第 8 章　文件系统接口和框架

8.1　简　　介

任何一个操作系统都必须要提供持久性存储和管理数据的手段。在 UNIX 系统中,“ 文件 ”用来保存数据，而 “ 文件系统 ” 可以让用户组织、操纵以及存取不同的文件。本章主要描述了文件系统和用户应用程序之间的接口，同时可以看到操作系统采用了怎样的框架来支持不同种类的文件系统。第 9，10，11 章讲述了几种不同的文件系统的实现，通过它们用户可以访问本地或者远程主机上的文件。

文件系统接口由一组系统调用和实用工具组成，用户可以使用这些系统调用和实用工具对文件进行存取操作。很长时期以来，文件系统的接口保持了一定的稳定性，即使变化也是向下兼容的。但是文件系统的框架结构发生丁彻底的变化。起初的框架结构只支持一种文件系统，并且所有的文件都必须存放在与系统物理连接的本地磁盘上，这种结构已经被彻底抛弃了，取而代之的是 vnode/vfs 接口，该接口可以使多种本地的或者远程的文件系统共存于同一台机器上。

早期的商业 UNIX 发行版都包含有一个现在称之为 s5fs(System 5 文件系统)的简单的文件系统[Thom 78]。4.2BSD 以前的 Berkely UNIX 版本，以及所有的 System V UNIX 版本都支持这种文件系统。4.2BSD 引进了一种新的文件系统——快速文件系统(FFS)[McKu84]，其系统性能比 s5fs 要优越。FFS 也因此赢得了广泛的支持，最终被包含到 SVR4 中。第日章将介绍 s5fs 和 FFS，同时也介绍其他一些基于 vhode/vfs 接口的专用文件系统。

注意： 术语 FFS 和 ufs(UNIX 文件系统)常常互换使用。为了不至于混淆，我们规定 FFS 特指 Berkeley 快速文件系统的最初实现，而 ufs 指的是在 vnode/vfs 框架下的 FFS 实现，本书以这两种方式使用它，随着计算机网络的日益昔及，人们开始想办法对远程节点上的文件进行访问。80 年代中期出现了几种技术可以将文件在互连的计算机上实现透明共享。第 10 章主要讲述了三个重要实现——网络文件系统(NFS)、远程文件系统(RFS)和 Andrew 文件系统(AFS)。

近些年来，为了提高 FFS 的性能或者满足某些特殊的需求，又出现了一些新的文件系统。为了有更好的性能，更高的可靠性以及可用性，这些系统大都使用了诸如日志、快照、卷管理等复杂的技术。第 1l 章讲述了一些现代文件系统。

8.2　文件的用户接口

UNIX 内核允许用户进程通过一个严格定义的过程性接口与文件系统进行交互。这个接口对用户屏蔽了文件系统的细节，同时指定了所有相关系统调用的行为和语义。该接口向用户提供了一组抽象概念：文件(files)，目录(directory)，文件描述符(filedescriptor)和文件系统(file systems)。

如前所述，现在有很多种类的文件系统，例如 s5fs 和 FFS。每一种文件系统都实现了相同的接口，因此给应用程序提供了一致的文件视图，不过每种文件系统在其接口的实现中有可能对某个方面加以限制。例如，在 s5fs 中文件名最多 14 个字符，而在 FFS 中最多可达 255 个字符。

8.2.1　文件和目录

“ 文件 ”在逻辑上是数据的容器。用户可以创建文件，并且把数据写入文件，从而达到保存数据的目的。对文件既可以顺序访问也可以随机访问。内核提供了几种控制操作用来命

名，组织和存取文件。内核并不解释文件的内容，因为在内核看来文件不过是一些字节的集合，所以它只负责对文件提供字节流控制。如果有的应用程序需要更加复杂的语义，比如要求可以通过记录或者索引来存取文件，那么它可以在内核的原语之上设计自己的机制来达到要求。

从用户的角度来看，UNIX 以一种层次的树状结构来管理文件的名字空间(图 8-1)。这棵“树”由文件和目录组成，其中所有的文件都在“叶子”的位置上①。目录的内容是所有在该目录底下的文件及其他子目录的名字信息。文件和目录的名字是由一些除去“/”和空字符的 ASCII 字符组成。不同的文件系统对文件和目录名字的长度有不同的限制。根目录常常被写做“/”，文件名字没有必要在整个文件系统中是唯一的，而只要求在该文件所在的目录中唯一即可。在图 8-1 中，在目录 bin 和目录 etc 下都有一个名为 passwd 的文件。为了确保唯一性，需要将文件的全路径名写出来。文件的全路径名由从根目录到该文件节点的所有分量组成,其中不同分量之间用‘/’隔离开。因此也图 8-1 中的两个文件可分别表示为/bin/passwd 和/etc/passwd。在 UNIX 中‘/’既可以表示根目录，也可以充当分隔符。

图 8-1　按目录树的方式组织文件

在 UNIX 中，每一进程中都有一个当前工作目录，这是进程状态的一部分。这样用户就可以用相对路径名来引用所需要的文件。在这里有两种特殊的路径分量：第一个是‘.’，这代表目录本身；第二个是“..”，代表父目录。根目录没有父目录，它的“..”指的是它自己。在图 8-1 中，假设有一个用户的当前工作目录是/usr/local，如果他想找到 lib 目录，那么他既可以使用/usr/lib(绝对路径名)，也可以使用../lib(相对路径名)。进程可以通过系统调用 chdir 来改变当前工作目录。

文件在目录中的项称为该文件的硬链接，简称链接。任何一个文件都可能有一个或者多个指向它的链接，它们可以在同一个目录下，也可能在不同的目录下。因此一个文件并不局限于在一个目录中，也并不一定只有一个名字。名字并不是文件的一个属性。只要一个文件的链接数大于零，这个文件就还存在。这些链接是完全对等的，只不过是同一个文件的不同名字而已。使用任何一个链接都可以访问到该文件，并且不知道到底哪一个是最初的联接。现代 UNIX 文件系统又提供了另外一种类型的链接——符号链接。我们将在第 8.4.1 小节中看到符号链接。

每一种文件系统都有其内部目录格式。因为应用程序员想要以一种可移植的方式来读取目录内容，POSlX.1 标准提供了一组标准库函数用以对目录进行操作：

```
dirp=opendir(const char * fllename);
direntp=readdir(dirp);
rewinddir(dirp);
status=closedir(dirp);
```

上面这些调用最初出现在 BSD 中，现在已被 SVR4 和大多数的商业 UNIX 版本所支持。当用户调用 opendir 时，系统将它同一个目录流(directory stream)关联起来，返回给用户一个流的句柄。流对象中维持着一个指向下一个要读的表项的偏移量。每次 readdir 调用都返回一个单独的目录表项，并且将偏移量向前移动。返回的表项格式是文件系统无关的，其格式定义如下：

```
struct dirent{
ind_t  d_ino;    /* inode numbe (see Section 8.2.2) */
char d_name[NAME_MAX+1];/* null-terminated filename */
};
```

不同的文件系统中 NAME_MAX 的值可能不一样。SVR4 还有一个 getdents 的系统调用，可以以文件系统无关的格式读取目录项。该调用返回的项的格式和 struct dirent 不同，因此

用户应该尽量使用可移植的 POSIX 函数。

8.2.2 文件属性

除了文件名字外，文件系统为每一个文件都维护了一组属性。这些属性并没有保存在目录项中，而是保存在一个叫做 i 节点(inode)的磁盘结构中。i 节点这个词是指索引 i 节点。不同的文件系统中 i 节点纳具体格式和内容并不完全相同。系统调用 stat 和 fstat 返回一些与文件系统无关的文件属性。最常见的文件属性如下：

·文件种类　除了普通文件外，UNIX 也识别其他一些特殊种类的文件，包括目录文件，先入先出文件(FIFO)，符号链接，代表字符设备或块设备的特殊文件。

·指向文件的硬链接数目。

·以字节计算的文件大小。

·设备 ID　用以标识该文件所在的设备。

·i 节点号　不管一个文件或目录有多少个链接，它只有一个与之相关的 i 节点。每一个处于一给定的磁盘分区之上的 i 节点有一个唯一的 i 节点号,因此使用设备 ID 和 i 节点号便可以唯一地确定一个文件。这些标识符(i 节点号)并不存储在 i 节点内。设备 ID 是文件系统的一个属性，因为一个文件系统中的所有文件都有相同的设备 ID。目录项存放 i 节点号以及文件名。

·文件所有者的用户 ID 和组 ID。

·时间戳　每一个文件都有三个时间戳：文件上次被访问的时间；上次被修改的时间；文件的属性(除了其他时间戳)上次被修改的时间。

v 权限和模式标志，如下所述。

针对每个文件都有三种不同的权限——读，写，执行。试图访问一个文件的用户也分为三类——文件的所有者②，跟文件的所有者在同一组内的用户，其他人(也就是 owner ;group 和 others)。这就意味着对一个文件的权限可以用 9 个比特来指定。目录的权限用另外的方法指定。如果对一个目录拥有写权限，那么你可以在该目录中创建和删除文件。执行权限允许你访问目录中的文件，即使用户有权限，也绝不允许直接对目录进行读写，目录的内容只有在创建文件和删除文件时才被改变。

