个节点，并将它作为新的当前节点，同时转到第3步。

这一算法是依次将每一个节点作为一棵树的根节点，并进行深度优先搜索。如果再次碰到已经遇到过的节点，那么就算找到了一个环。如果从任何给定的节点出发的弧都被穷举了，那么就回溯到前面的节点。如果回溯到根并且不能再深入下去，那么从当前节点出发的子图中就不包含任何环。如果所有的节点都是如此，那么整个图就不存在环，也就是说系统不存在死锁。

为了验证一下该算法是如何工作的，我们对图6-5a运用该算法。算法对节点次序的要求是任意的，所以可以选择从左到右、从上到下进行检测，首先从R节点开始运行该算法，然后依次从A、B、C、S、D、T、E、F开始。如果遇到了一个环，那么算法停止。

我们先从R节点开始，并将L初始化为空表。然后将R添加到空表中，并移动到惟一可能的节点A，将它添加到L中，变成L=[R，A]。从A我们到达S，并使L=[R，A，S]。S没有出发的弧，所以它是条死路，迫使我们回溯到A。既然A没有任何没有标记的出发弧，我们再回溯到R，从而完成了以R为起始点的检测。

现在我们重新以A为起始点启动该算法，并重置L为空表。这次检索也很快就结束了，所以我们又从B开始。从B节点我们顺着弧到达D，这时L=[B，T，E，V，G，U，D]。现在我们必须随机选择。如果选S点，那么走进了死胡同并回溯到D。接着选T并将L更新为[B，T，E，V，G，U，D，T]，在这一点上我们发现了环，算法结束。

这种算法远不是最佳算法，较好的一种算法参见（Even,1979）。但毫无疑问，该实例表明确实存在检测死锁的算法。

6.4.2 每种类型多个资源的死锁检测

如果有多种相同的资源存在，就需要采用另一种方法来检测死锁。现在我们提供一种基于矩阵的算法来检测从P_1到P_n这n个进程中的死锁。假设资源的类型数为m，E_1代表资源类型1，E_2代表资源类型2，E_i代表资源类型i($1 \leqslant i \leqslant m$)。$E$是*现有资源向量*（existing resource vector），代表每种已存在的资源总数。比如，如果资源类型1代表磁带机，那么E_1=2就表示系统有两台磁带机。

在任意时刻，某些资源已被分配所以不可用。假设A是*可用资源向量*（available resource vector），那么A_i表示当前可供使用的资源数（即没有被分配的资源）。如果仅有的两台磁带机都已经分配出去了，那么A_1的值为0。

现在我们需要两个数组：C代表*当前分配矩阵*（current allocation matrix），R代表*请求矩阵*（request matrix）。C的第i行代表P_i当前所持有的每一种类型资源的资源数。所以，C_{ij}代表进程i所持有的资源j的数量。同理，R_{ij}代表P_i所需要的资源j的数量。这四种数据结构如图6-6所示。

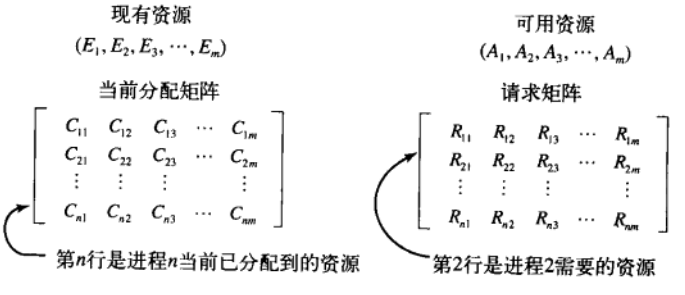


图6-6 死锁检测算法所需的四种数据结构

这四种数据结构之间有一个重要的恒等式。具体地说，某种资源要么已分配要么可用。这个结论意味着：

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

换言之，如果我们将所有已分配的资源j的数量加起来再和所有可供使用的资源数相加，结果就是该类资源的资源总数。

死锁检测算法就是基于向量的比较。我们定义向量A和向量B之间的关系为$A \leqslant B$以表明A的每一个分量要么等于要么小于和B向量相对应的分量。从数学上来说，$A \leqslant B$当且仅当且$A_i \leqslant B_i (0 \leqslant i \leqslant m)$。

每个进程起初都是没有标记过的。算法开始会对进程做标记，进程被标记后就表明它们能够被执行，不会进入死锁。当算法结束时，任何没有标记的进程都是死锁进程。该算法假定了一个最坏情形：所有的进程在退出以前都会不停地获取资源。

死锁检测算法如下：

1) 寻找一个没有标记的进程P_i，对于它而言R矩阵的第i行向量小于或等于A。

2) 如果找到了这样一个进程，那么将C矩阵的第i行向量加到A中，标记该进程，并转到第1步。

3) 如果没有这样的进程，那么算法终止。

算法结束时，所有没有标记过的进程（如果存在的话）都是死锁进程。

算法的第1步是寻找可以运行完毕的进程，该进程的特点是它有资源请求并且该请求可被当前的可用资源满足。这一选中的进程随后就被运行完毕，在这段时间内它释放自己持有的所有资源并将它们返回到可用资源库中。然后，这一进程被标记为完成。如果所有的进程最终都能运行完毕的话，就不存在死锁的情况。如果其中某些进程一直不能运行，那么它们就是死锁进程。虽然算法的运行过程是不确定的（因为进程可按任何行得通的次序执行），但结果总是相同的。

作为一个例子，在图6-7中展示了用该算法检测死锁的工作过程。这里我们有3个进程、4种资源（可以任意地将它们标记为磁带机、绘图仪、扫描仪和CD-ROM驱动器）。进程1有一台扫描仪。进程2有2台磁带机和1个CD-ROM驱动器。进程3有1个绘图仪和2台扫描仪。每一个进程都需要额外的资源，如矩阵R所示。

要运行死锁检测算法，首先找出哪一个进程的资源请求可被满足。第1个不能被满足，因为没有CD-ROM驱动器可供使用。第2个也不能被满足，由于没有打印机空闲。幸运的是，第3个可被满足，所以进程3运行并最终释放它所拥有的资源，给出

$$A = (2\ 2\ 2\ 0)$$

接下来，进程2也可运行并释放它所拥有的资源，给出

$$A = (4\ 2\ 2\ 1)$$

现在剩下的进程都能够运行，所以这个系统中不存在死锁。

假设图6-7的情况有所改变。进程2需要1个CD-ROM驱动器、2台磁带机和1台绘图仪。在这种情况下，所有的请求都不能得到满足，整个系统进入死锁。

现在我们知道了如何检测死锁（至少是在这种预先知道静态资源请求的情况下），但问题在于何时去检测它们。一种方法是每当有资源请求时去检测。毫无疑问越早发现越好，但这种方法会占用昂贵的CPU时间。另一种方法是每隔k分钟检测一次，或者当CPU的使用率降到某一域值时去检测。考虑到CPU使用效率的原因，如果死锁进程数达到一定数量，就没有多少进程可运行了，所以CPU会经常空闲。

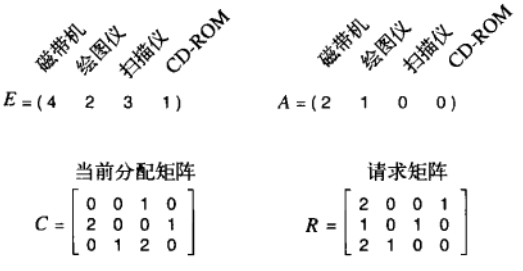


图6-7　死锁检测算法的一个例子

6.4.3　从死锁中恢复

假设我们的死锁检测算法已成功地检测到了死锁，那么下一步该怎么办？当然需要一些方法使系统重新正常工作。在本小节中，我们会讨论各种从死锁中恢复的方法，尽管这些方法看起来都不那么令人满意。

1. 利用抢占恢复

在某些情况下，可能会临时将某个资源从它的当前所有者那里转移到另一个进程。许多情况下，尤其是对运行在大型主机上的批处理操作系统来说，需要人工进行干预。

比如，要将激光打印机从它的持有进程那里拿走，管理员可以收集已打印好的文档并将其堆积在一

旁。然后，该进程被挂起（标记为不可运行）。接着，打印机被分配给另一个进程。当那个进程结束后，堆在一旁的文档再被重新放回原处，原进程可重新继续工作。

在不通知原进程的情况下，将某一资源从一个进程强行取走给另一个进程使用，接着又送回，这种做法是否可行主要取决于该资源本身的特性。用这种方法恢复通常比较困难或者说不太可能。若选择挂起某个进程，则在很大程度上取决于哪一个进程拥有比较容易收回的资源。

2. 利用回滚恢复

如果系统设计人员以及主机操作员了解到死锁有可能发生，他们就可以周期性地对进程进行检查点检查（checkpointed）。进程检查点检查就是将进程的状态写入一个文件以备以后重启。该检查点中不仅包括存储映像，还包括了资源状态，即哪些资源分配给了该进程。为了使这一过程更有效，新的检查点不应覆盖原有的文件，而应写到新文件中。这样，当进程执行时，将会有一系列的检查点文件被累积起来。

一旦检测到死锁，就很容易发现需要哪些资源。为了进行恢复，要从一个较早的检查点上开始，这样拥有所需要资源的进程会回滚到一个时间点，在此时间点之前该进程获得了一些其他的资源。在该检查点后所做的所有工作都丢失。（例如，检查点之后的输出必须丢弃，因为它们还会被重新输出。）实际上，是将该进程复位到一个更早的状态，那时它还没有取得所需的资源，接着就把这个资源分配给一个死锁进程。如果复位后的进程试图重新获得对该资源的控制，它就必须一直等到该资源可用时为止。

3. 通过杀死进程恢复

最直接也是最简单的解决死锁的方法是杀死一个或若干个进程。一种方法是杀掉环中的一个进程。如果走运的话，其他进程将可以继续。如果这样做行不通的话，就需要继续杀死别的进程直到打破死锁环。

另一种方法是选一个环外的进程作为牺牲品以释放该进程的资源。在使用这种方法时，选择一个要被杀死的进程要特别小心，它应该正好持有环中某些进程所需的资源。比如，一个进程可能持有一台绘图仪而需要一台打印机，而另一个进程可能持有一台打印机而需要一台绘图仪，因而这两个进程是死锁的。第三个进程可能持有另一台同样的打印机和另一台同样的绘图仪而且正在运行着。杀死第三个进程将释放这些资源，从而打破前两个进程的死锁。

有可能的话，最好杀死可以从头开始重新运行而且不会带来副作用的进程。比如，编译进程可以被重复运行，由于它只需要读入一个源文件和产生一个目标文件。如果将它中途杀死，它的第一次运行不会影响到第二次运行。

另一方面，更新数据库的进程在第二次运行时并非总是安全的。如果一个进程将数据库的某个记录加1，那么运行它一次，将它杀死后，再次执行，就会对该记录加2，这显然是错误的。

6.5 死锁避免

在讨论死锁检测时，我们假设当一个进程请求资源时，它一次就请求所有的资源（见图6-6中的矩阵R）。不过在大多数系统中，一次只请求一个资源。系统必须能够判断分配资源是否安全，并且只能在保证安全的条件下分配资源。问题是：是否存在一种算法总能做出正确的选择从而避免死锁？答案是肯定的，但条件是必须事先获得一些特定的信息。本节我们会讨论几种死锁避免的方法。

6.5.1 资源轨迹图

避免死锁的主要算法是基于一个安全状态的概念。在描述算法前，我们先讨论有关安全的概念。通过图的方式，能更容易理解。虽然图的方式不能被直接翻译成有用的算法，但它给出了一个解决问题的直观感受。

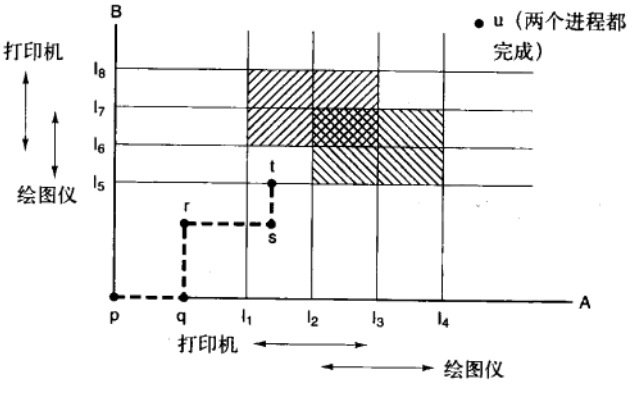


图6-8 两个进程的资源轨迹图

在图6-8中，我们看到一个处理两

个进程和两种资源（打印机和绘图仪）的模型。横轴表示进程A执行的指令，纵轴表示进程B执行的指令。进程A在I_1处请求一台打印机，在I_3处释放，在I_2处请求一台绘图仪，在I_4处释放。进程B在I_5到I_7之间需要绘图仪，在I_6到I_8之间需要打印机。

图6-8中的每一点都表示出两个进程的连接状态。初始点为p，没有进程执行任何指令。如果调度程序选中A先运行，那么在A执行一段指令后到达q，此时B没有执行任何指令。在q点，如果轨迹沿垂直方向移动，表示调度程序选中B运行。在单处理机情况下，所有路径都只能是水平或垂直方向的，不会出现斜向的。因此，运动方向一定是向上或向右，不会向左或向下，因为进程的执行不可能后退。

当进程A由r向s移动穿过I_1线时，它请求并获得打印机。当进程B到达t时，它请求绘图仪。

图中的阴影部分是我们感兴趣的，画着从左下到右上斜线的部分表示在该区域中两个进程都拥有打印机，而互斥使用的规则决定了不可能进入该区域。另一种斜线的区域表示两个进程都拥有绘图仪，且同样不可进入。

如果系统一旦进入由I_1、I_2和I_5、I_6组成的矩形区域，那么最后一定会到达I_2和I_6的交叉点，这时就产生死锁。在该点处，A请求绘图仪，B请求打印机，而且这两种资源均已被分配。这整个矩形区域都是不安全的，因此决不能进入这个区域。在点t处惟一的办法是运行进程A直到I_4，过了I_4后，可以按任何路线前进，直到终点u。

需要注意的是，在点t进程B请求资源。系统必须决定是否分配。如果系统把资源分配给B，系统进入不安全区域，最终形成死锁。要避免死锁，应该将B挂起，直到A请求并释放绘图仪。

6.5.2 安全状态和不安全状态

我们将要研究的死锁避免算法使用了图6-6中的有关信息。在任何时刻，当前状态包括了*E*、*A*、*C*和*R*。如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。通过使用一个资源的例子很容易说明这个概念。在图6-9a中有一个A拥有3个资源实例但最终可能会需要9个资源实例的状态。B当前拥有2个资源实例，将来共需要4个资源实例。同样，C拥有2个资源实例，还需要另外5个资源实例。总共有10个资源实例，其中有7个资源已经分配，还有3个资源是空闲的。

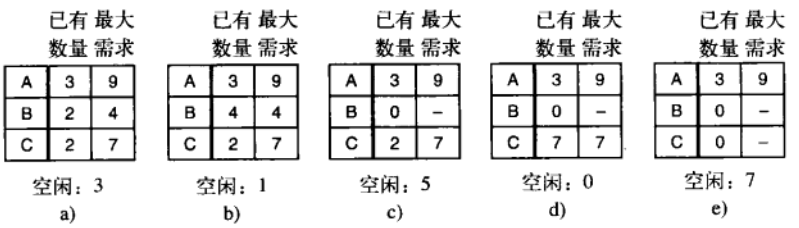


图6-9　说明a中的状态为安全状态

图6-9a的状态是安全的，这是由于存在一个分配序列使得所有的进程都能完成。也就是说，这个方案可以单独地运行B，直到它请求并获得另外两个资源实例，从而到达图6-9b的状态。当B完成后，就到达了图6-9c的状态。然后调度程序可以运行C，再到达图6-9d的状态。当C完成后，到达了图6-9e的状态。现在A可以获得它所需要的6个资源实例，并且完成。这样系统通过仔细的调度，就能够避免死锁，所以图6-9a的状态是安全的。

现在假设初始状态如图6-10a所示。但这次A请求并得到另一个资源，如图6-10b所示。我们还能找到一个序列来完成所有工作吗？我们来试一试。调度程序可以运行B，直到B获得所需资源，如图6-10c所示。

最终，进程B完成，状态如图6-10d所示，此时进入困境了。只有4个资源实例空闲，并且所有活动进程都需要5个资源实例。任何分配资源实例的序列都无法保证工作的完成。于是，从图6-10a到图6-10b的分配方案，从安全状态进入到了不安全状态。从图6-10c的状态出发运行进程A或C也都不行。回过头来再看，A的请求不应该满足。

值得注意的是，不安全状态并不是死锁。从图6-10b出发，系统能运行一段时间。实际上，甚至有

一个进程能够完成。而且，在A请求其他资源实例前，A可能先释放一个资源实例，这就可以让C先完成，从而避免了死锁。因而，安全状态和不安全状态的区别是：从安全状态出发，系统能够保证所有进程都能完成；而从不安全状态出发，就没有这样的保证。

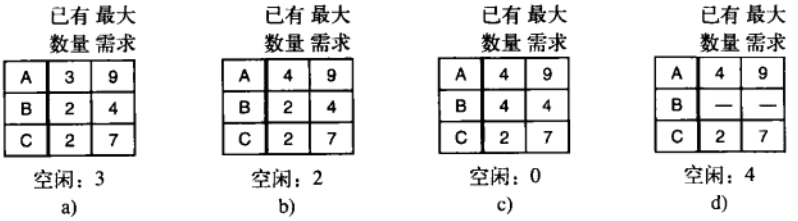


图6-10 说明b中的状态为不安全状态

6.5.3 单个资源的银行家算法

Dijkstra（1965）提出了一种能够避免死锁的调度算法，称为*银行家算法*（banker's algorithm），这是6.4.1节中给出的死锁检测算法的扩展。该模型基于一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度。算法要做的是判断对请求的满足是否会导致进入不安全状态。如果是，就拒绝请求；如果满足请求后系统仍然是安全的，就予以分配。在图6-11a中我们看到4个客户A、B、C、D，每个客户都被授予一定数量的贷款单位（比如1单位是1千美元），银行家知道不可能所有客户同时都需要最大贷款额，所以他只保留10个单位而不是22个单位的资金来为客户服务。这里将客户比作进程，贷款单位比作资源，银行家比作操作系统。

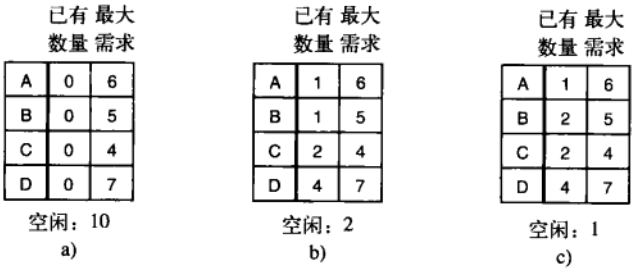


图6-11 三种资源分配状态：a) 安全；b) 安全；c) 不安全

客户们各自做自己的生意，在某些时刻需要贷款（相当于请求资源）。在某一时刻，具体情况如图6-11b所示。这个状态是安全的，由于保留着2个单位，银行家能够拖延除了C以外的其他请求。因而可以让C先完成，然后释放C所占的4个单位资源。有了这4个单位资源，银行家就可以给D或B分配所需的贷款单位，以此类推。

考虑假如向B提供了另一个他所请求的贷款单位，如图6-11b所示，那么我们就有如图6-11c所示的状态，该状态是不安全的。如果忽然所有的客户都请求最大的限额，而银行家无法满足其中任何一个的要求，那么就会产生死锁。不安全状态并不一定引起死锁，由于客户不一定需要其最大贷款额度，但银行家不敢抱这种侥幸心理。

银行家算法就是对每一个请求进行检查，检查如果满足这一请求是否会达到安全状态。若是，那么就满足该请求；若否，那么就推迟对这一请求的满足。为了看状态是否安全，银行家看他是否有足够的资源满足某一个客户。如果可以，那么这笔投资认为是能够收回的，并且接着检查最接近最大限额的一个客户，以此类推。如果所有投资最终都被收回，那么该状态是安全的，最初的请求可以批准。

6.5.4 多个资源的银行家算法

可以把银行家算法进行推广以处理多个资源。图6-12说明了多个资源的银行家算法如何工作。

在图6-12中我们看到两个矩阵。左边的矩阵显示出为5个进程分别已分配的各种资源数，右边的矩

阵显示了使各进程完成运行所需的各种资源数。这些矩阵就是图6-6中的C和R。和一个资源的情况一样，各进程在执行前给出其所需的全部资源量，所以在系统的每一步中都可以计算出右边的矩阵。

图6-12最右边的三个向量分别表示现有资源E、已分配资源P和可用资源A。由E可知系统中共有6 台磁带机、3台绘图仪、4台打印机和2台CD-ROM驱动器。由P可知当前已分配了5台磁带机、3台绘图仪、2台打印机和2台CD-ROM驱动器。该向量可通过将左边矩阵的各列相加获得，可用资源向量可通过从现有资源中减去已分配资源获得。

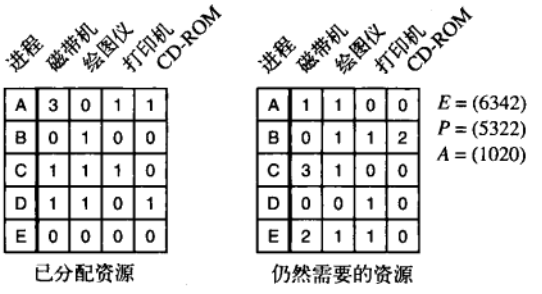


图6-12　多个资源的银行家算法

检查一个状态是否安全的算法如下：

1) 查找右边矩阵中是否有一行，其没有被满足的资源数均小于或等于A。如果不存在这样的行，那么系统将会死锁，因为任何进程都无法运行结束（假定进程会一直占有资源直到它们终止为止）。

2) 假若找到这样一行，那么可以假设它获得所需的资源并运行结束，将该进程标记为终止，并将其资源加到向量A上。

3) 重复以上两步，或者直到所有的进程都标记为终止，其初始状态是安全的；或者所有进程的资源需求都得不到满足，此时就是发生了死锁。

如果在第1步中同时有若干进程均符合条件，那么不管挑选哪一个运行都没有关系，因为可用资源或者会增多，或者至少保持不变。

图6-12中所示的状态是安全的，若进程B现在再请求一台打印机，可以满足它的请求，因为所得系统状态仍然是安全的（进程D可以结束，然后是A或E结束，剩下的进程相继结束）。

假设进程B获得两台可用打印机中的一台以后，E试图获得最后一台打印机，假若分配给E，可用资源向量会减到（1 0 0 0），这时会引起死锁。显然E的请求不能立即满足，必须延迟一段时间。

银行家算法最早由Dijkstra 于1965年发表。从那之后几乎每本操作系统的专著都详细地描述它，很多论文的内容也围绕该算法讨论了它的不同方面。但很少有作者指出该算法虽然很有意义但缺乏实用价值，因为很少有进程能够在运行前就知道其所需资源的最大值。而且进程数也不是固定的，往往在不断地变化（如新用户的登录或退出），况且原本可用的资源也可能突然间变成不可用（如磁带机可能会坏掉）。因此，在实际中，如果有，也只有极少的系统使用银行家算法来避免死锁。

6.6　死锁预防

通过前面的学习我们知道，死锁避免从本质上来说是不可能的，因为它需要获知未来的请求，而这些请求是不可知的。那么实际的系统又是如何避免死锁的呢？我们回顾Coffman 等人（1971）所述的四个条件，看是否能发现线索。如果能够保证四个条件中至少有一个不成立，那么死锁将不会产生（Havender，1968）。

6.6.1　破坏互斥条件

先考虑破坏互斥使用条件。如果资源不被一个进程所独占，那么死锁肯定不会产生。当然，允许两个进程同时使用打印机会造成混乱，通过采用假脱机打印机（spooling printer）技术可以允许若干个进程同时产生输出。该模型中惟一真正请求使用物理打印机的进程是打印机守护进程，由于守护进程决不会请求别的资源，所以不会因打印机而产生死锁。

假设守护进程被设计为在所有输出进入假脱机之前就开始打印，那么如果一个输出进程在头一轮打印之后决定等待几个小时，打印机就可能空置。为了避免这种现象，一般将守护进程设计成在完整的输出文件就绪后才开始打印。例如，若两个进程分别占用了可用的假脱机磁盘空间的一半用于输出，而任何一个也没有能够完成输出，那么会怎样？在这种情形下，就会有两个进程，其中每一个都完成了部分的输出，但不是它们的全部输出，于是无法继续进行下去。没有一个进程能够完成，结果在磁盘上出现了死锁。

不过，有一个小思路是经常可适用的。那就是，避免分配那些不是绝对必需的资源，尽量做到尽可能少的进程可以真正请求资源。

6.6.2 破坏占有和等待条件

Coffman 等表述的第二个条件似乎更有希望。只要禁止已持有资源的进程再等待其他资源便可以消除死锁。一种实现方法是规定所有进程在开始执行前请求所需的全部资源。如果所需的全部资源可用，那么就将它们分配给这个进程，于是该进程肯定能够运行结束。如果有一个或多个资源正被使用，那么就不进行分配，进程等待。

这种方法的一个直接问题是很多进程直到运行时才知道它需要多少资源。实际上，如果进程能够知道它需要多少资源，就可以使用银行家算法。另一个问题是这种方法的资源利用率不是最优的。例如，有一个进程先从输入磁带上读取数据，进行一小时的分析，最后会写到输出磁带上，同时会在绘图仪上绘出。如果所有资源都必须提前请求，这个进程就会把输出磁带机和绘图仪控制住一小时。

不过，一些大型机批处理系统要求用户在所提交的作业的第一行列出它们需要多少资源。然后，系统立即分配所需的全部资源，并且直到作业完成才回收资源。虽然这加重了编程人员的负担，也造成了资源的浪费，但这的确防止了死锁。

另一种破坏占有和等待条件的略有不同的方案是，要求当一个进程请求资源时，先暂时释放其当前占用的所有资源，然后再尝试一次获得所需的全部资源。

6.6.3 破坏不可抢占条件

破坏第三个条件（不可抢占）也是可能的。假若一个进程已分配到一台打印机，且正在进行打印输出，如果由于它需要的绘图仪无法获得而强制性地把它占有的打印机抢占掉，会引起一片混乱。但是，一些资源可以通过虚拟化的方式来避免发生这样的情况。假脱机打印机向磁盘输出，并且只允许打印机守护进程访问真正的物理打印机，这种方式可以消除涉及打印机的死锁，然而却可能带来由磁盘空间导致的死锁。但是对于大容量磁盘，要消耗完所有的磁盘空间一般是不可能的。

然而，并不是所有的资源都可以进行类似的虚拟化。例如，数据库中的记录或者操作系统中的表都必须被锁定，因此存在出现死锁的可能。

6.6.4 破坏环路等待条件

现在只剩下一个条件了。消除环路等待有几种方法。一种是保证每一个进程在任何时刻只能占用一个资源，如果要请求另外一个资源，它必须先释放第一个资源。但假若进程正在把一个大文件从磁带机上读入并送到打印机打印，那么这种限制是不可接受的。

另一种避免出现环路等待的方法是将所有资源统一编号，如图6-13a所示。现在的规则是：进程可以在任何时刻提出资源请求，但是所有请求必须按照资源编号的顺序（升序）提出。进程可以先请求打印机后请求磁带机，但不可以先请求绘图仪后请求打印机。

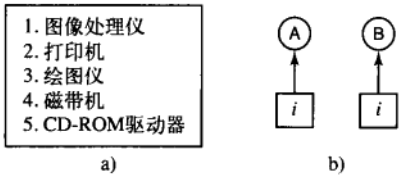


图6-13 a) 对资源排序编号；
b) 一个资源分配图

若按此规则，资源分配图中肯定不会出现环。让我们看看在有两个进程的情形下为何可行，参看图6-13b。只有在A请求资源j且B请求资源i的情况下会产生死锁。假设i和j是不同的资源，它们会具有不同的编号。若$i>j$，那么A不允许请求j，因为这个编号小于A已有资源的编号；若$i<j$，那么B不允许请求i，因为这个编号小于B已有资源的编号。不论哪种情况都不可能产生死锁。

对于多于两个进程的情况，同样的逻辑依然成立。在任何时候，总有一个已分配的资源是编号最高的。占用该资源的进程不可能请求其他已分配的各种资源。它或者会执行完毕，或者最坏的情形是去请求编号更高的资源，而编号更高的资源肯定是可用的。最终，它会结束并释放所有资源，这时其他占有最高编号资源的进程也可以执行完。简言之，存在一种所有进程都可以执行完毕的情景，所以不会产生死锁。

该算法的一个变种是摈弃必须按升序请求资源的限制，而仅仅要求不允许进程请求比当前所占有资

源编号低的资源。所以，若一个进程起初请求9号和10号资源，而随后释放两者，那么它实际上相当于从头开始，所以没有必要阻止它现在请求1号资源。

尽管对资源编号的方法消除了死锁的问题，但几乎找不出一种使每个人都满意的编号次序。当资源包括进程表项、假脱机磁盘空间、加锁的数据库记录及其他抽象资源时，潜在的资源及各种不同用途的数目会变得很大，以至于使编号方法根本无法使用。

条　件	处理方式
互斥	一切都使用假脱机技术
占有和等待	在开始就请求全部资源
不可抢占	抢占资源
环路等待	对资源按序编号

图6-14　死锁预防方法汇总

死锁预防的各种方法如图6-14所示。

6.7　其他问题

在本节中，我们会讨论一些和死锁相关的问题，包括两阶段加锁、通信死锁、活锁和饥饿。

6.7.1　两阶段加锁

虽然在一般情况下避免死锁和预防死锁并不是很有希望，但是在一些特殊的应用方面，有很多卓越的专用算法。例如，在很多数据库系统中，一个经常发生的操作是请求锁住一些记录，然后更新所有锁住的记录。当同时有多个进程运行时，就有出现死锁的危险。

