7.1 文件系统基本概念

文件系统是操作系统中最为可见的部分。它提供了在线存储和访问计算机操作系统和所有用户的程序与数据的机制。文件系统由两个不同部分组成：一组***文件***（文件用于存储相关数据）和一组***目录***（目录用于组织系统内的文件并提供有关文件的信息）结构。文件系统位于设备上。本章中，将考虑文件的许多方面和主要目录结构，讨论在多个进程、用户和计算机之间文件共享的语义；讨论各种文件保护方法，当多个用户访问文件和需要控制哪些用户按什么方式访问文件时，保护是很有必要的；讨论在最为常用的外存，即磁盘上，如何存储和访问文件的有关问题；讨论用各种方法来组织文件，分配磁盘空间等。

1. 文件概念

计算机能在多种不同媒介上（如磁盘、磁带和光盘）存储信息。为了方便地使用计算机系统，操作系统提供了信息存储的统一逻辑接口。操作系统对存储设备的各种属性加以抽象，从而定义了逻辑存储单元（文件），再将文件映射到物理设备上。这些物理设备通常为非易失性的，这样其内容在掉电和系统重启时也会一直保持。

文件是记录在外存上相关信息的具有名称的集合。从用户角度而言，文件是逻辑外存的最小分配单元，即数据除非在文件中，否则不能写到外存。通常，文件表示程序（源形式和目标形式）和数据。数据文件可以是数字、字符、字符数字或二进制。文件可以是自由形式，如文本文件，也可以是具有严格格式的。通常，文件由位、字节、行或记录组成，其具体意义是由文件创建者和使用者来定义的。因此，文件的概念极为广泛。

文件信息是由其创建者定义的。文件可存储许多不同类型的信息：源程序、目标程序、可执行程序、数字数据、文本、工资记录、图像、声音记录等。文件根据其类型具有一定结构。文本文件是由行（或页）组成，而行（或页）是由字符组成的。源文件由子程序和函数组成，而它们又是由声明和执行语句组成的。目标文件是一系列字节序列，它们按目标系统链接器所能理解的方式组成。可执行文件为一系列代码段，以供装入程序调入内存执行。

2. 文件属性

文件是有名称的，以方便用户通过名称对之加以引用。名称通常为字符串，如 example.c。有的系统区分名称中的大小写字符，而其他的则不加以区分。在文件被命名后，它就独立于进程、用户、甚至创建它的系统。例如，一个用户可能创建了文件 example.c，而另一用户就可通过此名称来编辑该文件。文件拥有者可能将文件写入到软盘、通过 email 发送、或通过网络拷贝，且在目的系统上它仍然被称为 example.c。

文件有一定的属性，这根据系统而有所不同，但是通常都包括如下属性：

- **名称**：文件符号名称是唯一的、按照人们容易读取的形式保存。
- **标识符**：标识文件系统内文件的唯一标签，通常为数字；这是对人而言不可读的文
- 件名称。
- **类型**：被支持不同类型的文件系统所使用。
- **位置**：该信息为指向设备和设备上文件位置的指针。
- **大小**：文件当前大小（以字节、字、或块来计），该属性也可包括文件可允许最大容量值。
- **保护**：决定谁能读、写、执行等的访问控制信息。
- **时间、日期和用户标识**：文件创建、上次修改和上次访问的相关信息。这些数据用于保护、安全和使用跟踪。

所有文件的信息都保存在目录结构中，而目录结构也保存在外存上。通常，目录条目包括文件名称及其唯一标识符。而标识符又定位文件其他属性信息。一个文件的这些信息可能需要1KB 多的空间来记录。在具有许多文件的系统中，目录本身大小可能有数 M 字节。因为目录如同文件一

样也必须是非易失性的，所以它们必须存放在设备上，并在需要时分若干次调入内存。

3. 文件操作

文件属于**抽象数据类型**。为了合适地定义文件，需要考虑有关文件的操作。操作系统提供系统调用对文件进行创建、写、读、定位、删除和截短。下面讨论操作系统要执行 6 个基本文件操作需要做哪些事，这样可以很容易地了解类似操作（如重命名文件）是如何实现的：

- **创建文件**：创建文件有两个必要步骤。第一，必须在文件系统中为文件找到空间。在下一章会讨论如何为文件分配空间。第二，在目录中为新文件创建一个条目。
- **写文件**：为了写文件，执行一个系统调用，其指明文件名称和要写入文件的内容。对于给定的文件名称，系统会搜索目录以查找该文件位置。系统必须为该文件维护一个写位置的指针。每当发生写操作时，必须更新写指针。
- **读文件**：为了读文件，使用一个系统调用，并指明文件名称和要读入文件块的内存位置。同样，需要搜索目录以找到相关目录项，系统要为该文件维护一个读位置的指针。每当发生读操作时，必须更新读指针。一个进程通常只对一个文件读或写，所以当前操作位置可作为每个进程**当前文件位置指针**。由于读和写操作都使用同一指针，节省了空间也降低系统复杂度。
- **在文件内重定位**：搜索目录相应条目，设置当前文件位置指针为给定值。重定位不需要真正 I/O 读写文件。该文件操作也称为**文件寻址**（seek）。
- **删除文件**：为了删除文件，在目录中搜索给定名称的文件。找到相关目录条目后，释放所有的文件空间以便其他文件使用，并删除相应目录条目。
- **截短文件**：用户可能只需要删除文件内容而保留其属性，而不是强制用户删除文件再创建文件。该函数允许所有文件属性都不变，而只是将其长度设为 0 并释放其空间。

这6个基本操作组成了所需文件操作的最小集合。其他常用操作包括向现有文件之后添加新信息和重命名现有文件。这些基本操作可以组合起来实现其他文件操作。例如，创建一个文件的复本，或复制文件到另一I/O设备，如打印机或显示器，可以这样来完成：创建一个新文件，从旧文件读入并写出到新文件。还希望有文件操作用于获取和设置文件的各种属性。例如，可能需要文件操作以允许用户确定文件属性，如文件长度；允许用户设置文件属性，如文件拥有者。