权限机制很有用，但是很原始。现在，大多数 UNIX 供应商或是在缺省实现中，或是在特殊的安全版本中③，提供了增强的安全特性。这些特性一般都涉及某种形式的访问控制列表，它可以对访问该文件的用户以及操作方式做更加详细的规定[Fern 88]。

在 UNIX 中有三种模式标志——suid，sgid 以及 sticky。其中 suid 和 sgid 适用于可执行文件。当一个用户执行文件时，如果 suid 标志被设置了，内核就会将用户的有效 UID 设置为该文件的所有者。sgid 以同样的方式影响着有效 GID(2.2.3 小节定义了不同的 UID 和 GID)。因为 sgid 标志在文件不是可执行文件时并没有什么用处，它可以被用作其他目的。如果文件并投有组执行许可并且其 sgid 标志被置位了，那么文件可能被强制文件/记录加锁(mandatory file/record locking)[UNIX 92]了。

sticky 标志也是用于可执行文件，它要求系统在文件执行完毕后将程序映象(image)保留在交换区中(见 13.2.4 小节)。为了提高系统性能，系统管理员常常把执行比较频繁的的文件的 sticky 标志置位。大多数的现代 UNIX 系统尽可能地将程序映象保留在交换区(使用一种最近最少使用的替换策略)，因此用不到 sticky 位。

sgid 和 sticky 用在目录中时其作用和用于文件时的作用是不同的。如果 sticky 标志被置位并且目录是可写的，那么当只有一个进程的有效 UID 是文件或目录的所有者或者进程对该文件有写权限时，该进程才能够删除或者重命名目录中的文件。如果 sticky 标志被清除，任何对目录有写权限的进程都可以删除或者重命名其中的文件。

当一个文件被创建时，它继承了创建进程的有效 UID。它的所有者 GID 可能取两个值中

的一个。在 SVR3 中，文件继承了创建者的有效 GID。在 4.3BSD 中，文件继承了它所在的目录的 GID。SVR4 使用父目录的 sgid 标志来决定要采取什么样的行为。如果一个目录的 sgid 标志被置位了，那么在该目录中创建的文件从父目录那里继承了 GID。如果 sgid 被清除，那么新文件继承了创建者的 GID。

UNIX 提供了一组系统调用来操纵文件属性。这些系统调以文件的路径名作为参数。系统调用 link 和 unlink 分别为文件创建和删除一个硬链接。只有当文件的所有硬链接都被清除并且没有人止在使用该文件时，内核才能将该文件删除。utimes 系统调用改变文件的“访问”和“修改”时间戳。chown 调用改变用户 UID 和 GID。chmod 系统调用改变文件的权限和模式标志。

8.2.3 文件描述符

进程要想对一个文件进行读写，必须首先将该文件打开。系统调用 open 的语法如下；

fd=open(path，oflag，mode)；

这里，path 是文件的绝对或相对路径名，如果必须创建文件，那么 mode 指定了该文件的权限。oflag 指定了文件是以读，写，写 -读或扩展方式打开，还是必须创建一个文件等等。create 系统调用也可以创建一个文件，在功能上 create 跟用 WRONLY，O_CREAT，O_TRUNC 作为参数打开一个文件等价(见 8.10.4 小节)。

每一个进程都有一个“文件创建掩码”(file creation mask)，这是一个权限位掩码，它不应该被赋予新创建的文件。当用户以某一指定的方式去打开或创建文件时，内核会自动清除缺省的位掩码。umask 系统调用改变缺省位掩码的值。在文件创建后用户可以调用 chmod 来重载文件的掩码。

当用户调用 open 时，内核创建一个“打开文件对象”(open file object)来表示文件的“打开实例”(open instance)。内核也分配一个文件描述符(fd)，文件描述符充当打开文件对象的句柄。系统调用 open 将一个文件描述符返回给调用者。一个用户可以多次打开同一个文件；多个用户也可以同时打开一个文件。在所有这些情况中，内核每次都为调用者创建一个打开文件对象和新的文件描述符。

文件描述符是一个进程内部的对象。如果在两个不同进程中有两个文件的描述符号相同，这两个文件描述符一般指的是两个文件。进程将文件描述符传递给 I/O 有关的系统调用，例如 read 和 write。内核利用该文件描述符迅速找到打开文件对象和其他与打开文件有关的数据结构。这样，内核就可以在 open 过程中将诸如路径解析和存取控制的工作一次做完，而不是在每次 I/O 操作时再去做这些工作。打开文件可以使用 close 系统调用关闭，当进程终止时打开文件也会自动关闭。

每一个描述符都代表与文件的一次独立会话。与之相联系的打开文件对象包含了该任务的上下文，其中也包含了文件被打开的方式以及下次读或写应该开始的偏移。在 UNIX 中，文件的访问方式缺省是顺序方式。当用户打开一个文件时，内核将偏移指针初始化为零。接下来，每次读或写之后内核在原来的偏移量上加上读出或写入的字节数。

将偏移指针存放在打开文件对象中可以使内核将对同一个文件的不同任务分隔开(见图 8-2)。如果两个进程打开同一个文件④，读进程或写进程仅仅移动它自己的偏移指针，并不影响其他的任务。这样就使得多个进程透明地共享一个文件成为可能。不过使用这个特性时要小心。在大多数情况下，多个同时访问同一个文件的进程应该对该文件进行同步操作，这可以使用 8.2.6 小节讲述的文件加锁工具。

图 8-2 一个文件打开两次

进程可以通过 dup 或 dup2 系统调用来复制文件描述符。这些系统调用创建一个指向同一个打开文件对象的新的描述符，这样该描述符跟原来的描述符共享同一个会话(图 8-3)。同样，系统调用 fork 将父进程中的所有文件描述符复制下来，将它们传到子进程中。当从 fork

调用返回后，父进程和子进程共享相同的打开文件。这种共享跟多个打开实例是不同的。因为此时两个描述共享文件的同一个任务，因此它们都可以看到相同的文件视图并且拥有同样的偏移指针。如果一个描述符的偏移指针被改变，则此改变对另外一个也是可见的。

图 8-3　通过 dup，dup2 或 fork 复制文件描述符

进程可以将文件描述符传递给另外的不相关进程，这和将指向打开文件对象的引用传递给另外一个进程是等价的。内核将文件描述符复制到接收进程的文件描述符表的第一个空闲表项中，这两个描述符共享同样的打开文件对象，因此也共享相同的偏移指针。一般情况下，发送进程在将描述符发送出去之后将该描述符关闭，这并不会关闭打开文件对象，即使接收进程此时还未接收到描述符，因为在第二个描述符传递过程中内核持有描述符。

对于描述符传递,其接口视不同的 UNIX 变体而不同,SVR4 使用带 I_SENDFD 命令的 ioctl 调用，通过 STREAMS pipe 传递描述符，其他进程通过 I_RECVFD ioctl 调用接收它。4.3BSD 使用 sendmsg 和 recvmsg 系统调用通过套接字连接在两个进程间传递描述符。描述符传递在某些类型的网络应用实现中很有用处。connection server 进程可以代表一个客户进程建立一个网络连接，接着将代表该连接的描述符返回给客户。10.1 小节讲述了在 4.4BSD 中的 portal 文件系统，该文件系统对上面说的概念进行了进一步的延伸。

8.2.4　文件 I/O

在 UNIX 中，文件既可以被顺序访问，又可被随机访问。缺省方式是顺序访问。内核在打开一个文件时将偏移指针设为零，在随后的操作中，内核维护着偏移指针。该指针代表当前文件中的某一位置，下一次读或写都将从该位置开始。每次进程读或写后，内核都会将指针值加上该次读或写传输的字节数。lseek 系统调用可以将偏移指针置为某一指定的值，从而实现随机存取。随后的 read 或 write 将从该指定的偏移开始传输数据。

系统调用 read 和 write 有类似的语义，我们以 read 为例⑤，其语法是：

nread＝read(fd,buf,count);

其中 fd 是文件描述符，buf 是在用户地址空间中指向某一缓冲区的指针，读取出来的数据将放到该缓冲区内，count 表示该次操作要读取多少字节的数据。

内核从与 fd 相联系的文件中并从保存在打开文件对象中的偏移处开始读取数据。如果文件到了结尾或者在 FIFO 及设备文件没有足够的数据时，open 调用读取的数据量可能小于 count。例如，当在一个处于 canonical 模式下的终端上进行 read 操作时，如果用户敲了回车键，read 必须返回，而不管该行的字符个数是否符合要求的个数。在任何情况下，内核都不会读取比 count 还要多的数据。用户必须保证 buf 要是够大以便能够容纳 count 个字节的数据。read 调用返回实际传输的字节数目 nread。read 调用同时把偏移指针向前移动 nread 个字节，以便下次读取或写入时能够从本次结束的地方开始。