常用的方法是*两阶段加锁*（two-phase locking）。在第一阶段，进程试图对所有所需的记录进行加锁，一次锁一个记录。如果第一阶段加锁成功，就开始第二阶段，完成更新然后释放锁。在第一阶段并没有做实际的工作。

如果在第一阶段某个进程需要的记录已经被加锁，那么该进程释放它所有加锁的记录，然后重新开始第一阶段。从某种意义上说，这种方法类似于提前或者至少是未实施一些不可逆的操作之前请求所有资源。在两阶段加锁的一些版本中，如果在第一阶段遇到了已加锁的记录，并不会释放锁然后重新开始，这就可能产生死锁。

不过，在一般意义下，这种策略并不通用。例如，在实时系统和进程控制系统中，由于一个进程缺少一个可用资源就半途中断它，并重新开始该进程，这是不可接受的。如果一个进程已经在网络上读写消息、更新文件或从事任何不能安全地重复做的事，那么重新运行进程也是不可接受的。只有当程序员仔细地安排了程序，使得在第一阶段程序可以在任意一点停下来，并重新开始而不会产生错误，这时这个算法才可行。但很多应用并不能按这种方式来设计。

6.7.2　通信死锁

到目前为止，我们所有的工作都着眼于资源死锁。一个进程需要使用另外一个进程拥有的资源，因此必须等待直至该进程停止使用这些资源。有时资源是硬件或者软件，比如说CD-ROM驱动器或者数据库记录，但是有时它们更加抽象。在图6-2中，可以看到当资源互斥时发生的资源死锁。这比CD-ROM驱动器更抽象一点，但是在这个例子中，每个进程都成功调用一个资源（互斥锁之一）而且死锁的进程尝试去调用另外的资源（另一个互斥锁）。这种情况是典型的资源死锁。

然而，正如我们在本章开始提到的，资源死锁是最普遍的一种类型，但不是惟一的一种。另一种死锁发生在通信系统中（比如说网络），即两个或两个以上进程利用发送信息来通信时。一种普遍的情形是进程A向进程B发送请求信息，然后阻塞直至B回复。假设请求信息丢失，A将阻塞以等待回复，而B会阻塞等待一个向其发送命令的请求，因此发生死锁。

仅仅如此并非经典的资源死锁。A没有占有B所需的资源，反之亦然。事实上，并没有完全可见的资源。但是，根据标准的定义，在一系列进程中，每个进程因为等待另外一个进程引发的事件而产生阻塞，这就是一种死锁。相比于更加常见的资源死锁，我们把上面这种情况叫做*通信死锁*（communication deadlock）。

通信死锁不能通过对资源排序（因为没有）或者通过仔细地安排调度来避免（因为任何时刻的请求都是不被允许延迟的）。幸运的是，另外一种技术通常可以用来中断通信死锁：超时。在大多数网络通信系统中，只要一个信息被发送至一个特定的地方，并等待其返回一个预期的回复，发送者就同时启动

计时器。若计时器在回复到达前计时就停止了，则信息的发送者可以认定信息已经丢失，并重新发送（如果需要，则一直重复）。通过这种方式，可以避免死锁。

当然，如果原始信息没有丢失，而仅仅是回复延时，接收者可能收到两次或者更多次信息，甚至导致意想不到的结果。想象电子银行系统中包含付款说明的信息。很明显，不应该仅仅因为网速缓慢或者超时设定太短，就重复（并执行）多次。应该将通信规则——通常称为协议（protocol）——设计为让所有事情都正确，这是一个复杂的课题，超出了本书的范围。对网络协议感兴趣的读者可以参考作者的另外一本书——《Computer Networks》(Tanenbaum，2003)。

并非所有在通信系统或者网络发生的死锁都是通信死锁。资源死锁也会发生，如图6-15中的网络。这张图是因特网的简化图（极其简化）。因特网由两类计算机组成：主机和路由器。主机（host）是一台用户计算机，可以是某人家里的PC机、公司的个人计算机，也可能是一个共享服务器。主机由人来操作。路由器（router）是专用的通信计算机，将数据包从源发送至目的地。每台主机都连接一个或更多的路由器，可以用一条DSL线、有线电视连接、局域网、拨号线路、无线网络、光纤等来连接。

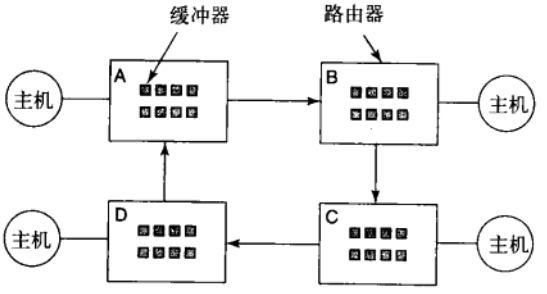


图6-15 一个网络中的资源死锁

当一个数据包从一个主机进入路由器时，它被放入一个缓冲器中，然后传输到另外一个路由器，再到另一个，直至目的地。这些缓冲器都是资源并且数目有限。在图6-15中，每个路由器都有8个缓冲器（实际应用中有数以百万计，但是并不能改变潜在死锁的本质，只是改变了它的频率）。假设路由器A的所有数据包需要发送到B，B的所有数据包需要发送到C，C的所有数据包需要发送到D，然后D的所有数据包需要发送到A。那么没有数据包可以移动，因为在另一端没有缓冲器。这就是一个典型的资源死锁，尽管它发生在通信系统中。

6.7.3 活锁

在某种情形下，轮询（忙等待）可用于进入临界区或存取资源。采用这一策略的主要原因是，相比所做的工作而言，互斥的时间很短而挂起等待的时间开销很大。考虑一个原语，通过该原语，调用进程测试一个互斥信号量，然后或者得到该信号量或者返回失败信息。如图2-26中的例子所示。

现在假设有一对进程使用两种资源，如图6-16所示。每个进程需要两种资源，它们利用轮询原语enter_region去尝试取得必要的锁，如果尝试失败，则该进程继续尝试。在图6-16中，如果进程A先运行并得到资源1，然后进程2运行并得到资源2，以后不管哪一个进程运行，都不会有任何进展，但是哪一个进程也没有被阻塞。结果是两个进程总是一再消耗完分配给它们的CPU配额，但是没有进展也没有阻塞。因此，没有出现死锁现象（因为没有进程阻塞），但是从现象上看好像死锁发生了，这就是活锁（livelock）。

```
void process_A(void) {
        enter_region(&resource_1);
        enter_region(&resource_2);
        use_both_resources( );
        leave_region(&resource_2);
        leave_region(&resource_1);
}

void process_B(void) {
        enter_region(&resource_2);
        enter_region(&resource_1);
        use_both_resources( );
        leave_region(&resource_1);
        leave_region(&resource_2);
}
```

图6-16 忙等待可能导致活锁

活锁也经常出人意料地产生。在一些系统中，进程表中容纳的进程数决定了系统允许的最大进程数量，因此进程表属于有限的资源。如果由于进程表满了而导致一次fork运行失败，那么一个合理的方法是：该程序等待一段随机长的时间，然后再次尝试运行fork。

现在假设一个UNIX系统有100个进程槽，10个程序正在运行，每个程序需要创建12个（子）进程。在每个进程创建了9个进程后，10个源进程和90个新的进程就已经占满了进程表。10个源进程此时便进

入了死锁——不停地进行分支循环和运行失败。发生这种情况的可能性是极小的，但是，这是可能发生的！我们是否应该放弃进程以及fork调用来消除这个问题呢？

限制打开文件的最大数量与限制索引节点表的大小的方式很相像，因此，当它被完全占用的时候，也会出现相似的问题。硬盘上的交换空间是另一个有限的资源。事实上，几乎操作系统中的每种表都代表了一种有限的资源。如果有n个进程，每个进程都申请了$1/n$的资源，然后每一个又试图申请更多的资源，这种情况下我们是不是应该禁掉所有的呢？也许这不是一个好主意。

大多数的操作系统（包括UNIX和Windows）都忽略了一个问题，即比起限制所有用户去使用一个进程、一个打开的文件或任意一种资源来说，大多数用户可能更愿意选择一次偶然的活锁（或者甚至是死锁）。如果这些问题能够免费消除，那就不会有争论。但问题是代价非常高，因而几乎都是给进程加上不便的限制来处理。因此我们面对的问题是从便捷性和正确性中做出取舍，以及一系列关于哪个更重要、对谁更重要的争论。

值得一提的是，一些人对饥饿（缺乏资源）和死锁并不作区分，因为在两种情况下都没有下一步操作了。还有些人认为它们从根本上不同，因为可以很轻易地编写一个进程，让它做某个操作n次，并且如果它们都失败了，再试试其他的就可以了。一个阻塞的进程就没有那样的选择了。

6.7.4 饥饿

与死锁和活锁非常相似的一个问题是饥饿（starvation）。在动态运行的系统中，在任何时刻都可能请求资源。这就需要一些策略来决定在什么时候谁获得什么资源。虽然这个策略表面上很有道理，但依然有可能使一些进程永远得不到服务，虽然它们并不是死锁进程。

作为一个例子，考虑打印机分配。设想系统采用某种算法来保证打印机分配不产生死锁。现在假设若干进程同时都请求打印机，究竟哪一个进程能获得打印机呢？

一个可能的分配方案是把打印机分配给打印最小文件的进程（假设这个信息可知）。这个方法让尽量多的顾客满意，并且看起来很公平。我们考虑下面的情况：在一个繁忙的系统中，有一个进程有一个很大的文件要打印，每当打印机空闲，系统纵观所有进程，并把打印机分配给打印最小文件的进程。如果存在一个固定的进程流，其中的进程都是只打印小文件，那么，要打印大文件的进程永远也得不到打印机。很简单，它会“饥饿而死”（无限制地推后，尽管它没有被阻塞）。

饥饿可以通过先来先服务资源分配策略来避免。在这种机制下，等待最久的进程会是下一个被调度的进程。随着时间的推移，所有进程都会变成最“老”的，因而，最终能够获得资源而完成。

6.8 有关死锁的研究

死锁在操作系统发展的早期就作为一个课题被详细地研究过。死锁的检测是一个经典的图论问题，任何对数学有兴趣的研究生都可以在其上做3～4年的研究。所有相关的算法都已经经过了反复修正，但每次修正总是得到更古怪、更不现实的算法。大部分工作已经结束了，但是仍然有很多关于死锁各方面内容的论文发表。这些论文包括由于错误使用锁和信号量而导致的死锁的运行时间检测（Agarwal和Stoller, 2006；Bensalem等人, 2006）；在Java线程中预防死锁（Permandia等人，2007；Williams等人，2005）；处理网络上的死锁（Jayasimha, 2003；Karol等人，2003；Schafer等人，2005）；数据流系统中的死锁建模（Zhou和Lee, 2006）；检测动态死锁（Li等人，2005）。Levine（2003a, 2003b）比较了文献中关于死锁各种不同的（经常相矛盾的）定义，从而提出了一个分类方案。她也从另外的角度分析了关于预防死锁和避免死锁的区别（Levine，2005）。而死锁的恢复也是一个正在研究的问题（David等人，2007）。

然而，还有一些（理论）研究是关于分布式死锁检测的，我们在这里不做表述，因为它超出了本书的范围，而且这些研究在实际系统中的应用非常少，似乎只是为了让一些图论家有事可做罢了。

6.9 小结

死锁是任何操作系统的潜在问题。在一组进程中，每个进程都因等待由该组进程中的另一进程所占有的资源而导致阻塞，死锁就发生了。这种情况会使所有的进程都处于无限等待的状态。一般来讲，这是进程一直等待被其他进程占用的某些资源释放的事件。死锁的另外一种可能的情况是一组通信进程都

在等待一个消息，而通信信道却是空的，并且也没有采用超时机制。

通过跟踪哪一个状态是安全状态，哪一个状态是不安全状态，可以避免死锁。安全状态就是这样一个状态：存在一个事件序列，保证所有的进程都能完成。不安全状态就不存在这样的保证。银行家算法可以通过拒绝可能引起不安全状态的请求来避免死锁。

也可以在设计系统时就不允许死锁发生，从而在系统结构上预防死锁的发生。例如，只允许进程在任何时刻最多占有一个资源，这就破坏了循环等待环路。也可以将所有的资源编号，规定进程按严格的升序请求资源，这样也能预防死锁。

资源死锁并不是惟一的一种死锁。尽管我们可以通过设置适当的超时机制来解决通信死锁，但它依然是某些系统中潜在的问题。

活锁和死锁的问题有些相似，那就是它也可以停止所有的转发进程，但是二者在技术上不同，由于活锁包含了一些实际上并没有锁住的进程，因此可以通过先来先服务的分配策略来避免饥饿。

习题

1. 给出一个由策略产生的死锁的例子。
2. 学生们在机房的个人计算机上将自己要打印的文件发送给服务器，服务器会将这些文件暂存在它的硬盘上。如果服务器磁盘空间有限，那么，在什么情况下会产生死锁？这样的死锁应该怎样避免？
3. 在图6-1中，资源释放的顺序与获得的顺序相反，以其他的顺序释放资源能否得到同样的结果？
4. 一个资源死锁的发生有四个必要条件（互斥使用资源、占有和等待资源、不可抢占资源和环路等待资源）。举一个例子说明这些条件对于一个资源死锁的发生不是充分的。何时这些条件对一个资源死锁的发生是充分条件？
5. 图6-3给出了资源分配图的概念，试问是否存在不合理的资源分配图，即资源分配图在结构上违反了使用资源的模型？如果存在，请给出一个例子。
6. 假设一个系统中存在一个资源死锁。举一个例子说明死锁的进程集合中可能包括了不在相应的资源分配图中循环链中的进程。
7. 鸵鸟算法中提到了填充进程表表项或者其他系统表的可能。能否给出一种能够使系统管理员从这种状况下恢复系统的方法？
8. 解释系统是如何从前面问题的死锁中恢复的，使用a) 抢占；b) 回滚；c) 终止进程。
9. 假设在图6-6中，对某个i，有$C_{ij} + R_{ij} > E_j$，这意味着什么？
10. 请说明表6-8中的模型与6.5.2节描述的安全状态和不安全状态有什么主要的差异。差异带来的后果是什么？
11. 图6-8所示的资源轨迹模式是否可用来说明三个进程和三个资源的死锁问题？如果可以，它是怎样说明的？如果不可以，请解释为什么。
12. 理论上，资源轨迹图可以用于避免死锁。通过合理的调度，操作系统可避免进入不安全区域。请列举一个在实际运用这种方法时会带来的问题。
13. 一个系统是否可能处于既非死锁也不安全的状态？如果可以，举出例子；如果不可以，请证明所有状态只能处于死锁或安全两种状态之一。
14. 考虑一个使用银行家算法避免死锁的系统。某个时刻一个进程P请求资源R，但即使R当前可用这个请求也被拒绝了。如果系统分配R给P，是否意味着系统将会死锁？
15. 银行家算法的一个主要限制就是需要知道所有进程的最大资源需求的信息。有没有可能设计一个不需要这些信息而避免死锁的算法？解释你的方法。
16. 仔细考察图6-11b，如果D再多请求1个单位，会导致安全状态还是不安全状态？如果换成C提出同样请求，情形会怎样？
17. 某一系统有两个进程和三个相同的资源。每个进程最多需要两个资源。这种情况下有没有可能发生死锁？为什么？
18. 再考虑上一个问题，但现在有p个进程，每个进程最多需要m个资源，并且有r个资源可用。什么样的条件可以保证死锁不会发生？
19. 假设图6-12中的进程A请求最后一台磁带机，这一操作会引起死锁吗？
20. 一个计算机有6台磁带机，由n个进程竞争使用，每个进程可能需要两台磁带机，那么n是多少时系统才没有死锁的危险？
21. 银行家算法在一个有m个资源类型和n个进程的系统中运行。当m和n都很大时，为检查状态是否安全而进行的操作次数正比于$m^a n^b$。a

和b的值是多少？

22. 一个系统有4个进程和5个可分配资源，当前分配和最大需求如下：

	已分配资源	最大需求量	可用资源
进程A	10211	11213	00×11
进程B	20110	22210	
进程C	11010	21310	
进程D	11110	11221	

若保持该状态是安全状态，x的最小值是多少？

23. 一个消除环路等待的方法是用规则说明一个进程在任意时刻只能得到一个资源。举例说明在很多情况下这个限制是不可接受的。

24. 两个进程A和B，每个进程都需要数据库中的3个记录1、2、3。如果A和B都以1、2、3的次序请求，将不会发生死锁。但是如果B以3、2、1的次序请求，那么死锁就有可能会发生。对于这3种资源，每个进程共有3!（即6）种次序请求，这些组合中有多大的可能可以保证不会发生死锁？

25. 一个使用信箱的分布式系统有两条IPC原语：send和receive。receive原语用于指定从哪个进程接收消息，并且如果指定的进程没有可用消息，即使有从其他进程发来的消息，该进程也等待。不存在共享资源，但是进程由于其他原因需要经常通信。死锁会产生吗？请讨论这一问题。

26. 在一个电子资金转账系统中，有很多相同进程按如下方式工作：每一进程读取一行输入，该输入给出一定数目的款项、贷方账户、借方账户。然后该进程锁定两个账户，传送这笔钱，完成后释放锁。由于很多进程并行运行，所以存在这样的危险：锁定x会无法锁定y，因为y已被一个正在等待x的进程锁定。设计一个方案来避免死锁。在没有完成事务处理前不要释放该账户记录。（换句话说，在锁定一个账户时，如果发现另一个账户不能被锁定就立即释放这个已锁定的账户。）

27. 一种预防死锁的方法是去除占有和等待条件。在本书中，假设在请求一个新的资源以前，进程必须释放所有它已经占有的资源（假设这是可能的）。然而，这样做会引入这样的危险性：使竞争的进程得到了新的资源但却丢失了原有的资源。请给出这一方法的改进。

28. 计算机系学生想到了下面这个消除死锁的方法。当某一进程请求一个资源时，规定一个时间限。如果进程由于得不到需要的资源而阻塞，定时器开始运行。当超过时间限时，进程会被释放掉，并且允许该进程重新运行。如果你是教授，你会给这样的学生多少分？为什么？

29. 解释死锁、活锁和饥饿的区别。

30. Cinderella 和Prince要离婚，为分割财产，他们商定了以下算法。每天早晨每个人发函给对方律师要求财产中的一项。由于邮递信件需要一天的时间，他们商定如果发现在同一天两人请求了同一项财产，第二天他们会发信取消这一要求。他们的财产包括狗Woofer、Woofer 的狗屋、金丝雀Tweeter和Tweeter的鸟笼。由于这些动物喜爱它们的房屋，所以又商定任何将动物和它们房屋分开的方案都无效，且整个分配从头开始。Cinderella和Prince都非常想要Woofer。于是他们分别去度假，并且每人都编写程序用一台个人计算机处理这一谈判工作。当他们度假回来时，发现计算机仍在谈判，为什么？产生死锁了吗？产生饥饿了吗？请讨论。

31. 一个主修人类学、辅修计算机科学的学生参加了一个研究课题，调查是否可以教会非洲狒狒理解死锁。他找到一处很深的峡谷，在上边固定了一根横跨峡谷的绳索，这样狒狒就可以攀住绳索越过峡谷。同一时刻，只要朝着相同的方向就可以有几只狒狒通过。但如果向东和向西的狒狒同时攀在绳索上那么会产生死锁（狒狒会被卡在中间），因为它们无法在绳索上从另一只的背上翻过去。如果一只狒狒想越过峡谷，它必须看当前是否有别的狒狒正在逆向通行。利用信号量编写一个避免死锁的程序来解决该问题。不考虑连续东行的狒狒会使得西行的狒狒无限制地等待的情况。

32. 重复上一个习题，但此次要避免饥饿。当一只想向东去的狒狒来到绳索跟前，但发现有别的狒狒正在向西越过峡谷时，它会一直等到绳索可用为止。但在至少有一只狒狒向东越过峡谷之前，不允许再有狒狒开始从东向西越过峡谷。

33. 编写银行家算法的模拟程序。该程序应该能够循环检查每一个提出请求的银行客户，并且能判断这一请求是否安全。请把有关请求和相应决定的列表输出到一个文件中。

34. 写一个程序实现每种类型多个资源的死锁检测算法。你的程序应该从一个文件中读取下面的

输入：进程数、资源类型数、每种存在类型的资源数（向量E）、当前分配矩阵C（第一行，接着第二行，以此类推）、需求矩阵R（第一行，接着第二行，以此类推）。你的程序输出应表明在此系统中是否有死锁。如果系统中有死锁，程序应该打印出所有死锁的进程id号。

35. 写一个程序使用资源分配图检测系统中是否存在死锁。你的程序应该从一个文件中读取下面的输入：进程数和资源数。对每个进程，你应该读取4个数：进程当前持有的资源数、它持有的资源的ID、它当前请求的资源数、它请求的资源ID。程序的输出应表明在此系统中是否有死锁。如果系统中有死锁，程序应该打印出所有死锁的进程id号。

第7章 多媒体操作系统

数字电影、视频剪辑和音乐正在日益成为用计算机表示信息和进行消遣娱乐的常用方式。音频和视频文件可以保存在磁盘上，并且在需要的时候进行回放。音频和视频文件的特征与传统的文本文件存在很大的差异，而目前的文件系统却是为文本文件设计的。因此，需要设计新型的文件系统来处理音频和视频文件。不仅如此，保存与回放音频和视频同样给调度程序以及操作系统的其他部分提出了新的要求。本章中，我们将研究这些问题以及它们与设计用来处理多媒体信息的操作系统之间的关系。

数字电影通常归于多媒体名下，多媒体的字面含义是一种以上的媒体。在这样的定义下，本书就是一部多媒体作品，毕竟它包含了两种媒体：文本和图像（插图）。然而，大多数人使用“多媒体”这一术语时所指的是包含两种或更多种连续媒体的文档，连续媒体也就是必须能够在某一时间间隔内回放的媒体。本书中，我们将在这样的意义下使用多媒体这一术语。

另一个有些模糊的术语是“视频”。在技术意义上，视频只是一部电影的图像部分（相对的是声音部分）。实际上，摄像机和电视机通常有两个连接器，一个标为“视频”，一个标为“音频”，因为这两个信号是分开的。然而，“数字视频”这一术语通常指的是完整的作品，既包含图像也包含声音。后面我们将使用“电影”这一术语指完整的作品。注意，在这种意义下一部电影不一定是好莱坞以超过一架波音747的代价制作的长达两小时的大片，一段通过因特网从CNN主页下载的30秒长的新闻剪辑在这一定义下也是一部电影。当我们提到非常短的电影时，也将其称为“视频剪辑”。

7.1 多媒体简介

在讨论多媒体技术之前，了解其目前和将来的用法可能有助于对问题的理解。在一台计算机上，多媒体通常意味着从数字通用光盘（Digital Versatile Disk，DVD）播放一段预先录好的电影。DVD是一种光盘，它使用与CD-ROM相同的120 mm聚碳酸脂（塑料）盘片，但是记录密度更高，容量在5GB到17GB之间（取决于记录格式）。

有两个候选者正在竞争成为DVD的后继者。一个是Blu-ray（蓝光）格式，其单层格式容量有25GB（双层格式有50GB）。另一个是HD DVD格式，其单层格式容量有15GB（双层格式30GB）。每一种格式都由一个不同的计算机和电影公司的财团支持。显然，电子与娱乐产业非常怀念在20世纪70年代到80年代Betamax与VHS的“格式大战”，因此他们决定重现这场战争。毋庸置疑，当消费者等着看哪家最终胜出时，这两个系统的普及也会因为这次“格式大战”而推迟好几年。

另一种多媒体的使用是从Internet上下载视频短片。许多网页都有可以点击下载短片的栏目。像YouTube一样的Web站点有成千上万可供欣赏的视频短片。随着有线电视与ADSL（非对称数字用户环线）的普及，更快的发布技术会占据市场，Internet中的视频短片将会像火箭升天一样猛增。

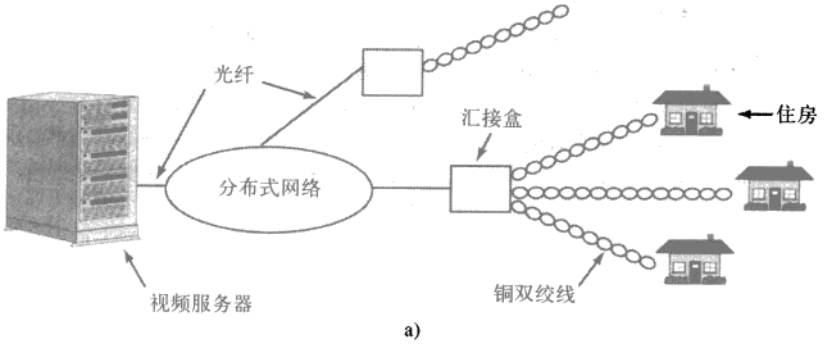
另一个必须支持多媒体的领域是视频本身的制作。目前有许多多媒体编辑系统，这些系统需要在不仅支持传统作业而且还支持多媒体的操作系统上运行，以获得最好的性能。

多媒体还在另一个领域中发挥着越来越重要的作用，这就是计算机游戏。计算机游戏经常要运行短暂的视频剪辑来描述某种活动。这些视频剪辑通常很短，但是数量非常多，并且还要根据用户采取的某些行动动态地选择正确的视频剪辑。计算机游戏正变得越来越复杂。当然，游戏本身也会生成大量的动画，不过，处理程序生成的视频与播放一段电影是不相同的。

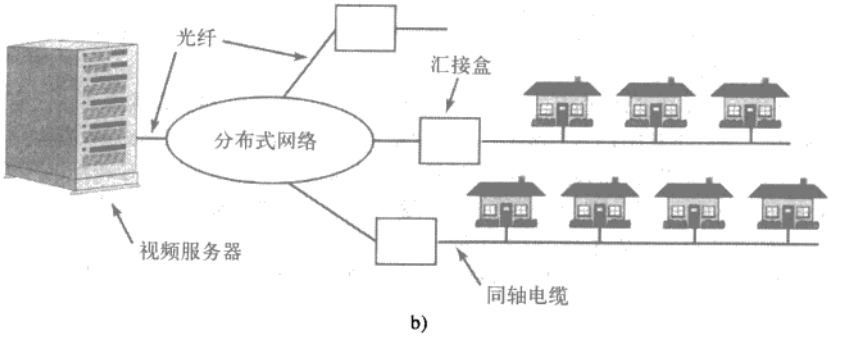
最后，多媒体世界的圣杯是视频点播（video on demand），这意味着消费者能够在家中使用电视遥控器（或鼠标）选择电影，并且立刻将其在电视机（或计算机显示器）上显示出来。视频点播要求有特殊的基础设施才能使用。图7-1所示为两种可能的视频点播基础设施，每种都包含三个基本的组件：一个或多个视频服务器、一个分布式网络以及一个在每个房间中用来对信号进行解码的机顶盒。视频服务器（video server）是一台功能强大的计算机，在其文件系统中存放着许多电影，并且可以按照点播请求

回放这些电影。大型机有时用来作为视频服务器，因为大型机连接1000个大容量的磁盘是一件轻而易举的事情；而在个人计算机上连接1000个容量不太大的磁盘也是一件很困难的事情。在本章后续各节中，有许多材料是关于视频服务器及其操作系统的。

用户和视频服务器之间的分布式网络必须能够高速实时地传输数据。这种网络的设计十分有趣也十分复杂，但是这超出了本书的范围。我们不想更多地讨论分布式网络，只想说明分布式网络总是使用光纤从视频服务器连接到客户居住的居民点的汇接盒。ADSL系统是由电话公司经营的，在ADSL系统中，现有的双绞电话线提供了最后一公里的数据传输。有线电视是由有线电视公司经营的，在有线电视系统中，现有的有线电视电缆用于信号的本地分送。ADSL的优点是为每个用户提供了专用数据通道，因此带宽有保证，但是由于现有电话线的局限其带宽比较低（只有几Mb/s）。有线电视使用高带宽的同轴电缆，带宽可以达到几Gb/s，但是许多用户必须共享相同的电缆，从而导致竞争，对于每个用户来说带宽没有保证。不过，为了与有线电视竞争，电话公司正在为住户铺设光缆，这样，光缆上的ADSL将比电视电缆有更大的带宽。



a)



b)

图7-1 视频点播使用不同的本地分布技术：a) ADSL；b) 有线电视

系统的最后一部分是机顶盒（set-top box），这是ADSL或电视电缆终结的地方。机顶盒实际上就是普通的计算机，只不过其中包含特殊的芯片用于视频解码和解压缩。机顶盒最少要包含CPU、RAM、ROM、与ADSL或电视电缆的接口，以及用于跟电视机连接的端子。

机顶盒的替代品是使用客户现有的PC机并且在显示器上显示电影。十分有趣的是，大多数客户可能都已经拥有一台计算机，为什么还要考虑机顶盒呢，这是因为视频点播的运营商认为人们更愿意在起居室中看电影，而起居室中通常放有电视机，很少有计算机。从技术角度看，使用个人计算机代替机顶盒更有道理，因为计算机的功能更加强大，拥有大容量的磁盘，并且拥有更高分辨率的显示器。不管采用的是机顶盒还是个人计算机，在解码并显示电影的用户端，我们通常都要区分视频服务器和客户机进程。然而，以系统设计的观点，客户机进程是在机顶盒上运行还是在PC机上运行并没有太大的关系。对于桌面视频编辑系统而言，所有的进程都运行在相同的计算机上，但是我们将继续使用服务器和客户这