以上所述的绝大多数文件操作都涉及到为给定文件搜索相关目录条目。为了避免这种不断的搜索操作，许多系统要求在首次使用文件时，需要使用系统调用 open()。操作系统维护一个包含所有打开文件的信息表（**打开文件表，*open-file table***）。当需要一个文件操作时，可通过该表的一个索引指定文件，而不需要搜索。当文件不再使用时，进程可关闭它，操作系统从打开文件表中删除这一条目。系统调用create和delete操作的是关闭文件而不是打开的文件。

有的系统在首次引用文件时，会隐式地打开它。在打开文件的作业或程序终止时会自动关闭它。然而，绝大多数操作系统要求程序员在使用文件之前，显式地打开它。操作 *open()*会根据文件名搜索目录，并将目录条目复制到打开文件表。调用*open()*也可接受访问模式参数：创建、只读、读写、添加等。该模式可以根据文件许可位进行检查。如果请求模式获得允许，进程就可打开文件。系统调用*open()*通常返回一个指向打开文件表中一个条目的指针。通过使用该指针，而不是真实文件名称，进行所有I/O操作，以避免进一步搜索和简化系统调用接口。

在多进程可能同时打开同一文件（这有可能发生在多个不同应用程序同时打开同一文件的系统中）的环境中，open()和close()操作的实现更为复杂。通常，操作系统采用两级内部表：单个进程的表和整个系统的表。单个进程表跟踪单个进程打开的所有文件。表内所存是该进程所使用的文件的信息。例如，每个文件的当前文件指针就保存在这里，另外还包括文件访问权限和记帐信息。

单个进程表的每一条目相应地指向整个系统的打开文件表。整个系统表包含进程无关信息，如文件在磁盘上的位置，访问日期和文件大小。一旦一个进程打开一个文件，系统打开文件表就会在表中为打开文件增加相应的条目。当另一个进程执行调用 *open()*，其结果只不过简单地在其进程打开表中增加一个条目，并指向整个系统表的相应条目。通常，系统打开文件表的每个文件还有一个

文件打开计数器*open count*，以记录多少进程打开了该文件。每个*close()*会递减*count*，当打开计数器为0时，表示该文件不再被使用，该文件条目可从系统打开文件表中删除。

总而言之，每个打开文件有如下关联信息：

- **文件指针**：对于没有将文件偏移量作为系统调用*read()*和*write()*参数的系统，系统必须跟踪上次读写位置以作为当前文件位置指针。这种指针对打开文件的某个进程来说是唯一的，因此必须与磁盘文件属性分开保存。
- **文件打开计数**：当文件关闭时，操作系统必须重用其打开文件表条目，否则表内的空间会不够用。因为多个进程可能打开一个文件，所以系统在删除打开文件条目之前，必须等待最后一个进程关闭文件。文件打开计数跟踪打开和关闭的数量，在最后关闭时计数为0。这时，系统可删除该条目。
- **文件磁盘位置**：绝大多数操作要求系统修改文件数据。用于定位磁盘上文件位置的信息保存在内存中以避免为每个操作从磁盘中读取该信息。
- **访问权限**：每个进程用一个访问模式打开文件。这种信息保存在单个进程打开文件表中，以便操作系统能允许或拒绝以后的I/O请求。

有的操作系统提供接口以锁闭文件（或部分文件）。文件锁允许一个进程锁闭文件，以防止其他进程访问它。文件锁可用于由多个进程共享的文件，例如，由系统内多个进程访问与修改的系统日志文件。

可执行文件	exe, com, bin or none	读并执行机器语言程序
目标文件	obj, o	已编译，机器语言，未链接
源代码文件	c, cc, java, pas, asm, a	各种语言的源代码
批处理文件	bat, sh	发送给命令解释器的命令
文本文件	txt, doc	文本数据，文档
文字处理文件	wp, tex, rtf, doc	各种文字处理器格式
库文件	lib, a, so, dll	程序库例程，供编程语言使用
打印或视图文件	ps,pdf,jpg	用于打印或视图的ASCII或二进制格式的文件
档案文件	arc, zip, tar	相关的几个文件组成在一个文件中，通常为了归档或存储而经过压缩
多媒体文件	mpeg, mov, rm mp3,avi	包含音频或A/V信息的二进制文件

4. 文件类型

当设计文件系统（事实上包括整个操作系统）时，总是要考虑操作系统是否应该识别和支持文件类型。如果操作系统识别文件类型，那么它就能按合理方式对文件进行操作。例如，一个常见错误是用户试图打印一个二进制目标形式的程序。这种尝试通常会产生垃圾，但是如果操作系统已知文件是二进程目标程序，那么就能阻止它被打印。

实现文件类型的常用技术是在文件名称内包含类型。这样，用户和操作系统仅仅通过文件名称就能确定文件类型是什么。名称可分为两部分：名称和扩展名（图7.1）。例如，绝大多数操作系统允许用户将文件名命名为一组字符，加上圆点，再加上扩展部分。文件名例子如*resume.doc*、*Server.java*、*ReaderThread.c*。操作系统采用扩展来表示文件类型及其可用的文件操作类型。例如，只有具有扩展名为*.com*、*.exe*、*.bat*的文件才可执行。*.com*和*.exe*是两种形式的二进制可执行文件，而*.bat*文件则是ASCII字符形式的**批处理文件**（batch file），包含操作系统的命令。虽然MS-DOS只识别少量扩展，但是应用程序也可使用扩展名表示其所感兴趣的的文件类型。例如，汇编程序认为其源文件具有*.asm*扩展，*Microsoft Word*字处理器认为其文件具有*.doc*扩展。这些扩展名并不是必须的，所以用户可以不用扩展名（节省打字）来指明文件，应用程序会根据给定名称和其所期待的扩展名来查找文件。因为这些扩展不是由操作系统所支持的，所以它们只作为给操作它们的应用程序的提示。

图**7.1** 通常的文件类型.