尽管内核允许多个进程同时打开和共享同一个文件，实际上的 I/O 操作还是有先后顺序的。例如，如果两个进程同时对一个文件发出写命令，内核将先完成一个写操作，然后再完成另外一个写操作。这样就保证了每一次操作都有一个一致的文件视图。

如果将 O_APPEND 标志作为参数传到 open 系统调用中，文件就会以扩展模式被打开。此时内核会通过描述符把偏移指针指向上次写调用时的文件结尾。值得注意的是，如果一个文件以扩展模式被打开，所有其他指向该文件的描述符对该文件都不会产生任何作用。

多线程系统必须处理由于多个线程之间共享文件描述符而带来的麻烦。例如，一个线程恰好在另一个线程发出一个读命令之前进行 lseek 调用，这将导致第一个线程从错误的位置开始读操作。有一些系统，比如 solaris，提供了 pread/pwrite 系统调用来进行原子查找以及读/写。3.6.6 小节详细地讨论了这个问题。

8.2.5　分散-聚集 I/O(Scatter-Gather I/O)

系统调用 read 和 write 可以在文件中一块逻辑上连续的部分和进程地址空间中一片连续

的地址空间中传递数据。但是有时候用户希望他们能够一次将进程地址空间中一些不连续的缓冲区中的数据写入到文件中去，或者从文件中将数据读到其中。read 和 write 并不能达到这些要求，因为它们要求进程中存放数据的空间在逻辑上是连续的，UNIX 提供了两个系统调用 writev 和 readv，可以提供分散-聚集 I/O，也就是在文件和进程的不连续地址空间之间传递数据。

举一个例子，考虑一个网络协议，它从远程节点那里接受文件，然后把它们写入本地磁盘中。数据是以网络数据包的形式到达本地节点的，其中每一个数据包中都包含有文件的呆一部分。如果没有分散，聚集 I/O 方式的话，协议必须首先将所有的数据包放到一个连续的缓冲区中去，然后才能再写入文件中去。而有了 writev，协议便可以一次将那些数据包中的数据写到文件中去，并不需要事先收集。writev 的语法如下(readv 类似)：

nbytes = writev(fd，iov，iovcnt);

其中 fd 是文件描述符，iov 是指向一个<base，length>偶对(struct iovee)的数组，这个序列描述数据源缓冲区，iovcnt 是在数组中的元素数目，跟系统调用 write 一样，nbytes 代表该次操作实际写入的字节数目。fd 中的偏移量指针决定了这次操作的文件中的起始地址，写入操作结束后，内核会将偏移量加上 nbytes。

图 8-4 显示了分散-聚集写操作的效果，内核创建一个 uio 结构，用从系统调用参数和打开文件对象那里得到的信息来初始化该结构。然后内核将指向 uio 结构的指针传递给更低层的函数来进行 I/O 操作。内核自动将离散缓冲区中的所有数据写入到文件中去。

图 8-4　分散-聚集 I/O

8.2.6　文件力口锁

缺省方式下，UNIX 允许多个进程并发地对同一个文件读或者写。每一个 read 和 write 调用都是原子性的，但是在不同的读和写调用中并没有进行同步化操作。结果，如果一个进程通过多个 read 调用来读取一个文件,另外一个进程可能会在两个 read 之间改变文件内容。

如果有一个应用程序，它要求在多个存取操作过程中文件视图保持一致，很显然，如果不采取额外的措施，根本达不到这个要求。因此，UNIX 提供了文件加锁工具。文件加锁既可以是建议性的也可以是强制性的。如果选择了建议性加锁，内核并不强制加锁，并且只有当协作进程显式地检查文件锁时此文件锁才能保护文件。如果选择了强制性加锁，内核会为文件强制加锁，任何跟锁冲突的操作都会被拒绝掉，加锁请求既可以是阻塞的，也可以是非阻塞的；在后者中，如果锁没有被建立起来，内核会返回一个 EWOULDBLOCK 的错误代码。

4BSD 提供了系统调用 flock，它只支持打开文件时建议性文件加锁，但是却可以允许共享的和互斥加锁。System V UNIX 中的文件加锁功能根据版本的不同而有所不同。SVR2 只支持对文件和记录(文件中的一个数据块)的建议性加锁。SVR3 加上了强制性文件加锁，但是要求文件首先使用 chmod 系统调用使得文件能够被强制性文件加锁，见 8.2.2 小节。这个特性是 XENIX 二进制兼容的产物。SVR4 加上了 BSD 兼容部分并且支持单写者，多读者(single-writeer)的读写锁。系统调用 fcntl 提供了加锁功能，然而大多数应用程序都使用由 C 库函数 lockf 提供的更加简单的编程接口。

8.3　文件系统

尽管 UNIX 文件层次树看起来是一个整体，实际上却是由一个或多个独立于树组成，其中每一个子树包含一个完整的，自包含的文件系统。其中一个文件系统被配置为根文件系统，它的根目录被称为系统根目录。其余的文件系统可以通过被安装到已知文件树的一个目录上.从而与已知文件结构联系起来。一旦被安装上，被安装文件系统便覆盖了安装点上的目录，任何对安装点目录的访问都会自动转换到被安装文件系统根目录的访问。一直到该文件系统被卸出，它一直都是可见的。

图 8-5 显示了由两个文件系统组成的文件层次树。在这个例子中，fs0 被当作根文件系统安装到机器上,文件系统 fs1 被安装到 fs0 的/usr 目录上。/usr 被称为安装目录或安装点，任何试图对/usr 的访问都会转换到对安装在/usr 的文件系统根目录的访问。

图 8-5　在另一个文件系统上安装文件系统

如果 fs0 的/usr 目录中包含文件，那么当 fs1 安装到该点时，这些文件变得不可见了或称为被覆盖了，用户再也不能访问它们了。当 fs1 被卸出之后，这些文件才重新对用户可见，并且也能够访问。内核的路径分析程序必须要正确地理解安装点，并且当遍历安装点时必须确定正确的路径方向。最初的 s5fs 和 FFS 实现使用了安装表来跟踪安装的文件系统。现代 UNIX 系统使用了一种称为 vfs 列表(虚拟文件系统列表)的形式，vfs 列表将在 8.9 节讲述。

可安装子系统的概念可以对用户隐蔽存储细节，文件名字空间是同构的，用户并不需要把磁盘作为文件名字的一部分(而在 MS-DOS 和 VMS 中必须要把磁盘号作为名字的一部分)。文件系统可以被离线备份，压缩，以及修复。系统管理员也可以对每一个文件系统设置不同的保护措施，比如可以将它们中的一些设置为只读。

可安装文件系统也会给文件层次树加上一些限制。文件不能跨越文件系统，其长度要受到它所在的文件系统的大小的限制。重命名以及建立硬链接的操作也不能跨越文件系统。每一个文件系统必须要驻留在一个单独的逻辑磁盘上并且要受到该磁盘大小的限制。

8.3.1　逻辑磁盘

逻辑磁盘是抽象的存储概念，内核将其视为一些有固定大小可随机存取的块的线性序列。磁盘设备驱动程序将这些块映射到物理存储介质上。newfs 以及 mkfs 等工具可以在磁盘上创建一个 UNIX 文件系统，每一个文件系统只能在单独一个逻辑磁盘卜。一个逻辑磁盘可能只包含一个文件系统，有些逻辑磁盘不包含任何文件系统，但是可以被存储子系统用作交换区。

内核可以使用多种方式将逻辑磁盘映射到物理存储。最简单的情况是一个逻辑磁盘占据了一整个物理磁盘。一般情况下，一个物理磁盘被分成物理上连续的几个分区，每个分区就是一个逻辑磁盘。旧的 UNIX 系统仅仅提供这种分区方式。因此，分区一词就常用来描述一个文件系统的物理存储介质。

现代 UNIX 系统支持很多其他的存储配置。很多物理磁盘可以结合起来成为一个逻辑磁盘，从而使逻辑磁盘可以容纳比单个物理磁盘还要大的文件。磁盘镜像使得所有的数据都可以有一份冗余备份，从而提高了文件系统的可靠性。带区集(stripe set)通过把数据跨磁盘组分成带区增加了文件系统的吞吐虽。一些类型的 RAID(廉价磁盘冗余阵列)配置可以增强可靠性并提高性能，从而达到不同种类的安装需求[Patt 88]。

8.4　特殊文件

UNIX 文件系统的一个显著特点是可以用文件抽象来代表所有的 I/O 对象，其中包括目录，符号链接，像磁盘，终端和打印机等硬件设备，诸如系统内存之等的伪设备，还有像管道和套接字等通信抽象。上面这些设备都是通过文件描述符来访问的，对普通文件使用的系统调用同样也适用于操纵特殊文件。例如，一个用户想把数据送入打印机，那么他只需打开跟该打印机相连的文件并且将数据写入即可。