样的术语，以便搞清楚哪个进程正在做什么事情。

回到多媒体本身，要想成功地处理多媒体则必须很好地理解它所具有的两个关键特征：

1) 多媒体使用极高的数据率。

2) 多媒体要求实时回放。

高数据率来自视觉与听觉信息的本性。眼睛和耳朵每秒可以处理巨大数量的信息，必须以这样的速率为眼睛和耳朵提供信息才能产生可以接受的观察体验。图7-2列举了几种数字多媒体源和某些常见硬件设备的数据率。在本章后面我们将讨论这些编码格式。需要注意的是，多媒体需要的数据率越高，则越需要进行压缩，并且需要的存储量也就越大。例如，一部未压缩的2小时长的HDTV电影将填满一个570GB的文件。存放1000部这种电影的视频服务器需要570TB的磁盘空间，按照目前的标准这可是难以想象的数量。还需要注意的是，没有数据压缩，目前的硬件不可能跟上这样的数据率。我们将在本章后面讨论视频压缩。

数据源	Mbps	GB/hr
电话（PCM）	0.064	0.03
MP3音乐	0.14	0.06
音频CD	1.4	0.62
MPEG-2电影（640×480）	4	1.76
数字便携式摄像机（720×480）	25	11
未压缩电视（640×480）	221	97
未压缩HDTV（1280×720）	648	288

设备	Mbps
快速以太网	100
EIDE磁盘	133
ATM OC-3网络	156
IEEE 1394b(FireWire)	800
千兆位以太网	1000
SATA磁盘	3000
SCSI Ultra-640磁盘	5120

图7-2 某些多媒体和高性能I/O设备的数据率（1 Mbps = 10^6 位/秒，1 GB =2^{30} 字节）

多媒体对系统提出的第二个要求是需要实时数据传输。数字电影的视频部分每秒包含某一数目的帧。北美、南美和日本采用的NTSC制式每秒包含30帧（对纯粹主义者而言是29.97帧），世界上其他大部分地区采用的PAL和SECAM制式每秒包含25帧（对纯粹主义者而言是25.00帧）。帧必须分别以33.3ms或40ms的精确时间间隔传输，否则电影看起来将会有起伏。

NTSC代表美国国家电视标准委员会（National Television Standards Committee），但是当彩色电视发明时将彩色引入该标准的拙劣方法产生了业界的一个笑话，戏称NTSC代表决不复现相同的颜色（Never Twice the Same Color）。PAL代表相位交错排列（Phase Alternating Line），它是技术上最好的制式。SECAM代表顺序与存储彩色（SEquentiel Couleur Avec Memoire），该制式被法国采用，意在保护法国的电视制造商免受国外竞争。SECAM还被东欧国家所采用。

耳朵比眼睛更加敏感，传输时间中即使存在几毫秒的变动也会被察觉到。传输率的变动称为*颤动*（jitter），必须严格限制颤动以获得良好的性能。注意，颤动不同于延迟。如果图7-1中的分布式网络均匀地将所有的位准确地延迟5s，电影将开始得稍稍晚一些，但是看起来却不错。但从另一方面来说，如果分布式网络在100~200ms之间随机地延迟各帧，那么不论是谁主演的电影，看起来都像是查理·卓别林的老片。

让人可以接受地回放多媒体所要求的实时性通常通过*服务质量*（quality of service）参数来描述，这些参数包括可用平均带宽、可用峰值带宽、最小和最大延迟（两者一起限制了颤动）以及位丢失概率。例如，网络运营商提供的服务可以保证4 Mbps的平均带宽、105~110ms时间间隔之内99%的传输延迟以及10^{-10}的位丢失概率，那么这样的服务质量对于MPEG-2电影来说是非常好的。网络运营商还可以提供价格更为低廉等级也比较低的服务，例如平均带宽为1 Mbps（如ADSL），在这种情况下，服务质量就不得不打些折扣，可能是降低分辨率、降低帧率或者是放弃颜色信息以黑白方式播放电影。

提供服务质量保证的最常见的方法是预先为每一个新到的客户预留容量，预留的资源包括CPU、内存缓冲区、磁盘传输容量以及网络带宽。如果一位新的客户到来并且想观看一部电影，但是视频服务器或网络计算出不具有为另一位客户提供服务的容量，那么它将拒绝新的客户，以避免降低向当前客户提供的服务质量。因此，多媒体服务器需要有资源预留方案和*进入控制算法*（admission control algorithm），以判定什么时候能够处理更多的任务。

7.2 多媒体文件

在大多数系统中，普通的文本文件由字节的线性序列组成，没有操作系统了解或关心的任何结构。对于多媒体而言，情况就复杂多了。首先，视频与音频是完全不同的。它们由不同的设备捕获（视频为CCD芯片，音频为麦克风），具有不同的内部结构（视频每秒有25~30帧，音频每秒有44 100个样本），并且它们通过不同的设备来回放（视频为显示器，音频为扩音器）。

此外，大多数好莱坞电影现在针对的是全世界的观众，而这些观众大多不讲英语。这一情况有两种处理方法。对于某些国家，需要产生一个额外的声音轨迹，用当地语言进行配音，但是不包含音效。在日本，所有的电视都具有两个声道，电视观众看外国影片时可以听原声语言也可以听日语，遥控器上有一个按钮可以用来进行语言选择。在其他国家中，使用的是原始的声音轨迹，配以当地语言的字幕。

除此之外，许多在电视中播放的电影现在也提供英文字幕，使讲英语但是听力较弱的人可以观看。结果，数字电影实际上可能由多个文件组成：一个视频文件、多个音频文件以及多个包含各种语言字幕的文本文件。DVD能够存放至多32种语言的字幕文件。简单的一组多媒体文件如图7-3所示，我们将在本章后面解释快进和快倒的含义。

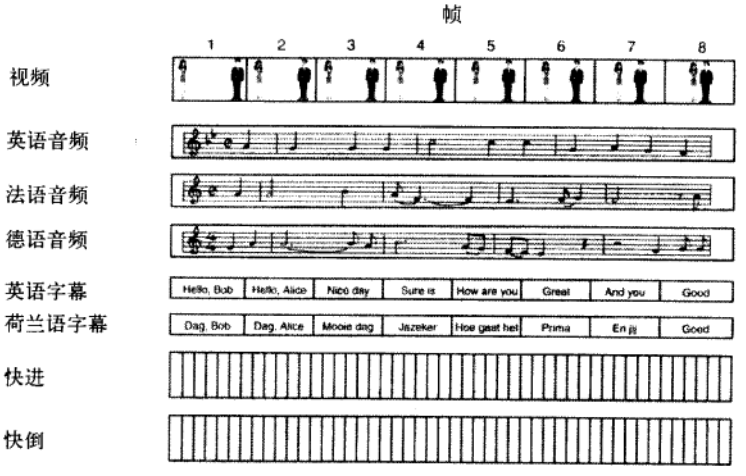


图7-3 电影可能由若干文件组成

因此，文件系统需要跟踪每个文件的多个“子文件”。一种可能的方案是像传统的文件一样管理每个子文件（例如，使用i节点来跟踪文件的块），并且要有一个新的数据结构列出每个多媒体文件的全部子文件。另一方法是创造一种二维的i节点，使每一列列出每个子文件的全部块。一般而言，其组织必须能够使观众观看电影时可以动态地选择使用哪个音频及字幕轨迹。

在各种情况下，还必须有保持子文件同步的某种方法，这样才能保证当选中的音频轨迹回放时与视频保持同步。如果音频与视频存在即使是轻微的不同步，观众可能会在演员的嘴唇运动之前或之后才听到他说的话，这很容易察觉到，相当让人扫兴。

为了更好地理解多媒体文件是如何组织的，有必要在某种细节程度上了解数字音频与数字视频是如何工作的，下面我们将介绍这些主题。

7.2.1 视频编码

人类的眼睛具有这样的特性：当一幅图像闪现在视网膜上时，在它衰退之前将保持几毫秒的时间。如果一个图像序列以每秒50或更多张图像闪现，眼睛就不会注意到它看到的是不连续的图像。所有基于视频或影片胶片的运动图像系统都利用了这一原理产生活动的画面。

要想理解视频系统，最好从简单且过时的黑白电视开始。为了将二维图像表示为作为时间函数的一维电压，摄像机用一个电子束对图像进行横向扫描并缓慢地向下移动，记录下电子束经过处光的强度。在扫描的终点处，电子束折回，称为一帧（frame）。这一作为时间函数的光的强度以广播方式传播出去，

接收机则重复扫描过程以重构图像。摄像机与接收机采用的扫描模式如图7-4所示。(CCD摄像机的成像方式是积分而非扫描，但是某些摄像机及所有的CRT显示器采用的都是扫描方式。)

精确的扫描参数随国家的不同而有所不同。NTSC有525条扫描线，水平与垂直方向的纵横比为4:3，每秒为30（实际为29.97）帧。欧洲的PAL和SECAM制式有625条扫描线，纵横比也是4:3，每秒为25帧。在两种制式中，顶端和底端的几条线是不显示的(为的是在原始的圆形CRT上显示一个近似矩形的图像)。525条NTSC扫描线中只显示483条，625条PAL/SECAM扫描线中只显示576条。

虽然每秒25帧足以捕获平滑的运动，但是在这样的帧率下，有许多人特别是老年人会感觉到图像闪烁（因为新的图像尚未出现以前旧的图像就已经在视网膜上消失)。增加帧率就会对稀缺的带宽提出更多的要求，因此要采取不同的方法。这一方法不是按从上到下的顺序显示扫描线，而是首先显示所有的奇数扫描线，接着再显示所有的偶数扫描线。此处的半帧称为一个*场*（field)。实验表明，尽管人们在每秒25帧时感觉到闪烁，但是在每秒50场时却感觉不到。这一技术被称为*隔行扫描*（interlacing)。非隔行扫描的电视或视频被称为*逐行扫描*（progressive)。

彩色视频采用与单色（黑白）视频相同的扫描模式，只不过使用了三个同时运动的电子束而不是一个运动电子束来显示图像，对于红、绿和蓝（RGB）这三个加性原色中的每一颜色使用一个电子束。这一技术能够工作是因为任何颜色都可以由红、绿和蓝以适当的强度线性叠加而构造出来。然而，为了在一个信道上进行传输，三个彩色信号必须组合成一个*复合*（composite）信号。

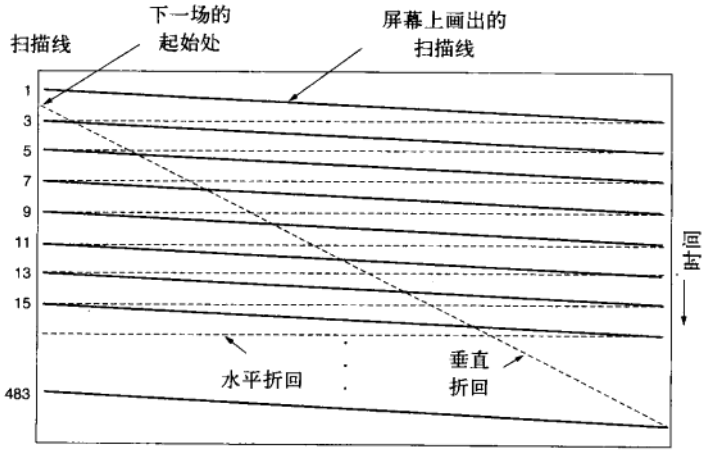


图7-4 NTSC视频和电视使用的扫描模式

为了使黑白接收机可以显示传输的彩色电视节目，NTSC、PAL和SECAM三种制式均将RGB信号线性组合为一个*亮度*（luminance）信号和两个*色度*（chrominance）信号，但是不同的制式使用不同的系数从RGB信号构造这些信号。说来也奇怪，人的眼睛对亮度信号比对色度信号敏感得多，故色度信号倒不必非要精确地进行传输。因此，亮度信号应该用与旧的黑白信号相同的频率进行广播，从而使其可以被黑白电视机接收。两个色度信号则可以以更高的频率用较窄的波段进行广播。某些电视机有标着亮度、色调和饱和度（或者是亮度、色彩和颜色）字样的旋钮或调节装置，可以分别控制这三个信号。理解亮度和色度对于理解视频压缩的工作原理是十分必要的。

到目前为止我们介绍的都是模拟视频，现在让我们转向数字视频。数字视频最简单的表示方法是帧的序列，每一帧由呈矩形栅格的图像要素即*像素*（pixel）组成。对于彩色视频，每一像素RGB三色中的每种颜色用8个二进制位来表示，这样可以表示$2^{24} \approx 1600$万种不同的颜色，已经足够了。人的眼睛没有能力区分这么多颜色，更不用说更多的颜色了。

要产生平滑的运动效果，数字视频像模拟视频一样必须每秒至少显示25帧。然而，由于高质量的计算机显示器通常用存放在视频RAM中的图像每秒钟扫描屏幕75次或更多次，隔行扫描是不必要的，因此所有计算机显示器都采用逐行扫描。仅仅连续刷新（也就是重绘）相同的帧三次就足以消除闪烁。

换言之，运动的平滑性是由每秒不同的图像数决定的，而闪烁则是由每秒刷新屏幕的次数决定的。这两个参数是不同的。一幅静止的图像以每秒20帧的频率显示不会表现出断断续续的运动，但是却会出现闪烁，因为当一帧画面在视网膜上消退时下一帧还没有出现。一部电影每秒有20个不同的帧，在80 Hz的刷新率下每一帧将连续绘制4次，这样不会出现闪烁，但是运动将是断断续续的。

当我们考虑在网络上传输数字视频所需要的带宽时，这两个参数的重要性就十分清楚了。目前许多计算机显示器都采用4:3的纵横比，所以可以使用便宜的并且大量生产的显像管，这样的显像管本来是为电视市场的消费者设计的。显示器常用的配置有640×480（VGA）、800×600（SVGA）、1024×768（XGA）以及1600×1200（UXGA）。每像素24位的UXGA显示以及25帧/秒，需要1.2Gbps的带宽，即使VGA显示也需要184Mbps。将这些速率加倍以避免闪烁是没有吸引力的，更好的解决方案是每秒传输25帧，同时让计算机保存每一帧并将其绘制两次。广播方式的电视没有使用这一策略，因为电视机没有存储器，并且模拟信号如果不首先转换成数字形式无论如何也无法存放在RAM中，而模数转换则需要额外的硬件。因此，隔行扫描对于广播方式的电视而言是需要的，但是对数字视频则不需要。

7.2.2 音频编码

音频（声音）波是一维的声（压）波。当声波进入人耳的时候，鼓膜将振动，导致内耳的小骨随之振动，将神经脉冲送入大脑，这些脉冲被收听者感知为声音。类似地，当声波冲击麦克风的时候，麦克风将产生电信号，将声音的振幅表示为时间的函数。

人耳可以听到的声音的频率范围从20 Hz到20 000Hz，而某些动物，特别是狗，能够听到更高频率的声音。耳朵是以对数规律听声音的，所以两个振幅为A和B的声音的比率习惯以dB（分贝）为单位来表示，公式为

$$\mathrm{dB} = 20 \log_{10}(A / B)$$

如果我们定义1 kHz正弦波可听度的下限（压力大约为0.0003 dyne/ cm^2）为0 dB，那么日常谈话大约为50 dB，而使人感到痛苦的阈值大约为120 dB，动态范围为一百万量级。为避免混淆，上面公式中的A和B是振幅。如果我们使用的是功率水平，则上面公式中对数前面的系数应该为10，而不是20，因为功率与振幅的平方成正比。

音频波可以通过*模数转换器*（Analog Digital Converter，ADC）转换成数字形式。ADC以电压作为输入，并且生成二进制数作为输出。图7-5a中为一个正弦波的例子。为了数字化地表示该信号，我们可以每隔ΔT秒对其进行采样，如图7-5b中的条棒高度所示。如果一个声波不是纯粹的正弦波，而是正弦波的线性叠加，其中存在的最高频率成分为f，那么以$2f$的频率进行采样就足够了。1924年贝尔实验室的一位物理学家Harry Nyquist从数学上证明了这一结果，这就是著名的*Nyquist抽样定理*。更多地进行采样是没有价值的，因为如此采样可以检测到的更高的频率并不存在。

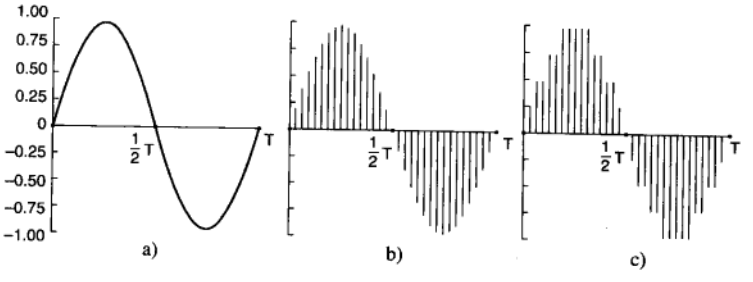


图7-5 a) 正弦波；b) 对正弦波进行采样；c) 对样本进行4位量化

数字样本是不准确的。图7-5c中的样本只允许9个值，从−1.00到1.00，步长为0.25，因此，需要4个二进制位来表示它们。8位样本可以有256个不同的值，16位样本可以有65 536个不同的值。由于每一样本的位数有限而引入的误差称为*量化噪声*（quantization noise）。如果量化噪声太大，耳朵就会感觉到。

对声音进行采样的两个著名的例子是电话和音频CD。电话系统使用的是*脉冲编码调制*（pulse code modulation），脉冲编码调制每秒以7位（北美和日本）或8位（欧洲）对声音采样8000次，故这一系统的

数据率为56 000 bps或64 000 bps。由于每秒只有8000个样本，所以4 kHz以上的频率就丢失了。

音频CD是以每秒44 100个样本的采样率进行数字化的，足以捕获最高达到22 050 Hz的频率，这对于人而言是很好的，但是对于狗而言却是很差的。每一样本在其振幅范围内以16位进行线性量化。注意，16位样本只有65 536个不同的值，而人耳以最小可听度为步长进行测量时的动态范围大约为一百万。所以每个样本只有16位引入了某些量化噪声（尽管没有覆盖全部动态范围，但是人们并不认为CD的质量受到损害）。以每秒44 100个样本、每个样本16位计算，音频CD需要的带宽单声道为705.6 Kbps，立体声为1.411 Mbps（参见图7-2）。音频压缩也许要以描述人类听觉如何工作的心理声学模型为基础。使用MPEG第3层（MP3）系统进行10倍的压缩是可能的。采用这一格式的便携式音乐播放器近年来已经十分普遍。

数字化的声音可以十分容易地在计算机上用软件进行处理。有许许多多的个人计算机程序可以让用户从多个信号源记录、显示、编辑、混合和存储声波。事实上，所有专业的声音记录与编辑系统如今都是数字化的。模拟方式基本上过时了。

7.3 视频压缩

现在我们已经十分清楚，以非压缩格式处理多媒体信息是完全不可能的——它的数据量太大了，惟一的希望是有可能进行大比例的数据压缩。幸运的是，在过去几十年，大量的研究群体已经发明了许多压缩技术和算法，使多媒体传输成为可能。在下面几节中，我们将研究一些多媒体数据（特别是图像）的压缩方法，更多的细节请参见（Fluckiger, 1995；Steinmetz和Nahrstedt, 1995）。

所有的压缩系统都需要两个算法：一个用于在源端对数据进行压缩，另一个用于在目的端对数据进行解压缩。在文献中，这两个算法分别被称为*编码*（encoding）算法和*解码*（decoding）算法，我们在本书中也使用这样的术语。

这些算法具有某些不对称性，这一不对称性对于理解数据压缩是十分重要的。首先，对于许多应用而言，一个多媒体文档（比如说一部电影）只需要编码一次（当该文档存储在多媒体服务器上时），但是需要解码数千次（当该文档被客户观看时）。这一不对称性意味着，假若解码算法速度快并且不需要昂贵的硬件，那么编码算法速度慢并且需要昂贵的硬件也是可以接受的。从另一方面来说，对于诸如视频会议这样的实时多媒体而言，编码速度慢是不可接受的，在这样的场合，编码必须即时完成。

第二个不对称性是编码/解码过程不必是100%可逆的。也就是说，当对一个文件进行压缩并进行传输，然后对其进行解压缩时，用户可以期望取回原始的文件，准确到最后一位。对于多媒体，这样的要求是不存在的。视频信号经过编码和解码之后与原始信号只存在轻微的差异通常就是可以接受的。当解码输出不与原始输入严格相等时，系统被称为是*有损的*（lossy）。所有用于多媒体的压缩系统都是有损的，因为这样可以获得更好的压缩效果。

7.3.1 JPEG标准

用于压缩连续色调静止图像（例如照片）的JPEG（Joint Photographic Experts Group，*联合摄影专家组*）标准是由摄影专家在ITU、ISO和IEC等其他标准组织的支持下开发出来的。JPEG标准对于多媒体而言是十分重要的，因为用于压缩运动图像的标准MPEG不过是分别对每一帧进行JPEG编码，再加上某些帧间压缩和运动补偿等额外的特征。JPEG定义在10918号国际标准中。它具有4种模式和许多选项，但是我们在这里只关心用于24位RGB视频的方法，并且省略了许多细节。

用JPEG对一幅图像进行编码的第一步是块预制。为明确起见，我们假设JPEG输入是一幅640 × 480的RGB图像，每个像素24位，如图7-6a所示。由于使用亮度和色度可以获得更好的压缩效果，所以从RGB值中计算出一个亮度信号和两个色度信号，对于NTSC制式，分别将其记作Y、I和Q，对于PAL制式，分别将其记作Y、U和V，两种制式的计算公式是不同的。下面我们将使用NTSC的符号，但是压缩算法是相同的。

对Y、I和Q构造不同的矩阵，每个矩阵其元素的取值范围在0到255之间。接下来，在I和Q矩阵中对由4个元素组成的方块进行平均，将矩阵缩小至320 × 240。这一缩小是有损的，但是眼睛几乎注意不到，因为眼睛对亮度比对色度更加敏感，然而这样做的结果是将数据压缩了2倍。现在将所有三个矩阵的每

个元素减去128，从而将0置于取值范围的中间。最后将每个矩阵划分成8×8的块，Y矩阵有4800块，其他两个矩阵每个有1200块，如图7-6b所示。

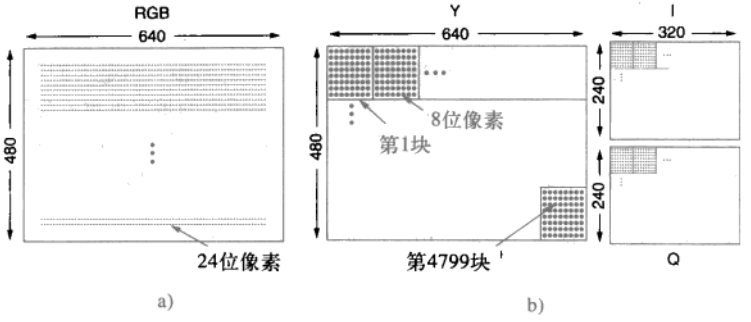


图7-6 a) RGB输入数据；b) 块预制之后

JPEG的第2步是分别对7200块中的每一块应用DCT（离散余弦变换）。每一DCT的输出是一个8×8的DCT系数矩阵。DCT矩阵的（0, 0）元素是块的平均值，其他元素表明每一空间频率存在多大的谱功率。对于熟悉傅立叶变换的读者而言，DCT则是一种二维的空间傅立叶变换。在理论上，DCT是无损的，但是在实践中由于使用浮点数和超越函数总要引入某些舍入误差，从而导致轻微的信息损失。通常这些元素随着到（0,0）元素距离的增加而迅速衰减，如图7-7b所示。

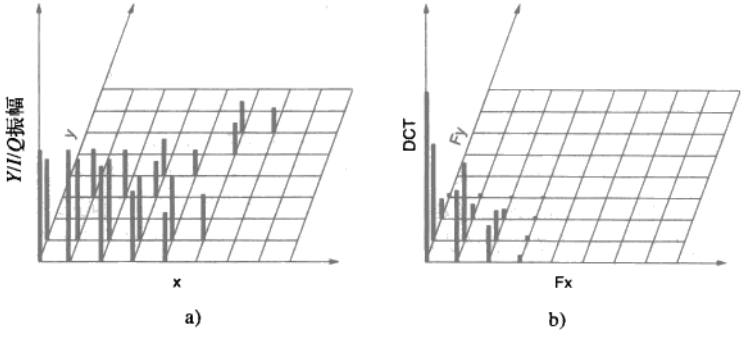


图7-7 a) Y矩阵的一块；b) DCT系数

DCT完成之后，JPEG进入到第3步，称为量化（quantization），在量化过程中不重要的DCT系数将被去除。这一（有损）变换是通过将8×8 DCT矩阵中的每个元素除以取自一张表中的权值而实现的。如果所有权值都是1，那么该变换将不做任何事情。然而，如果权值随着离原点的距离而急剧增加，那么较高的空间频率将迅速衰落。

图7-8给出了这一步的一个例子，在图7-8中我们可以看到初始DCT矩阵、量化表和通过将每个DCT元素除以相应量化表元素所获得的结果。量化表中的值不是JPEG标准的一部分。每一应用必须提供自己的量化表，这样就给应用以控制自身压缩损失权衡的能力。

DCT系数

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

量化系数

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

量化表

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

图7-8 量化DCT系数的计算

第4步通过将每一块的（0,0）值（左上角元素）以它与前一块中相应元素相差的量替换而减小。由于这些元素是各自所在块的平均，它们应该变化得比较缓慢，所以采用差值可以将这些元素中的大部分缩减为较小的值。对于其他元素不计算差值。(0,0）值称为DC分量，其他值称为AC分量。

第5步是将64个元素线性化并且对线性化得到的列表进行行程长度编码。从左到右然后从上到下地对块进行扫描不能将零集中在一起，所以采用了Z字形的扫描模式，如图7-9所示。在本例中，Z字形模式最终在矩阵的尾部产生了38个连续的0，这一串0可以缩减为一个计数表明有38个0。

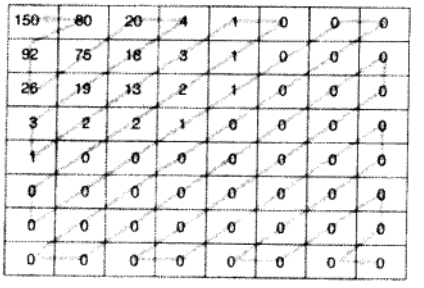


图7-9 量化值传送的顺序

现在我们得到一个代表图像的数字列表（在变换空间中），第6步将采用Huffman编码对列表中的数字进行编码以用于存储或传输。

JPEG看来似乎十分复杂，这是因为它确实很复杂。尽管如此，由于它通常可以获得20:1或更好的压缩效果，所以获得广泛的应用。解码一幅JPEG图像需要反过来运行上述算法。JPEG大体上是对称的：解码一幅图像花费的时间与编码基本相同。

7.3.2 MPEG标准

最后，我们讨论问题的核心：MPEG（Motion Picture Experts Group，运动图像专家组）标准。这是用于压缩视频的主要算法，并于1993年成为国际标准。MPEG-1（第11172号国际标准）设计用于视频录像机质量的输出（对NTSC制式为352×240），它使用的位率为1.2 Mbps。MPEG-2（第13818号国际标准）设计用于将广播质量的视频压缩至4 Mbps到6 Mbps，这样就可以适应NTSC或PAL制式的广播频道。

MPEG的两个版本均利用了在电影中存在的两类冗余：空间冗余和时间冗余。空间冗余可以通过简单地用JPEG分别对每一帧进行编码而得到利用。互相连续的帧常常几乎是完全相同的，这就是时间冗余，利用这一事实可以达到额外的压缩效果。数字便携式摄像机使用的数字视频（Digital Video，DV）系统只使用类JPEG的方案，这是因为只单独对每一帧进行编码可以达到更快的速度，从而使编码可以实时完成。这一论断的因果关系可以从图7-2看出：尽管数字便携式摄像机与未压缩电视相比具有较低的数据率，但是却远不及MPEG-2。（为了使比较公平，请注意DV便携式摄像机以8位对亮度、以2位对每一色度进行采样，使用类JPEG编码仍然存在5倍的压缩率。）

对于摄像机和背景绝对静止，而有一两个演员在四周缓慢移动的场景而言，帧与帧之间几乎所有的像素都是相同的。此时，仅仅将每一帧减去前一帧并且在差值图像上运行JPEG就相当不错。然而，对于摇动或缩放摄像机镜头的场景而言，这一技术将变得非常糟糕。此时需要某种方法对这一运动进行补偿，这正是MPEG要做的事情；实际上，这就是MPEG和JPEG之间的主要差别。

MPEG-2输出由三种不同的帧组成，观看程序必须对它们进行处理，这三种帧为：

1) I帧：自包含的JPEG编码静止图像。

2) P帧：与上一帧逐块的差。

3) B帧：与上一帧和下一帧的差。

I帧只是用JPEG编码的静止图像，沿着每一轴还使用了全分辨率的亮度和半分辨率的色度。在输出流中使I帧周期性地出现是十分必要的，其原因有三。首先，MPEG可以用于电视广播，而观众收看是随意的。如果所有的帧都依赖于其前驱直到第一帧，那么错过了第一帧的人就再也无法对随后的帧进行解码，这样使观众在电影开始之后就不能再进行收看。第二，如果任何一帧在接收时出现错误，那么进一步的解码就不可能再进行。第三，没有I帧，在进行快进或倒带时，解码器将不得不计算经过的每一帧，只有这样才能知道快进或倒带停止时帧的全部值。有了I帧，就可以向前或向后跳过若干帧直到找到一个I帧并从那里开始观看。由于上述原因，MPEG每秒将I帧插入到输出中一次或两次。

与此相对照，P帧是对帧间差进行编码。P帧基于宏块（macroblock）的思想，宏块覆盖亮度空间中16×16个像素和色度空间中8×8个像素。通过在前一帧中搜索宏块或者与其只存在轻微差异的宏块实现

对一个宏块的编码。

P帧的用途在图7-10所示的例子中可以看出。在图7-10中我们看到三个连续的帧具有相同的背景，但是在一个人所在的位置上存在差异。对于摄像机固定在三脚架上，而演员在摄像机面前活动的情形中，这种场景是常见的。包含背景的宏块是严格匹配的，但是包含人的宏块在位置上存在某一未知数量的偏移，编码时必须追踪到前一帧中相应的宏块。

