7.3 文件系统实现

1. 层次化的文件系统

磁盘提供大量的外存空间来维持文件系统。磁盘的下述两个特点，使其成为存储多个文件的方便媒介：

1.磁盘可以原地重写。可以从磁盘上读一块，修改该块，并将它写回到原来的位置。

2.可以直接访问磁盘上的任意一块信息。因此，可以方便地按顺序或随机地访问文件，从一个文件切换到另一个文件只需要简单地移动读写磁头并等待磁盘转动就可以完成。

为了改善 I/O 效率，内存与磁盘之间的 I/O 转移是以块为单位而不是以字节为单位来进行的。每块为一个或多个扇区。根据磁盘驱动器的不同，扇区从 32 字节到 4096 字节不等，通常为 512 字节。

为了提供对磁盘的高效且便捷的访问，操作系统通过**文件系统**来轻松地存储、定位、提取数据。文件系统有两个不同的设计问题。第一个问题是如何定义文件系统对用户的接口。这个任务涉及到定义文件及其属性、文件所允许的操作、组织文件的目录结构。第二个问题是创建数据结构和算法来将逻辑文件系统映射到物理外存设备上。

文件系统本身通常由许多不同的层组成。图 7.15 所示的结构是一个分层设计的例子。设计中的每层利用底层的功能创建新的功能来为更高层服务。

I/O 控制为最底层，由**设备驱动程序**和中断处理程序组成，实现内存与磁盘之间的信息传输。设备驱动程序可以作为翻译器。其输入由高层命令组成，如“提取块 123”。其输出由底层的、硬件特定的命令组成，这些命令用于控制硬件控制器，通过硬件控制器可以使I/O 设备与系统其他部分相连。设备驱动程序通常在 I/O 控制器的特定位置写入特定位格式来通知控制器在什么位置采取什么动作。

图 7.15分层设计的文件系统

基本文件系统只需要向合适的设备驱动程序发送一般命令就可对磁盘上的物理块进行读写。每个块由其数值磁盘地址来标识（例如，驱动器 1，柱面（cylinder） 73，磁道（track）3，扇区（sector）10）。

文件组织模块（file-organization module）知道文件及其逻辑块和物理块。由于知道所使用的文件分配类型和文件的位置，文件组织模块可以将逻辑块地址转换成基本文件系统所用的物理块地址。每个文件的逻辑块按从 0 或 1 到 N 来编号，而包含数据的物理块并不与逻辑号匹配，因此需要通过翻译来定位块。文件组织模块也包括空闲空间管理器，用来跟踪未分配的块并根据要求提供给文件组织模块。

最后，**逻辑文件系统**管理元数据。元数据包括文件系统的所有结构数据，而不包括实际数据（或文件内容）。逻辑文件系统根据给定符号文件名来管理目录结构，并提供给文件组织模块所需要的信息。逻辑文件系统通过文件控制块来维护文件结构。**文件控制块（file control block，FCB）**包含文件的信息，如拥有者、权限、文件内容的位置。

采用分层的结构实现文件系统，能够最大限度地减少重复的代码。相同的 I/O 控制代码（有时候是基本文件系统的代码）可以被多个文件系统采用。然后每个文件系统有自己的逻辑文件系统和文件组织模块。

现在有许多文件系统正在被使用。绝大多数操作系统都支持多个文件系统。例如，绝大多数 CD-ROM 都是按 ISO 9660 的格式来写的，这一格式是 CD-ROM 制造商所遵循的标准格式。除可移动媒介文件系统外，每个操作系统还有一个或多个基于磁盘的文件系统。UNIX使用 UNIX 文件系统（UFS），它是基于伯克利快速文件系统（FFS）的。Windows NT、2000、XP 支持磁盘文件系统 FAT、FAT32 和 NTFS（或 Windows NT File System），还有 CD-ROM、DVD 和软盘文件系统。虽然 Linux 支持超过 40 种不同的文件系统，标准的 Linux 文件系统是可扩展文件系统（extended file system），它最常见的版本是 ext2 和 ext3。同样还存在一些分布式操作系统，即服务器上的文件系统能够被一个或多个客户端加载。