另一个使用文件类型的例子来自TOPS-20操作系统。如果用户试图执行目标程序，且其源文件自从生成目标文件后已被修改或编辑，那么源文件就会自动被编译。这种功能确保用户总是运行最新的目标文件。否则，用户可能会浪费大量时间来执行旧的目标文件。为了实现这种功能，操作系统必须能区分目标文件和源文件，检查每个文件的创建或修改的时间，确定源程序的语言（以便使用正确的编译器）。

下面考虑Mac OS X操作系统。对这种系统，每个文件都有其类型，如表示文本文件的*TEXT*或表示应用程序的*APPL*。每个文件也有一个创建者属性，用来包含创建它的程序名称。这种属性是由操作系统在调用create()时设置的，因此系统支持并强制其使用。例如，由字处理器创建的文件会用字处理器名称作为其创建者。当用户用鼠标双击表示该文件的图标来打开文件时，就会自动调用字处理器，并将文件装入以便编辑。

UNIX 系统采用**幻数**（magic number）（保存在文件的开始部分）大致表明文件类型：可执行程序、批处理文件（**shell 脚本**），*postscript* 文件等。不是所有文件都具有幻数，所以系统特性不能只根据这种信息。UNIX 也不记录文件创建程序的名称。UNIX 确实允许文件名称扩展提示来帮助确定文件内容的类型，这些扩展可以被给定应用程序所使用或忽视，这是由应用程序开发者所决定的。

5. 文件结构

文件类型也可用于表示文件的内部结构。如7.4 所述，源文件和目标文件具有一定结构，以适应相应处理程序的要求。而且，有些文件必须符合操作系统所要求的结构。例如，操作系统可能要求可执行文件具有特定结构，以便它能确定将文件装入到哪里以及第一个指令的位置是什么。有的操作系统将这种思想扩展到系统支持的一组文件结构中，采用特殊操作处理具有这些结构的文件。例如，DEC 的 VMS 操作系统有一个可支持三种文件结构定义的文件系统。

以上讨论中的可支持多个文件结构的操作系统不可避免地出现了一个缺点：操作系统会变大。如果操作系统定义了 5 个不同文件结构，那么它需要包含代码以支持这些文件结构。另外，每个文件可能都需要被定义成操作系统所支持的某一种类型。如果新应用程序要求按操作系统所不支持的结构来组织信息，那么就会出问题。

例如，假设一个系统支持两种文件类型：文本文件（由回车和换行所分开的 ASCII 字符组成）和可执行二进制文件。现在，如果（作为用户）想要定义加密文件以保护内容不被未经授权的用户读取，那么会发现两种文件类型均不合适。加密文件不是 ASCII 文本行，而是随机位组合。虽然加密文件看起来是二进制文件，但是它不是可执行的。因此，要么必须绕过或错误地使用操作系统文件类型机制，要么放弃的加密方案。

有的操作系统强加（和支持）了最少数量的文件结构。这种方法为 UNIX、MS-DOS 和其他操作系统所采用。UNIX 认为每个文件是由 8 位字节序列所组成的；操作系统并不解释这些位。这种方案提供最大程度的灵活性，但是什么也不支持。每个应用程序必须有自己的代码对输入文件进行合适的解释。当然，所有操作系统必须至少支持一种结构，即可执行文件结构，以便能装入和运行程序。

Macintosh 操作系统也支持最少数量的文件结构。它要求每个文件包括两个部分：**资源叉（source fork）**和**数据叉（data fork）**。资源叉包括用户所感兴趣的信息。例如，它包含程序显示按钮的标签。外国用户可按照自己使用的语言来重新标识这些按钮，Macintosh 操作系统提供工具以允许对资源叉中的数据进行修改。数据叉包括程序代码或数据，即传统的文件内容。为了在 UNIX 或 MS-DOS 上能实现同样任务，程序员必须要修改和重新编译源程序，除非他创建了用户可修改的数据文件。显然，操作系统支持常用结构是有用的，这能节省程序员大量的劳动。太少结构会使编程不够灵活，然而太多结构会使操作系统过大且会使程序员混淆。

6. 内部文件结构

从内部而言，定位文件偏移量对操作系统来说可能是比较复杂的。磁盘系统通常具有明确定义的块大小，这是由扇区大小决定的。所有磁盘 I/O 是按块（物理记录）来执行的，且所有块都是同样大小。物理记录大小不太可能刚好与所需逻辑记录大小一样长，而且逻辑记录的长度是可变的。对这个问题的常用解决方法是先将若干个逻辑记录**打包**，再放入物理记录。

例如，UNIX 操作系统定义所有文件为字节流。每个字节可以从它到文件首（或尾）的偏移量来访问。在这种情况下，逻辑记录大小为 1 个字节。文件系统通常会自动将字节打包以存入物理磁盘块，或从磁盘块中解包得到字节（如按需要可能为每块 512 字节）。

逻辑记录大小、物理块大小和打包技术决定了多少逻辑记录可保存在每个物理块中。打包可由用户应用程序或操作系统来执行。

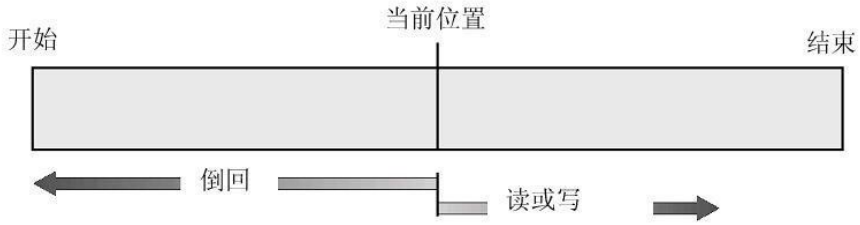
不管如何，文件都可当作一系列块的组合，所有基本 I/O 操作都是按块来进行的。逻辑记录与物理块之间的转换相对来说是个简单的软件问题。