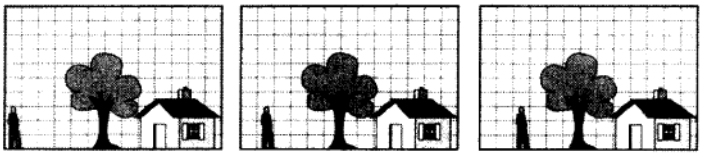


图7-10 三个连续的视频帧

MPEG标准没有规定如何搜索、搜索多远以及如何计算一个匹配的好坏，这些都留给每一具体的实现。例如，一种实现可能在前一帧中的当前位置以及所有在x方向偏移$\pm\Delta x$、在y方向偏移$\pm\Delta y$的位置搜索一个宏块。对于每个位置，可以计算出亮度矩阵中匹配的数目。具有最高得分的位置将成为获胜者，只要其得分高于某一预设的阈值。否则，宏块就被称为失配。当然，更复杂的算法也是可能的。

如果一个宏块被找到，则通过以其值与前一帧中的值求差对其进行编码（针对亮度和两个色度），然后，对这些差值矩阵进行JPEG编码。输出流中宏块的值是运动矢量（宏块在每一方向从其前一位置移动多远的距离），随后是以JPEG进行编码的与前一帧的差值。如果宏块在前一帧中查找不到，则当前值以JPEG进行编码，如同在I帧中一样。

B帧与P帧相类似，不同的是它允许参考宏块既可以在前一帧中，也可以在后续的帧中，既可以在I帧中，也可以在P帧中。这一额外的自由可以改进运动补偿，并且在物体从前面（或后面）经过其他物体时非常有用。例如，在一场垒球比赛中，当三垒手将球掷向一垒时，可能存在某些帧其中垒球遮蔽了在背景中移动的二垒手的头部。在下一帧中，二垒手的头部可能在垒球的左面有一部分可见，头部的下一个近似可以从垒球已经通过了头部的后续的帧中导出。B帧允许一个帧基于未来的帧。

要进行B帧编码，编码器需要在内存中同时保存三个解码的帧：过去的一帧、当前的一帧和未来的一帧。为了简化解码，各帧必须以依赖的顺序而不是以显示的顺序出现在MPEG流中。因而，当一段视频通过网络被观看时，即使有完美的定时，在用户的机器上也需要进行缓冲，对帧进行记录以便正常地显示。由于这一依赖顺序和显示顺序间的差异，试图反向播放一部电影而没有相当可观的缓冲和复杂的算法是无法工作的。

有许多动作以及快速剪切（比如战争电影）的电影需要许多I型帧。而那种在导演对准了摄像机之后便出去喝咖啡，只留下演员背台词（比如爱情故事）的电影，就可以使用长段的P帧与B帧，而这两种帧结构与I帧相比使用很少的存储空间。从磁盘效率的角度来看，一个运营多媒体服务的公司应该尝试得到尽可能多的女性消费群体。

7.4 音频压缩

就像我们刚刚看到的，CD品质的音频需要一个1.411 Mbps带宽的传送。很清楚，在Internet的实际传送中，需要有效的压缩。正是因为这一点，已经发展起来许多不同的音频压缩算法。或许最流行的算法是拥有三个层（变体）的MPEG音频，其中，MP3(MPEG 音频层3)是功能最强大也是最出名的。在Internet上随处可见大量MP3格式的音乐，它们并非都合法，因此引发了许多来自艺术家与版权拥有者的案件。MP3属于MPEG视频压缩标准里的音频部分。

音频压缩可以用两种方法完成。在*波形编码*技术中，信号通过傅立叶变换（Fourier transform）变换成频率分量。图7-11给出一个时间与它最初的15个傅立叶振幅的实例函数。然后每一个分量的振幅用最简短的方法编码。目标是在另一端用尽可能少的二进制位精确地重建其波形。

另一种方法是*感知编码*，这种技术是在人类听觉系统中寻找某种细纹，用来对信号编码，这种信号听起来与人的正常收听相同，尽管在示波器上看起来却大相径庭。感知编码是基于*心理声学*的——人们如何感知声音的科学。MP3正是基于感知编码。

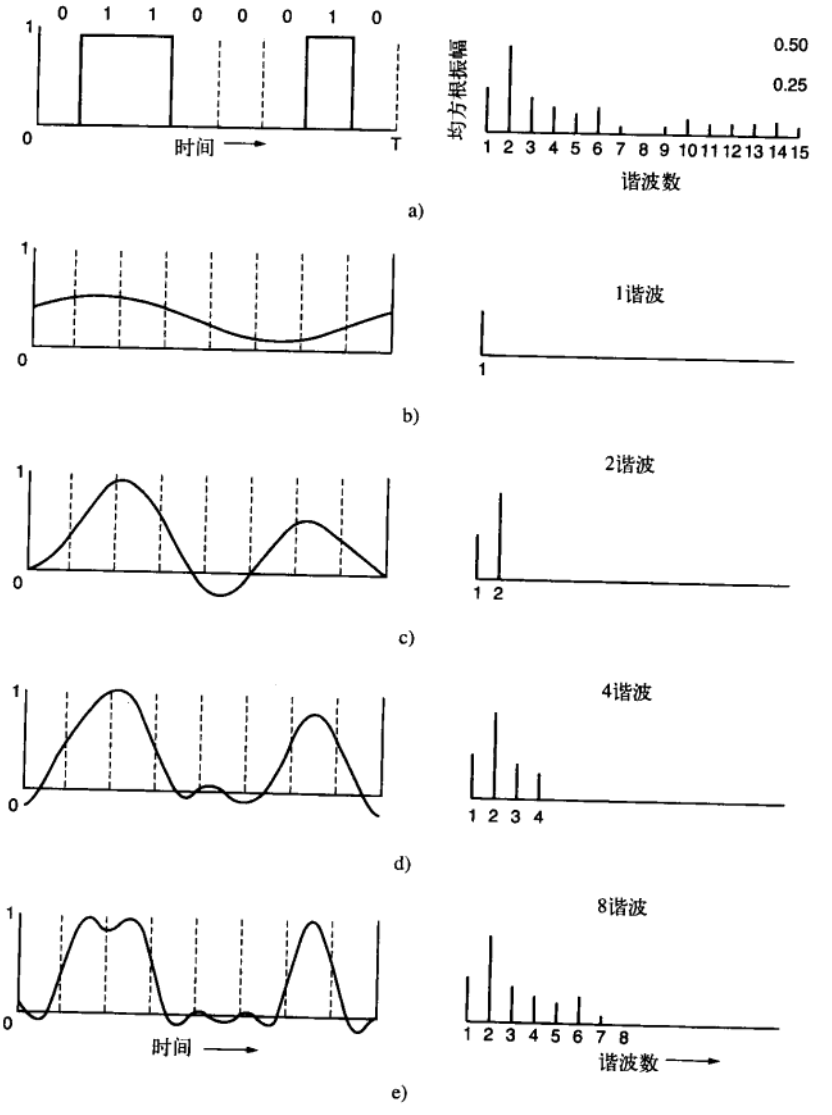


图7-11 a) 二进制信号和它的均方根傅立叶振幅；b)～e)成功逼近原始信号

感知编码的关键特性在于一些声音可以*掩盖*住其他声音。想象在一个温暖夏天举办的现场直播的长笛音乐会，突然间，附近的一群工人打开他们的风镐开始挖掘街道。这时没有人可以再听到笛子的声音，因为它已经被风镐的声音给掩盖了。从传送角度看，只编码风镐的频段就足够了，因为听众无论如何都听不到笛子的声音。这种技术就叫做*频段屏蔽*——在一个频段里响亮的声音掩盖住另一频段中较柔和声音的能力，这种较柔和声音只有在没有响亮声音时才可以听到。事实上，即使风镐停止工作，在一个短时间内笛子的声音也很难再被听到，因为耳朵在开始工作时已经调低了增益，并且需要在一段时间之后才会再次调高增益。这种效果称为*暂时屏蔽*。

为了使得这些影响能尽量被量化，设想实验1。某个人在一间安静的屋子里戴着与计算机声卡相连的耳机。计算机产生最低100Hz但功率逐渐增加的纯正弦波。这个人被命令在他/她听到一个音调的时候敲击一个键。计算机在记录当前功率级之后，以200Hz、300hz以及其他所有不超过人类听力极限的频率重复之前的实验。在把许多实验者的实验平均值计算出来后，一张关于“需要多大功率才能使人们听到

一个音调”的对数-对数图就展现出来了，如图7-12a所示。图中曲线的给出直接结果是：人们并没有必要对那些功率在可听阈值之下的频率编码。例如，在图7-12a中，如果100Hz的功率是20dB，那么在输出上就可以忽略掉，而且不会感觉到声音质量的损失，因为在100Hz处20dB是低于可听水平的。

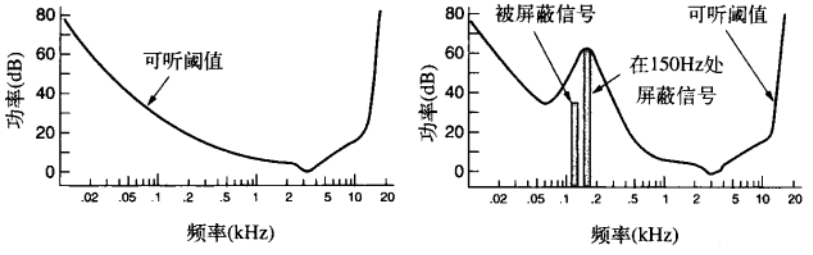


图7-12 a) 作为频率函数的可听阈值；b) 屏蔽效应

现在考虑实验2。计算机再次运行实验1，但是这次却一有个大约150赫兹的等幅正弦波叠加在实验频率上。我们发现，在150Hz频率附近的可听阈值上升了，如图7-12b所示。

这一新实验的结果表明：通过跟踪那些被附近频段能量更强的信号所屏蔽的信号，可以省略越来越多的编码信号中的频率，以此来节约二进制位。在图7-12中，125Hz信号的输出是可以完全忽略掉的，并且没有人能够听出其中的不同。甚至当某个频段中的一个强大信号停止后，出于对暂时屏蔽这一知识的了解，也会让我们在耳朵恢复期的时间段内省略掉那些被屏蔽的频率。MP3编码的实质就是对声音做傅立叶变换从而得到每个频率的能量，之后只传递那些不被屏蔽掉的频率，并且用尽可能少的二进制位数来编码这些频率。

有了这些信息作为背景，现在来考察有关编码是如何完成的。通过抽取32kHz、44.1kHz或者48 kHz的波形，完成声音压缩。第一个和最后一个都是四舍五入的整数。44.1kHz是用于Audio CD的，因为这个值能很好地捕获人耳可听到的所有音频信息。可以在以下四个配置中任选一个，用一或两个通道完成抽样：

1) 单声道（一个输入流）。

2) 双声道（例如，一个英语的和一个日语的音轨）。

3) 分立立体声（每个通道分开压缩）。

4) 联合立体声（完全利用通道间的冗余）。

首先，选择输出的比特率。MP3可以将摇滚CD的立体声降低到96kbps，并且在质量上几乎没有任何失真，甚至连摇滚迷都听不出差别。而对于一场钢琴音乐会，至少需要128kbps。造成这样不同的原因是因为摇滚的信噪比要比一场钢琴音乐会要高得多（至少从工程角度上看）。也可以选择稍低一点的输出比率，接受质量上的少许失真。

然后将这些样本处理成1152（大概26ms）的一些组，每组首先通过32个数字滤波器，获得32个频率波段。同时，将输入放进一个心理声学的模型中，测定被屏蔽的频率。接下来，进一步转换32频率波段中的每一个，以提供一个更精确的频谱解决方案。

接着，将现有的二进制位分配到各个波段中，大部分二进制位分配给拥有多数频谱能量的未屏蔽波段，小部分二进制位分配给拥有较少频谱能量的未屏蔽波段，已屏蔽的波段不分配二进制位。最后，用霍夫曼编码来对这些二进制位进行编码，它可以将经常出现的数字赋予较短的代码，而对不常出现的数字赋予较长的代码。

实际的工作过程更复杂。为了减少噪音，消除混淆，以及利用通道间冗余，需要各种各样的技术，不过这些内容超出了本书的范围。

7.5 多媒体进程调度

支持多媒体的操作系统与传统的操作系统在三个主要的方面有所区别：进程调度、文件系统和磁盘调度。本节中我们开始讨论进程调度，在后面的各节中接着讨论其他主题。

7.5.1 调度同质进程

最简单的一种视频服务器可以支持显示固定数目的电影，所有电影使用相同的帧率、视频分辨率、数据率以及其他参数。在这样的情况下，可以采用下述简单但是有效的调度算法。对每一部电影，存在一个进程（或线程），其工作是每次从磁盘中读取电影的一帧然后将该帧传送给用户。由于所有的进程同等重要，每一帧有相同的工作量要做，并且当它们完成当前帧的处理时将阻塞，所以采用轮转调度可以很好地做这样的工作。将调度算法标准化的惟一的额外要求是定时机制，以确保每一进程以恰当的频率运行。

实现适当定时的一种方式是有一个主控时钟，该时钟每秒滴答适当的次数，例如针对NTSC制式，每秒滴答30次。在时钟的每一滴答，所有的进程以相同的次序相继运行。当一个进程完成其工作时，它将发出suspend系统调用释放CPU直到主控时钟再次滴答。当主控时钟再次滴答时，所有的进程再次以相同的次序运行。只要进程数足够少，所有的工作都可以在一帧的时间内完成，采用轮转调度就足够了。

7.5.2 一般实时调度

不幸的是，这一模型在实践中几乎没有什么用处。随着观众的来来去去，用户的数目不断发生变化，由于视频压缩的本性（I帧比P帧或B帧大得多），帧的大小剧烈变化，并且不同的电影可能有不同的分辨率。因此，不同的进程可能必须以不同的频率运行，具有不同的工作量，并且具有不同的最终时限（在此之前所有工作必须完成）。

这些考虑导致一个不同的模型：多个进程竞争CPU，每个进程有自己的工作量和最终时限。在下面的模型中，我们将假设系统知道每个进程必须以什么样的频率运行、有多少工作要做以及下一个最终时限是什么。（磁盘调度也是一个问题，但我们将在后面考虑。）多个相互竞争的进程，其中若干进程或全部进程具有必须满足的最终时限的调度称为*实时调度*（real-time scheduling）。

作为实时多媒体调度程序工作环境的一个例子，我们考虑三个进程A、B和C，如图7-13所示。进程A每30ms运行一次（近似NTSC制式速度），每一帧需要10ms的CPU时间。在不存在竞争的情况下，进程A将在突发A1、A2、A3等中运行，每一突发在前一突发的30ms之后开始。每个CPU突发处理一帧并且具有一个最终时限：它必须在下一个突发开始之前完成。

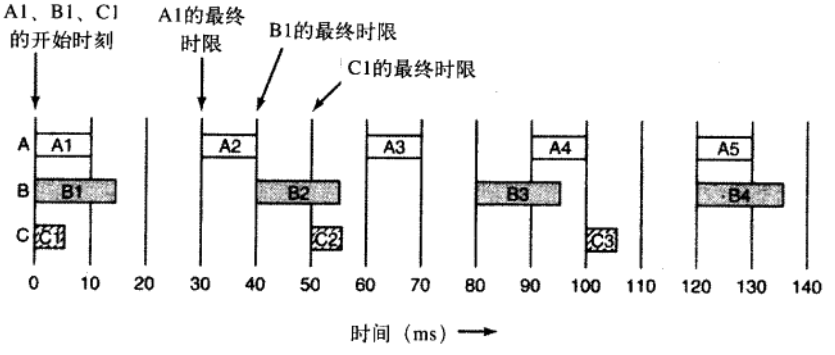


图7-13 三个周期性的进程，每个进程播放一部电影；每一电影的帧率以及每帧的处理需求有所不同

图7-13中还有另外两个进程：B和C。进程B每秒运行25次（例如PAL制式），进程C每秒运行20次（例如一个慢下来的NTSC或PAL流，意在使一个低带宽的用户连接到视频服务器）。每一帧的计算时间如图7-13中所示，进程B为15ms，进程C为5ms，没有使它们都具有相同的时间只是为了使调度问题更加一般化。

现在调度问题是如何调度A、B和C以确保它们满足各自的最终时限。在寻找调度算法之前，我们必须看一看这一组进程究竟是不是可调度的。回想2.4.4节，如果进程i具有P_i ms的周期并且需要$C_i$ms的CPU时间，那么系统是可调度的当且仅当

$$\sum_{i=1}^{m}\frac{C_i}{P_i}\leqslant 1$$

其中m是进程数，在本例中，$m = 3$。注意，C_i / P_i只是CPU被进程i使用的部分。就图7-13所示的例子而言，进程A用掉CPU的10/30，进程B用掉CPU的15/40，进程C用掉CPU的5/50。将这些分数加在一起为CPU的0.808，所以该系统是可调度的。

到目前为止我们假设每个影片流有一个进程，实际上，每个影片流可能有两个（或更多个）进程，例如，一个用于音频，一个用于视频。它们可能以不同的速率运行并且每一脉冲可能消耗不同数量的CPU时间。然而，将音频进程加入到系统中并没有改变一般模型，因为我们的全部假设是存在m个进程，每个进程以一个固定的频率运行，对每一CPU突发有固定的工作量要求。

在某些实时系统中，进程是可抢占的，在其他的系统中，进程是不可抢占的。在多媒体系统中，进程通常是可抢占的，这意味着允许有危险错过其最终时限的进程在正在运行的进程完成工作以前将其中断，然后当它完成工作之后，被中断的前一个进程再继续运行。这一行为只不过是多道程序设计，正如我们在前面已经看过的。我们要研究的是可抢占的实时调度算法，因为在多媒体系统中没有拒绝它们的理由并且它们比不可抢占的调度算法具有更好的性能。惟一要关心的是如果传输缓冲区在很少的几个突发中被填充，那么在最终时限到来之前该缓冲区应该是完全满的，这样它就可以在一次操作中传递给用户，否则就会引起颤动。

实时算法可以是静态的也可以是动态的。静态算法预先分配给每个进程一个固定的优先级，然后使用这些优先级做基于优先级的抢占调度。动态算法没有固定的优先级。下面我们将研究每种类型的一个例子。

7.5.3 速率单调调度

适用于可抢占的周期性进程的经典静态实时调度算法是*速率单调调度*（Rate Monotonic Scheduling，RMS）（Liu和Layland, 1973）。它可以用于满足下列条件的进程：

1) 每个周期性进程必须在其周期内完成。
2) 没有进程依赖于任何其他进程。
3) 每一进程在一次突发中需要相同的CPU时间量。
4) 任何非周期性进程都没有最终时限。
5) 进程抢占即刻发生而没有系统开销。

前四个条件是合理的。当然，最后一个不是，但是该条件使系统建模更加容易。RMS分配给每个进程一个固定的优先级，优先级等于进程触发事件发生的频率。例如，必须每30ms运行一次（每秒33次）的进程获得的优先级为33，必须每40ms运行一次（每秒25次）的进程获得的优先级为25，必须每50ms运行一次（每秒20次）的进程获得的优先级为20。所以，优先级与进程的速率（每秒运行进程的次数）成线性关系，这正是为什么将其称为速率单调的原因。在运行时，调度程序总是运行优先级最高的就绪进程，如果需要则抢占正在运行的进程。Liu和Layland证明了在静态调度算法种类中RMS是最优的。

图7-14演示了在图7-13所示的例子中速率单调调度是如何工作的。进程A、B和C分别具有静态优先级33、25和20，这意味着只要A需要运行，它就可以运行，抢占任何当前正在使用CPU的其他进程。进程B可以抢占C，但不能抢占A。进程C必须等待直到CPU空闲才能运行。

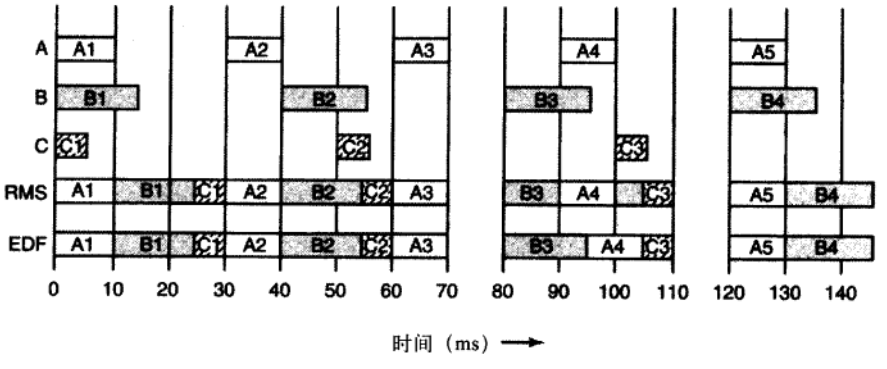


图7-14 RMS和EDF实时调度的一个例子

在图7-14中，最初所有三个进程都就绪要运行，优先级最高的进程A被选中，并准许它运行直到它在10ms时完成，如图7-14中的RMS一行所示。在进程A完成之后，进程B和C以先后次序运行。合起来，这些进程花费了30ms的时间运行，所以当C完成的时候，正是该A再次运行的时候。这一轮换持续进行直到t = 70时系统变为空闲。

在t = 80时，进程B就绪并开始运行。然而，在t = 90时，优先级更高的进程A变为就绪，所以它抢占B并运行，直到在t = 100时完成。在这一时刻，系统可以在结束进程B或者开始进程C之间进行选择，所以它选择优先级最高的进程B。

7.5.4 最早最终时限优先调度

另一个流行的实时调度算法是*最早最终时限优先*（Earliest Deadline First，EDF）算法。EDF是一个动态算法，它不像速率单调算法那样要求进程是周期性的。它也不像RMS那样要求每个CPU突发有相同的运行时间。只要一个进程需要CPU时间，它就宣布它的到来和最终时限。调度程序维持一个可运行进程的列表，该列表按最终时限排序。EDF算法运行列表中的第一个进程，也就是具有最近最终时限的进程。当一个新的进程就绪时，系统进行检查以了解其最终时限是否发生在当前运行的进程结束之前。如果是这样，新的进程就抢占当前正在运行的进程。

图7-14给出了EDF的一个例子。最初所有三个进程都是就绪的，它们按其最终时限的次序运行。进程A必须在t = 30之前结束，B必须在t = 40之前结束，C必须在t = 50之前结束，所以A具有最早的最终时限并因此而先运行。直到t = 90，选择都与RMS相同。在t = 90时，A再次就绪，并且其最终时限为t = 120，与B的最终时限相同。调度程序可以合理地选择其中任何一个运行，但是由于抢占B具有某些非零的代价与之相联系，所以最好是让B继续运行，而不去承担切换的代价。

为了消除RMS和EDF总是给出相同结果的想法，现在让我们看一看另外一个例子，如图7-15所示。在这个例子中，进程A、B和C的周期与前面的例子相同，但是现在A每次突发需要15ms的CPU时间，而不是只有10ms。可调度性测试计算CPU的利用率为0.500 + 0.375 + 0.100 = 0.975。CPU只留下了2.5%，但是在理论上CPU并没有被超额预定，找到一个合理的调度应该是可能的。

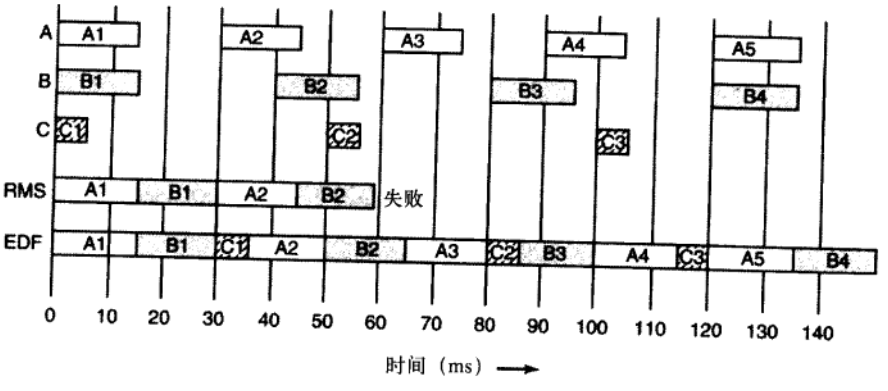


图7-15 以RMS和EDF进行实时调度的另一个例子

对于RMS，三个进程的优先级仍为33、25和20，因为优先级只与周期有关系，而与运行时间没有关系。这一次，进程B直到t = 30才结束，在这一时刻，进程A再次就绪要运行。等到A结束时，t = 45，此时B再次就绪，由于它的优先级高于C，所以B运行而C则错过了其最终时限。RMS失败。

现在看一看EDF如何处理这种情况。当t = 30时，在A2和C1之间存在竞争。因为C1的最终时限是50，而A2的最终时限是60，所以C被调度。这就不同于RMS，在RMS中A由于较高的优先级而成为赢家。

当t = 90时，A第四次就绪。A的最终时限与当前进程相同（同为120），所以调度程序面临抢占与否的选择。如前所述，如果不是必要最好不要抢占，所以B3被允许完成。

在图7-15所示的例子中，直到t = 150，CPU都是100%被占用的。然而，因为CPU只有97.5%被利用，所以最终将会出现间隙。由于所有开始和结束时间都是5ms的倍数，所以间隙将是5ms。为了获得要求

的2.5%的空闲时间，5ms的间隙必须每200ms出现一次，这就是间隙为什么没有在图7-15中出现的原因。

一个有趣的问题是RMS为什么会失败。根本上，使用静态优先级只有在CPU的利用率不太高的时候才能工作。Liu和Layland（1973）证明了对于任何周期性进程系统，如果

$$\sum_{i=1}^{m}\frac{C_i}{P_i}\leqslant m(2^{1/m}-1)$$

那么就可以保证RMS工作。对于m = 3、4、5、10、20和100，最大允许利用率为0.780、0.757、0.743、0.718、0.705和0.696。随着$m\to\infty$，最大利用率逼近ln 2。换句话说，Liu和Layland证明了，对于三个进程，如果CPU利用率等于或小于0.780，那么RMS总是可以工作的。在第一个例子中，CPU利用率为0.808而RMS工作正常，但那只不过是幸运罢了。对于不同的周期和运行时间，利用率为0.808很可能会失败。在第二个例子中，CPU利用率如此之高（0.975），根本不存在RMS能够工作的希望。

与此相对照，EDF对于任意一组可调度的进程总是可以工作的，它可以达到100%的CPU利用率，付出的代价是更为复杂的算法。因而，在一个实际的视频服务器中，如果CPU利用率低于RMS限度，可以使用RMS，否则，应该选择EDF。

7.6 多媒体文件系统范型

至此我们已经讨论了多媒体系统中的进程调度，下面继续我们的研究，看一看多媒体文件系统。这样的文件系统使用了与传统文件系统不同的范型。我们首先回顾传统的文件I/O，然后将注意力转向多媒体文件服务器是如何组织的。进程要访问一个文件时，首先要发出open系统调用。如果该调用成功，则调用者被给予某种令牌以便在未来的调用中使用，该令牌在UNIX中被称为文件描述符，在Windows中被称为句柄。这时，进程可以发出read系统调用，提供令牌、缓冲区地址和字节计数作为参数。操作系统则在缓冲区中返回请求的数据。以后还可以发出另外的read调用，直到进程结束，在进程结束时它将调用close以关闭文件并返回其资源。

由于实时行为的需要，这一模型对于多媒体并不能很好地工作。在显示来自远程视频服务器的多媒体文件时，该模型的工作尤为拙劣。第一个问题是用户必须以相当精确的时间间隔进行read调用。第二个问题是视频服务器必须能够没有延迟地提供数据块，当请求没有计划地到来并且预先没有保留资源时，做到这一点是十分困难的。

为解决这些问题，多媒体文件服务器使用了一个完全不同的范型：像录像机（Video Cassette Recorder，VCR）一样工作。为了读取一个多媒体文件，用户进程发出start系统调用，指定要读的文件和各种其他参数，例如，要使用哪些音频和字幕轨迹。接着，视频服务器开始以必要的速率送出帧。然后用户进程以帧进来的速率对它们进行处理。如果用户对所看的电影感到厌烦，那么发出stop系统调用可以将数据流终止。具有这种数据流模型的文件服务器通常被称为*推送型服务器*（push server），因为它将数据推送给用户；与此相对照的是传统的*拉取型服务器*（pull server），用户不得不通过重复地调用read一块接一块地取得数据，每调用一次可以拉取出一块数据。这两个模型之间的区别如图7-16所示。

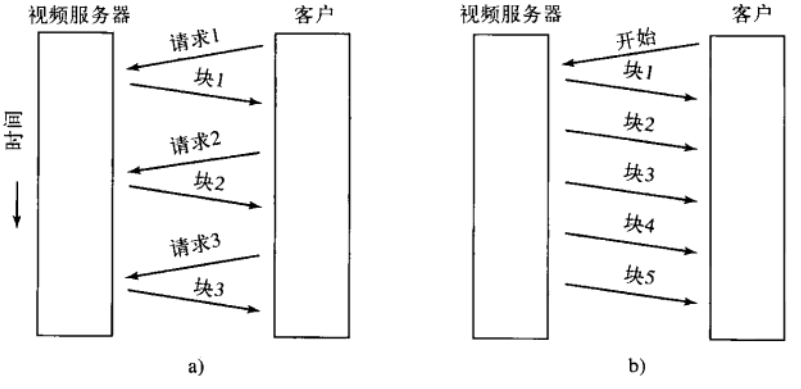


图7-16 a) 拉取型服务器；b) 推送型服务器

7.6.1 VCR控制功能

大多数视频服务器也实现了标准的VCR控制功能，包括暂停、快进和倒带。暂停是相当简单的。用户发送一个消息给视频服务器，告诉它停止。视频服务器此时要做的全部事情是记住下一次要送出的是哪一帧。当用户要求服务器恢复播放时，服务器只要从它停止的地方继续就可以了。