2. 文件系统结构

实现文件系统要使用多个磁盘和内存结构。虽然这些结构因操作系统和文件系统而异，但是还是有一些通用规律的。

在磁盘上，文件系统可能包括如下信息：如何启动所存储的操作系统、总的块数、空闲块的数目和位置、目录结构以及各个具体文件等。上述的许多结构会在本章余下的部分详细讨论，这里简要的描述一下：

- **（每个卷的）引导控制块（boot control block）**包括系统从该卷引导操作系统所需要的信息。如果磁盘没有操作系统，那么这块的内容为空。它通常为卷的第一块。UFS 称之为**引导块（boot block）**，NTFS 称之为**分区引导扇区（partition boot sector）**。
- **（每个卷的）卷控制块（volume control block）**包括卷（或分区）的详细信息，如分区的块数、块的大小、空闲块的数量和指针、空闲 FCB 的数量和指针等。UFS称之为**超级块（superblock）**，而在 NTFS 中它存储在**主控文件表（Master File Table）**中。
- 每个文件系统的目录结构用来组织文件。UFS 中它包含文件名和相关的**索引节点**

（inode）号。NTFS 中它存储在主控文件表（Master File Table）。

- 每个文件的 FCB 包括很多该文件的详细信息，如文件权限、拥有者、大小和数据块的位置。UFS 称之为索引节点（inode）。NTFS 将这些信息存在主控文件表中，主控文件表采用关系数据库结构，每个文件占一行。

内存内信息用于文件系统管理并通过缓存来提高性能。这些数据在文件系统安装的时候被加载，卸载的时候被丢弃。这些结构可能包括：

- 一个内存中的安装表，包括所有安装卷的信息。
- 一个内存中的目录结构缓存，用来保存近来访问过的目录信息。（对于卷所加载的目录，可以包括一个指向卷表的指针）
- **系统范围内的打开文件表**包括每个打开文件的 FCB 拷贝和其他信息。
- **单个进程的打开文件表**包括一个指向系统范围内已打开文件表中合适条目的指针和其信息。

为了创建一个新文件，应用程序调用逻辑文件系统。逻辑文件系统知道目录结构形式。为了创建一个新文件，它将分配一个新的 FCB。（如果文件系统实现在文件系统被创建的时候已经创建了所有的 FCB，那么只是从空闲的 FCB 集合中分配一个）。然后系统把相应目录信息读入内存，用新的文件名更新该目录和 FCB，并将结果写回到磁盘。图7.16 显示了一个典型的 FCB。

文件权限
文件日期（创建、访问、写）
文件所有者、组、访问控制列表（ACL）
文件大小
文件数据块或者文件数据块指针

图 7.16 一个典型的文件控制块

有些操作系统，包括 UNIX，将目录按文件来处理，用一个类型域来表示是否为目录。其他操作系统如 Windows NT 为文件和目录提供了分开的系统调用，对文件和目录采用了不同的处理。不管结构如何，逻辑文件系统都能够调用文件组织模块来将目录 I/O 映射成磁盘块的号，再进而传递给基本文件系统和 I/O 控制系统。一旦文件被创建，它就能用于 I/O。不过，首先应打开文件。调用 open()将文件名传给文件系统。系统调用 open()会首先搜索系统范围内的打开文件表以确定某文件是否已被其他进程所使用。如果是，就在单个进程的打开文件表中创建一项，并指向现有系统范围内的打开文件表。该算法能节省大量开销。当打开文件时，根据给定文件名来搜索目录结构。部分目录结构通常缓存在内存中以加快目录操作。一旦找到文件，其 FCB 就复制到系统范围内的打开文件表。该表不但存储 FCB，而且还跟踪打开该文件的进程数量。

接着，在单个进程的打开文件表中会增加一个条目，并通过指针将系统范围内的打开文件表的条目和其它域相连。这些其它域可以包括文件当前位置的指针（用于下一次的读写操作）和文件打开模式等。调用 open()返回一个指向单个进程的打开文件表中合适条目的指针。所有之后的文件操作都是通过该指针进行的。文件名不必是打开文件表的一部分，因为一旦完成对 FCB 在磁盘上的定位，系统就不再使用文件名了。然而它可以被缓存起来以节省后续打开相同文件的时间。