另外，在 UNIX 中文件是一种字节流模型。很多现实世界中的应用需要更加幸富的数据抽象，诸如基于记录或者顺序索引的存取模型抽象等。UNIX 对于该类的应用显得无能为力，这就需要用户在字节流模型之上加上他们自己的访问模型。另外并不是所有的 I/O 对象都支持所有类型的文件操作。例如，对于终端和打印机来说，就没有随机访问或查找的概念。通常，应用程序需要确认(一般通过 fstat)它们正在对什么类型的文件操作。

8.4.1　符号链接

像 SVR3 以及 4.1BSD 等早期的 UNIX 版本仅支持磁盘硬链接。磁盘硬链接很有用，但是也有很多限制。一个磁盘硬链接不能跨越不同的文件系统。而且创建磁盘硬链接可能会受到系统管理员的限制，系统管理员不鼓励创建磁盘硬链接。这是因为此类链接有可能造成目录树的循环，从而使得一些诸如 du 或 find 等使用迭代方法在目录树中遍历的程序造成混乱。

特权用户，可以给目录创建磁盘硬链接，这仅仅是因为早期版本(SVR2 以前的 UNIX)中没有 mkdir 系统调用。为了创建一个目录，用户必须首先调用 mknod 来创建一个目录特殊文件，接着再调用两次 link，为“.”和“..”加入项。这导致很多问题，并且会现竞争，因为三个操作的执行并不是原子性的[Bach 86]。SVR3 加入了系统调用 mkdir 和 rmdir，但是为了向前兼容一些旧的应用程序，该版本仍然允许目录创建链接。

硬链接也会产生控制上的问题.假设用户 X 有一个名为/usr/X/file 的文件。另一个用户想对该文件创建一个磁盘硬链接，并且将其命名为/usr/Y/link1(图 8-6)。为了达到这个目的，Y 用户只需对该路径的目录有执行权限并且对目录/usr/Y 有写权限。随后，用户 X 可能想与 filel 断开链接并且认为该文件已被删除掉了(一般情况下用户并不经常检查他们自己的文件的链接数)。但是该文件却依然通过别的链接而存在。

图 8-6　文件的硬链接

当然，/usr/Y/link1 仍然由用户 X 所拥有，即使该链接是由用户 Y 创建的。如果 X 已经对文件施加了写保护，Y 将不能修改它。然而，X 可能不希望文件长久保存下去。在有磁盘使用定量的系统中，系统却仍然认为用户 X 对该文件负责。而且，X 没有任何方法发现链接的位置，特别是当 Y 对目录/usr/Y 加了读保护时(或者 X 不再知道文件的 i 节点号)。

为了解决上述问题，4.2BSD 引入了符号链接。这个概念很快被大多数供应商所接受，并且在 SVR4 中桩集成到 s5fs 中。系统调用 syslink 创建一个符号联接。符号链接是一个指向其他文件的特殊文件，该类文件可以很容易被辨认出来，因为其属性明显的标明了这一点。很多系统将一些小路径名存储在符号链接的 i 节点中。这种优化策略最初是在 ULTRIX 中引进的。

包含在符号链接中的路径名既可以是绝对路径名，也可以是相对路径名。路径转换程序会识别符号链接，并且将符合链接自动转换到它所指向的文件。如果是相对路径名，那么转换程序会将该链接所在的目录名加到链接前作为一个完整的链接。尽管符号链接对大多数程序是透明的，有些程序还是需要检测并且处理符号链接。系统调用 lstat 可以完成该项工作，lstat 抑制了将符号链接转换为路径名的过程。因此如果 mylink 是一个指向 myfile 的符号链接，那么 lstat(mylink…)返回 mylink 的属性，而 stat(mylink…)返回 myfile 的属性。当使用 lstat 辨认出一个文件是符号链接后，用户可以使用 readlink 来检索该链接的内容。

8.4.2　管道和 FIFO

管道和 FIFO 都是可以提供先入先出数据流的文件抽象符。它们之间的区别在于创建的方法。FIFO 由系统调用 mknod 创建，接着便可以被知道其名字和对其拥有权限的进程打开和访问。除非被显式地删除，否则 FIFO 会一直存在下去。而管道是由系统调用 pipe 创建，返回一个读描述符和一个写描述符。创建管道的进程可以将管道描述符通过 fork 传递给它的子进程，这样两个进程便可以共享同一个管道⑥。一个管道可能有多个读者和多个写者。如果读者和写者都不存在了，内核会自动将管道删除。

管道和 FIFO 的 I/O 操作十分类似。写操作是将数据在后面，而读操作则是将前面的数据移去。一旦数据被读出，它就会被删除掉，而且其他的读者都将读不到该数据。内核定义了一个称为 PIPE_BUF 的参数(缺省值为 5120 字节)用来限制一个管道可以容纳数据的字节数。如果一次写操作会导致管道或 FIFO 溢出，写进程将阻塞，直到有数据被读出而空出一定的空间。如果一个进程试图在一次写操作中写入多于 PIPE_BUF 的数据，内核将不能保持其原子性。这些数据可能同其他进程写入的数据随意交织在一起。

读取操作稍微有些不同。当一个进程要读取的字节数大于管道或 FIFO 中现存的数据时，内核将把管道中的所有数据读出来，将实际读取的字节数返回给调用者。如果其中没有数据，那么读者进程将阻塞在该管道或者 FIFO 上，直到有数据送入为止。O_NDELAY 选项使得管道和 FIFO 处于非阻塞模式下：读和写不阻塞便可完成，并且可以传输任意数目的数据。

管道维护着当前读者和写者的数目。当最后一个写者关闭管道后，内核便唤醒所有正在等待的读者。它们便开始读取管道中的数据。一旦管道空之后，下一个读者进程便得到一个返回值，说明管道已经空了。当最后一个读者关闭管道，内核向阻塞的写者进程发送一个 SIGPIPE 信号。随后的写操作将返回一个 EPIPE 错误(因为已经有一个写者进程正在进行写操作)。

管道的最初实现使用了文件系统，为每一个管道赋予了一个 i 节点和块列表。很多基于 BSD 的 UNIX 变体使用了套接字(sockets，见 17.10.3 小节)来实现管道。SVR4 使用 STREAMS 来建立管道和 FIFO。第 17.9 节描述了其实现的细节。SVR4 的管道是双向的，内核为每一个向都维持了一个单独的数据沉。这样就极大的方便了那些需要全双工以及需要进程间通信机制的应用程序。

8.5 文件系统框架

传统的 UNIX 内核中只有一个文件系统。它只支持一种类型的文件系统——s5fs。在 4.2BSD 中引入的 FFS(快速文件系统)使供应商们多了一种选择，但是其基本框架却使得两个文件系统不能共存。尽管很多人喜欢 FFS 的高性能和特性，其他人却为了向前兼容而保留了 s5fs。然而不论哪种方式都不是很好的方案。

而且，尽管 s5fs 和 FFS 对于一般的分时应用是足够的，很多应用程序开发者却发现它们都不能很好地满足他们某方面的需求。例如，数据库方面的应用就要求操作系统对事务处理有更好的支持。有些应用程序的读取操作占大多数，甚至完全是读取操作，它们希望文件系统能够提供基于内容的分配支持，从而提高顺序读取的性能。如果不彻底修改内核，早期的 UNIX 系统不能使供应商很方便地加入一个用户定制的文件系统。这就限制了 UNIX 在多种不同的环境中的使用。

当然也有一种不断增长的需求，要求 UNIX 系统支持非 UNIX 的文件系统。这样运行在个人计算机上的 UNIX 系统可以访问同一台机器上 DOS 分区以及用 DOS 写入的软盘。

更重要的是，随着计算机网络的出现，在计算机之间共享文件的需求越来越迫切。80 年代中期出现了一些分布式文件系统，例如 AT&T 的远程文件共享(RFS)和 sun 的网络文件系统(NFS)，这些文件系统可以对远程主机上的文件提供透明的访问。

以上的发展要求对 UNIX 文件系统的框架进行彻底的改变以便能支持多种文件系统。与其他情况类似，解决该问题也有许多不同的方案。例如 AT&T 的文件系统开关(file sys-tem switch)[Rifk 86]，Sun 公司的 vnode/vis 体系结构[Klei 86l 以及 DEC 公司酌 gnode 体系结构[Rodr 86]，曾几何时，这些技术为了获得广泛的接受而斗争过。最后，AT&T 把 Sun 的 vnode/vfs 以及 NFS 技术被集成到 SVR4 中，使它们成为事实工业标准。

vnode/vfs 接口主要从它最初的实现演化而来。尽管所有的主要变体都接受它作为基本的结构，每一变体实际上都提供了不同的接口和实现。本章主要集中于 SVR4 中的接口部分。8.11 节介绍了其他的几种实现。

8.6 vnode/vfs 体系结构

Sun 引入了 vnode/vfs 接口作为支持多种文件系统的框架结构。它已被广泛接受并在 SVR4 中成为 System V UNIX 的一部分。

8.6.1 目标

vnode/vfs 体系结构有几个重要的目标：

·该系统应该同时支持几种文件系统类型。其中包括 UNIX 文件系统(s5fs 或 ufs)和非 UNIX 文件系统(DOS，A/UX 等等)。

·不同的磁盘分区可以包含不同类型的文件系统。然而，一旦安装在其他的文件系统上，它们应该和传统的单一文件系统没有区别。用户对整个文件系统的视图应该一致，而意识不到子树在磁盘结构上的差别。