然而，这里存在着一个复杂因素。为了获得可接受的性能，服务器应该为每个流出的数据流保留诸如磁盘带宽和内存缓冲区等资源。当电影暂停时继续占用这些资源将造成浪费，特别是如果用户打算到厨房中找到一块冷冻的比萨饼（或许是特大号的）、用微波炉烹调并且美餐一顿的时候。当然，在暂停的时候可以很容易地将资源释放，但是这引入了风险：当用户试图恢复播放的时候，有可能无法重新获得这些资源。

真正的倒带实际上非常简单，没有任何复杂性。服务器要做的全部事情是注意到下一次要送出的帧是第0帧。还有比这更容易的吗？然而，快进和快倒（也就是在倒带的同时播放）就难处理多了。如果没有压缩，那么以10倍的速度前进的一种方法是每10帧只显示一帧，以20倍的速度前进则要求每20帧显示一帧。实际上，在不存在压缩的情况下，以任意速度前进和后退都是十分容易的。要以正常速度的k倍运行，只要每k帧显示一帧就可以了。要以正常速度的k倍后退，只要沿另一个方向做相同的事情就可以了。这一方法在推送型服务器和拉取型服务器上工作得同样好。

压缩则使快进和快倒复杂起来。对于便携式摄像机的DV磁带，由于其每一帧都是独立于其他帧而压缩的，所以只要能够快速地找到所需要的帧，使用这一策略还是有可能的。由于视其内容不同每一帧的压缩量也有所不同，所以每一帧具有不同的大小，因而在文件中向前跳过k帧并不能通过数字计算来完成。此外，音频压缩是独立于视频压缩的，所以对于在高速模式中显示的每一视频帧，还必须找到正确的音频帧（除非在高于正常速度播放时将声音关闭）。因此，对一个DV文件进行快进操作需要有一个索引，该索引可以使帧的查找快速地实现，但是至少在理论上这样做是可行的。

对于MPEG，由于使用I帧、P帧和B帧，这一方案即使在理论上也是不能工作的。向前跳过k帧（就算假设能这样做）可能落在一个P帧上，而这个P帧则基于刚刚跳过的一个I帧。没有基本帧，只有从基本帧发生的增量变化（这正是P帧所包含的）是无用的。MPEG要求按顺序播放文件。

攻克这一难题的另一个方法是实际尝试以10倍的速度顺序地播放文件。然而，这样做就要求以10倍的速度将数据拉出磁盘。此时，服务器可能试图将帧解压缩（这是正常情况下服务器不需要做的事情），判定需要哪一帧，然后每隔10帧重新压缩成一个I帧。然而，这样做给服务器增加了沉重的负担。这一方法还要求服务器了解压缩格式，正常情况下服务器不必了解这些东西。

作为替代，可以通过网络实际发送所有的数据给用户，并在用户端选出正确的帧，这样做就要求网络以10倍的速度运行，这或许是可行的，但是在这么高的速度下正常操作肯定不是一件容易的事情。

总而言之，不存在容易的方法。惟一可行的策略要求预先规划。可以做的事情是建立一个特殊的文件，包含每隔10帧中的一帧，并且将该文件以通常的MPEG算法进行压缩。这个文件正是在图7-3中注为“快进”的那个文件。要切换到快进模式，服务器必须判定在快进文件中用户当前所在的位置。例如，如果当前帧是48 210并且快进文件以10倍的速度运行，那么服务器在快进文件中必须定位到4821帧并且在此处以正常速度开始播放。当然，这一帧可能是P帧或B帧，但是客户端的解码进程可以简单地跳过若干帧直到看见一个I帧。利用特别准备的快倒文件，可以用类似的方法实现快倒。

当用户切换回到正常速度时，必须使用相反的技巧。如果在快进文件中当前帧是5734，服务器只要切换回到常规文件并且从57 340帧处继续播放。同样，如果这一帧不是一个I帧，客户端的解码进程必须忽略所有的帧直到看见一个I帧。

尽管有了这两个额外的文件可以做这些工作，这一方案还是有某些缺点。首先，需要某些额外的磁盘空间来存放额外的文件。其次，快进和倒带只能以对应于特别文件的速度进行。第三，在常规文件、快进文件和快倒文件之间来回切换需要额外的复杂算法。

7.6.2 近似视频点播

有k个用户取得相同的电影和这些用户取得k部不同的电影在本质上给服务器施加了相同的工作量。然而，通过对模型做一个小小的修改，就可能获得巨大的性能改进。视频点播面临的问题是用户可能在

任意时刻开始观看一部电影，所以，如果有100个用户全部在晚8点左右开始观看某个新电影，很可能不会有两个用户在完全相同的时刻开始，所以他们无法共享一个数据流。使优化成为可能的修改是，通知所有用户电影只在整点和随后每隔（例如）5分钟开始。因此，如果一个用户想在8:02看一部电影，那么他必须等到8:05。

这样做的收益是，不管存在多少客户，对于一部2小时的电影，只需要24个数据流。如图7-17所示，第一个数据流开始于8:00。在8:05，当第一个数据流处于第9000帧时，第二个数据流开始。在8:10，当第一个数据流处于第18 000帧并且第二个数据流处于第9000帧时，第三个数据流开始，以此类推直到第24个数据流开始于9:55。在10:00，第一个数据流终止并且再一次从第0帧开始。这一方案称为近似视频点播（near video on demand），因为视频并不是完全随着点播而开始，而是在点播之后不久开始。

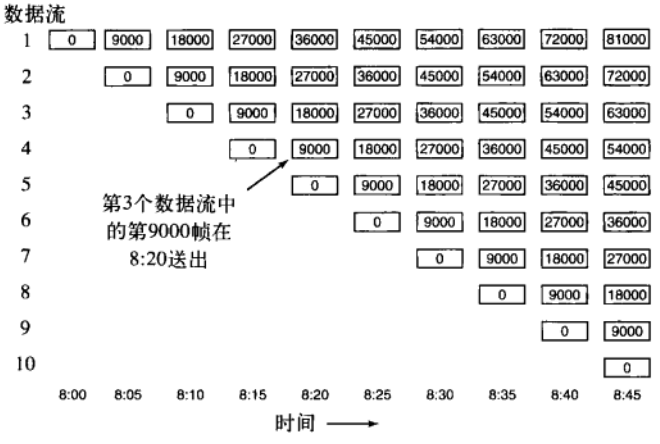


图7-17 近似视频点播以规则的间隔开始一个新的数据流，在本例中时间间隔为5分钟（9000帧）

这里的关键参数是多长时间开始一个数据流。如果每2分钟开始一个数据流，那么对于一部2小时的电影来说就需要60个数据流，但是开始观看的最大等待时间是2分钟。运营商必须判定人们愿意等待多长时间，因为人们愿意等待的时间越长，系统效率就越高，并且同时能够被观看的电影就越多。一个替代的策略是同时提供不用等待的选择权，在这种情况下，新的数据流可以立刻开始，但是需要对系统做更多的修改以支持即时启动。

在某种意义上，视频点播如同使用出租车：一招手它就来。近似视频点播如同使用公共汽车：它有着固定的时刻表，乘客必须等待下一辆。但是大众交通只有在存在大众的时候才有意义。在曼哈顿中心区，每5分钟一辆的公共汽车加起来至少还可以拉上一些乘客；而在怀俄明州乡间公路上旅行的公共汽车，可能在所有的时间几乎都是空空的。类似地，播放史蒂文·斯皮尔伯格的最新大片可能吸引足够多的客户，从而保证每5分钟开始一个新的数据流；但是对于《乱世佳人》这样的经典影片，最好还是简单地在点播的基础上播映。

对于近似视频点播，用户不具有VCR控制能力。没有用户能够暂停一部电影而去一趟厨房。他们所能做的最好的事情不过是当他们从厨房中返回时，向后退到随后开始的一个数据流，从而使漏过的几分钟资料重现。

实际上，近似视频点播还有另外一个模型。在这个模型中，人们可以在他们需要的任意时候预订电影，而不是预先宣布每隔5分钟将开演某部电影。每隔5分钟，系统要查看哪些电影已经被预订并且开始这些电影。采用这一方案时，根据点播的情况，一部电影可能在8:00、8:10、8:15和8:25开始，但不会在中间的时间开始。结果，没有观众的数据流就不会被传输，节约了磁盘带宽、内存和网络容量。另一方面，现在到厨房去制作冰淇淋就有点冒险，因为不能保证在观众正在观看的电影之后5分钟还有另一个数据流正在运行。当然，运营商可以给用户提供一个选项，以便显示所有同时发生的数据流的一个列表，但是大多数人觉得他们的电视机遥控器按钮已经太多，不大可能会热情地欢迎更多的几个按钮。

7.6.3 具有VCR功能的近似视频点播

将近似视频点播（为的是效率）加上每个个体观众完全的VCR控制（为的是方便用户）是一种理想的组合。通过对模型进行略微的修正，这样的设计是有可能的。下面我们将介绍为达到这一目标所采用的一种方法（Abram-Profeta和Shin, 1998），我们给出的是略微简化了的描述。

我们将以图7-17所示的标准近似视频点播模式为开端。可是，我们要增加要求，即要求每个客户机在本地缓冲前ΔT分钟以及即将来临的ΔT分钟。缓冲前ΔT分钟是十分容易的：只要在显示之后将其保存下来即可。缓冲即将来临的ΔT分钟是比较困难的，但是如果客户机有一次读两个数据流的能力也是可以实现的。

可以用一个例子来说明建立缓冲区的一种方法。如果一个用户在8:15开始观看电影，那么客户机读入并显示8:15的数据流（该数据流正处于第0帧）。与此并行，客户机读入并保存8:10的数据流，该数据流当前正处于5分钟的标记处（也就是第9000帧）。在8:20时，第0帧至第17 999帧已经被保存下来，并且用户下面将要看到的应该是第9000帧。从此刻开始，8:15的数据流被放弃，缓冲区用8:10的数据流（该数据流正处于第18 000帧）来填充，而显示则从缓冲区的中间点（第9000帧）驱动。当每一新的帧被读入时，在缓冲区的终点处添加一帧，而在缓冲区的起点处丢弃一帧。当前被显示的帧称为播放点（play point），它总是处于缓冲区的中间点。图7-18a所示为电影播放到第75分钟时的情形。此时，70分钟到80分钟之间所有的帧都在缓冲区中。如果数据率是4 Mbps，则10分钟的缓冲区需要300M字节的存储容量。以目前的价格，这样的缓冲区肯定可以在磁盘中保持，并且在RAM中保持也是可能的。如果希望使用RAM，但是300M字节又太大，那么可以使用小一些的缓冲区。

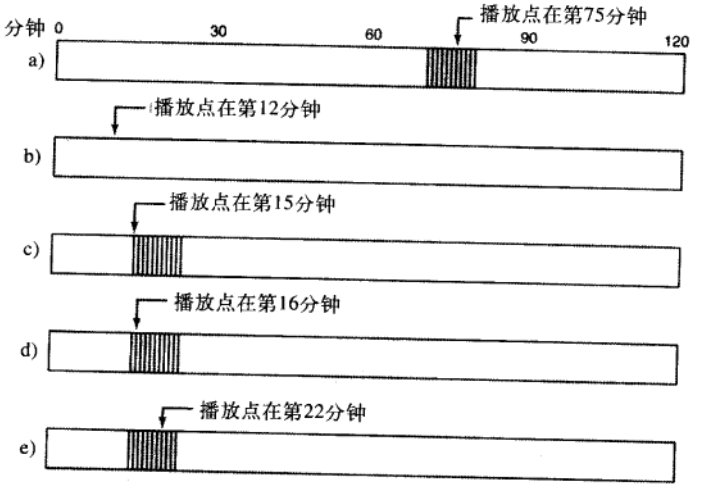


图7-18 a) 初始情形；b) 倒带至12分钟之后；c) 等待3分钟之后；d) 开始重填充缓冲区之后；e) 缓冲区满

现在假设用户决定要快进或者快倒。只要播放点保持在70到80分钟的范围之内，显示就可以从缓冲区馈入。然而，如果播放点在某个方向离开了这一区间，我们就遇到了问题。解决方法是开启一个私有（也就是视频点播）数据流以服务于用户。沿着某个方向快速运动可以用前面讨论过的技术来处理。

通常，在某一时刻用户可能会安下心来决定再次以正常速度观看电影。此时，我们可以考虑将用户迁移到某一近似视频点播数据流，这样私有数据流就可以被放弃。例如，假设用户决定返回到12分钟标号处，如图7-18b所示。这一点远远超出了缓冲区的范围，所以显示不可能从缓冲区馈入。此外，由于切换（立刻）发生在第75分钟，系统中存在着正在显示电影第5、10、15和20分钟那一帧的数据流，但是没有显示电影第12分钟那一帧的数据流。

解决方法是继续观看私有数据流，但是开始从当前正播放电影第15分钟那一帧的数据流填充缓冲区。经过3分钟之后的情形如图7-18c所示。播放点现在是第15分钟，缓冲区包含了15到18分钟的帧，而近似视频点播数据流正处在第8、13、18和23分钟。在这一时刻，私有数据流可以被放弃，显示可以从缓冲区馈入。缓冲区继续从现在正处于第18分钟的数据流填充。经过另一分钟之后，播放点是第16分钟，缓

冲区包含了15到19分钟的帧，并且数据流在第19分钟处将数据馈入缓冲区，如图7-18d所示。

经过另外6分钟之后，缓冲区变满，并且播放点是在第22分钟。播放点不是处于缓冲区的中间点，但是如果必要可以进行这样的整理。

7.7 文件存放

多媒体文件非常庞大，通常只写一次而读许多次，并且倾向于被顺序访问。它们的回放还必须满足严格的服务质量标准。总而言之，这些要求暗示着不同于传统操作系统使用的文件系统布局。我们在下面将讨论某些这样的问题，首先针对单个磁盘，然后是多个磁盘。

7.7.1 在单个磁盘上存放文件

最为重要的要求是数据能够以必要的速度流出到网络或输出设备上，并且没有颤动。为此，在传输一帧的过程中有多次寻道是极度不受欢迎的。在视频服务器上消除文件内寻道的一种方法是使用连续的文件。通常，使文件为连续的工作做得并不十分好，但是在预先精心装载了电影的视频服务器上它工作得还是不错的，因为这些电影后来不会再发生变化。

然而，视频、音频和文本的存在是一个复杂因素，如图7-3所示。即使视频、音频和文本每个都存储为单独的连续文件，从视频文件到音频文件，再从音频文件到文本文件的寻道在需要的时候还是免不了的。这使人想起第二种可能的存储排列，使视频、音频和文本交叉存放，但是整个文件还是连续的，如图7-19所示。此处，直接跟随第1帧视频的是第1帧的各种音频轨迹，然后是第1帧的各种文本轨迹。根据存在多少音频和文本轨迹，最简单的可能是在一次磁盘读操作中读入每一帧的全部内容，然后只将需要的部分传输给用户。

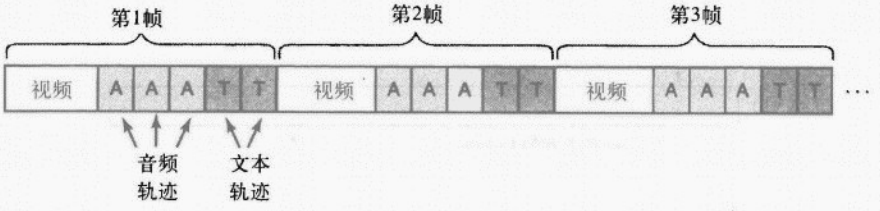


图7-19 每部电影在一个连续文件中交叉存放视频、音频和文本

这一组织需要额外的磁盘I/O读入不必要的音频和文本，在内存中还需要额外的缓冲区空间存放它们。可是它消除了所有的寻道（在单用户系统上），并且不需要任何系统开销跟踪哪一帧在磁盘上的什么地方，因为整部电影存放在一个连续文件中。以这样的布局，随机访问是不可能的，但是如果不需要随机访问，这点损失并不严重。类似地，如果没有额外的数据结构和复杂性，快进和快倒也是不可能的。

在具有多个并发输出流的视频服务器上，使整部电影成为一个连续文件的优点就失去了，因为从一部电影读取一帧之后，磁盘可能不得不从许多其他电影读入帧，然后才能返回到第一部电影。同样，对于一部电影既可以读也可以写的系统（例如用于视频生产或编辑的系统）来说，使用巨大的连续文件是很困难的，因而也是没有用的。

7.7.2 两个替代的文件组织策略

这些考虑导致两个针对多媒体文件的其他文件存放组织。第一个是小块模型，如图7-20a所示。在这种组织中，选定磁盘块的大小比帧的平均大小，甚至是比P帧和B帧的大小，要小得多。对于每秒30帧以4 Mbps速率传输的MPEG-2而言，帧的平均大小为16KB，所以一个磁盘块的大小为1KB或2KB工作得比较好。这里的思想是每部电影有一个帧索引，这是一个数据结构，每一帧有一个帧索引项，指向帧的开始。每一帧本身是一连串连续的块，包含该帧所有的视频、音频和文本轨迹，如图7-20中所示。这样，读第k帧时首先要在帧索引中找到第k个索引项，然后在一次磁盘操作中将整个帧读入。由于不同的帧具有不同的大小，所以在帧索引中需要有表示帧大小的字段（以块为单位），即便对于1KB大小的磁盘块，8位的字段也可以处理最大为255KB的帧，这对于一个未压缩NTSC帧来说，就算它有许多音频轨迹也已经足够了。

存放电影的另一个方法是使用大磁盘块（比如256KB），并且在每一块中放入多个帧，如图7-20b所示。这里仍然需要一个索引，但是这次不是帧索引而是块索引。实际上，该索引与图6-15中的i节点基本相同，只是可能还有额外的信息表明哪一帧处于每一块的开始，这样就有可能快速地找到指定的帧。一般而言，一个磁盘块拥有的帧的数目不见得是整数，所以需要做某些机制来处理这一问题。解决这一问题有两种选择。

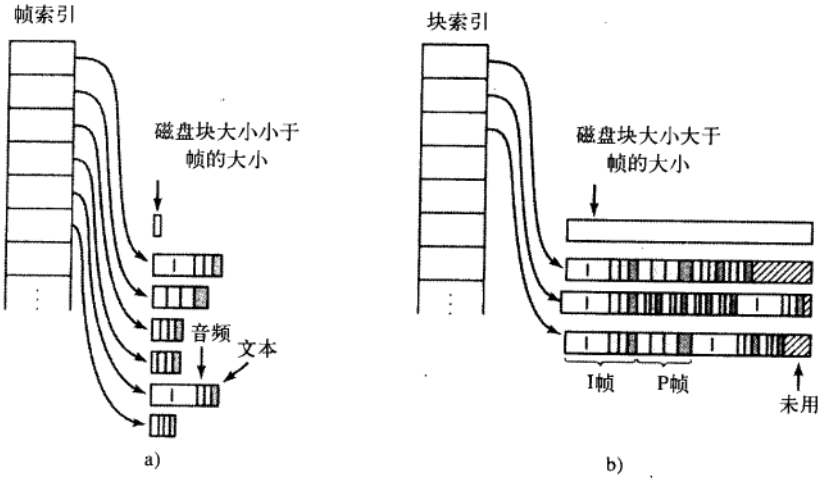


图7-20 不连续的电影存储：a) 小磁盘块；b) 大磁盘块

第一种选择如图7-20b所示，当下一帧填不满当前磁盘块的时候，则磁盘块剩余的部分就保持空闲状态。这一浪费的空间就是内部碎片，与具有固定大小页面的虚拟内存系统中的内部碎片相同。但是，这样做在一帧的中间决不需要进行寻道。

另一种选择是填充每一磁盘块到尽头，将帧分裂开使其跨越磁盘块。这一选择在帧的中间引入寻道的需要，这将损害性能，但是由于消除了内部碎片而节约了磁盘空间。

作为对比，图7-20a中小块的使用也会浪费某些磁盘空间，因为在每一帧的最后一块可能有一小部分未被使用。对于1KB的磁盘块和一部由216 000帧组成的2小时的NTSC电影，浪费的磁盘空间总共只有3.6GB中的大约108KB。图7-20b浪费的磁盘空间计算起来非常困难，但是肯定多很多，因为在一个磁盘块的尽头有时会留下100KB的空间，而下一帧是一个比它大的I帧。

另一方面，块索引比帧索引要小很多。对于256KB的块，如果帧的平均大小为16KB，那么一个块大约可以装下16个帧，所以一部由216 000帧组成的电影在块索引中只需要有13 500个索引项，与此相对比，对于帧索引则需要216 000个索引项。因为性能的原因，在这两种情形中索引都应该列出所有的帧或磁盘块（也就是说不像UNIX那样有间接块），所以块索引在内存中占用了13 500个8字节的项（4个字节用于磁盘地址，1个字节用于帧的大小，3个字节用于起始帧的帧号），帧索引则在内存中占用了216 000个5字节的项（只有磁盘地址和帧的大小），比较起来，当电影在播放时，块索引比帧索引节省了接近1MB的RAM空间。

这些考虑导出了如下的权衡：

1) 帧索引：电影在播放时使用大量的RAM；磁盘浪费小。

2) 块索引（禁止分裂帧跨越磁盘块）：RAM用量低；磁盘浪费较大。

3) 块索引（允许分裂帧跨越磁盘块）：RAM用量低；无磁盘浪费；需要额外寻道。

因此，这里的权衡涉及回放时RAM的使用量、自始至终浪费的磁盘空间以及由于额外寻道造成的回放时的性能损失。但是，这些问题可以用各种方法来解决。采用分页操作在需要的时候及时将帧索引装入内存，可以减少RAM的使用量。通过足够的缓冲可以屏蔽在帧传输过程中的寻道，但是这需要额外的内存并且可能还需要额外的复制操作。好的设计必须仔细分析所有这些因素，并且为即将投入的应

用做出良好的选择。

这里的另一个因素是图7-20a中的磁盘存储管理更加复杂，因为存放一帧需要找到大小合适的一连串连续的磁盘块。理想情况下，这一连串磁盘块不应该跨越一个磁道的边界，但是通过磁头偏斜，这一损失并不严重。然而，跨越一个柱面的边界则应该避免。这些要求意味着，磁盘的自由存储空间必须组织成变长孔洞的列表，而不是简单的块列表或者位图。与此相对照，块列表或者位图都可以用在图7-20b中。

在上述所有情况下，还要说明的是，只要可能应该把一部电影的所有块或者帧放置在一个狭窄的范围之内，比如说几个柱面。这样的存放方式意味着寻道可以更快，从而留下更多的时间用于其他（非实时）活动，或者可以支持更多的视频流。这种受约束的存放可以通过将磁盘划分成柱面组来实现，每个组保持单独的空闲块列表或位图。如果使用孔洞，可能存在一个1KB孔洞的列表、一个2KB孔洞的列表、一个3KB到4KB孔洞的列表、一个5KB到8KB孔洞的列表等。以这种方法在一个给定的柱面组中找到一个给定大小的孔洞是十分容易的。

这两种方法之间的另一个区别是缓冲。对于小块方法，每次读操作正好读取一帧。因此，采用简单的双缓冲策略就工作得相当好：一个缓冲区用于回放当前帧，另一个用于提取下一帧。如果使用固定大小的缓冲区，则每个缓冲区必须足够大以装得下最大可能的I帧。另一方面，如果针对每一帧从一个池中分配不同的缓冲区，并且当帧在被读入之前其大小未知，那么对于P帧和B帧就可以选择一个较小的缓冲区。

使用大磁盘块时，因为每一块包含多个帧，并且在每一块的尽头还可能包含帧的片段（取决于选定前面提到的是哪种选择），因而需要更加复杂的策略。如果显示或传输帧时要求它们是连续的，那么它们就必须被复制，但是复制是一个代价高昂的操作，应该尽可能避免。如果连续性是不必要的，那么跨越块边界的帧可以分两次送出到网络上或者送出到显示设备上。

双缓冲也可以用于大磁盘块，但是使用两个大磁盘块会浪费内存。解决浪费内存问题的一种方法是使用比为网络或显示器提供数据的磁盘块（每个数据流）稍大一些的循环传输缓冲区。当缓冲区的内容低于某个阈值时，从磁盘读入一个新的大磁盘块，将其内容复制到传输缓冲区，并且将大磁盘块缓冲区返还给通用池。循环缓冲区大小的选取必须使得在它达到阈值时，还有空间能够容纳另一个完整的磁盘块。因为传输缓冲区可能要环绕，所以磁盘读操作不能直接达到传输缓冲区。这里复制和内存的使用量相互之间存在着权衡。

在比较这两种方法时，还有另一个因素就是磁盘性能。使用大磁盘块时磁盘可以以全速运转，这经常是主要关心的事情。作为单独的单位读入小的P帧和B帧效率是比较低的。此外，将大磁盘块分解在多个驱动器上（下面将讨论）是可能的，而将单独的帧分解在多个驱动器上是不可能的。

图7-20a的小块组织有时称为*恒定时间长度*（constant time length），因为索引中的每个指针代表着相同的播放时间毫秒数。相反，图7-20b的组织有时称为*恒定数据长度*（constant data length），因为数据块的大小相同。

两种文件组织间的另一个区别是，如果帧的类型存储在图7-20a的索引中，那么有可能通过仅仅显示I帧实现快进。然而，根据I帧出现在数据流中的频度，人们可能会察觉到播放的速率太快或太慢。在任何情况下，以图7-20b的组织，这样的快进都是不可能的。实际上连续地读文件以选出希望的帧需要大量的磁盘I/O。

第二种方法是使用一个特殊的文件给人以10倍速度快进的感觉，而这个特殊的文件是以正常速度播放的。这个文件可以用与其他文件相同的方法构造，可以使用帧索引也可以使用块索引。打开一个文件的时候，如果需要，系统必须能够找到快进文件。如果用户按下快进按钮，系统必须立即找到并且打开快进文件，然后跳到文件中正确的地方。系统所知道的是当前所在帧的帧号，但是它所需要的是能够在快进文件中定位到相应的帧。如果系统当前所在的帧号是4816，并且知道快进文件是10倍速，那么它必须在快进文件中定位到第482帧并且从那里开始播放。

如果使用了帧索引，那么定位一个特定的帧是十分容易的，只要检索帧索引即可。如果使用的是块索引，那么每个索引项中需要有额外的信息以识别哪一帧在哪一块中，并且必须对块索引执行二分搜索。快倒的工作方式与快进相类似。

7.7.3 近似视频点播的文件存放

到目前为止我们已经了解了视频点播的文件存放策略。对于近似视频点播，采用不同的文件存放策

略可以获得更高的效率。我们还记得，近似视频点播将同一部电影作为多个交错的数据流送出。即使电影是作为连续文件存放的，每个数据流也需要进行寻道。Chen和Thapar（1997）设计了一种文件存放策略几乎可以消除全部这样的寻道。图7-21说明了这一方法的应用，图7-21中的电影以每秒30帧的速率播放，每隔5分钟开始一个新的数据流（参见图7-17）。根据这些参数，2小时长的电影需要24个当前数据流。

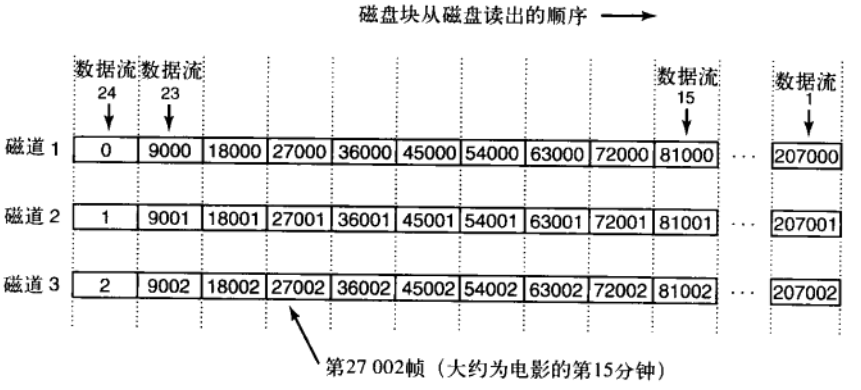


图7-21 针对近似视频点播的优化帧存放策略

在这一存放策略中，由24个帧组成的帧集合连成一串并且作为一个记录写入磁盘。它们还可以在一个读操作中被读回。考虑这样一个瞬间，数据流24恰好开始，它需要的是第0帧，5分钟前开始的数据流23需要的是第9000帧；数据流22需要的是第18 000帧，以此类推，直到数据流0，它需要的是第20 700帧。通过将这些帧连续地存放在一个磁道上，视频服务器只用一次寻道（到第0帧）就可以以相反的顺序满足全部24个数据流的需要。当然，如果存在某一原因要以升序为数据流提供服务，这些帧也可以以相反的顺序存放在磁盘上。完成对最后一个数据流的服务之后，磁盘臂可以移到磁道2准备再次为这些数据流服务。这一方法不要求整个文件是连续的，但是对于若干个同时的数据流仍然给予了良好的性能。

简单的缓冲策略是使用双缓冲。当一个缓冲区正在向外播放24个数据流的时候，另一个缓冲区正在预先加载数据。当前操作结束时，两个缓冲区进行交换，刚才用于回放的缓冲区现在在一个磁盘操作中加载数据。

一个有趣的问题是构造多大的缓冲区。显然，它必须能够装下24个帧。然而，由于帧的长度是变化的，选取正确大小的缓冲区并不完全是无足轻重的事情。使缓冲区大到足以装下24个I帧是不必要的过度行为，但是使缓冲区大小为24个平均帧则要冒风险。