对于访问打开文件表的索引有各种名称。UNIX 称之为**文件描述符（file descriptor）**，Windows 称之为**文件句柄（file handle）**。因此，只要文件没有被关闭，所有文件操作都是通过打开文件表来进行的。

当一个进程关闭文件，就删除一个相应的单个进程打开文件表的条目，系统范围内打开文件表相应文件条目的打开数也会递减。当打开文件的所有用户都关闭一个文件时，更新的文件元数据会复制到磁盘的目录结构中，系统范围内的打开文件表的相应条目也将删除。 有的系统更加复杂，它们将文件系统作为对其他系统方面的访问接口，如网络。例如，UFS 的系统范围的打开文件表有关于文件和目录的索引节点（inode）和其他信息。它也有关于网络连接和设备的类似信息。采用这种方式，一种机制满足了多个目的。

文件系统结构的缓存也不应被忽视。大多数系统在内存中保留了打开文件的所有信息（除了实际的数据块）。BSD UNIX 系统在使用缓存方面比较典型，哪里能节省磁盘 I/O 哪里就使用缓存。其 85% 的缓存平均命中率说明了这些技术的实现是值得的。BSD UNIX 系统在附录 A 中详细描述。

图 7.17 总结了内存中的文件系统结构。

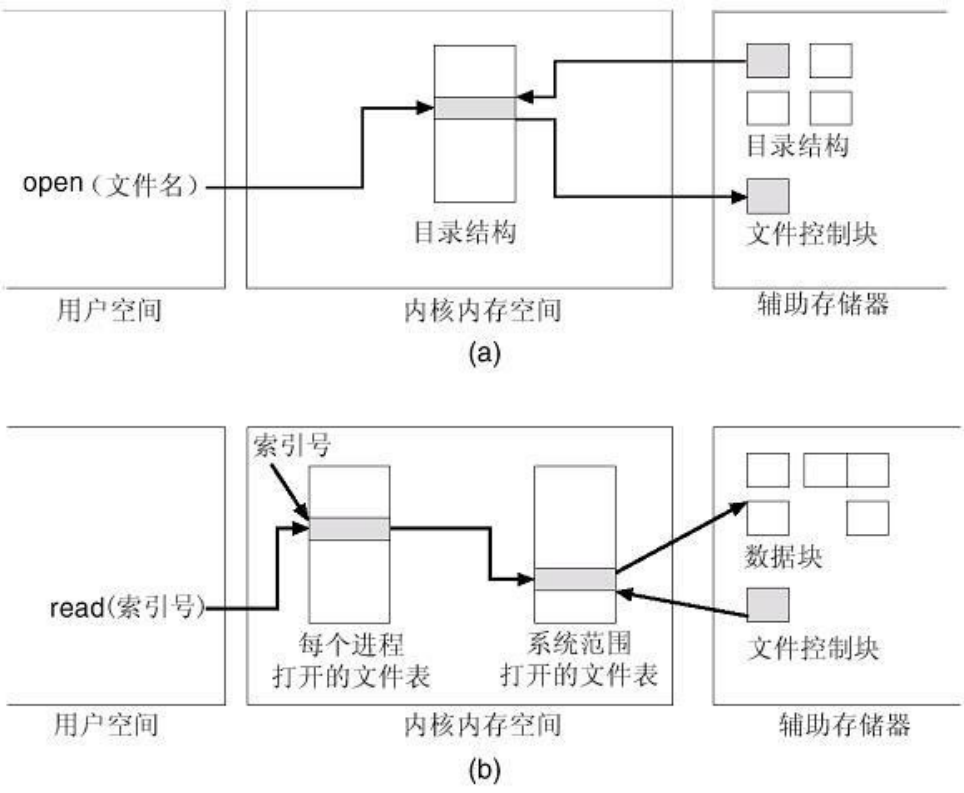

图7.17 内存中的文件系统结构 (a) 打开文件 (b) 读文件

3. 虚拟文件系统

上一节清楚地说明了现代操作系统必须同时支持多个文件系统类型。但是操作系统如何才能把多个文件系统整合成一个目录结构？用户如何在访问文件系统空间时，可以无缝地在文件系统