由于磁盘空间总是按块来分配的，所以文件的最后一块的部分空间通常会有浪费。如果每块为 512 字节，一个大小为 1949 的文件会分配到 4 块（2048 字节）；最后 99 字节就浪费了。

按块分配所浪费的字节称为**内部碎片**，块越大，内部碎片也越大。

7. 访问方法

文件用来存储信息。当使用时，必须访问和将这些信息读入到计算机内存。文件信息可按多种方式来进行访问。有的系统只提供了一种文件访问方式。其他系统，如 IBM，支持多种访问方式，因此为特定应用选择合适的访问方式是个重要的设计问题。



（1）顺序访问

最为简单的访问方式是顺序访问。文件信息按顺序，一个记录接着一个记录地加以处理。这种访问模式最为常用，例如，编辑器和编译器通常按这种方式访问文件。

大量的文件操作是读和写。读操作读取下一文件部分，并自动前移文件指针，以跟踪 I/O位置。类似地，写操作会向文件尾部增加内容，相应的文件指针移到新增数据之后（新文件结尾）。文件也可重新设置到开始位置，有的系统允许向前或向后跳过 n 个（这里 n 为整数，有时只能为1）记录，。顺序访问如图 10.3 所示。顺序访问基于文件的磁带模型，不但适用于顺序访问设备，也适用于随机访问设备。

图 **7.2** 顺序访问文件

（2）直接访问

另一方式是直接访问（或相对访问）。文件由固定长度的逻辑记录组成，以允许程序按任意顺序进行快速读和写。直接访问方式是基于文件的磁盘模型，这是因为磁盘允许对任意文件块进行随机读和写。对直接访问，文件可作为块或记录的编号序列。因此，可先读取块14，再读块53，最后再写块7。对于直接访问文件，读写顺序是没有限制的。

直接访问文件可立即访问大量信息，所以极为有用。数据库通常使用这种类型的文件。当有关特定主题的查询到达时，计算哪块包含答案，并直接读取相应块来提供所需信息。

举一个简单的例子，在一个航班订票系统上，可以将所有特定航班（如航班 713）的信息，保存在由航班号码所标识的块上。因此，航班713的空位数量保存在订票文件的块713上。为了存储关于某个更大集合如人的信息，可以根据人名计算出一个 hash 函数，或者搜索位于内存中的索引以确定需要读和搜索的块。

对于直接访问方式，文件操作必须经过修改从而能将块号作为其参数。因此，有读 n 的操作，其中 n 是块号，而不是读下一个；有写 n 的操作，而不是写下一个。另外一种方法是像顺序访问一样，保留读下一个和写下一个，但是增加定位文件到 n，其中 n 是块号。这样，要实现读 n，只要定位到 n 并再执行读下一个。

由用户向操作系统所提供的块号通常为相对块号。相对块号是相对于文件开始的索引。因此，文件的第一块的号码是0，下一块为 1，依次类推，而第一块的真正绝对磁盘地址可能为 14703，

顺序存取	直接存取的实现
reset	*cp = 0;*
read next	*read cp;* *cp = cp+1;*
write next	*write cp;* *cp = cp+1;*

下一块为3192等。使用相对块号允许操作系统决定该文件放在哪里（称为分配问题，将在下一章中讨论），以阻止用户访问不属于其文件的其他的文件系统部分。有的系统的相对块号从0开始，其他的从1开始。

那么系统如何满足对某个文件的第N个记录的请求呢？设逻辑记录长度为L，记录N的请求可转换为从文件位置$L \times N$开始的L字节的请求（设第一记录为$N=0$）。由于逻辑记录为固定大小，所以它也容易读、写和删除记录。

不是所有操作系统都支持文件的顺序和直接访问。有的系统只允许顺序文件访问，也有的只允许直接访问。有的系统要求在创建文件时指定文件究竟是顺序访问还是直接访问，这样的文件只能按照所声明的方式进行访问。然而，对直接访问文件，可容易地模拟顺序访问。可简单地定义一个变量*cp*以表示当前位置，以按照图7.3所示的方式模拟顺序文件操作。另一方面，在顺序访问文件上，模拟直接访问是极为低效和笨重的。

图**7.3** 在直接访问文件上模拟顺序访问.

（3）其他访问方式

其他访问方式可建立在直接访问方式之上。这些访问通常涉及创建文件索引。**索引**，如同本书最后的索引，包括各块的指针。为了查找文件中的记录，首先搜索索引，再根据指针直接访问文件，以查找所需要的记录。

例如，一个零售价格文件可能列出每个产品的通用商品编码（universal product codes, UPC）及价格。每个记录包括10位数UPC和6位数价格，所以每个记录为16字节。如果每个磁盘块有1024字节，那么每块存储64个记录。一个具有120000个记录的文件可能占有2000块（2MB字节）。通过将文件按UPC排序，可将索引定义为包括每块的第一个UPC。该索引有2000个条目，每个条目为10个数字，共计20000字节，因此可保存在内存中。为查找某一产品的价格，可以构建一个索引的对分搜索（binary search）。通过这个搜索，可以精确地知道哪个块包括所要的记录并访问该块。这种结构允许只通过少量I/O就能搜索大文件。

对于大文件，索引本身可能太大以致于不能保存在内存中。解决方法之一是为索引文件再创建索引。初级索引文件包括二级索引文件的指针，而二级索引再包括真正指向数据项的指针。

例如，IBM的ISAM就使用小的主索引文件以指向二级索引的磁盘块，二级索引块再指向实际文件块。该文件按定义的键排序。当查找一特定项时，首先对主索引进行对分搜索，以得到二级索引的块号。读入该块，再通过对分搜索以查找包括所要记录的块。最后，按顺序搜索该块。这样，通过键入最多不超过两次直接访问就可定位记录。图7.4显示了一个类似情况，这是由VMS索引和相对文件所实现的。