·应该对通过网络共享文件提供完全的支持。访问远程节点上文件系统应该和访问本地节点的文件系统完全一样。

·厂家应该可以开发他们自己需要的文件系统并且以模块方式加入到内核中去。主要目标是在内核里提供一个对文件进行访问和操纵的框架，以及在内核与实现专门
文件系统的模块之间提供一个严格定义的接口。

8.6.2 设备 I/O 的经验

虽然早期的 UNIX 版本只支持一种文件系统类型，却支持许多不同类的文件。除了普通文件外，UNIX 还支持许多通过特殊文件进行访问的设备。虽然每一种设备驱动程序在底层 I/O 实现细节上很不相同，但是我们向用户提供一个一致的文件接口，这样用户就可以像访问普通文件一样访问设备文件。

因此，队设备 I/O 中可以看出支持多种文件系统的必要性。16.3 节详细地描述了设备驱动程序的框架结构。在这里，我们只总结一下与我们的讨论有关的一部分。UNIX 将设备分为字符设备和块设备两种。它们跟内核之间的接口在很多方面都不同；但是基本的框架足相同的。我们在下面都使用字符设备作例子。

UNIX 要求每一种字符设备支持一组标准的操作。这些操作被封装在 cdevsw 结构中，cdevsw 是一函数指针的向量，其结构如下：

```
struct cdevsw{
int (* d_open)();
int (* d_close)();
int (* d_read)();
int (* d_write)();
}cdevsw[];
```

cdevsw[]是结构 cdevsw 的一个全局数组，被称为字符设备开关表。该结构的各个域定义了同抽象字符设备的接口。每一个不同类型的设备都给出实现这个接口的一组函数。例如，行式打印机可能提供函数 lpopen()，lpclose()等等。每一种设备有一个不同的与之联系的主设备号(major device number)。可以以这个设备号为索引，在 cdevsw[]数组中找到它所代表的设备的表项。表项中的各域被初始化为指向由设备所提供的函数。

假设一个用户对一个字符设备文件发出一个读系统调用，在传统的 UNIX 系统中，内核将进行如下的工作：

1.使用文件描述符得到打开文件对象。

2.检查项查看该文件是否被读打开。

3.从该项得到指向内存 inode 的指针。内存 i 节点是文件系统数据结构，用来保存在内存中的活动文件的属性。这将在 9.3.1 小节中详细讨论。

4.锁定 i 节点从而使对文件的访问串行化。

5.检测 i 节点的模式域发现该文件是字符设备文件。

6.用主设备号(保存在 i 节点中)为索引到字符设备开关表中左、检索，得到该没备的 cdevsw 表项。该表项是一个指向实现该设备特殊操作的函数数组。

7.从 cdevsw 中得到指向该设备的 d_read 例程的指针。

8.调用 d_read 操作，根据相应的设备对读请求进行处理。其代码如下：

result=(*(cdevsw[major].d_read))(...);

其中 major 是设备的主要设备号。

9.将 i 节点解锁，返回到用户进程。

正如我们所看到的，上面处理过程的大部分步骤是与设备无关的。从第 1 步到第 4 步以及第 9 步既可应用于普通文件也可应用于特殊设备文件，因此是与文件类型无关的。第 5 步到第 7 步表示了内核和设备之间的接口，它被封装在 cdevsw 表中。所有与特定设备有关的操作都在第 8 步中。

cdevsw 中的像 d_read 之类的数据域定义了一个抽象接口。每一种设备都要对接口用其特定的函数加以实现，例如，在行式打印机中用 lpread()实现，在终端中用 ttread()实现。主设备号作用类似于一把钥匙，将一般的 d_read 函数转化为设备相关的函数。

同样的原则可以用于解决多文件系统支持的问题。我们需要将文件子系统代码分为文件系统相关代码和文件系统无关代码两部分。这两部分之间的接口由一组函数定义，文件系统无关代码可以使用这些函数对文件进行各种操作。与文件系统相关的代码随文件系统类型不同而不同，它主要是对这些接口进行定义。这个框架提供了一些机制，使得用户可以加入新的文件系统，并且可以将抽象操作转化为与被访问文件相关的特定函数。

面向对象设计

vnode/vfs 接口的设计利用了面向对象的编程概念。这些概念现在已被广泛用于 UNIX 内核的其他部分，比如内存管理，基于消息的通信，进程调度等等。大致回顾一下面向对象的基本原则可以更好地了解这些概念是怎样在 UNIX 内核中被应用的。尽管这些技术更加适合于像 C++之类的面向对象语言，但设计者仍然选择 C 来实现，以便使文件系统与内核的其他部分协调地工作。

面向对象方法是建立在类和对象概念之上的。类是包含有数据成员域和函数成员集的复杂数据类型。对象是类的实例，并且为该类分配了存储空间。类的成员函数能对类中的对象(数据和函数)进行操作。类中的成员(包括数据和函数)既可以是私有的，也可以是公有的。只有公有数据才对类用户外部可见。私有数据和函数只能被类中的其他函数所使用。

从一个类可以得到其一个或多个派生类，也叫子类(见图 8-7)。同样子类也可以再派生出它自己的子类，这样就形成了类的层次树。子类从基类中继承了所有的属性(包括数据和函数)。同时，子类也可以在类宁加上自己所需要的数据域和函数。子类还可以对父类中的某些函数进行重载，并给出其实现。

因为子类中包含了基类的所有属性，所以一个子类的对象也是其基类的对象。例如，目录类是基类文件类的派生类，这意味着每一个目录也是一个文件。当然反过来是不成立的——一个文件不是一个目录。这样，指向目录对象的指针也是一个指向文件对象的指针。因为子类中新加上的属性对基类是不可见的，所以指向基类对象的指针不能访问于类对象中子类专有的信息。

我们经常使用基类来表达一些抽象内容以及定义一些抽象接口，而使用派生类来对成员函数给出它自己的实现。比如，文件类中可能定义了一个 create 函数，但当用户对任一类型的文件调用谊函数时，我们希望根据该文件的具体类型来调用不同的例程，这是因为该文件既可能是常规文件，也有可能是目录，符号链接或者设备文件等等，创建这些文件的过程绝不可能是统一的。像 create()这样的函数称为纯虚函数(pure virtual functions)。

面向对象的语言提供了这种设施。例如，在 C++语言中，我们将其中有至少一个纯虚函数的类定义为抽象基类。因为在基类中并没有纯虚函数的实现部分，所以它不能够被实例化，而只能用来派生子类，而子类则会对纯虚函数给出具体实现。所有的对象都是子类的实例，但用户却可以使用一个指向基类的指针来访问它们，而不用知道它究竟属于哪一个子类。在

这样的对象中，当一个虚函数被调用时，实现会根据该对象实际属于的予类来决定调用哪一个函数。如前所述，像 C++和 SmallTalk 之类的语言内嵌了可以描述类和虚函数等概念的结构。然而 C 语言对这些概念的支持却近乎为零，用户必须自己采用手段来支持。下面，我们将可以看到 vnode/vfs 层是如何以面向对象的方式实现的。

图 8-7　基类与其子类间的关系

8.6.3　vhode/vfs 接口概述

v 节点(虚拟节点)代表 UNIX 内核中的一个文件；vfs(虚拟文件系统)代表内核中的一个文件系统，它们都可被视为抽象基类。我们可以从中派生出不同的子类，支持不同的文件系统，比如 s5fs，ufs，NFS 和 FAT(MS-DOS 文件系统)等等。

图 8-8 所示的是 SVR4 中的 v 节点类。v 节点基类的数据域包含了不依赖于具体文件系统类型的信息。其成员函数可以分为两部分，第一部分是一组定义文件系统相关接口的虚函数，每一种不同的文件系统对这些函数可能都要给出不同的实现。第二部分是一组可被内核其他子系统用来操纵文件的高级应用例程。这些函数然后再调用文件系统相关例程完成底层任务。

v 节点基类有两个域必须要在子类中加以实现。第一个 v_data，这是个指向私有数据结构的指针(caddr_t 类型)，其中包含有文件系统相关的 v 节点数据。对于 s5fs 和 vfs 文件来说,这个结构就是传统 s5fs 和 ufs 中的 i 节点结构。在 NFS 中使用了 rnode 结构,在 tmpfs(见 9.10.2)中使用了 tmpnode 结构，等等。因为这个结构是通过 v_data 间接访问的，它对基类 v 节点来说是不可见的，它的域只对具体的文件系统的内部函数才是可见的。

v_op 域指向一个结构 vnodeops，该结构中包含子一组实现 v 节点的虚拟接口的函数指针。当 v 节点被初始化时，v_data 和 v_op 域都被赋以初值，当系统调用 open 和 create 被执行时尤为如此。当文件系统无关的代码针对某一类型 v 节点调用某一虚函数时，内核会间接调用 v_op 并调用相应的文件系统实现的函数。例如，操作 VOP_CLOSE 允许调用者关闭与 v 节点相关连的文件。这可以通过如下的宏来实现：