类型之间移动？现在来讨论这些实现细节。

实现多个类型文件系统的一个明显但不十分满意的方法是为每个类型编写目录和文件程序。但是，绝大多数操作系统包括 UNIX 都使用面向对象技术来简化、组织和模块化实现过程。使用这些方法允许不同文件系统类型可通过同样结构来实现，这也包括网络文件系统类型如 NFS。户可以访问位于本地磁盘的多个文件系统类型，甚至位于网络上的文件系统。

采用数据结构和子程序，可以分开基本系统调用的功能和实现细节。因此，文件系统实现包括三个主要层次，如图7.18所示。第一层为文件系统接口，包括open()，read(), write()和close()调用及文件描述符。

第二层称为**虚拟文件系统（VFS）**层，它有两个目的：

1. VFS 层通过定义一个清晰的 VFS 接口，以将文件系统的通用操作和具体实现分开。多个 VFS 接口的实现可以共存在同一台机器上，它允许访问已装在本地的多个类型的文件系统。
2. VFS 提供了在网络上唯一标识一个文件的机制。VFS 基于称为 vnode 的文件表示结构，该结构包括一个数值标志符以表示位于整个网络范围内的唯一文件。Unix 索引节点 inode 在文件系统内是唯一的）。该网络范围的唯一性用来支持网络文件系统。内核中为每个活动节点（文件或目录）保存一个 vnode 结构。

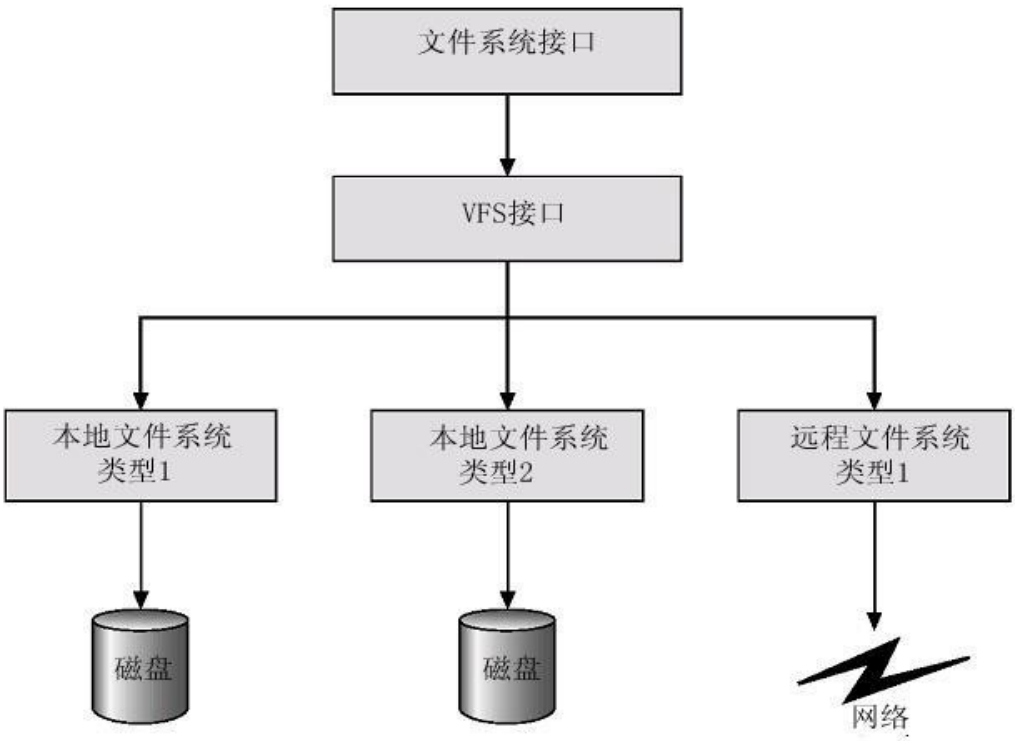

图 7.18 虚拟文件系统示意图

因此，VFS 区分本地文件和远程文件，根据文件系统类型可以进一步区分不同本地文件。

VFS 根据文件系统类型调用特定文件类型操作以处理本地请求，通过调用 NFS 协议程序来处

理远程请求。文件句柄可以从相应的 vnode 中构造，并作为参数传递给程序。结构中的第三层实现文件系统类型或远程文件系统协议。

下面简要的讨论一下 Linux 中的 VFS 结构。Linux VFS 定义的四种主要对象类型是：

- **索引节点对象**（inode object），表示一个单独的文件
- **文件对象**(file object)，表示一个打开的文件
- **超级块对象**(superblock object)，表示整个文件系统
- **目录条目对象**(dentry object)，表示一个单独的目录条目