图 7.4 索引文件和相关文件的例子

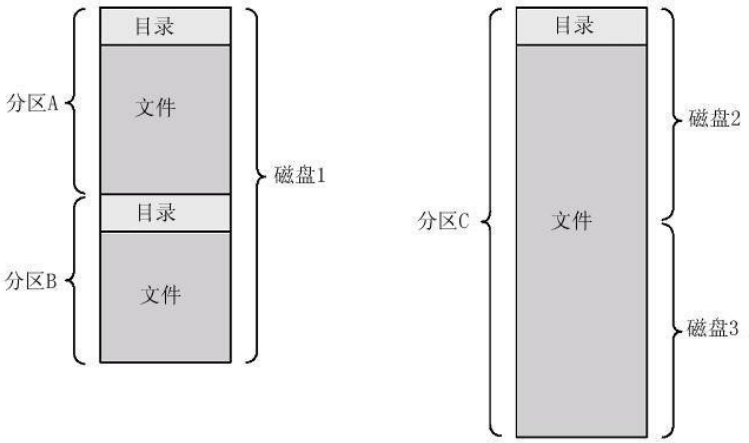
8. 目录结构

到目前，一直在讨论“文件系统”。实际上，系统可有若干文件系统，且文件系统可有不同类型。例如，一个典型的Solaris系统可有数个UFS文件系统，一个VFS文件系统，及一些NFS文件系统。文件系统实现的细节将在下一章讨论。

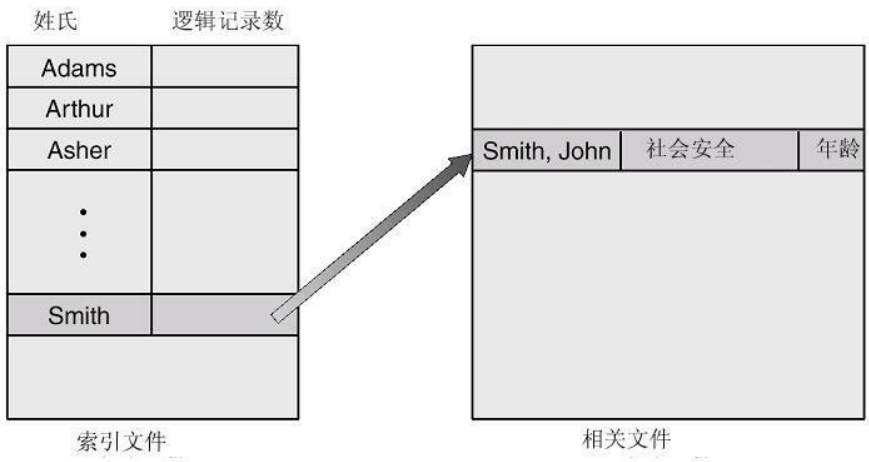
计算机的文件系统可以非常大。有的系统在数 T 磁盘上保存了数以百万计的文件。为了管理所有这些数据，需要组织它们。这种管理涉及到使用目录。本节将探讨目录结构。首先，需要解释存储结构的基本特点。

（1）存储结构

磁盘（或任何足够大的存储设备）可以整体地用于一个文件系统。但是，有时需要在一个磁盘上放多种文件系统，或一部分用于文件系统而另一部分用于其他如交换空间或非格式化的磁盘空间。这些部分称为分区，或片，或称为小型磁盘（IBM的说法）。每个磁盘分区可以创建一个文件系统。如下一章所述，这些部分可以组合成更大的可称为卷（volume）的结构，也可以在其上



创建文件系统。现在，为简单起见，可以将存储文件系统的一大块存储空间作为卷。卷可以存放多



个操作系统，使系统启动和运行多个OS。

包含文件系统的每个卷还必须包含系统上文件的信息。这些信息保存在设备目录或卷表中。设备目录（常称为目录）记录卷上所有文件的信息如名称、位置、大小和类型等。图7.5显示了一个典型文件系统的结构。

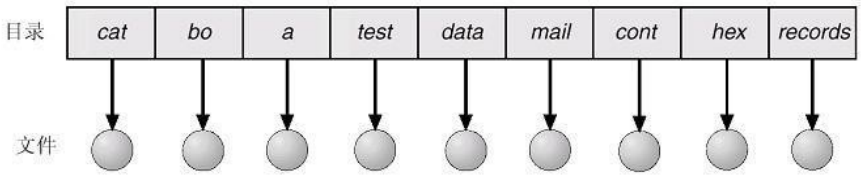
图7.5 典型的文件系统组成

（2）目录概述

目录可看作符号表，它能将文件名称转换成目录条目。如果采用这种观点，那么目录可按许多方式来加以组织。对目录，需要能够插入条目、删除条目、搜索给定条目、列出所有目录条目。这里，讨论用于定义目录系统的逻辑结构的若干方案。

在考虑特定目录结构时，需要记住目录相关操作：

- 搜索文件：需要能够搜索目录结构以查找特定文件的条目。因为文件具有符号名称，且类似的名称可能表示文件之间的关系，所以可能要能查找文件名和某个模式相匹配的所有文件。
- 创建文件：可以创建新文件并增加到目录中。



- 删除文件：当不再需要文件时，可以从目录中删除它。
- 遍历目录：需要能遍历目录内的文件，及其目录中每个文件条目的内容。
- 重命名文件：因为文件名称可向用户表示其内容，当文件内容和用途改变时名称必须改变。重新命名文件也允许改变该文件在目录结构中的位置。
- 跟踪文件系统：可能希望访问每个目录和每个目录的每个文件。为了可靠，定期备份整个文件系统的内容和结构是个不错的方法。这种备份通常将所有文件复制到磁带上。这种技术提供了备份拷贝以防止系统出错。除此之外，当文件不再使用时候，这个文件被复制到磁带上，该文件的原来占用磁盘空间可以释放以供其他文件所用。