幸运的是，对于任何一部给定的电影，电影中最大的磁道（在图7-21的意义上说）事先是已知的，所以可以选择缓冲区恰好为这一大小。然而，很有可能发生这样的事情，最大的磁道有16个I帧，而第二大的磁道只有9个I帧。选择缓冲区的大小能够足以装下第二大的磁道可能更为明智。做出这样的选择意味着要截断最大的磁道，因此对某些数据流将舍弃电影中的一帧。为避免低频干扰，前一帧可以再次显示，没有人会注意到这一问题。

进一步运用这一方法，如果第三大的磁道只有4个I帧，使用能够保存4个I帧和20个P帧的缓冲区是值得的。对某些数据流在电影中两次引入两个重复的帧可能是可以接受的。这样做下去何处是头呢？也许是缓冲区大小对于99%的帧而言足够大就行了。显然，在缓冲区使用的内存和电影的质量之间存在着权衡。注意，同时存在的数据流越多，统计数据就越好并且帧集合也越均匀。

7.7.4 在单个磁盘上存放多个文件

到目前为止我们还只考虑了单部电影的存放。在视频服务器上，当然存在着许多电影。如果它们随机地散布在磁盘上，那么当多部电影被不同的客户同时观看时，时间将浪费在磁头在电影之间来回移动上。

通过观察到某些电影比其他电影更为流行并且在磁盘上存放电影时将流行性考虑进去，可以改进这一情况。尽管总的来说有关个别电影的流行性并没有多少可说的（除了有大腕明星似乎有所帮助以外），但是大体上关于电影的相对流行性总还是可以说出一些规律。

对于许多种类的流行性比赛，诸如出租的电影、从图书馆借出的图书、访问的Web网页，甚至一部小说中使用的英文单词或者特大城市居住的人口，相对流行性的一个合理的近似遵循着一种令人惊奇的可预测模式。这一模式是哈佛大学的一位语言学教授George Zipf（1902—1950）发现的，现在被称为Zipf定律。该定律说的是，如果电影、图书、Web网页或者单词按其流行性进行排名，那么下一个客户选择排行榜中排名为k的项的概率是C/k，其中C是一个归一化常数。

因而，前三部电影的命中率分别是$C/1$、$C/2$和$C/3$，其中C的计算要使全部项的和为1。换句话说，如果有N部电影，那么

$$C/1 + C/2 + C/3 + C/4 + \cdots + C/N = 1$$

从这一公式，C可以被计算出来。对于具有10个、100个、1000个和10 000个项的总体，C的值分别是0.341、0.193、0.134和0.102。例如，对于1000部电影，前5部电影的概率分别是0.134、0.067、0.045、0.034和0.027。

图7-22说明了Zipf定律。只是为了娱乐，该定律被应用于美国20座最大城市的人口。Zipf定律预测第二大城市应该具有最大城市一半的人口，第三大城市应该具有最大城市三分之一的人口，以此类推。虽然不尽完美，该定律令人惊奇地吻合。

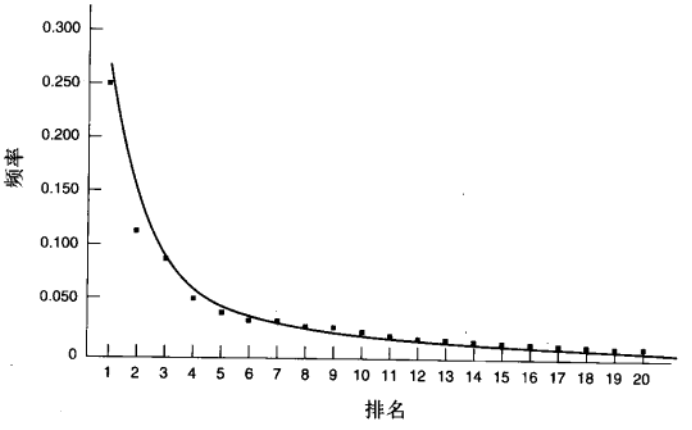


图7-22 当$N = 20$时的Zipf定律曲线。方块表示美国20座最大城市的人口，按排名顺序排列（纽约第一、洛杉矶第二、芝加哥第三等）

对于视频服务器上的电影而言，Zipf定律表明最流行的电影被选择的次数是第二流行的电影的两倍，是第三流行的电影的三倍，以此类推。尽管分布在开始时下降得相当快，但是它有着一个长长的尾部。例如，排名50的电影拥有$C/50$的流行性，排名51的电影拥有$C/51$的流行性，所以排名51的电影的流行性是排名50的电影的50/51，只有大约2%的差额。随着尾部进一步延伸，相邻电影间的百分比差额变得越来越小。一个结论就是，服务器需要大量的电影，因为对于前10名以外的电影存在着潜在的需求。

了解不同电影的相对流行性，使得对视频服务器的性能进行建模以及将该信息应用于存放文件成为可能。研究已经表明，最佳的策略令人惊奇地简单并且独立于分布。这一策略称为管风琴算法（organ-pipe algorithm）（Grossman和Silverman, 1973; Wong, 1983）。该算法将最流行的电影存放在磁盘的中央，第二和第三流行的电影存放在最流行的电影的两边，在这几部电影的外边是排名第四和第五的电影，以此类推，如图7-23所示。如果每一部电影是如图7-19所示类型的连续文件，这样的存放方式工作得最好；如果每一部电影被约束在一个狭窄的柱面范围之内，这样的存放方式也可以扩大其使用的范围。该算法的名字来自这样的事实——概率直方图看起来像是一个稍稍不对称的管风琴。

该算法所做的是试图将磁头保持在磁盘的中央。当服务器上的电影有1000部时，根据Zipf定律分布，排在前5名的电影代表了0.307的总概率，这意味着大约30%的时间磁头停留在为排在前5名的电影分配的柱面中，如果有1000部电影可用，这是一个惊人的数量。

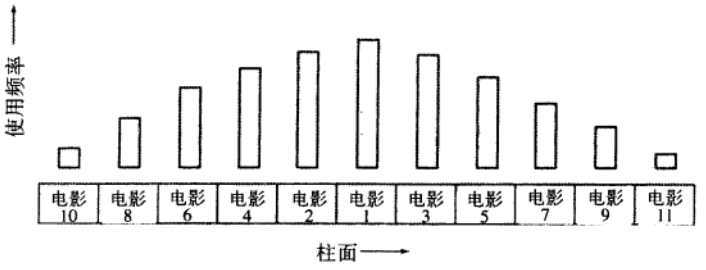


图7-23 视频服务器上文件的管风琴分布

7.7.5 在多个磁盘上存放文件

为了获得更高的性能，视频服务器经常拥有可以并行运转的很多磁盘。RAID有时会被用到，但是通常并不是因为RAID以性能为代价提供了更高的可靠性。视频服务器通常希望高的性能而对于校正传输错误不怎么太关心。除此之外，如果RAID控制器有太多的磁盘要同时处理，那么RAID控制器可能会成为一个瓶颈。

更为普通的配置只是数目很多的磁盘，有时被称为*磁盘囤*（disk farm）。这些磁盘不像RAID那样以同步方式旋转，也不像RAID那样包含奇偶校验位。一种可能的配置是将电影A存放在磁盘1上，将电影B存放在磁盘2上，以此类推，如图7-24a所示。实际上，使用新式的磁盘，每个磁盘上可以存放若干部电影。

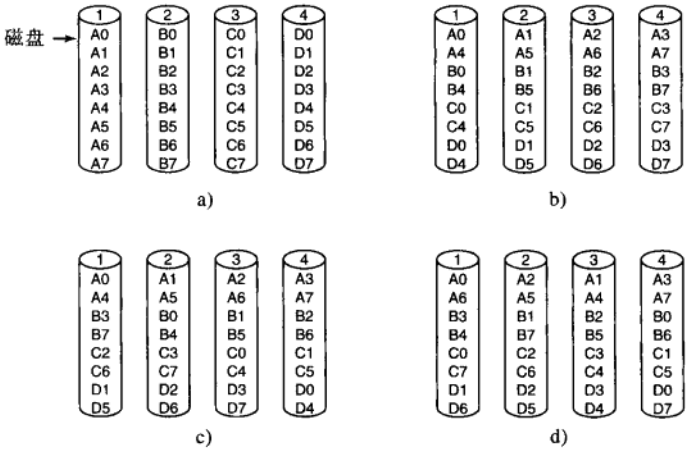


图7-24 在多个磁盘上组织多媒体文件的四种方式：a) 无条带；b) 所有文件采用相同的条带模式；c) 交错条带；d) 随机条带

这一组织方式实现起来很简单，并且具有简单明了的故障特性：如果一块磁盘发生故障，其上的所有电影都将不再可用。注意，一家公司损失了一块装满了电影的磁盘并没有一家公司损失了一块装满了数据的磁盘那么糟糕，因为电影还可以从DVD重新装载到一块空闲的磁盘中。这一方法的缺点是负载可能没有很好地平衡，如果某些磁盘上装载的是目前十分热门的电影，而另外的磁盘上装载的是不太流行的电影，则系统就没有被充分利用。当然，一旦知道了电影的使用频率，那么手工移动某些电影以平衡负载也是可能的。

第二种可能的组织方式是将每一部电影在多块磁盘上分成条带，图7-24b所示为4部电影的例子。让我们暂时假设所有的帧大小相同（也就是未压缩）。固定的字节数从电影A写入磁盘1，然后相同的字节数写入磁盘2，直到到达最后一块磁盘（在本例的情形中是A3单元）。然后，再次在第一块磁盘处继续分条带操作，写入A4单元，这样进行下去直到整个文件被写完。电影B、C和D以同样的模式分成条带。

由于所有的电影在第一块磁盘开始，这一条带模式的一个可能的缺点是跨磁盘的负载可能不平衡。一种更好地分散负载的方法是交错起始磁盘，如图7-24c所示。还有一种试图平衡负载的方法是对每一文件使用随机的条带模式，如图7-24d所示。

到目前为止，我们一直假设所有的帧大小相同，而对于MPEG-2电影，这一假设是错误的：I帧比P帧要大得多。有两种方法可以处理这一新出现的问题：按帧分条带或按块分条带。按帧分条带时，电影A的第一帧作为连续的单位存放在磁盘1上，不管它有多大。下一帧存放在磁盘2上，以此类推。电影B以类似的方式分条带，或者在同一块磁盘上开始，或者在下一块磁盘上开始（如果是交错条带），或者是在随机的一块磁盘上开始。因为每次读入一帧，这一条带形式并没有加快任何给定电影的读入，然而它比图7-24a更好地在磁盘间分散了负载，如果有许多人决定今晚观看电影A而没有人想看电影C，图7-24a的表现将很糟糕。总的来说，在所有的磁盘间分散负载将更好地利用总的磁盘带宽，并因此而增加能够服务的顾客数目。

分条带的另一种方法是按块分条带。对于每部电影，固定大小的单元连续（或随机）写到每块磁盘上。每个块包含一个或多个帧或者其中的碎片。对于同一部电影，系统现在可以发出对多个块的请求，每个请求要求读数据到不同的内存缓冲区，但是以这样的方式，当所有的请求都完成时，一个连续的电影片断（包含多个帧）在内存中将被连续地组装好。这些请求可以并行处理。当最后一个请求被满足时，可以用信号通知请求进程工作已经完成了，此时它就可以将数据传送给用户。许多帧过后，当缓冲区下降到最后几帧时，更多的请求将被发出，以便预装载另外一个缓冲区。这一方法使用了大量的内存作为缓冲区，从而使磁盘保持忙碌。在一个具有1000个活跃用户和1MB缓冲区的系统上（例如，在4块磁盘中的每块上使用256KB的磁盘块），将需要1GB 的RAM作为缓冲区。在1000个用户的服务器上，这样的内存用量只是“小意思”，应该不会有问题。

关于条带的最后一个问题是在多少个磁盘上分条带。在一个极端，每部电影将在所有的磁盘上分成条带。例如，对于2GB的电影和1000块磁盘，可以将2MB的磁盘块写在每块磁盘上，这样就没有电影两次使用同一块磁盘。在另一个极端，磁盘被分区为小的组（如同图7-24那样），并且每部电影被限制在一个分区中。前者称为*宽条带*（wide striping），它在平衡磁盘间负载方面工作良好。它的主要问题是每部电影使用了所有磁盘，如果一块磁盘出现故障，那么就没有电影可以观看了。后者称为*窄条带*（narrow striping），它将遭遇热点（广受欢迎的分区）的问题，但是损失一块磁盘将只是葬送存放在其分区中的电影。对于可变大小帧的划分条带，Shenoy和Vin（1999）在数学上进行了详细的分析。

7.8 高速缓存

传统的LRU文件高速缓存对于多媒体文件而言工作得并不好，这是因为电影的访问模式与文本文件有所不同。在传统的LRU缓冲区高速缓存背后的思想是，当一个块被使用之后，应该将其保存在高速缓存中，以防很快再次需要访问它。例如，在编辑一个文件的时候，文件被写入的一组磁盘块很可能反复地被用到，直到编辑过程结束。换言之，如果一个磁盘块在短暂的时间间隔内存在比较高的可能性要被重用的话，它就值得保存在高速缓存之中，以免将来对磁盘的访问。

对于多媒体而言，通常的访问模式是按顺序从头到尾观看一部电影。一个块不太可能被使用两次，除非用户对电影进行倒带操作以再次观看某一场景。因此，通常的高速缓存技术是行不通的。然而，高速缓存仍然是可以有帮助的，只不过是要以不同的方式使用。在下面几小节，我们来看一看适用于多媒体的高速缓存技术。

7.8.1 块高速缓存

尽管只是将一个块保存起来期望它可能很快再次被用到是没有意义的，但是可以利用多媒体系统的可预测性，使高速缓存再度成为十分有益的技术。假设两个用户正在观看同一部电影，其中一个用户在另一个用户2秒钟之后开始观看。当第一个用户取出并观看了任何一个给定的块之后，很有可能第二个用户在2秒钟后将需要相同的块。系统很容易跟踪哪些电影只有一个观众，哪些电影有两个或更多个在时间上相隔很近的观众。

因此，只要一部电影中的一个块读出后很快会再次需要，对其进行高速缓存就是有意义的，当然是否进行高速缓存还取决于它要被高速缓存多长时间以及内存有多紧张。这里应该使用不同的策略，而不

是将所有磁盘块保留在高速缓存之中并且在高速缓存被填满之后淘汰最近最少使用的。对于在第一个观众之后ΔT时间之内有第二个观众的每一部电影，可以将其标记为可高速缓存的，并且高速缓存其所有磁盘块直到第二个观众（也可能是第三个观众）使用。对于其他的电影，根本不需要进行高速缓存。

这一思想还可以进一步发挥。在某些情况下合并两个视频流是可行的。假设两个用户正在观看同一部电影，但是在两个用户之间存在10秒钟的延迟。在高速缓存中保留10秒钟的磁盘块是有可能的，但是要浪费内存。一种替代的方法是试图使两部电影同步，这一方法可以通过改变两部电影的帧率实现，图7-25演示了这一思想。

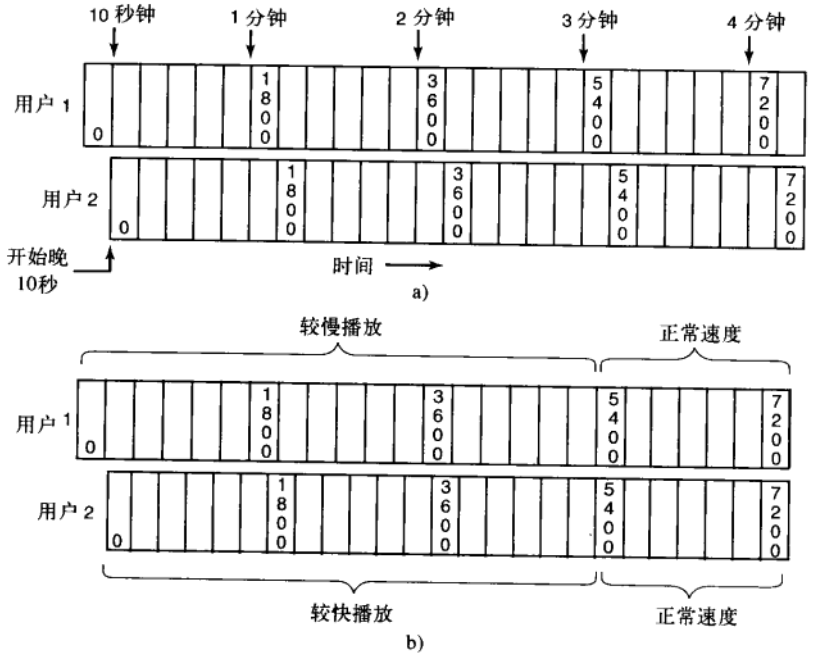


图7-25 a) 两个用户观看失步10秒钟的同一部电影；b) 将两个视频流合并为一个

在图7-25a中，两部电影均以每分钟1800帧的NTSC速率播放，由于用户2开始晚了10秒钟，他将在整部电影播放过程中落后10秒钟。然而，在图7-25b中，当用户2到来时，用户1的视频流将放慢，在接下来的3分钟里，它不是以每分钟1800帧的速率播放，而是以每分钟1750帧的速率播放，3分钟后，它正处于第5550帧。与此同时，用户2的视频流在最初的3分钟里以每分钟1850帧的速率播放，3分钟后，它同样也处于第5550帧。从此刻之后，两个视频流均以正常速度播放。

在追赶阶段，用户1的视频流运行速度慢了2.8%，而用户2的视频流运行速度快了2.8%。用户不太可能会注意到这一点。然而，如果对此有所担心，那么追赶阶段可以在比3分钟更长的时间间隔上展开。

一种降低一个用户的速度以便与另一个视频流合并的可选方法是，给用户以在他们的电影中包含广告的选项，与无广告的电影相比，其观看价格比较低。用户还可以选择产品门类，这样广告的侵扰就会小一些而更有可能被观看。通过对广告的数目、长度和时间安排进行巧妙的操作，视频流就可以被阻滞足够长的时间，以便与期望的视频流取得同步（Krishnan，1999）。

7.8.2 文件高速缓存

在多媒体系统中高速缓存还能够以不同的方式提供帮助。由于大多数电影都非常大（3～6GB），视频服务器通常不能在磁盘上存放所有这些文件，所以要将它们存放在DVD或磁带上。当需要一部电影的时候，它总是可以被复制到磁盘上，但是存在大量的启动时间来查找电影并将其复制到磁盘上。因此，大多数视频服务器维护着一个请求最频繁的电影的磁盘高速缓存。流行的电影将完整地存放在磁盘上。

使用高速缓存的另一种方法是在磁盘上保存每部电影的最初几分钟。这样，当一部电影被请求时，可以立刻从磁盘文件开始回放，与此同时，电影从DVD或磁带复制到磁盘上。通过始终在磁盘上存放电影足够长的部分，电影的下一个片断在它需要之前就已经取到磁盘上的概率会很高。如果一切都进行得很好，整部电影将在它需要之前就已经在磁盘上了，然后它将进入高速缓存并且停留在磁盘上以备随后有更多的请求。如果太多的时间过去而没有另外的请求，电影将从高速缓存中删除，以便为更为流行的电影腾出空间。

7.9 多媒体磁盘调度

多媒体对磁盘提出了与传统的、面向文本的应用程序（例如编译器或字处理器）有所不同的要求。特别是，多媒体要求极高的数据率和数据的实时传输。这些都不是轻易就能够提供的。此外，在视频服务器的情形中，让一个服务器同时处理几千个客户还存在着经济压力。这些需求影响着整个系统。上面我们了解了文件系统，现在让我们来看一看多媒体磁盘调度。

7.9.1 静态磁盘调度

尽管多媒体对系统的所有部分提出了巨大的实时和数据率要求，但是它还有一个特性使其比传统的系统更加容易处理，这就是可预测性。在传统的操作系统中，对磁盘块的请求是以相当不可预测的方式发出的。磁盘子系统所能做的最好不过是对于每个打开的文件执行一个磁盘块的预读，除此之外，它能够做的全部事情就是等待请求的到来，并且在请求时对它们进行处理。多媒体就不同了，每个活动的视频流对系统施加明确的负载，使系统成为高度可预测的。就NTSC回放而言，每33.3ms，每个客户将需要其文件中的下一帧，并且系统有33.3ms的时间提供所有的帧（系统对每个视频流需要缓冲至少一帧，所以取第$k+1$帧可以与第k帧的回放并行处理）。

这一可预测的负载可以用来使用为多媒体剪裁的算法对磁盘进行调度。下面我们将只考虑一个磁盘，但是其思想也可以运用于多个磁盘。就这个例子而言，我们将假设存在10个用户，每个用户观看不同的电影。此外，我们还将假设所有的电影具有相同的分辨率、帧率和其他特性。

根据系统的其他部分，计算机可能有10个进程，每个视频流一个进程，或者有一个具有10个线程的进程，或者甚至只有一个具有一个线程的进程，以轮转方式处理10个视频流。细节并不重要，重要的是，时间被分割成回环（round），在这里一个回环是一帧的时间（对于NTSC是33.3ms，对于PAL是40ms）。在每一回环的开始，为每个用户生成一个磁盘请求，如图7-26所示。

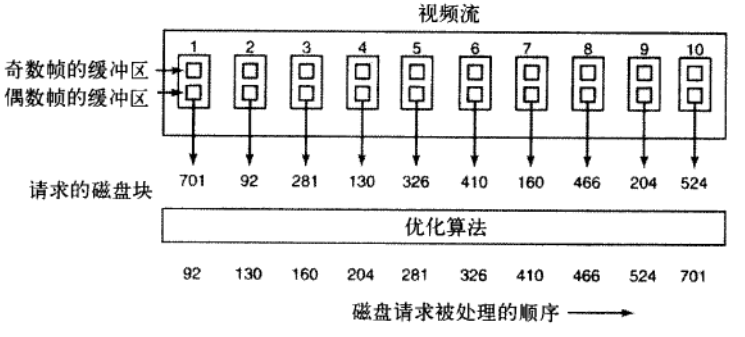


图7-26 在一个回环中，每部电影请求一帧

在回环的起始处，当所有的请求都进来之后，磁盘就会知道在那个回环期间它必须做什么，它还知道直到处理完这些请求并且下一个回环开始，不会有其他的请求进来。因此，它能够以优化的方法对请求排序，可能是以柱面顺序（可以想象在某些情形也可能以扇区顺序）排序，然后以优化的顺序对它们进行处理。在图7-26中，显示的请求是以柱面顺序排序的。

乍一看，人们可能会认为以这样的方式优化磁盘没有什么价值，因为只要磁盘满足最终时限，那么它是以1ms的富余满足还是以10ms的富余满足并没有什么关系。然而，这一结论是错误的。通过以这样的方式优化寻道，处理每一请求的平均时间就缩短了，这意味着一般来说每一回环磁盘可以处理更多的

视频流。换句话说，像这样优化磁盘请求增加了服务器可以同时传送的电影数。回环末尾的富余时间还可以用来服务可能存在的任何非实时请求。

如果服务器有太多的视频流，偶尔也会出现当要求从磁盘的边缘部分读取帧时错过了最终时限的情况。但是，只要错过最终时限的情况足够稀少，以此换取同时处理更多的视频流还是可以容忍的。注意，要紧的是读取的视频流的数目，每个视频流有两个或更多个客户并不影响磁盘性能或调度。

为了保持输出给客户的数据流运行流畅，在服务器中采用双缓冲是必要的。在第1个回环期间，使用一组缓冲区，每个视频流一个缓冲区。在这个回环结束的时候，输出进程或进程组被解除阻塞并且被告知传输第1帧。与此同时，新的请求进来请求每部电影的第2帧（每部电影或许有一个磁盘线程和一个输出线程）。这些请求必须用第二组缓冲区来满足，因为第一组缓冲区仍然在忙碌中。当第3个回环开始的时候，第一组缓冲区已经空闲，可以重新用来读取第3帧。

我们一直在假设每一帧只有一个回环，这一限制并不是严格必需的。每一帧也可以有两个回环，以便减少所需缓冲区空间的数量，其代价是磁盘操作的次数增加了一倍。类似地，每一回环可以从磁盘中读取两帧（假设一对帧连续地存放在磁盘上）。这一设计将磁盘操作的数目减少了一半，其代价是所需缓冲区空间的数量增加了一倍。依靠相对可利用率、性能和内存费用与磁盘I/O的对比，可以计算并使用优化策略。

7.9.2 动态磁盘调度

在上面的例子中，我们假设所有的视频流具有相同的分辨率、帧率和其他特性，现在让我们放弃这一假设。不同的电影现在可能具有不同的数据率，所以不可能每33.3ms有一个回环并且为每个视频流读取一帧。对磁盘的请求或多或少是随机到来的。

每一读请求需要指定要读的是哪一磁盘块，另外还要指定什么时间需要该磁盘块，也就是最终时限。为简单起见，我们假设对于每次请求实际的服务时间是相同的（尽管这肯定是不真实的）。以这种方法，我们可以从每次请求减去固定的服务时间，得到请求能够发出并且还能满足最终时限的最近的时间。因为磁盘调度程序所关心的是对请求进行调度的最终时限，所以这样做使模型更为简洁。

当系统启动的时候，还没有挂起的磁盘请求。当第一个请求到来的时候，它立即得到服务。当第一次寻道发生的时候，其他请求可能到来，所以当第一次请求结束的时候，磁盘驱动器可能要选择下一次处理哪个请求。某个请求被选中并开始得到处理。当该请求结束的时候，再一次有一组可能的请求：它们是第一次没有被选中的请求和第二个请求正在被处理的时候新到来的请求。一般而言，只要一个磁盘请求完成，磁盘驱动器就有若干组挂起的请求，必须从中做出选择。问题是："使用什么算法选择下一个要服务的请求？"

在选择下一个磁盘请求时，有两个因素起着重要的作用：最终时限和柱面。从性能的观点来看，保持请求存放在柱面上并且使用电梯算法可以将总寻道时间最小化，但是可能导致存放在边缘柱面上的请求错过其最终时限。从实时的观点来看，将请求按照最终时限排序并且以最终时限的顺序对它们进行处理，可以将错过最终时限的机会最小化，但是可能增加总寻道时间。

使用scan-EDF算法（scan-EDF algorithm）（Reddy和Wyllie，1994）可以将这两个因素结合起来。这一算法的思想是，将最终时限比较接近的请求收集在一起分成若干批，并且以柱面的顺序对其进行处理。作为一个例子，我们考虑图7-27当t = 700时的情形。磁盘驱动器知道它有11个挂起的请求，这些请求具有不同的最终时限和不同的柱面。它可以决定将具有最早的最终时限的5个请求视为一批，将它们按照柱面号排序，并且使用电梯算法以柱面顺序对它们进行服务。于是，顺序将是110、330、440、676和680。只要每个请求能够在其最终时限之前完成，这些请求就可以安全地重新排列，从而将所需的总寻道时间最小化。

如果不同的视频流具有不同的数据率，那么当一个新的客户出现时将引起一个严重的问题：该客户是否应该被接纳？如果接纳该客户会导致其他的视频流频繁地错过它们的最终时限，那么答案可能就是否。存在两种方法计算是否接纳新的客户。一种方法是假设每个客户平均地需要某些数量的资源，如磁盘带宽、内存缓冲区、CPU时间等。如果剩下的每一资源对于一个平均的顾客来说都是足够的，则接纳新的客户。

另一种算法更为复杂。它要关注新顾客想要看的特定的电影，查找该电影的（预先计算的）数据率，而对于黑白片和彩色片、卡通片和故事片、爱情片和战争片，数据率都不相同。爱情片运动缓慢，具有较长的场景和缓慢的淡入淡出，所有这些都会充分得到压缩，而战争片具有许多快速的切换和迅速的运动，因此具有许多I帧和较大的P帧。如果服务器对于新客户想要看的电影而言具有足够的容量，那么就准许接纳，否则就拒绝接纳。

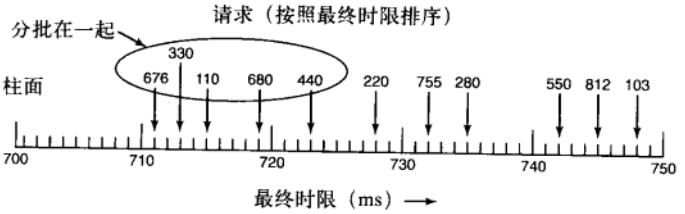


图7-27 scan-EDF算法使用最终时限和柱面号进行调度

7.10 有关多媒体的研究

多媒体是近些年的热门课题，所以有相当多数量关于多媒体的研究。这些研究中有许多是关于内容、构造工具和应用的，而这些都超出了本书的范围。另一个热门领域是多媒体与网络，这也超出了本书范围。但是关于多媒体服务器的研究，尤其是分布式服务器与操作系统是相关的（Sarhan和Das, 2004；Matthur和Mundur, 2004；Zaia等人，2004）；支持多媒体的文件系统也是与操作系统相关的研究（Ahn等人，2004；Cheng 等人，2005；Kang 等人，2006；Park和Ohm，2006）。

优秀的音频和视频编码（尤其是3D应用）对于高性能是很关键的。因此，这些课题也引起了相当程度的关注（Chattopadhyay 等人，2006；Hari等人，2006；Kum和Mayer-Patel,2006）。

服务质量对多媒体系统非常重要，所以吸引了相当的关注（Childs和Ingram，2001；Tamai 等人，2004）。与服务质量有关的还有调度，以及 CPU（Etsion 等人，2004；Etsion 等人，2006；Nieh和Lam，2003；Yuan和Nahrstedt，2006）和硬盘（Lund和Goebel，2003；Reddy 等人，2005）。

在为付费客户提供多媒体广播编排服务时，安全就变得很重要了，所以这个课题也受到了关注Barni，2006。

7.11 小结

多媒体是一种非常有前途的计算机应用。由于多媒体文件的巨大和苛刻的实时回放要求，为文本而设计的操作系统对于多媒体而言不是最理想的。多媒体文件包含多重平行的轨迹，通常有一个视频轨迹和至少一个音频轨迹，有时还有一些字幕轨迹。在回放期间，这些轨迹都必须保持同步。