VFS 对每种类型的对象都定义了一组必须实现的操作。这些类型的每一个对象都包含了一个指向函数表的指针。函数表列出了实际上实现特定对象的操作函数。比如，文件对象的一些操作的缩写 API 包括：

- int open(...) 打开一个文件
- ssize_t read(...) 读文件
- ssize_t write(...) 写文件
- int mmap(...) 内存映射一个文件

一个特定文件类型的文件对象的实现必须实现文件对象定义中的每个函数。（文件对象的完整定义在文件/usr/include/linux/fs.h 中的 struct file_operations 中）

因此，VFS 软件层能够通过调用对象函数表中的合适函数来对对象进行操作，而不需要事先知道对象的实现类型。VFS 不知道，也不关心一个索引节点是代表一个磁盘文件、一个目录文件还是一个远程文件。实现文件 read()操作的合适函数总是被放在函数表中的相同位置，VFS 软件层调用这些函数，而不关心数据是如何被读取的。

4. 目录实现

目录分配和目录管理算法的选择对文件系统的效率、性能和可靠性有很大影响。这一部分讨论这些算法的优缺点。

（1）.线性列表

最为简单的目录实现方法是使用存储文件名和数据块指针的线性列表。这种方法编程简单但运行费时。要创建新文件，必须首先搜索目录以确定没有同样名称的文件存在。接着，在目录后增加一个新条目。要删除文件时，根据给定文件名搜索目录，接着释放分配给它的空间。如果要重用目录条目，可以有许多办法。可以将目录条目标记为不再使用（赋予它一个特定的文件，如全为空的名称或为每个条目增加一个使用-非使用位），或者可以将它加到空闲目录条目列表上。第三方法是将目录的最后一个条目拷贝到空闲位置上，并减少目录长度。链表可以用来减少删除文件的时间。

目录条目的线性列表的真正缺点是查找文件需要线性搜索。目录信息需要经常使用，用户在访问文件时会注意到现实的快慢。事实上，许多操作系统采用软件缓存来存储最近访问过的目录信息。缓存命中避免了不断地从磁盘读取信息。排序列表可以使用二分搜索，并减少平均搜索时间。不过，列表始终需要排序的要求会使文件的创建和删除复杂化，这是因为可能需要移动不少的目录信息来保持目录的排序。一个更为复杂的树数据结构，如 B 树，可能更为有用。已排序列表的一

个优点是不需要排序步骤就可生成排序目录信息。

（2）.哈希表

用于文件目录的另一个数据结构是**哈希表**。采用这种方法时，除了使用线性列表存储目录条目外，还使用了哈希数据结构。哈希表根据文件名得到一个值，并返回一个指向线性列表中元素的指针。因此，它大大地减少目录搜索时间。插入和删除也较简单，不过需要一些预备措施来避免冲突（两个文件名哈希到相同的位置）。

哈希表的最大困难是其通常固定的大小和哈希函数对大小的依赖性。例如，假设使用线性哈希表来存储 64 个条目。哈希函数可以将文件名转换为 0 到 63 的整数，例如采用除以64 的余数。如果后来设法创建第 65 个文件，那么必须扩大目录哈希表，比如到 128 个条目。因此，需要一个新的哈希函数来将文件名映射到 0 到 127 的范围，而且必须重新组织现有目录条目以反映它们新的哈希函数值。

或者，可以使用 chained-overflow 哈希表。每个哈希条目可以是链表而不是单个值，可以采用向链表增加一项来解决冲突。由于查找一个名称可能需要搜索冲突条目组成的链表，因而查找可能变慢；但是，这比线性搜索整个目录可能还是要快很多。