下面，将讨论定义目录逻辑结构的常用方案。

（3）单层结构目录

最简单的目录结构是单层结构目录。所有文件都包含在同一目录中，其特点是便于理解和支持。（图7.6）。

图7.6单层结构目录

然而，在文件类型增加时或系统有多个用户时，单层结构目录有严重限制。由于所有文件位于同一目录，它们必须具有唯一名称。如果两个用户称其数据文件为*test*，那么就违背了唯一名称规则。例如，在一个编程班级中，有 23 个学生将其第 2 作业称为 *prog*2，而另11 位则称其为 *assign*2。虽然文件名称通常经过选择以反映文件内容，但是它们的长度通常有限制。MS-DOS 操作系统只允许11个字符的文件名，UNIX 允许255个字符。

随着文件数量增加，单层结构目录的单个用户会发现难以记住所有文件的名称。通常，一个用户在一个计算机系统上有数百个文件，在另外一个系统上也有同样数量的其他文件。在这种环境上，记住如此之多的文件是令人畏缩的。

（4）双层结构目录

单层结构目录通常会在不同用户之间引起文件名称的混淆。标准解决方法是为每个用户创建独立目录。

对于双层目录结构，每个用户都有自己的**用户文件目录（UFD，User File Directory）**。每个 UFD 都有相似的结构，但只列出了单个用户的文件。当一个用户作业开始执行或一个用户注册时，就搜索系统的**主文件目录（MFD，Master File Directory）**。通过用户名或帐号可索引MFD，每个条目指向用户的UFD（图7.7）。

图7.7 双层目录结构

当一个用户引用特定文件时，只需搜索他自己的UFD。因此，不同用户可拥有具有相同名称的文件，只要每个 UFD 内的所有文件名称唯一即可。当一个用户创建文件时，操作系统只搜索该用户的 UFD 以确定具有同样名称的文件是否存在。当删除文件时，操作系统只在局部 UFD 中对其进行搜索；因此，它并不会删除另一用户的具有相同名称的文件。

用户目录本身必须根据需要加以创建和删除。这可通过运行一个特别的系统程序进行，再加上合适用户名称和帐号信息。该程序创建新的 UFD，并在 MFD 中为之增加一项。该程序可能只能由系统管理员执行。用户目录的磁盘空间分配可以采用下一章所讨论的技术来处理。

虽然两层结构目录解决了名称冲突问题，但是它仍有缺点。这种结构有效地对用户加以隔离。这种隔离在用户需要完全独立时是优点，但是在用户需要在某个任务上进行合作和访问其他文件时却是个缺点。有的系统简单地不允许本地用户文件被其他用户所访问。

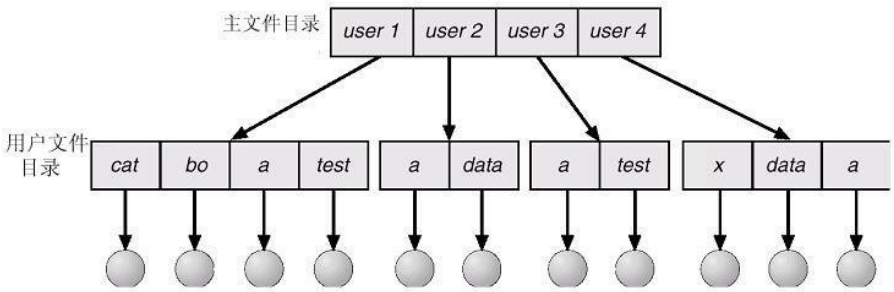
如果允许访问，那么一个用户必须能够指定另一用户目录内的文件。为了唯一指定位于两层目录内的特定文件，必须给出用户名和文件名。两层结构目录可作为高度为 2 的树或倒置树。树根为 MFD，其直接后代为 UFD。UFD 的后代为文件本身，文件为树的叶。在树中指定用户名和文件名就定义了从根（MFD）到叶（指定文件）的路径。因此，用户名和文件名定义了路径名。系统内的每个文件都有路径名。为了唯一指定文件，用户必须知道所要访问的文件的路径名。

（5）树型结构目录

一旦理解了如何将两层结构目录作为两层树看待，那么将目录结构扩展为任意高度的树就显得自然了（图7.8）。这种推广允许用户创建自己的子目录，相应地组织文件。事实上，树是最为常用的目录结构。树有根目录，系统内的每个文件都有唯一路径名。

图 **7.8** 树型目录结构

目录（或子目录）包括一组文件和子目录。一个目录只不过是一个需要按照特定方式访问的文件。所有目录具有同样的内部格式。每个目录条目都用一位来定义其为文件(0)或子目录(1)。创建



和删除目录条目需要使用特定的系统调用。

在通常情况下，每个进程都有一个当前目录。**当前目录**包括进程当前感兴趣的绝大多数文件。当需要引用一个文件时，就搜索当前目录。如果所需文件不在当前目录中，那么用户必须指定路径名或将当前目录改变为包括所需文件的目录。为了改变目录，用户可使用系统调用以重新定义

当前目录，该系统调用需要有一个目录名作为参数。这样，用户需要时就可以改变当前目录。从当前改变直到下次改变，系统调用 open 搜索当前目录以打开所指定的文件。注意搜索目录可能有、也可能没有一个特殊条目表示当前目录。

用户的初始当前目录是在用户进程开始时或用户登录时指定的。操作系统搜索账户文件（或其他预先定义的位置）以得到该用户的相关条目（以便于记帐）。账户文件有用户初始目录的指针（或名称）。该指针可复制到局部变量以指定用户初始的当前目录。从该 shell 开始，其他进程也纷纷启动。子进程的当前目录通常就是创建子进程时的父进程的当前目录。

路径名有两种形式：**绝对路径名**或**相对路径名**。**绝对路径名**从根开始并给出路径上的目录名直到所指定的文件。**相对路径名**从当前目录开始定义的一个路径。例如在图 10.9 所示树型结构文件系统中，如果当前目录是 root/spell/mail，那么相对路径名 prt/first 与绝对路径名 root/spell/mail/prt/frst 指向同一文件。