音频通过周期性地对音量进行采样而得以记录下来，通常每秒采样44 100次（针对CD质量的声音）。压缩可以应用于音频信号，得到大约10倍的均匀的压缩率。视频压缩可以使用帧内压缩（JPEG），也可以使用帧间压缩（MPEG）。后者将P帧表示为与前一帧的差，而B帧则既可以基于前面的帧，也可以基于后面的帧。

多媒体需要实时调度以便满足其最终时限。通常使用的算法有两个。第一个算法是速率单调调度，它是一个静态抢先算法，它根据进程的周期将固定的优先级分配给进程。第二个算法是最早最终时限优先调度，它是一个动态算法，总是选择具有最近最终时限的进程。EDF更复杂一些，但是它可以达到100%的利用率，而RMS有时不能达到。

多媒体文件系统通常使用推送型模型而不是拉取型模型。一旦开始一个视频流，则数据位就从服务器不断流出而不需要用户进一步请求。这一方法从根本上不同于常规的操作系统，但是为了满足实时要求这样做是必要的。

文件可以连续存放也可以不连续存放。在后一种情况下，存储单位可以是可变长度的（一个磁盘块是一帧），也可以是固定长度的（一个磁盘块是多个帧）。这些方法具有不同的权衡。

磁盘上文件的存放格局影响着系统的性能。当存在多个文件时，有时使用风琴管算法。横跨多个磁盘将文件分成条带（无论是宽条带还是窄条带）也是常用的。为了改进性能，磁盘块与文件高速缓存策略也得到了广泛的利用。

习题

1. 未压缩的黑白NTSC电视能否通过快速以太网发送？如果可以的话，同时可以发送多少个频道？
2. HDTV的水平分辨率是常规电视的两倍（1280像素对640像素）。利用正文中提供的信息，它需要的带宽比标准电视多多少？
3. 在图7-3中，对于快进和快倒存在着单独的文件。如果一台视频服务器还打算支持慢动作，那么对于前进方向的慢动作是否需要另一个文件？后倒的方向如何？
4. 声音信号用16位有符号数（1个符号位，15个数值位）采样。以百分比表示的最大量化噪声是多少？对于长笛协奏曲或摇滚音乐这是不是一个大问题，或者对于两者是不是问题相同？请解释你的答案。
5. 唱片公司能够使用20位采样制作数字唱片的母片，最终发行给听众的唱片使用的是16位采样。请提出一种方法来减少量化噪声的影响，并讨论你的方案的优点和缺点。
6. DCT变换使用8 × 8的块，但是用于运动补偿的算法使用16 × 16的块。这一差异是否会导致问题？如果是的话，在MPEG中这个问题是怎样解决的？
7. 在图7-10中，我们看到对于静止的背景和运动角色MPEG是如何工作的。假设一个MPEG视频是由这样的场景制作的：在该场景中摄像机被安装在一个三脚架上并且从左到右摇动镜头，摇动的速度使得没有两幅连续的帧是相同的。现在是不是所有的帧都必须是I帧？为什么？
8. 假设图7-13中的三个进程中的每个进程都伴随一个进程，该进程支持一个音频流按照与视频进程相同的周期播放，那么音频缓冲区可以在视频帧之间得到更新。所有这三个音频进程都是完全相同的。对于一个音频进程的每一次突发，有多少可用的CPU时间？
9. 两个实时进程在一台计算机上运行，第一个进程每25ms运行10ms，第二个进程每40ms运行15ms。RMS对于它们是否总是起作用？
10. 一台视频服务器的CPU利用率是65%。采用RMS调度该服务器可以放映多少部电影？
11. 在图7-15中，EDF算法保持CPU 100%忙碌直到t = 150的时刻，它不能保持CPU无限期地忙碌，因为CPU每秒只有975ms的工作要做。扩展该图到150ms之后并确定采用EDF算法CPU何时首次变为空闲。
12. DVD可以保存足够的数据用于全长的电影并且传输率足够显示电视质量的节目。为什么视频服务器不采用许多DVD驱动器的”储存库”作为数据源？
13. 近似视频点播系统的操作员发现某个城市的人们不愿意为电影开始而等待超过6分钟的时间。对于一部3小时的电影，需要多少个并行的数据流？
14. 考虑一个采用Abram-Profeta与Shin提出的方法的系统，在这个系统中视频服务器操作员希望客户能够完全在本地向前或向后搜索1分钟。假设视频流是速率为4 Mbps的MPEG-2，每个客户在本地必须有多大的缓冲区空间？
15. 考虑Abam-Profeta和Shin方法。如果用户用大小为50MB的RAM用来缓冲，那么一个2Mbps的视频流的ΔT是多少？
16. 一个HDTV视频点播系统使用图7-20a的小块模型，磁盘块大小为1KB。如果视频分辨率为1280 × 720并且数据流速率为12 Mbps，那么在一部采用NTSC制式的2小时长的电影中有多少磁盘空间浪费在内部碎片上？
17. 针对NTSC和PAL思考图7-20a的存储分配方法。对于给定的磁盘块和影片大小，是否一种制式比另一种制式具有更多的内部碎片？如果是的话，哪一种制式要好一些？为什么？
18. 考虑图7-20所示的两种选择。向HDTV的转换是否更有利于其中的一种系统？请讨论。
19. 考虑一个系统，它有2KB磁盘块，能存储2小时的PAL制电影，平均每帧16KB。那么用小磁盘块存储方法平均浪费空间是多少？
20. 上例中，如果每帧项需要8字节，其中有1字节用来指示每帧的磁盘块号，那么可能存储的最长的电影大小是多少？
21. 当每一个帧集合的大小相同时，Chen与Thapar的近似视频点播方法工作得最为出色。假设一部电影正在用同时发出的24个数据流播放，并且10帧中有1帧是I帧。再假设I帧的大小是P帧的10倍，B帧的大小与P帧相同。一个等于4个I帧和20个P帧的缓冲区不够大的概率是多少？你认为这样的一个缓冲区大小是可接受的吗？为了使问题易于处理，假设在数据流中帧的类型是随机且独立分布的。
22. 对于Chen和Thapar方法，假设有5个轨道需要8I-帧，35个轨道需要5I-帧，45个轨道需要3I-帧，15个帧从1到2帧中选择，如果我们想保证95帧能被缓冲器容纳，那么缓冲器的大小应该是多少？

23. 对于Chen和Thapar方法，假定有一个用PAL制格式编码的3小时电影，需要在每个15分钟内流出。那么需要多少个并发流？
24. 图7-18的最终结果是播放点不再处于缓冲区的中间。设计一个方案，最少在播放点之后有5分钟并且在播放点之前有5分钟。你可以做出任何合理的假设，但是陈述要清楚。
25. 图7-19的设计要求所有语言轨迹在每一帧上读出。假设视频服务器的设计者必须支持大量的语言，但是不想将这么多的RAM投入给缓冲区以保存每一帧。其他可利用的选择是什么？每一种选择的优点和缺点是什么？
26. 一台小的视频服务器具有8部电影。对于最流行的电影、第二流行的电影，直到最不流行的电影，Zipf定律预测的概率是多少？
27. 一块具有1000个柱面的14GB的磁盘用于保存以4 Mbps速率流动的1000个30秒的MPEG-2视频剪辑。这些视频剪辑根据管风琴算法存放。依照Zipf定律，磁盘臂花在中间10个柱面的时间比例是多少？
28. 假设对于影片A、B、C和D的相对需求由Zipf定律所描述，对于如图7-24中所示的四种划分条带的方法，四块磁盘的期望相对利用率是多少？
29. 两个视频点播客户相隔6秒钟开始观看同一部PAL电影。如果系统加快一个数据流并且减慢另一个数据流以便使它们合并，为了在3分钟内将它们合并，需要的加速/减速百分比是多少？
30. 一台MPEG-2视频服务器对于NTSC视频使用图7-26的回环方法。所有的视频流出自一个转速为10 800 rpm的UltraWide SCSI磁盘，磁盘的平均寻道时间是3ms。能够得到支持的数据流有多少？
31. 重做前一个习题，但是现在假设scan-EDF算法将平均寻道时间减少了20%。现在能够得到支持的数据流有多少？
32. 考虑到下面一系列对磁盘的需求，每个需求由一个元组（截止时间（ms），柱面）代表。使用scan_EDF算法后，四个即将到期的需求聚集在一起得到服务。如果服务每个请求的平均时间是6ms，那么有没有错过的终止时间？
(32 300)；(36 500)；(40 210)；(34 310)
假定当前时间是15ms。
33. 再次重做前一个习题，但是现在假设每一帧在四块磁盘上分成条带，在每块磁盘上scan-EDF算法将平均寻道时间减少了20%。现在能够得到支持的数据流有多少？
34. 正文描述了使用五个数据请求为一批来调度在图7-27a中所描述的情形。如果所有请求需要等量的时间，在这个例子中每个请求可以允许的最大时间是多少？
35. 供生成计算机”墙纸”的许多位图图像使用很少的颜色并且十分容易压缩。一种简单的压缩方法是：选择一个不在输入文件中出现的数据值，并且将其用作一个标志。一个字节一个字节地读取文件，寻找重复的字节值。将单个值和最多重复三次的字节直接复制到输出文件。当4个或更多字节的重复串被发现时，将一个由3个字节组成的串写到输出文件，这3个字节的串包括标志字节、指示从4到255计数的字节和在输入文件中发现的实际的值。使用该算法编写一个压缩程序，以及一个能够恢复原始文件的解压缩程序。额外要求：如何处理在数据中包含标志字节的文件？
36. 计算机动画是通过显示具有微小差异的图像序列实现的。编写一个程序，计算两幅具有相同尺寸的未压缩位图图像之间的字节和字节的差。当然，输出文件应该与输入文件具有相同的大小。使用这一差值文件作为前一个习题中的压缩程序的输入，并且将这一方法的效率和压缩单个图像的情况进行比较。
37. 实现教材中的基本RMS和EDF算法。程序的主要输入是一个有若干行的文件，每行代表一个进程的CPU请求，并且有如下的参数：周期（秒）、计算时间（秒）、开始时间（秒）、结束时间（秒）。在以下方面对比两个算法：a) 由于CPU的不可调度性导致平均被阻塞的CPU请求数；b) 平均CPU使用率；c) 每个CPU请求的平均等待时间；d) 错过截止时间的请求平均数量。
38. 实现存储多媒体文件的常量时间长度和常量数据长度的技术。程序的主要输入是一系列文件，每个文件包含一个MPEG-2压缩多媒体文件（如电影）的每帧元数据。元数据包括帧类型（I/P/B）、帧长、相关联的音频帧等。对于不同文件块大小，就需要的总存储空间大小、浪费的磁盘存储空间和平均RAM需求三个方面比较两种技术。
39. 在上面的程序上添加一个“读者”程序，它随机地从上面的输入列表中选择文件，使用VCR功能的视频点播或准视频点播。实现scan-EDF算法以便能够给出磁盘读请求。就每个文件的平均寻找磁盘次数比较常量时间长度和常量数据长度这两个方法。

第 8 章　多处理机系统

从计算机诞生之日起，人们对更强计算能力的无休止的追求就一直驱使着计算机工业的发展。ENIAC可以完成每秒300次的运算，它一下子就比以往任何计算器都快1000多倍，但是人们并不满足。我们现在有了比ENIAC快数百万倍的机器，但是还有对更强大机器的需求。天文学家们正在了解宇宙，生物学家正在试图理解人类基因的含义，航空工程师们致力于建造更安全和速度更快的飞机，而所有这一切都需要更多的CPU周期。然而，即使有更多运算能力，仍然不能满足需求。

过去的解决方案是使时钟走得更快。但是，现在开始遇到对时钟速度的限制了。按照爱因斯坦的相对论，电子信号的速度不可能超过光速，这个速度在真空中大约是30cm/ns，而在铜线或光纤中约是20cm/ns。这在计算机中意味着10GHz 的时钟，信号的传送距离总共不会超过2cm。对于100GHz的计算机，整个传送路径长度最多为2mm。而在一台1THz（1000GHz）的计算机中，传送距离就不足100μm了，这在一个时钟周期内正好让信号从一端到另一端并返回。

让计算机变得如此之小是可能的，但是这会遇到另一个基本问题：散热。计算机运行得越快，产生的热量就越多，而计算机越小就越难散热。在高端Pentium系统中，CPU的散热器已经比CPU自身还要大了。总而言之，从1MHz到1GHz需要的是更好的芯片制造工艺，而从1GHz到1THz 则需要完全不同的方法。

获得更高速度的一种处理方式是大规模使用并行计算机。这些机器有许多CPU，每一个都以“通常”的速度（在一个给定年份中的速度）运行，但是总体上会有比单个CPU强大得多的计算能力。具有1000个CPU的系统已经商业化了。在未来十年中，可能会建造出具有100万个CPU的系统。当然为了获得更高的速度，还有其他潜在的处理方式，如生物计算机，但在本章中，我们将专注于有多个普通CPU的系统。

在高强度的数据处理中经常采用高度并行计算机。如天气预测、围绕机翼的气流建模、世界经济模拟或理解大脑中药物-受体的相互作用等问题都是计算密集型的。解决这些问题需要多个CPU同时长时间运行。在本章中讨论的多处理机系统被广泛地用于解决这些问题以及在其他科学、工程领域中的类似问题。

另一个相关的进展是因特网不可思议地快速增长。因特网最初被设计为一个军用的容错控制系统的原型，然后在从事学术研究的计算机科学家中流行开来，并且在过去它已经获得了许多新用途。其中一种用途是，把全世界的数千台计算机连接起来，共同处理大型的科学问题。在某种意义上，一个包含有分布在全世界的1000台计算机的系统与在一个房间中有1000台计算机的系统之间没有差别，尽管这两个系统在延时和其他技术特征方面会有所不同。在本章中我们也将讨论这些系统。

假如有足够多的资金和足够大的房间，把一百万台无关的计算机放到一个房间中很容易做到。把一百万台无关的计算机放到全世界就更容易了，因为不存在第二个问题了。当要在一个房间中使这些计算机相互通信，以便共同处理一个问题时，问题就出现了。结果，人们在互连技术方面做了大量工作，而且不同的互连技术已经导致了不同性质的系统以及不同的软件组织。

在电子（或光学）部件之间的所有通信，归根结底是在它们之间发送消息——具有良好定义的位串(bit string)。其差别在于所涉及的时间范围、距离范围和逻辑组织。一个极端的例子是共享存储器多处理机，系统中有从2个到1000个的CPU通过一个共享存储器通信。在这个模型中，每个CPU可同样访问整个物理存储器，可使用指令LOAD和STORE读写单个的字。访问一个存储器字通常需要2~10ns。尽管这个模型，如图8-1a所示，看来很简单，但是实际上要实现它并不那么简单，而且通常涉及底层大量的消息传递，这一点我们会简要地加以说明。不过，该消息传递对于程序员来说是不可见的。

其次是图8-1b中的系统，许多CPU-存储器通过某种高速互连网络连接在一起。这种系统称为消息传递型多计算机。每个存储器局部对应一个CPU，且只能被该CPU访问。这些CPU通过互连网络发送多字消息通信。存在良好的连接时，一条短消息可在10~50μs之内发出，但是这仍然比图8-1a中系统的存

储器访问时间长。在这种设计中没有全局共享的存储器。多计算机（消息传递系统）比（共享存储器）多处理机系统容易构建，但是编程比较困难。可见，每种类型各有其优点。

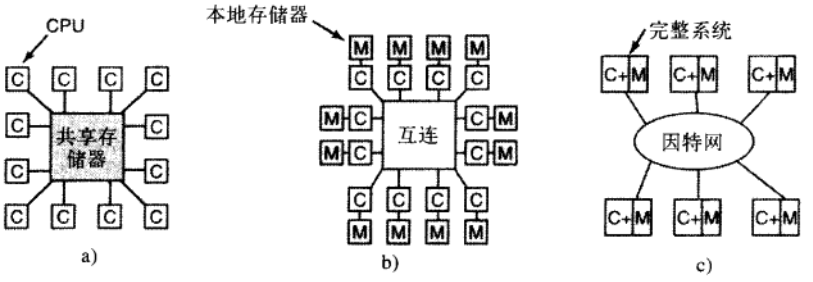


图8-1 a) 共享存储器多处理机；b) 消息传递多计算机；c) 广域分布式系统

第三种模型参见图8-1c，所有的计算机系统都通过一个广域网连接起来，如因特网，构成了一个*分布式系统*（distributed system）。每台计算机有自己的存储器，当然，通过消息传递进行系统通信。图8-1b和图8-1c之间真正惟一的差别是，后者使用了完整的计算机而且消息传递时间通常需要10~100ms。如此长的延迟造成使用这类*松散耦合*系统的方式和图8-1b中的*紧密耦合*系统不同。三种类型的系统在通信延迟上各不相同，分别有三个数量级的差别。类似于一天和三年的差别。

本章有四个主要部分，分别对应于图8-1中的三个模型再加上虚拟化技术（一种通过软件创造出更多虚拟CPU的方法）。在每一部分中，我们先简要地介绍相关的硬件。然后，讨论软件，特别是与这种系统类型有关的操作系统问题。我们会发现，每种情况都面临着不同的问题并且需要不同的解决方法。

8.1 多处理机

共享存储器多处理机（或以后简称为多处理机，multiprocessor）是这样一种计算机系统，其两个或更多的CPU全部共享访问一个公用的RAM。运行在任何一个CPU上的程序都看到一个普通（通常是分页）的虚拟地址空间。这个系统惟一特别的性质是，CPU可对存储器字写入某个值，然后读回该字，并得到一个不同的值（因为另一个CPU改写了它）。在进行恰当组织时，这种性质构成了处理器间通信的基础：一个CPU向存储器写入某些数据而另一个读取这些数据。

至于最重要的部分，多处理机操作系统只是通常的操作系统。它们处理系统调用，进行存储器管理，提供文件系统并管理I/O设备。不过，在某些领域里它们还是有一些独特的性质。这包括进程同步、资源管理以及调度。下面首先概要地介绍多处理机的硬件，然后进入有关操作系统的问题。

8.1.1 多处理机硬件

所有的多处理机都具有每个CPU可访问全部存储器的性质，而有些多处理机仍有一些其他的特性，即读出每个存储器字的速度是一样快的。这些机器称为UMA（Uniform Memory Access，*统一存储器访问*）多处理机。相反，NUMA（Nonuniform Memory Access，*非一致存储器访问*）多处理机就没有这种特性。至于为何有这种差别，稍后会加以说明。我们将首先考察UMA多处理机，然后讨论NUMA多处理机。

1. 基于总线的UMA多处理机体系结构

最简单的多处理机是基于单总线的，参见图8-2a。两个或更多的CPU以及一个或多个存储器模块都使用同一个总线进行通信。当一个CPU需要读一个存储器字（memory word）时，它首先检查总线忙否。如果总线空闲，该CPU把所需字的地址放到总线上，发出若干控制信号，然后等待存储器把所需的字放到总线上。

当某个CPU需要读写存储器时，如果总线忙，CPU只是等待，直到总线空闲。这种设计存在问题。在只有两三个CPU时，对总线的争夺还可以管理；若有32个或64个CPU时，就不可忍受了。这种系统完全受到总线带宽的限制，多数CPU在大部分时间里是空闲的。

这一问题的解决方案是为每个CPU添加一个高速缓存（cache），如图8-2b所示。这个高速缓存可以位于CPU芯片的内部、CPU附近、在处理器板上或所有这三种方式的组合。由于许多读操作可以从本地

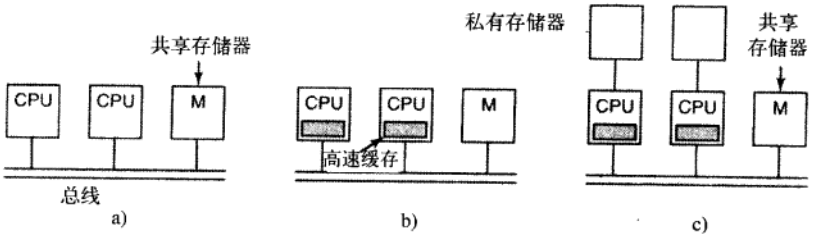


图8-2 三类基于总线的多处理机：a) 没有高速缓存；b) 有高速缓存；c) 有高速缓存与私有存储器

高速缓存上得到满足，总线流量就大大减少了，这样系统就能够支持更多的CPU。一般而言，高速缓存不以单个字为基础，而是以32字节或64字节块为基础。当引用一个字时，它所在的整个数据块（叫做一个cache行）被取到使用它的CPU的高速缓存当中。

每一个高速缓存块或者被标记为只读（在这种情况下，它可以同时存在于多个高速缓存中），或者标记为读写（在这种情况下，它不能在其他高速缓存中存在）。如果CPU试图在一个或多个远程高速缓存中写入一个字，总线硬件检测到写，并把一个信号放到总线上通知所有其他的高速缓存。如果其他高速缓存有个“干净”的副本，也就是同存储器内容完全一样的副本，那么它们可以丢弃该副本并让写者在修改之前从存储器取出高速缓存块。如果某些其他高速缓存有“脏”（被修改过）副本，它必须在处理写之前把数据写回存储器或者把它通过总线直接传送到写者上。高速缓存这一套规则被称为*高速缓存一致性协议*，它是诸多协议之一。

还有另一种可能性就是图8-2c中的设计，在这种设计中每个CPU不止有一个高速缓存，还有一个本地的私有存储器，它通过一条专门的（私有）总线访问。为了优化使用这一配置，编译器应该把所有程序的代码、字符串、常量以及其他只读数据、栈和局部变量放进私有存储器中。而共享存储器只用于可写的共享变量。在多数情况下，这种仔细的放置会极大地减少总线流量，但是这样做需要编译器的积极配合。

2. 使用交叉开关的UMA多处理机

即使有最好的高速缓存，单个总线的使用还是把UMA多处理机的数量限制在16至32个CPU。要超过这个数量，需要不同类型的互连网络。连接n个CPU到k个存储器的最简单的电路是*交叉开关*，参见图8-3。交叉开关在电话交换系统中已经采用了几十年，用于把一组进线以任意方式连接到一组出线上。

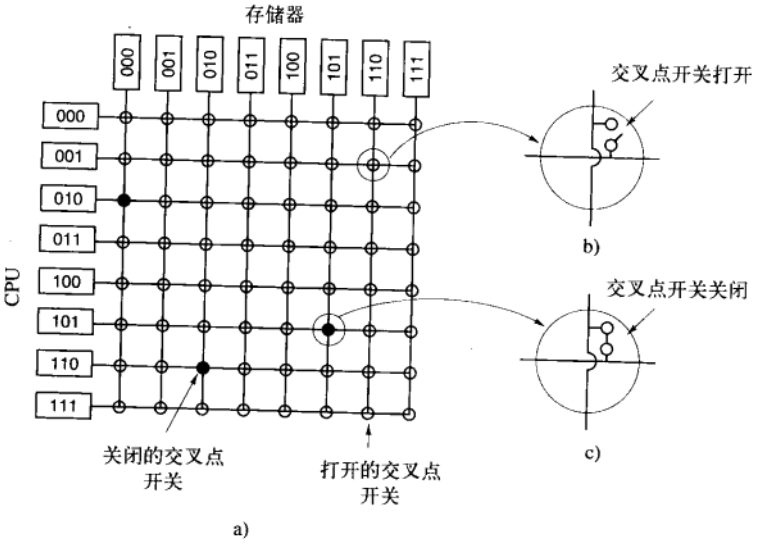


图8-3 a) 8×8交叉开关；b) 打开的交叉点；c) 闭合的交叉点

水平线（进线）和垂直线（出线）的每个相交位置上是一个交叉点（crosspoint）。交叉点是一个以电子方式开关的小开关，具体取决于水平线和垂直线是否需要连接。在图8-3a中我们看到有三个交叉点同时闭合，允许（CPU，存储器）对（010，000）、（101，101）和（110，010）同时连接。其他的连接也是可能的。事实上，组合的数量等于象棋盘上8个棋子安全放置方式的数量（8皇后问题）。

交叉开关最好的一个特性是它是一个非阻塞网络，即不会因有些交叉点或连线已经被占据了而拒绝连接（假设存储器模块自身是可用的）。而且并不需要预先的规划。即使已经设置了7个任意的连接，还有可能把剩余的CPU连接到剩余的存储器上。

当然，当两个CPU同时试图访问同一个模块的时候，对内存的争夺还是可能的。不过，通过将内存分为n个单元，与图8-2的模型相比，这样的争夺概率可以降至$1/n$。

交叉开关最差的一个特性是，交叉点的数量以n^2方式增长。若有1000个CPU和1000个存储器我们就需要一百万个交叉点。这样大数量的交叉开关是不可行的。不过，无论如何对于中等规模的系统而言，交叉开关的设计是可用的。

3. 使用多级交换的UMA 多处理机

有一种完全不同的、基于简单2×2开关的多处理机设计，参见图8-4a。这个开关有两个输入和两个输出。到达任意一个输入线的消息可以被交换至任意一个输出线上。就我们的目标而言，消息可由四个部分组成，参见图8-4b。Module（模块）域指明使用哪个存储器。Address（地址）域指定在模块中的地址。Opcode（操作码）给定了操作，如READ或WRITE。最后，在可选的Value（值）域中可包含一个操作数，比如一个要被WRITE写入的32位字。该开关检查Module域并利用它确定消息是应该送给X还是发送给Y。

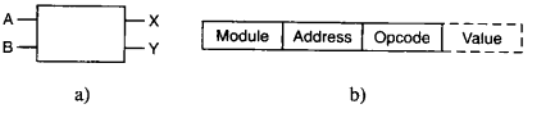


图8-4 a) 一个带有A和B两个输入线以及X和Y两个输出线的2×2的开关；b) 消息格式

这个2×2开关可有多种使用方式，用以构建大型的多级交换网络（Adams 等人，1987； Bhuyan 等人，1989；Kuman和Reddy，1987）。有一种是简单经济的Omega网络，见图8-5。这里采用了12个开关，把8个CPU连接到8个存储器上。推而广之，对于n个CPU和n个存储器，我们将需要$\log_2 n$级，每级$n/2$个开关，总数为$(n/2)\log_2 n$个开关，比n^2个交叉点要好得多，特别是当n值很大时。

Omega网络的接线模式常被称作全混洗（perfect shuffle），因为每一级信号的混合就像把一副牌分成两半，然后再把牌一张张混合起来。接着看看Omega网络是如何工作的，假设CPU 011打算从存储器模块110读取一个字。CPU发送READ消息给开关1D，它在Module域包含110。1D开关取110的首位（最左位）并用它进行路由处理。0路由到上端输出，而1的路由到下端，由于该位为1，所以消息通过低端输出被路由到2D。

所有的第二级开关，包括2D，取用第二个比特位进行路由。这一位还是1，所以消息通过低端输出转发到3D。在这里对第三位进行测试，结果发现是0。于是，消息送往上端输出，并达到所期望的存储器110。该消息的路径在图8-5中由字母a标出。

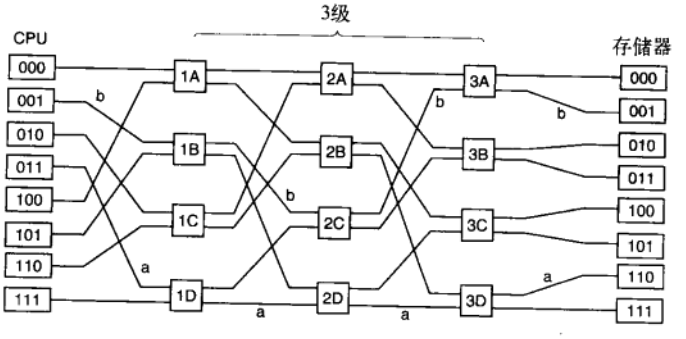


图8-5 Omega交换网络

在消息通过交换网络之后，模块号的左端的位就不再需要了。它们可以有很好的用途，可以用来记录入线编号，这样，应答消息可以找到返回路径。对于路径a，入线编号分别是0（向上输入到1D）、1（低输入到2D）和1（低输入到3D）。使用011作为应答路由，只要从右向左读出每位即可。

在上述这一切进行的同时，CPU 001需要往存储器001里写入一个字。这里发生的情况与上面的类似，消息分别通过上、上、下端输出路由，由字母b标出。当消息到达时，从Module域读出001，代表了对应的路径。由于这两个请求不使用任何相同的开关、连线或存储器模块，所以它们可以并行工作。

现在考虑如果CPU 000同时也请求访问存储器模块000会发生什么情况。这个请求会与CPU 001的请求在开关3A处发生冲突。它们中的一个就必须等待。和交叉开关不同，Omega网络是一种阻塞网络，并不是每组请求都可被同时处理。冲突可在一条连线或一个开关中发生，也可在对存储器的请求和来自存储器的应答中产生。

显然，很有必要在多个模块间均匀地分散对存储器的引用。一种常用的技术是把低位作为模块号。例如，考虑一台经常访问32位字的计算机中面向字节的地址空间，低位通常是00，但接下来的3位会均匀地分布。将这3位作为模块号，连续的字会放在连续的模块中。而连续字被放在不同模块里的存储器系统被称作交叉（interleaved）存储器系统。交叉存储器将并行运行的效率最大化了，这是因为多数对存储器的引用是连续编址的。设计非阻塞的交换网络也是有可能的，在这种网络中，提供了多条从每个CPU到每个存储器的路径，从而可以更好地分散流量。

4. NUMA多处理机

单总线UMA多处理机通常不超过几十个CPU，而交叉开关或交换网络多处理机需要许多（昂贵）的硬件，所以规模也不是那么大。要想超过100个CPU还必须做些让步。通常，一种让步就是所有的存储器模块都具有相同的访问时间。这种让步导致了前面所说的NUMA 多处理机的出现。像UMA一样，这种机器为所有的CPU提供了一个统一的地址空间，但与UMA 机器不同的是，访问本地存储器模块快于访问远程存储器模块。因此，在NUMA 机器上运行的所有UMA程序无须做任何改变，但在相同的时钟速率下其性能不如UMA 机器上的性能。