图 8-8　v 节点抽象

```
#define VOP_CLOSE(vp, ...) (*((vp)->v_op->vop_close))(vp, ...)
```

其中省略号代表 close 例程的其他参数。当 v 节点被正确地初始化后，该宏便可以确保当施加 VOP_CLOSE 操作时会对 ufs 文件调用 ufs_close()例程，而对一个 NFS 文件施加同样的操作却调用 nfs_close()例程，等等。

类似的，基类 vfs 也是通过两个域 vfs_data 和 vfs_op 而将数据和函数与实现特定文件系统的数据和函数链接起来。图 8-9 中显示了 vfs 抽象的各个组成部分。

在 C 中，基类是由一个结构加上一组定义公共非虚函数的全局内核函数(或者宏)组成。基类中有一个指向另一个结构的指针，该结构中包含有一组指向虚函数的指针。v_op 和 v_data(对 vfs 类来说是 vfs_op 和 vfs_data)可以将子类链接起来，从而可以在运行时访问文件系统相关的函数和数据。

8.7　实现概述

下面，我们将对 vnode/vfs 接口作更加详细的讨论，并且介绍这个接口是如何实现不同文件操作的。

8.7.1　目标

提供一个可被许多不同种类的文件系统灵活使用的接口，必须达到以以下目标：

·每一个操作必须由当前进程执行，如果在执行过程中由于函数等待资源和事件而被阻塞时，该操作进入睡眠状态。

·某些操作可能需要将对文件的访问串行化。这些操作可能将文件系统相关层的数据结构锁定，并且在操作结束之前解锁。

·接口必须是无状态的。不允许使用像 u 区那样的全局变量在操作之间传递状态信息。

·该接口必须是可重入的，这要求不能使用像 u_error 和 u_rvall 之类的全局变量来保存错误代码和返回值。实际上，所有的操作都将错误代码作为返回值返回。

·文件系统的实现可以使用像高速缓存之类的全局资源，但是不一定非要如此。

·在一远程文件系统中，为了满足客户端的请求，该接口在服务器端必须是可用的。

·坚决避免使用定长的静态表。应该尽量使用动态存储分配。

图 8-9　vfs 抽象

8.7.2　v 节点和打开文件

v 节点是内核中的一个活动文件的基本抽象。它定义了对文件的接口，并且将所有对文件的操作定向到相应的特定文件系统函数上。内核有两种方法访问一个 v 节点。首先，就像本节所讲的那样，与 I/O 相关的系统调用通过文件描述符来定位 v 节点。其次，路径名遍历例程使用文件系统相关的数据结构来查找 v 节点。8.10.1 小节更加详细地时论了路径转换。

在读写文件之前，进程必须首先要打开文件。系统调用 open 返回给用户一个文件描述符。文件描述符通常是一个小整数，它充当打开文件的句柄，代表了对该文件的一次独立的会话(或者流 stream)。在随后的 read 和 write 调用中，进程必须将该描述符作为参数调用这些函数。

图 8-10 中给出了有关的数据结构。文件描述符是一个进程级对象，其中包含有指向一个打开文件对象的指针和一组每个描述符都需要的标志。目前支持的标志有 FCLOSEXEC 和 U_FDLOCK，前者的作用是当进程调用 exec 时要求内核关闭描述符，后者用于文件加锁。

打开文件对象中有为管理文件的一次会话所需要的上下文。如果多个用户打开同一个文件(或者一个用户多次打开同一个文件)，每一个用户都有他自己的打开文件对象。打开文件对象中主要有如下数据：

图 8-10　文件系统无关数据结构

·下一次读写开始的位置在文件中的偏移量。

·指向该文件打开对象的文件描述符的引用次数。通常该值为 1，但如果使用 dup 或 fork 复制描述符的话，此值会大于 1。

·指向文件 v 节点的指针。

·文件打开方式。每一次 I/O 操作时内核都要检查这个方式。因此如果用户以只读方式打开一个文件得到一个描述符，即使他对该文件有写权限，也不能用这个描述符对文件进行写操作。

传统 UNIX 系统使用存放在 u 区的一个静态的，大小固定的文件描述符表。返回给用户的文件描述符是这个表的索引。显而易见，这样的表会限制用户同时打开文件的最多数目{一般是 64 个)。在现代 UNIX 系统中，文件描述符表在大小上没有限制，而是可以任意扩展⑦(其实仍然有限制，受到 RLIMIT_NOFILE 资源的限制)。

在一些系统的实现中，文件描述符是以有 32 个表项的块的形式被分配的，这些块被保存在一个链接的列表中，其中链表头在 u 区中，像 SVR4 和 SunOS 就是这样。这使得对描述符间接引用的任务变得更加复杂了。这是因为内核必须首先找到合适的块，然后再去索引这个块，而不是直接使用文件描述符作为索引去查找。这种方法的优点是取消了对一个进程同时打开文件的数目的限制，其代价却是增加了代码的复杂性，影响了性能。

一些新的基于 SVR4 的系统动态地分配描述符表。每当需要时就可以调用 kmem_real-loc() 来扩展它。扩展的方式有两种，一种是就地扩展，另一种是将表复制到一个更大的能扩展的内存空间去。这样，描述符就可以动态增长并且可以快速转换描述符，不过其代价是在分配时必须首先将描述符表复制。

8.7.3　v 节点

v 节点的数据结构表示如下：

```
struct vnodo {
u_short v_flag;     /* V_ROOT, etc. */
u_short v_count;     /* reference count */
struct vfs *vfsmountedhere; /* for mount points */
struct vnodeops *v_op;  /* vnode operations vector */
struct vfs *vfsp;     /*  file system to which it belongs  */
struct stdata *v_stream;  /* pointer to associated stream, if any */
struct page *v_page;     /* resident page list */
enum  vtype v_type;     /* file type */
dav_t  v_rdev;     /* device ID for device files */
ceddr_t v_data;     /* pointer to private data structure */
};
```

下面，我们将更加详细地对每个数据域加以描述。

8.7.4 v 节点引用计数

v 节点中的数据域 v_count 保存了一个称为引用计数的值，它可以决定 v 节点必须要在内核中呆多久。当文件第一次被访问时，内核便分配一个 v 节点并将其赋给文件。这样，其他对象可以保存一个指向该 v 节点的指针或引用，并且可用该指针或引用来对 v 节点进行访问。这就意味着只要有这样一个指针或者引用存在，内核就必须保留 v 节点，并且不能将其再分配给其他文件。

引用计数是 v 节点的一个通用属性，由文件系统无关代码操作。使用两个宏 VN_HOLD 和 VN_RELE 可以分别增加或者减少引用计数。如果计数减到零，文件会变为失效，同时内核便可以将 v 节点删除或者将其再分配给其他的文件。

把引用(或保持)同加锁区分开是很重要的。锁定一个对象可以以某种方式阻止别人访问它，具体什么方式要视是排斥性加锁还是读，写加锁而定。而保持(holding)对一个对象的引用仅仅足为了确保对象的持久性。文件系统相关代码会将 v 节点锁定一小段时间，特别是在单个 v 节点操作期间。而引用则有可能会保持很长时间，不仅覆盖几个 v 节点操作，还有可能覆盖多个系统调用。下面是可以得到 v 节点的引用的操作：

·打开一个文件便得到对 v 节点的引用(增加引用计数)。关闭文件释放引用(将引用计数减少)。

·一个进程会保持对其当前工作目录的引用。当进程改变当前工作目录时，它将得到新目录的引用，并且将旧目录的引用释放掉。

·当一个新的文件系统被安装时，该新文件系统得到安装点目录的引用。当卸载文件系统时会将引用释放掉。

·路径名遍历例程对在其中间经过的每一个中间目录都会获得对其的引用。当搜寻目录时，它保持引用，而当它得到下一个目录的引用后便将该引用释放掉。

引用计数可以确保 v 节点和底层文件的持久性。当一个进程试图删除另一个进程(有可能是同一个进程)正在打开的文件时，该文件并没有被物理地删除，只是在目录表项中对应于该文件的项被删除掉。这样，其他进程便不能访问该文件。但是由于这个文件的引用计数不为零，所以它依然存在。当前正在打开该文件的所有进程仍然可以继续访问它，直到它们将文件关闭。这就相当为文件加上了删除的标记。当对该文件的最后一个引用被释放后，文件系统无关代码便会调用 VOP_INACTIVE 操作，完成文件删除操作。在 sSfs 和 ufs 文件中，i 节点和数据块便是在此时被删除的。