允许用户定义自己的子目录结构可以使其能按一定结构组织文件。这种结构可以是按不同主题来组织文件（例如，可创建一个目录以包括本书的内容），或按不同信息类型来组织文件（例如，目录 programs 可以包含源程序；而目录 bin 可存储所有二进制文件）。

对于树型结构目录，一个有趣的策略决定是如何处理删除目录。如果目录为空，那么可简单地加以删除。然而，假如要删除的目录不为空，而是包括多个文件或子目录，那么可有两个选择。有的系统，如 MS-DOS，如果目录不为空就不能删除。因此，为了删除一个目录，用户必须首先删除其中的内容。如果有子目录存在，那么必须先删除其子目录的内容。这种方案可能会导致大量的工作。另一种方法，如 UNIX 的 rm 命令，提供了选择：当需要删除一个目录时，所有该目录的文件和子目录也可删除。这两种方法的实现均不难，这是策略的问题。后一种策略更方便，但是更危险，因为用一个命令可以删除整个目录结构。如果错误地使用了这个命令，那么就必须从备份磁带中恢复大量文件和目录（假设存在备份）。

（6）无环图目录

考虑一下两个在进行一个合作项目的程序员。与该项目相关的文件可保存在一个子目录中，以区分两程序员的其他项目和文件。但是，由于两程序员都负责该项目，所以都希望该子目录在他自己的目录内。这种共同子目录应该共享。共享目录或文件可同时位于文件系统的两（或多）处。

树型结构禁止共享文件和目录。**无环图**（acyclic graph）允许目录含有共享子目录和文件（图 7.9）。同一文件或子目录可出现在两个不同目录中。无环图是树型结构目录方案的扩展。

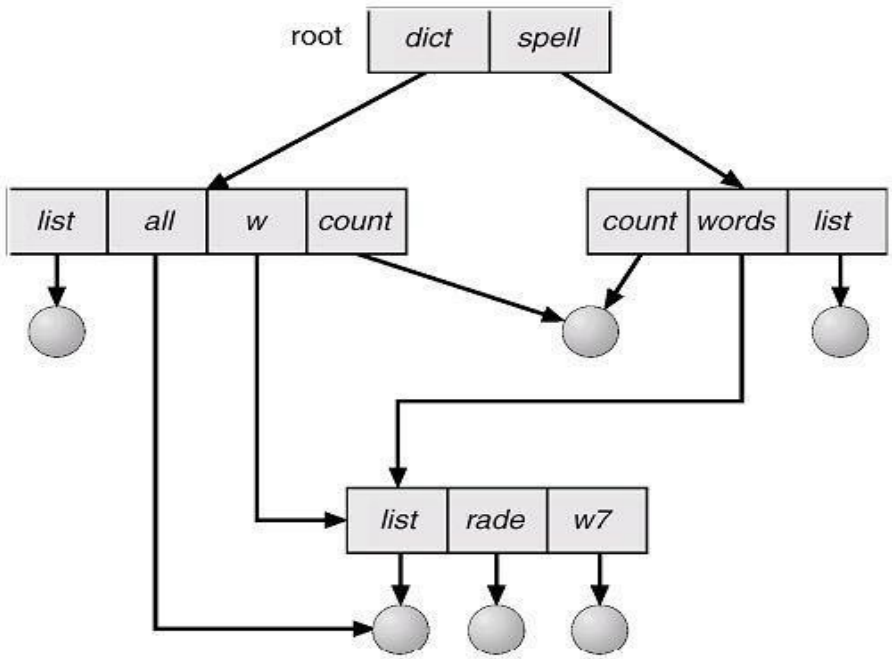


图 **7.9** 无环图目录结构

请注意，共享文件（或目录）不同于文件拷贝。如果有两个拷贝，每个程序员看到的是拷贝而不是原件；如果一个程序员改变了文件，那么这一改变不会出现在另一程序员的拷贝中。对于共享文件，只存在一个真正文件，所以任何改变都会为其他用户所见。共享对于子目录尤其重要，由一个用户创建的文件可自动出现在所有共享目录。

当人们作为一个组工作时，所共享的文件可放在一个目录中。所有组员的 UFD 可以将该共享文件目录作为其子目录。即使对于单个用户，也可能会要求有的文件出现在不同子目录中。例如，某个特定项目的程序不但可位于所有程序目录中，也可位于该项目的目录中。

实现共享文件和目录有许多方法。一个为许多 UNIX 系统所采用的常用方法是创建一个称为链接的新目录条目。**链接**实际上是另一文件或目录的指针。例如，链接可用绝对路径或相对路径名称来实现。当需要访问一个文件时，就可搜索目录。如果目录条目标记为链接，那么就可获得真正文件（或目录）的名称。链接可以通过使用路径名定位真正的文件来获得**解析**。链接可通过目录条目格式（或通过特殊类型）而容易地加以标识，它实际上是具有名称的间接指针。在遍历目录树时，操作系统忽略这些链接以维护系统的无环结构。

实现共享文件的另一个常用方法是简单地在共享目录中重复所有（被）共享文件的信息，因此两个目录条目完全相同。链接显然不同于原来的目录条目，因此，两者是不相同的。然而，重复目录条目会使原来的文件和复制品无法区分。复制目录条目的存在的主要问题是在修改文件时要维护一致性。

无环结构目录比树型结构目录更加灵活，但是也更为复杂。有些问题必须加以仔细考虑。现在一个文件可有多个绝对路径名。这样，不同文件名可能表示同一文件。这类似于程序设计语言的别名问题。如果试图遍历整个文件系统，如查找文件，计算所有文件的统计数据，复制所有文件到备份存储，那么这个问题就重要了，这是因为人们不希望多次重复地遍历共享目录。

另一问题是删除。分配给共享文件的空间什么时候可删除和重新使用？一种可能是每当用户删除文件时就删除，但是这样会留下悬挂指针指向不再存在的文件。更为糟糕的是，这些剩余文