所有NUMA 机器都具有以下三种关键特性，它们使得NUMA机器与其他多处理机相区别：

1) 具有对所有CPU都可见的单个地址空间。

2) 通过LOAD和STORE指令访问远程存储器。

3) 访问远程存储器慢于访问本地存储器。

在对远程存储器的访问时间不被隐藏时（因为没有高速缓存），系统被称为NC- NUMA（No Cache NUMA，无高速缓存NUMA）。在有一致性高速缓存时，系统被称为CC- NUMA（Cache-Coherent NUMA，高速缓存一致NUMA）。

目前构造大型CC-NUMA多处理机最常见的方法是基于目录的多处理机（directory-based multiprocessor）。其基本思想是，维护一个数据库来记录高速缓存行的位置及其状态。当一个高速缓存行被引用时，就查询数据库找出高速缓存行的位置以及它是“干净”的还是“脏”（被修改过）的。由于每条访问存储器的指令都必须查询这个数据库，所以它必须配有极高速的专用硬件，从而可以在一个总线周期的几分之一内作出响应。

要使基于目录的多处理机的想法更具体，让我们考虑一个简单的（假想）例子，一个256个节点的系统，每个节点包括一个CPU和通过局部总线连接到CPU上的16MB的RAM。整个存储器有2^{32}字节，被划分成2^{26}个64字节大小的高速缓存行。存储器被静态地在节点间分配，节点0 是0~16M，节点1是16~32M，以此类推。节点通过互连网络连接，参见图8-6a。每个节点还有用于构成其2^{24}字节存储器的2^{18}个64字节高速缓存行的目录项。此刻，我们假定一行最多被一个高速缓存使用。

为了了解目录是如何工作的，让我们跟踪引用了一个高速缓存行的发自CPU 20的LOAD指令。首先，发出该指令的CPU把它交给自己的MMU，被翻译成物理地址，比如说，0x24000108。MMU将这个地址拆分为三个部分，如图8-6b所示。这三个部分按十进制是节点36、第4行和偏移量8。MMU看到引用的存储器字来自节点36，而不是节点20，所以它把请求消息通过互连网络发送到该高速缓存行的的主节点（home node）36上，询问行4是否被高速缓存，如果是，高速缓存在何处。

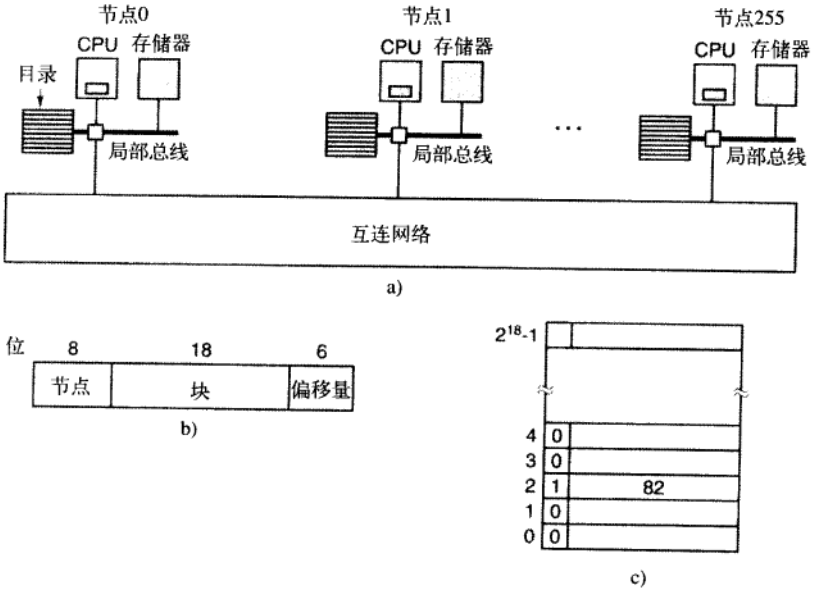


图8-6 a) 256个节点的基于目录的多处理机；b) 32位存储器地址划分的域；c) 节点36中的目录

当请求通过互连网络到达节点36时，它被路由至目录硬件。硬件检索其包含2^{18}个表项的目录表（其中的每个表项代表一个高速缓存行）并解析到项4。从图8-6c中，我们可以看到该行没有被高速缓存，所以硬件从本地RAM中取出第4行，送回给节点20，更新目录项4，指出该行目前被高速缓存在节点20处。

现在来考虑第二个请求，这次访问节点36的第2行。在图8-6c中，我们可以看到这一行在节点82处被高速缓存。此刻硬件可以更新目录项2，指出该行现在在节点20上，然后送一条消息给节点82，指示把该行传给节点20并且使其自身的高速缓存无效。注意，即使一个所谓“共享存储器多处理机”，在下层仍然有大量的消息传递。

让我们顺便计算一下有多少存储器单元被目录占用。每个节点有16 MB的RAM，并且有2^{18}个9位的目录项记录该RAM。这样目录上的开支大约是9×2^{18}位除以16 MB，即约1.76%，一般而言这是可接受的（尽管这些都是高速存储器，会增加成本）。即使对于32字节的高速缓存行，开销也只有4%。至于128字节的高速缓存行，它的开销不到1%。

该设计有一个明显的限制，即一行只能被一个节点高速缓存。要想允许一行能够在多个节点上被高速缓存，我们需要某种对所有行定位的方法，例如，在写操作时使其无效或更新。要允许同时在若干节点上进行高速缓存，有几种选择方案，不过对它们的讨论已超出了本书的范围。

5. 多核芯片

随着芯片制造技术的发展，晶体管的体积越来越小，从而有可能将越来越多的晶体管放入一个芯片中。这个基于经验的发现通常称为摩尔定律（Moore's Law），得名于首次发现该规律的Intel公司创始人之一Gordon Moore。Intel Core 2 Duo系列芯片已包含了3亿数量级的晶体管。

随之一个显而易见的问题是：“你怎么利用这些晶体管？”按照我们在第1.3.1小节的讨论，一个选择是给芯片添加数兆字节的高速缓存。这个选择是认真的，带有4兆字节片上高速缓存的芯片现在已经很常见，并且带有更多片上高速缓存的芯片也即将出现。但是到了某种程度，再增加高速缓存的大小只能将命中率从99%提高到99.5%，而这样的改进并不能显著提升应用的性能。

另一个选择是将两个或者多个完整的CPU，通常称为核（core），放到同一个芯片上（技术上来说是同一个小硅片）。双核和四核的芯片已经普及，八十核的芯片已经被制造出来，而带有上百个核的芯片也即将出现。

虽然CPU可能共享高速缓存或者不共享（如图1-8所示），但是它们都共享内存。考虑到每个内存字总是有惟一的值，这些内存是一致的。特殊的硬件电路可以确保在一个字同时出现在两个或者多个的高

速缓存中的情况下，当其中某个CPU修改了该字，所有其他高速缓存中的该字都会被自动地并且原子性地删除来确保一致性。这个过程称为窥探（snooping）。

这样设计的结果是多核芯片就相当于小的多处理机。实际上，多核芯片时常被称为片级多处理机（Chip-level MultiProcessors，CMP）。从软件的角度来看，CMP与基于总线的多处理机和使用交换网络的多处理机并没有太大的差别。不过，它们还是存在着若干的差别。例如，对基于总线的多处理机，每个CPU拥有自己的高速缓存，如图8-2b以及图1-8b的AMD设计所示。在图1-8a所示的Intel使用的共享高速缓存的设计并没有出现在其他的多处理机中。共享二级高速缓存会影响性能。如果一个核需要很多高速缓存空间，而另一个核不需要，这样的设计允许它们各自使用所需的高速缓存。但另一方面，共享高速缓存也让一个贪婪的核损害其他核的性能成为了可能。

CMP与其他更大的多处理机之间的另一个差异是容错。因为CPU之间的连接非常紧密，一个共享模块的失效可能导致许多CPU同时出错。而这样的情况在传统的多处理机中是很少出现的。

除了所有核都是对等的对称多核芯片之外，还有一类多核芯片被称为片上系统（system on a chip）。这些芯片含有一个或者多个主CPU，但是同时还包含若干个专用核，例如视频与音频解码器、加密芯片、网络接口等。这些核共同构成了完整的片上计算机系统。

正如过去已经发生的，硬件的发展常常领先于软件。多核的时代已经来临，但是我们还不具备为它们编写应用程序的能力。现有的编程语言并不适合编写高度并行的代码，同时适用的编译器和调试工具还很匮乏。几乎没有几个程序员有编写并行程序的经验，而大部分程序员对于如何将工作划分为若干可以并行执行的块（package）知之甚少。同步、消除竞争、避免死锁成为了程序员的噩梦，同时也影响到了性能。信号量（semaphore）并不能解决问题。除了这些问题，什么样的应用真的需要使用数百个核尚不明确。自然语言语音识别可能需要大量的计算能力，但这里的问题并不是缺少时钟周期，而是缺少可行的算法。简而言之，或许硬件开发人员正在发布软件开发人员不知道如何使用而用户也并不需要的产品。

8.1.2 多处理机操作系统类型

让我们从对多处理机硬件的讨论转到多处理机软件，特别是多处理机操作系统上来。这里有各种可能的方法。接下来将讨论其中的三种。需要强调的是所有这些方法除了适用于多核系统之外，同样适用于包含多个分离CPU的系统。

1. 每个CPU有自己的操作系统

组织一个多处理机操作系统的可能的最简单的方法是，静态地把存储器划分成和CPU一样多的各个部分，为每个CPU提供其私有存储器以及操作系统的各自私有副本。实际上n个CPU以n个独立计算机的形式运行。这样做一个明显的优点是，允许所有的CPU共享操作系统的代码，而且只需要提供数据的私有副本，如图8-7所示。

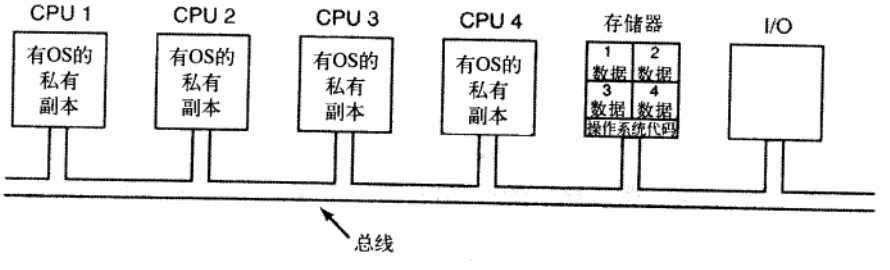


图8-7 在4个CPU中划分多处理机存储器，但共享一个操作系统代码的副本。标有“数据”字样的方框是每个CPU的操作系统私有数据

这一机制比有n个分离的计算机要好，因为它允许所有的机器共享一套磁盘及其他的I/O设备，它还允许灵活地共享存储器。例如，即便使用静态内存分配，一个CPU也可以获得极大的一块内存，从而高效地执行代码。另外，由于生产者能够直接把数据写入存储器，从而使得消费者从生产者写入的位置取出数据，因此进程之间可以高效地通信。况且，从操作系统的角度看，每个CPU都有自己的操作系统非常自然。

值得提及该设计看来不明显的四个方面。首先，在一个进程进行系统调用时，该系统调用是在本机的CPU上被捕获并处理的，并使用操作系统表中的数据结构。

其次，因为每个操作系统都有自己的表，那么它也有自己的进程集合，通过自身调度这些进程。这里没有进程共享。如果一个用户登录到CPU 1，那么他的所有进程都在CPU 1上运行。因此，在CPU 2有负载运行而CPU 1空载的情形是会发生的。

第三，没有页面共享。会出现如下的情形：在CPU2不断地进行页面调度时CPU 1却有多余的页面。由于内存分配是固定的，所以CPU 2无法向CPU 1借用页面。

第四，也是最坏的情形，如果操作系统维护近期使用过的磁盘块的缓冲区高速缓存，每个操作系统都独自进行这种维护工作，因此，可能出现某一修改过的磁盘块同时存在于多个缓冲区高速缓存的情况，这将会导致不一致性的结果。避免这一问题的惟一途径是，取消缓冲区高速缓存。这样做并不难，但是会显著降低性能。

由于这些原因，上述模型已很少使用，尽管在早期的多处理机中它一度被采用，那时的目标是把已有的操作系统尽可能快地移植到新的多处理机上。

2. 主从多处理机

图8-8中给出的是第二种模型。在这种模型中，操作系统的一个副本及其数据表都在CPU 1上，而不是在其他所有CPU上。为了在该CPU 1上进行处理，所有的系统调用都重定向到CPU 1上。如果有剩余的CPU时间，还可以在 CPU 1上运行用户进程。这种模型称为主从模型（master-slave），因为CPU 1是主CPU，而其他的都是从属CPU。

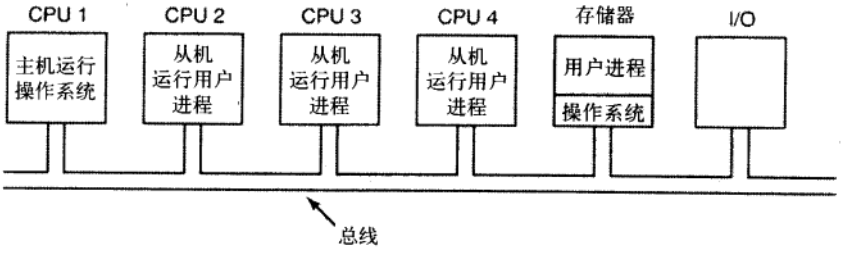


图8-8 主从多处理机模型

主从模型解决了在第一种模型中的多数问题。有单一的数据结构（如一个链表或者一组优先级链表）用来记录就绪进程。当某个CPU空闲下来时，它向CPU 1上的操作系统请求一个进程运行，并被分配一个进程。这样，就不会出现一个CPU空闲而另一个过载的情形。类似地，可在所有的进程中动态地分配页面，而且只有一个缓冲区高速缓存，所以决不会出现不一致的情形。

这个模型的问题是，如果有很多的CPU，主CPU 会变成一个瓶颈。毕竟，它要处理来自所有CPU的系统调用。如果全部时间的10%用来处理系统调用，那么10个CPU就会使主CPU饱和，而20个CPU就会使主CPU彻底过载。可见，这个模型虽然简单，而且对小型多处理机是可行的，但不能用于大型多处理机。

3. 对称多处理机

我们的第三种模型，即对称多处理机（Symmetric MultiProcessor，SMP），消除了上述的不对称性。在存储器中有操作系统的一个副本，但任何CPU都可以运行它。在有系统调用时，进行系统调用的CPU陷入内核并处理系统调用。图8-9是对SMP模式的说明。

这个模型动态地平衡进程和存储器，因为它只有一套操作系统数据表。它还消除了主CPU的瓶颈，因为不存在主CPU；但是这个模型也带来了自身的问题。特别是，当两个或更多的CPU同时运行操作系统代码时，就会出现灾难。想象有两个CPU同时选择相同的进程运行或请求同一个空闲存储器页面。处理这些问题的最简单方法是在操作系统中使用互斥信号量（锁），使整个系统成为一个大临界区。当一个CPU要运行操作系统时，它必须首先获得互斥信号量。如果互斥信号量被锁住，就得等待。按照这种方式，任何CPU都可以运行操作系统，但在任一时刻只有一个CPU可运行操作系统。

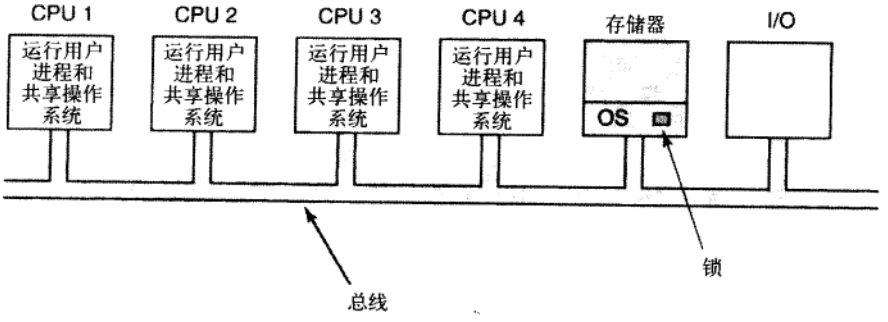


图8-9 SMP多处理机模型

这个模型是可以工作的，但是它几乎同主从模式一样糟糕。同样假设，如果所有时间的10%花费在操作系统内部。那么在有20个CPU 时，会出现等待进入的CPU长队。幸运的是，比较容易进行改进。操作系统中的很多部分是彼此独立的。例如，在一个CPU运行调度程序时，另一个CPU则处理文件系统的调用，而第三个在处理一个缺页异常，这种运行方式是没有问题的。

这一事实使得把操作系统分割成互不影响的临界区。每个临界区由其互斥信号量保护，所以一次只有一个CPU可执行它。采用这种方式，可以实现更多的并行操作。而某些表格，如进程表，可能恰巧被多个临界区使用。例如，在调度时需要进程表，在系统fork调用和信号处理时也都需要进程表。多临界区使用的每个表格，都需要有各自的互斥信号量。通过这种方式，可以做到每个临界区在任一个时刻只被一个CPU执行，而且在任一个时刻每个临界表（critical table）也只被一个CPU访问。

大多数的现代多处理机都采用这种安排。为这类机器编写操作系统的困难，不在于其实际的代码与普通的操作系统有多大的不同，而在于如何将其划分为可以由不同的CPU并行执行的临界区而互不干扰，即使以细小的、间接的方式。另外，对于被两个或多个临界区使用的表必须通过互斥信号量分别加以保护，而且使用这些表的代码必须正确地运用互斥信号量。

更进一步，必须格外小心地避免死锁。如果两个临界区都需要表A和表B，其中一个首先申请A，另一个首先申请B，那么迟早会发生死锁，而且没有人知道为什么会发生死锁。理论上，所有的表可以被赋予整数值，而且所有的临界区都应该以升序的方式获得表。这一策略避免了死锁，但是需要程序员非常仔细地考虑每个临界区需要哪个表，以便按照正确的次序安排请求。

由于代码是随着时间演化的，所以也许有个临界区需要一张过去不需要的新表。如果程序员是新接手工作的，他不了解系统的整个逻辑，那么可能只是在他需要的时候获得表，并且在不需要时释放掉。尽管这看起来是合理的，但是可能会导致死锁，即用户会觉察到系统被凝固住了。要做正确并不容易，而且要在程序员不断更换的数年时间之内始终保持正确性太困难了。

8.1.3 多处理机同步

在多处理机中CPU经常需要同步。这里刚刚看到了内核临界区和表被互斥信号量保护的情形。现在让我们仔细看看在多处理机中这种同步是如何工作的。正如我们将看到的，它远不是那么无足轻重。

开始讨论之前，还需要引入同步原语。如果一个进程在单处理机（仅含一个CPU）中需要访问一些内核临界表的系统调用，那么内核代码在接触该表之前可以先禁止中断。然后它继续工作，在相关工作完成之前，不会有任何其他的进程溜进来访问该表。在多处理机中，禁止中断的操作只影响到完成禁止中断操作的这个CPU，其他的CPU继续运行并且可以访问临界表。因此，必须采用一种合适的互斥信号量协议，而且所有的CPU都遵守该协议以保证互斥工作的进行。

任何实用的互斥信号量协议的核心都是一条特殊指令，该指令允许检测一个存储器字并以一种不可见的操作设置。我们来看看在图2-22中使用的指令TSL（Test and Set Lock）是如何实现临界区的。正如我们先前讨论的，这条指令做的是，读出一个存储器字并把它存储在一个寄存器中。同时，它对该存储器字写入一个1（或某些非零值）。当然，这需要两个总线周期来完成存储器的读写。在单处理机中，只要该指令不被中途中断，TSL指令就始终照常工作。

现在考虑在一个多处理机中发生的情况。在图8-10中我们看到了最坏情况的时序，其中存储器字1000，被用作一个初始化为0的锁。第1步，CPU 1读出该字得到一个0。第2步，在CPU 1有机会把该字写为1之前，CPU 2进入，并且也读出该字为0。第3步，CPU 1把1写入该字。第4步，CPU 2也把1写入该字。两个CPU都由TSL指令得到0，所以两者都对临界区进行访问，并且互斥失败。

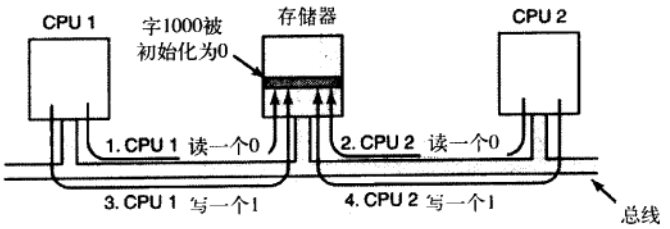


图8-10 如果不能锁住总线，TSL指令会失效。这里的四步解释了失效情况

为了阻止这种情况的发生，TSL指令必须首先锁住总线，阻止其他的CPU访问它，然后进行存储器的读写访问，再解锁总线。对总线加锁的典型做法是，先使用通常的总线协议请求总线，并申明（设置一个逻辑1）已拥有某些特定的总线线路，直到两个周期全部完成。只要始终保持拥有这一特定的总线线路，那么其他CPU就不会得到总线的访问权。这个指令只有在拥有必要的线路和和使用它们的（硬件）协议上才能实现。现代总线有这些功能，但是早期的一些总线不具备，它们不能正确地实现TSL指令。这就是Peterson 协议（完全用软件实现同步）会产生的原因（Peterson，1981）。

如果正确地实现和使用TSL，它能够保证互斥机制正常工作。但是这种互斥方法使用了自旋锁（spin lock），因为请求的CPU只是在原地尽可能快地对锁进行循环测试。这样做不仅完全浪费了提出请求的各个CPU的时间，而且还给总线或存储器增加了大量的负载，严重地降低了所有其他CPU从事正常工作的速度。

乍一看，高速缓存的实现也许能够消除总线竞争的问题，但事实并非如此。理论上，只要提出请求的CPU已经读取了锁字（lock word），它就可在其高速缓存中得到一个副本。只要没有其他CPU试图使用该锁，提出请求的CPU就能够用完其高速缓存。当拥有锁的CPU写入一个1到高速缓存并释放它时，高速缓存协议会自动地将它在远程高速缓存中的所有副本失效，要求再次读取正确的值。

问题是，高速缓存操作是在32或64字节的块中进行的。通常，拥有锁的CPU也需要这个锁周围的字。由于TSL指令是一个写指令（因为它修改了锁），所以它需要互斥地访问含有锁的高速缓存块。这样，每一个TSL都使锁持有者的高速缓存中的块失效，并且为请求的CPU取一个私有的、惟一的副本。只要锁拥有者访问到该锁的邻接字，该高速缓存块就被送进其机器。这样一来，整个包含锁的高速缓存块就会不断地在锁的拥有者和锁的请求者之间来回穿梭，导致了比单个读取一个锁字更大的总线流量。

如果能消除在请求一侧的所有由TSL引起的写操作，我们就可以明显地减少这种开销。使提出请求的CPU首先进行一个纯读操作来观察锁是否空闲，就可以实现这个目标。只有在锁看来是空闲时，TSL才真正去获取它。这种小小变化的结果是，大多数的行为变成读而不是写。如果拥有锁的CPU只是在同一个高速缓存块中读取各种变量，那么它们每个都可以以共享只读方式拥有一个高速缓存块的副本，这就消除了所有的高速缓存块传送。当锁最终被释放时，锁的所有者进行写操作，这需要排它访问，也就使远程高速缓存中的所有其他副本失效。在提出请求的CPU的下一个读请求中，高速缓存块会被重新装载。注意，如果两个或更多的CPU竞争同一个锁，那么有可能出现这样的情况，两者同时看到锁是空闲的，于是同时用TSL指令去获得它。只有其中的一个会成功，所以这里没有竞争条件，因为真正的获取是由TSL指令进行的，而且这条指令是原子性的。即使看到了锁空闲，然后立即用TSL指令试图获得它，也不能保证真正得到它。其他CPU可能会取胜，不过对于该算法的正确性来说，谁得到了锁并不重要。纯读出操作的成功只是意味着这可能是一个获得锁的好时机，但并不能确保能成功地得到锁。

另一个减少总线流量的方式是使用著名的以太网二进制指数补偿算法（binary exponential backoff algorithm）（Anderson，1990）。不是采用连续轮询，参考图2-22，而是把一个延迟循环插入轮询之间。初始的延迟是一条指令。如果锁仍然忙，延迟被加倍成为两条指令，然后，四条指令，如此这样进行，直到某个最大值。当锁释放时，较低的最大值会产生快速的响应。但是会浪费较多的总线周期在高速缓存的颠簸上。而较高的最大值可减少高速缓存的颠簸，但是其代价是不会注意到锁如此迅速地成为空闲。二进制指数补偿算法无论在有或无TSL指令前的纯读的情况下都适用。

一个更好的思想是，让每个打算获得互斥信号量的CPU都拥有各自用于测试的私有锁变量，如图8-11所示（Mellor-Crummey和 Scott，1991）。有关的变量应该存放在未使用的高速缓存块中以避免冲突。对这种算法的描述如下：给一个未能获得锁的CPU分配一个锁变量并且把它附在等待该锁的CPU链表的末端。在当前锁的持有者退出临界区时，它释放链表中的首个CPU正在测试的私有锁（在自己的高速缓存中）。然后该CPU进入临界区。操作完成之后，该CPU释放锁。其后继者接着使用，以此类推。尽管这个协议有些复杂（为了避免两个CPU同时把它们自己加在链表的末端），但它能够有效工作，而且消除了饥饿问题。具体细节，读者可以参考有关论文。

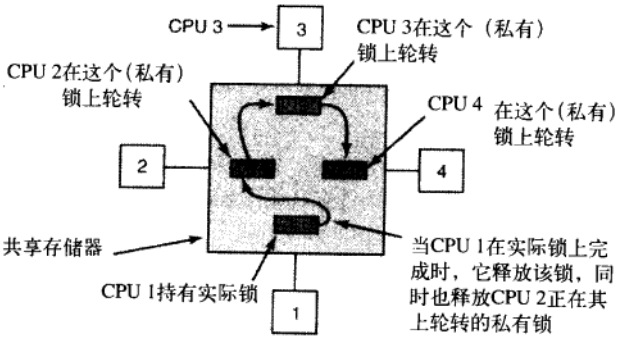


图8-11 使用多个锁以防止高速缓存颠簸

自旋与切换

到目前为止，不论是连续轮询方式、间歇轮询方式，还是把自己附在进行等候CPU链表中的方式，我们都假定需要加锁的互斥信号量的CPU只是保持等待。有时对于提出请求的CPU而言，只有等待，不存在其他替代的办法。例如，假设一些CPU是空闲的，需要访问共享的就绪链表（ready list）以便选择一个进程运行。如果就绪链表被锁住了，那么CPU就不能够只是决定暂停其正在进行的工作，而去运行另一个进程，因为这样做需要访问就绪链表。CPU必须保持等待直到能够访问该就绪链表。

然而，在另外一些情形中，却存在着别的选择。例如，如果在一个CPU中的某些线程需要访问文件系统缓冲区高速缓存，而该文件系统缓冲区高速缓存正好锁住了，那么CPU可以决定切换至另外一个线程而不是等待。有关是进行自旋还是进行线程切换的问题则是许多研究课题的内容，下面会讨论其中的一部分。请注意，这类问题在单处理机中是不存在的，因为没有另一个CPU释放锁，那么自旋就没有任何意义。如果一个线程试图取得锁并且失败，那么它总是被阻塞，这样锁的所有者有机会运行和释放该锁。

假设自旋和进行线程切换都是可行的选择，则可进行如下的权衡。自旋直接浪费了CPU周期。重复地测试锁并不是高效的工作。不过，切换也浪费了CPU周期，因为必须保存当前线程的状态，必须获得保护就绪链表的锁，还必须选择一个线程，必须装入其状态，并且使其开始运行。更进一步来说，该CPU高速缓存还将包含所有不合适的高速缓存块，因此在线程开始运行的时候会发生很多代价昂贵的高速缓存未命中。TLB的失效也是可能的。最后，会发生返回至原来线程的切换，随之而来的是更多的高速缓存未命中。花费在这两个线程间来回切换和所有高速缓存未命中的周期时间都浪费了。

如果预先知道互斥信号量通常被持有的时间，比如是50μs，而从当前线程切换需要1ms，稍后切换返回还需1ms，那么在互斥信号量上自旋则更为有效。另一方面，如果互斥信号量的平均保持时间是10ms，那就值得忍受线程切换的麻烦。问题在于，临界区在这个期间会发生相当大的变化，所以，哪一种方法更好些呢？

有一种设计是总是进行自旋。第二种设计方案则总是进行切换。而第三种设计方案是每当遇到一个锁住的互斥信号量时，就单独做出决定。在必须做出决定的时刻，并不知道自旋和切换哪一种方案更好，但是对于任何给定的系统，有可能对其所有的有关活动进行跟踪，并且随后进行离线分析。然后就可以确定哪个决定最好及在最好情形下所浪费的时间。这种事后算法（hindsight algorithm）成为对可行算法进行测量的基准评测标准。

已有研究人员对上述这一问题进行了研究（Karlin 等人，1989；Karlin等人，1991；Ousterhout，1982）。多数的研究工作使用了这样一个模型：一个未能获得互斥信号量的线程自旋一段时间。如果时间超过某个阈值，则进行切换。在某些情形下，该阈值是一个定值，典型值是切换至另一个线程再切换回来的开销。在另一些情形下，该阈值是动态变化的，它取决于所观察到的等待互斥信号量的历史信息。

在系统跟踪若干最新的自旋时间并且假定当前的情形可能会同先前的情形类似时，就可以得到最好