这个特性在创建临时文件时非常有用，像编译器之类的程序需要用一些临时文件来保存

中间状态。如果程序正常终止，这些临时文件应该被删除掉。应用程序可以通过先打开文件然后马上将其切断链接(unlink)的方法做到这一点。文件的链接计数变为零，内核会将其目录项删掉。这样就可以防止其他用户看到并试图访问这些文件。因为内存索引计数为 1，所以该文件仍然存在，应用程序仍然可以对其进行读写操作。当程序关闭文件，或者当进程显式或非显式地终止后，该文件引用计数变为零。内核完成文件删除操作，将其数据块和 i 节点释放掉。很多 UNIX 系统有一个叫做 tmpfile 的标准库函数，可以用来创建临时文件。

8.7.5 vfs 对象

vfs 对象(vfs 结构)表示一个文件系统。内核为每一活动文件系统分配一个 vfs。其数据结构如下：

```
struct vfs{
struct vfs * vfs_next;    /* next VFS in list */
struct vfsops * vfs_op;    /* operations vector */
struct vnode * vfs_vnodecovered;    /* vnode mounted on */
int vfs_fstype;    /* file system type index */
caddr_t vfs_data;    /* private data */
dev_t vfs_dev;    /* device ID */
};
```

图 8-11 显示了在一个有着两个文件系统的系统中 v 节点对象和 vfs 对象之间的关系。第二个文件系统安装在根文件系统的/usr 目录下。全局变量 rootvfs 指向一个由所有 vfs 对象组成的链表的头部，其中根文件系统的 vfs 位于该链表的头部，数据域 vfs_vnodecovered 指向第二个文件系统的安装点。

图 8-11 v 节点与 vfs 对象间的关系

每一个 v 节点的 v_vfsp 域都指向它所属的 vfs。每一个文件系统的根 v 节点有 VROOT 标志。如果一个 v 节点是一个安装点,那么它的 v_vfsmountedhere 域指向安装在它上面的文件系统的 vfs 对象。注意根文件系统并不安装在任何地方，因此也不覆盖任何 v 节点。

8.8 文件系统相关对象

在本节，我们将描述 vnode/vfs 接口中的文件系统相关对象，同时会看到文件系统无关层是怎样访问这些对象的。

8.8.1 每个文件的私有数据

v 节点是一个抽象对象，它不能单独存在，总是在特定文件的上下文环境中被实例化。文件所属的文件系统为抽象 v 节点接口给出它自己的实现 oY 节点中的 v_op 和 v_data 域将 v 节点跟该节点的文件系统相关部分联系起来。v_data 指向一个保持有文件的文件系统相关信息的私有数据结构。具体使用什么的数据结构依赖于文件所属的文件系统的类型。例如，在 ufs 和 s5fs 中使用 i 节点结构⑧，在 NFS 中使用 modes 等等。

v_data 是一个不透明的指针，这意味着文件系统无关代码不能直接访问与文件系统相关的对象。而文件系统相关代码却可以访问并且也的确要访问基类 v 节点对象。因此我们需要一种可以通过私有数据对象找到 v 节点的方法。由于这两个对象总是在一起分配的，将它们合在一起会效率更高。在 v 节点层的参考实现中，v 节点仅仅是文件系统相关对象的一部分。请注意，这仅仅是一个比较流行的实现方法。只要 v_data 部分能正确地被初始化，将 v 节点和文件系统相关部分数据结构分立开来将更好。图 8-12 展示了上面所说两种不同的实现方式。

8.8.2 vnedeop,s 向量

v 节点接口定义了在一般文件上的一组操作，文件系统无关代码只能通过这些操作来拱

纵文件，它不能直接访问文件系统相关对象。结构 vnodeops 定义了这个接口。它可以描述如下：

```
struct vnodeops{
    int (* vop_open)();
    int (* vop_close)();
    int (* vop_read)();
    int (* vop_write)();
    int (* vop_ioctl)();
    int (* vop_getattr)();
    int (* vop_setattr)();
    int (* vop_access)();
    int (* vop_lookup)();
    int (* vop_create)();
    int (* vop_remove)();
    int (* vop_link)();
    int (* vop_rename)();
    int (* vop_mkdir)();
    int (* vop_rmdir)();
    int (* vop_reacldir)();
    int (* vop_symlink)();
    int (* vop_readlink)();
    int (* vop_inactive)();
    int (* vop_rwlook)();
    int (* vop_rwunlock)();
    int (* vop_realvp)();
    int (* vop_getpage)();
    int (* vop_putpage)();
    int (* vop_map)();
    int (* vop_poll)();
};
```

每一个文件系统对上述接口都以它自己的方式加以实现，并提供一组函数。例如，在 ufs 中，VOP_READ 操作是通过从本地磁盘读取数据来实现的，而在 NFS 中是通过向远程文件服务器发送请求以得到数据来实现的。因此，每一个文件系统都为结构 vnodeops 提供一个实例。例如在 ufs 中，对象定义如下：

```
struct vnodeops ufs_vnodeops {
ufs_open,
ufs_close,
...
};
```

v 节点的 v_op 域指向相关的文件系统的 vnodeops 结构。就像图 8-13 所示，同一个文件系统的所有文件共享该结构的同一个实例，并且访问同一组函数。

图 8-13　文件系统相关 v 节点对象

8.8.3　vfs 层中的文件系统相关部分

就像 v 节点那样，vfs 对象也有指向它自己私有数据和操作向量的指针。vfs_data 指向一个不透明的，为每个文件系统配制的数据结构。同 v 节点不同的是，vfs 对象和它的私有数据结构通常是分别分配的。vfs_op 指向一个称为 vfsops 的结构，该结构描述如下：

```
struct vfsops{
int (* vfs_mount)();
int (* vfs_unmount)();
int (* vfs_root)();
int (* vfs_statvfs)();
int (* vfs_sync)();
};
```

每一种文件系统都为这些操作给出它们自己的实现。因此每种文件系统都有与该文件系统相关的一个 vfsops 结构的实例。例如，ufs 的实例是 ufs_vfsnode，NFS 的实例是 nfs_vf-sops。图 8-14 所示为有两个 ufs 和一个 NFS 的系统的 vfs 层数据结构。

8.9 安装一个文件系统

为了支持现有的多种文件系统类型，vfs 接口的实现必须修改系统调用 mount。在 SVR4 中的语法如下：

```
mount(spec, dir,flags, type, dataptr, datalen);
```

其中，spec 是表示文件系统的设备文件名，dir 是安装点目录路径名，type 是一个字符

图 8-14 vfs 层数据结构

串,用来指定它是何种文件系统,dataptr 是一个指向附加文件系统相关参数的指针,datalen 是这些附加参数的总长度。在本节中，我们看一下内核是如何实现系统调用 mount 的。

8.9.1 虚拟文件系统转换

为了正确地将 v 节点和 vfs 的操作定向到对应的特定文件系统的实现上，必须正确的配置系统上的每一个文件系统。内核需要一种机制来决定怎样访问每个文件系统的接口函数。

SVR4 使用了虚拟文件系统开关表，这是一个全局表，其中每一文件系统类型都在里面有一个表项。每一个表项内容如下：

```
stlruct vfssw{
char * vsw_name;    /* file system type name */
int (* vsw_init)();    /* address of initialization routine */
struct vfsops * vsw_vfsops;  /* vfs operations vector for this fs */
}vfssw[];
```

8.9.2 mount 的实现

mount 系统调用通过调用 lookuppn()来获得安装点目录的 v 节点。系统调用首先确认这个 v 节点代表一个目录，并且没有其他的文件系统在该点安装(注意，loopuppn()会得到对该目录的一个引用，这个引用直到文件系统被卸载后才被删除)。接着 mount 会到 vfssw[]表去找与 type 名字相应的表项。

找到相应的转换表项后，内该调用它的 vsw_init 操作。该操作会调用一个特定文件系统的初始化例程，用来分配文件系统所需要的数据结构和资源。接着，内核会分配一个新的 vfs 结构并且初始化：

1.将该结构加入到以 rootvfs 为头部的链表中去。

2.设置 vfs_op 域，使其指向在开关表项中指定的 vfsops 向量。

3.设置 vfs_vnodecovered 域，使其指向安装点目录的 v 节点。

接着内核将指向 vfs 结构的指针存放到被覆盖的目录的 v 节点的 v_vfsmountedhere 域

中。最后，内核调用虚拟文件系统的 vfs_mount 操作，完成文件系统相关的 mount 调用处理过程。

8.9.3 VFS_MOUNT 处理

每一种文件系统都为 VFS_MOUNT 操作提供自己的实现函数。这个函数必须完成如下的操作：

1.确认执行该操作所需要的权限。

2.分配并且初始化文件系统的私有数据对象。

3.将指向该私有数据对象的指针存放到 vfs 对象的 vfs_data 域中。

4.访问文件系统的根目录并在内存中初始化其 v 节点。内核访问被安装文件系统的根目录的唯一方法是通过 VFS_ROOT 操作。vfs 的文件系统相关部分必须要保存定位根目录所需的必要信息。

一般情况下,本地文件系统通过从磁盘上读取文件系统元数据(像 s5fs 中的超级块)来实现 VFS_MOUNT，而分布式文件系统可能通过向远程文件服务器发送请求来实现。