件指针可能包括实际磁盘地址，而该空间可能又被其他文件使用，这样悬挂指针就可能指向其他文件的中间部分。

对于采用符号链接实现共享的系统，这种情况较为容易处理。删除链接并不需要影响原文件，而只是链接被删除。如果文件条目本身被删除，那么文件空间就释放，并使链接指针无效。可以搜索这些链接并删除它们，但是除非每个文件都保留相关链接列表，否则这种搜索可能会费时。或者，可以暂时不管这些指针，直到试图使用它们为止。到时，可以确定由链接所给定名称的文件不再存在，从而不能解析链接名称，将这种访问与其他非法文件名一样处理（在这种情况下，系统设计人员需要仔细考虑如下问题：当一个删除文件，而在使用其符号链接之前，创建了另一个具有同样名称的文件）。对于 UNIX 系统，当删除文件时，其符号链接并不删除，需要由用户认识到原来文件已被删除或替换。微软公司的 Windows（所有版本）也使用同样方法。

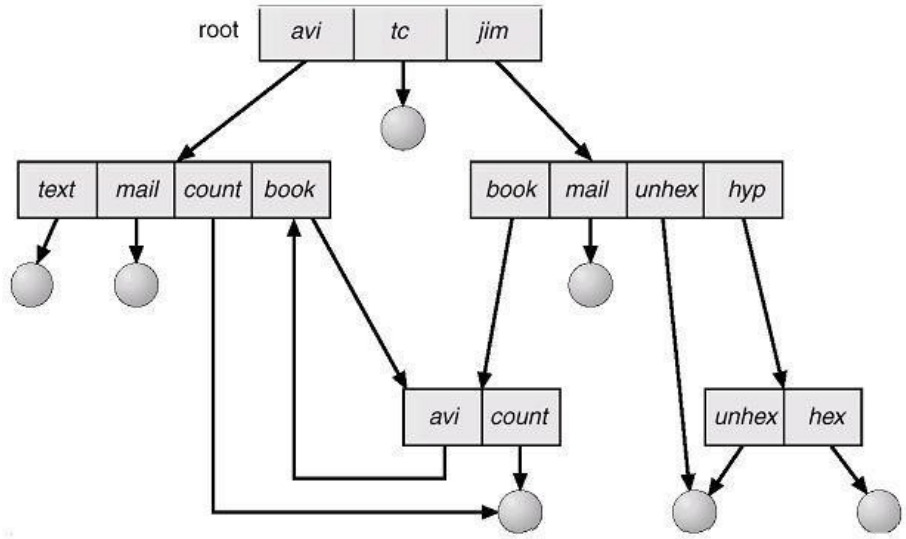
删除的另一方法是保留文件直到删除其所有引用为止。为了实现这种方法，必须有一种机制来确定最后文件引用已删除。可以为每个文件保留一个引用列表（目录条目或符号链接）。在建立一个目录条目的链接或复制时，需要将新条目增加到文件引用列表。当删除链接或目录条目时，删除列表上的相应条目。当其文件引用列表为空时，就删除文件本身。

这种方法的麻烦是可能会出现可变的、并可能很大的文件引用列表。然而，并不需要保留整个文件列表，只需要保留文件引用的数量。增加一个新链接或目录条目就增加引用计数，删除链接或条目就降低计数。当计数为 0 时，没有它的其他引用就能删除该文件。UNIX 操作系统对非符号链接（或**硬链接**）采用了这种方法，即在文件信息块中保留一个引用计数。通过禁止对目录的多重引用，可以维护无环图结构。 为了避免这些问题，有的系统不允许共享目录和链接。例如，对 MS-DOS，目录结构是树结构而不是无环图。

（7）通用图目录

采用无环图结构的一个特别重要的问题是要确保没有环。如果从两层目录开始，并允许用户创建子目录，那么就产生了树结构目录。可以容易地看出对已存在的树结构目录简单地增加新文件和子目录将会保持树结构性质。然而，当对已存在的树结构目录增加链接时，树结构就被破坏了，产生了简单的图结构（图 7.10）。

图 7.10 通用图目录



在考虑这种简单的图结构时，需要认识到无环图的主要优点是可用简单算法遍历图，并确定是否存在文件引用。主要是因为性能原因，大家希望避免多次遍历无环图的共享部分。如果搜索一共享子目录以查找特定文件，但没有找到，那么需要避免再次搜索该子目录，第二次搜索只能浪费时间。

如果在目录中允许有环存在，那么无论是从正确性还是从性能角度而言，同样需要避免多次搜索同一部分。一个设计欠佳的算法可能会无穷地搜索循环而不终止。解决办法之一是可以强制限制在搜索时所访问目录的次数。

当试图确定一个文件什么时候可删除时，会存在另一个类似的问题。与无环结构目录一样，引用计数为 0 意味着没有对文件或目录的引用，那么文件可删除。然而，当存在环时，即使不存在对文件或目录的引用，其引用计数也可能不为 0。这种异常是由于在目录中可能存在自我引用的原故。在这种情况下，通常需要使用垃圾收集方案以确定什么时候可删除最后引用，释放其空间。垃圾收集涉及遍历整个文件系统，并标记所有可访问的空间。然后，第二次遍历将所有没有标记的收集到空闲空间链表上。（一个类似标记步骤可用于确保只对文件系统内的文件或目录进行一次遍历或搜索）。然而，用于磁盘文件系统的垃圾收集是极为费时的，因此很少使用。

垃圾收集是需要的，这仅是因为图中可能存在环。因此，无环图结构更加便于使用。问题是如何在创建新链接时避免环。如何知道什么时候新链接会形成环呢？有些算法可用于检测图中的环，然而，这些算法极为费时，尤其当图位于磁盘上时。在处理目录和链接的特殊情况下，一个简单算法是在遍历目录时避开链接。这样，既避免了环，又没有其他开销。