8.10 对文件的操作

在本节，我们将介绍在 vnode/vfs 设计中几个重要的文件操作是如何处理的，其中包括路径转换和系统调用 open 和 read。

8.10.1 路径名遍历

文件系统无关函数 Lookuppn()对一个文件路径名进行转换，并返回一个指向所需文件的 v 节点指针。同时也得到对该 v 节点的一个引用。搜索的起点要视路径名是相对还是绝对路径名而定。如果是相对路径名，lookuppn()从当前目录开始，从 u 区的 u_cdir 域中获得指向当前目录的 v 节点的指针。如果是绝对路径名，则从根目录开始。它的 v 节点指针在全局变量 roodir 中⑨。

lookuppn()首先保留住起始 v 节点(增加它的引用计数)，接着执行一个循环，每次解析路径名的一个分量。循环的每一次迭代都要执行如下的任务：

1.确认 v 节点的确代表一个目录(除非已经到达最后一个分量)。v 节点的 v_type 域中包含有该信息。

2.如果分量是“..”并且当前目录是系统根目录，那么直接移向下一个目录。根目录的父目录就是它自己。

3.如果分量是“..”并且当前目录是一个安装文件系统的根目录，那么应该去访问安装点目录。所有的根目录都有 VROOT 标志，v_vfsp 指向那个文件系统的 vfs 结构，在该结构的 vfs_vnodecovered 中有一个指向安装点的指针。

4.调用在这个 v 节点上的 VOP_LOOKUP 操作。这将会调用该特定文件系统的 lookup 函数(s5lookup()，ufs_lookup()等等)。这个函数在目录中搜索当前分量，如果找到了，则返回一个指向那个文件的 v 节点的指针(如果 v 节点没有在内核中，需要先分配一个)。这也需要保留 v 节点。

5.如果在当前目录中没有找到分量，位查一下该分量是不是最后一个分量。如果是，返回成功(调用者可能想创建一个文件)，并且在不释放对父目录的情况下，将指向父目录的指针返回。否则返回 ENOENT 错误。

6.如果新分量是一个安装点(v_vfsmountedhere != NULL)，根据指针得到安装文件系统的 vfs 对象，并且调用 vfs_root 操作，将该文件系统的根 v 节点返回。

7.如果新的分量是一个符号链接(v_type＝VLNK)，调用它的 VOP_SYMLINK 操作将符号链接加以转换。将路径名的其他部分扩展到符号链接的后面，重新开始迭代。如果链接包含有绝对路径名，则解析必须从系统根目录开始。lookuppn()的调用者可能传入一个参数，抑制

将路径名的最后一个分量转换为符号链接。这主要是为了适应一些像 lstat 之类的系统调用，如果路径名的最后一个分量是符号链接，这些系统调用不希望转换该符号连接。还有，一个叫做 MAXSYMLINKS 的全局参数(一般被设为 20)会限制在一次 lookuppn()调用中符号链接的最大数目。这样可以防止函数陷入无限循环，比如当/x/y 是/x 的符号链接的时候就会出现这种情况。

8.将刚刚搜索过的目录释放掉。对目录的保持可以通过 VOP_LOOKUP 操作得到。对于起始点，可以显式地通过 lookuppn()得到。

9.最后，返回循环顶部，开始在由新的 v 节点代表的新的分量中搜索。

10.当所有的分量都用完，或者其中一个分量没有找到，搜索过程将终止。如果搜索成功，不要将最后的 v 节点的保持释放，将指向它的一个指针返回给调用者。

8.10.2 目录查找缓存

目录名字查找高速缓存是一个中心资源，可以被所有想使用它的文件系统实现使用。它包含一个 LRU(最近最少使用顺序)的对象缓存，其中的内容有三部分，目录的 v 节点指针，目录中的文件名以及指向该文件的 v 节点的指针。该缓存以基于文件所在的目录和文件名的哈希表组织它的表项，这样就可以快速查找某一表项。

如果文件系统使用了名字缓存，其 lookup 函数(实现 VOP_LOOKUP 操作的函数)首先在名字缓存中查找所需要的文件名。如果找到了，该函数将对其 v 节点的引用数增加 1，将 v 节点返回给调用者。这样就避免了目录搜索，从而可以节省几次磁盘读取的时间。因为编程者通常对经常使用的文件和目录一连发出几次请求，所以在名字缓存中的查找命中率是很高的。如果找不到，查找程序就搜索目录。如果分量在该目录中找到后，该函数便在缓存中增加一个表项以备后用。

因为文件系统无需遍历路径名就可以访问 v 节点，它可能进行一些操作，使高速缓冲项变为无效。例如，用户可能断开与一个文件的链接，内核在以后有可能将该文件的 v 节点赋给其他的文件。如果事先没有准备，在其后，可能需要搜索这个旧文件，搜索程序就有可能找到错误的项，从而把不属于该文件的 v 节点返回来。因此缓存必须提供一种手段来检查其中的项是否有效。4.3BSD 和 SVR4 都实现了目录查找缓存，它们使用了不同技术来解决这个问题。

4.3BSD 没有使用 vnode/vfs 接口，它的名字查找高速缓存直接找到文件的内存 i 节点。i 节点有一个代号，每当 i 节点被赋给不同的文件时该代号便增加一个值。名字查找缓存是基于线索的。当往高速缓存里增加一个表项时，文件系统将文件的 i 节点代号复制到高速缓存项中。高速缓存查找程序将该数字同文件 i 节点的代号比较。如果这两个数字不同，表明表项是无效的，也就是在高速缓存没有 i 节点。

在 SVR4 中，高速缓存中保留了被缓存文件 v 节点的引用，当项被刷新或赋给别的 v 节点时，该保持被释放。尽管这种方法可以保证缓存项一直是有效的，也有缺点。比如，内核可能会保留一些不活动的 v 节点，仅仅因为在名字高速缓冲中有它们的一项。而且，这种方法也使得内核的其他部分不能独占文件和设备。

8.10.3 VOP_LOOKUP 操作

VOP_LOOKUP 是文件系统特定函数的一个接口，用于在一个目录中查找一个文件名分量。它由一个宏调用，该宏形式如下：

```
errer=VOP_LOOKUP(vp，compname，&tvp，...);
```

其中，vp 是指向父目录 v 节点的指针。compname 是分量名。如果成功返回，tvp 必须指向 compname 的 v 节点，且其引用计数(reference count)必须增加。

像该接口的其他操作一样，上述宏将调用特定文件系统的 lookup 函数。一般地该函数首先在名字查找高速缓存中查找，如果找到了，它将引用计数增加，将指向 v 节点的指针返回。

如果没有找到，它在文件所在的父目录中搜索该名字分量。本地文件系统一块一块地遍历所有目录项完成搜索功能，而分布式文件系统则向服务器节点发送搜索请求。

如果目录中包含有分量的一个合法匹配，查找函数检查一下该文件的 v 节点是否已经在内存中(每一种文件系统都有它自己的跟踪内存对象的方法)。例如，在 ufs 中，目录搜索导致产生一个 i 节点号，ufs 使用该 i 节点号到一个哈希表去搜索 i 节点。内存中 i 节点包含有 v 节点。如果发现 v 节点在内存中，查找函数增加其引用次数，将它返回给调用者。

有时候目录搜索会产生对分量的匹配，但是 v 节点不在内存中。查找函数必须分配一个 v 节点，并且将其和文件系统相关的私有数据结构初始化。一般情况下，v 节点是文件系统相关的私有数据结构的一部分，因此它们可以一起分配内存。这两个对象(v 节点和私有数据结构)是通过读取文件属性而被初始化的。查找函数将 v 节点的 v_op 域指向该文件系统的 vnodeops 向量，然后将对该 v 节点增加一个保持。最后查找程序在目录名字查询高速缓存中增加一项，将其放入到缓存中的 LRU 列表的最后。

8.10.4 打开文件

系统调用 open 的参数包括一个路径名，一组标志，以及文件被创建时赋予该文件的权限。标志位包括 O_READ，O_WRITE，O_APPEND，O_TRUNC(截断到零长度)，O_CREAT(如果文件不存在则创建)，和 O_-EXCL(该标志和 O_CREAT 一同使用，如果文件存在则返回一个错误信息)。open 的实现几乎都是在文件系统无关层进行的。其算法如下：

1.分配一个文件描述符(见 8.2.3 小节)，如果 open 成功返回，其返回值是文件描述符在块列表中的标识。

2.分配一个打开文件对象(struct file)，将指向它的指针存放在文件描述符中。SVR4 动态地分配该对象，而较早的实现中使用的是静态的、固定长度的表。

3.调用 lookuppn()遍历路径名，返回指向要打开文件的 v 节点的指针。lookuppn()同时也返回一个指向文件目录的 v 节点的指针。

4.检查 v 节点(通过调用 VOP_ACCESS 操作来实现)以保证调用者有权限对文件进行所需要的操作。

5.检查并且拒绝一些非法的操作。例如试图以写方式打开目录文件和活动的可执行文件(否则，执行该程序的用户将得到错误的结果)。

6.如果文件不存在，检查是否有 O_CREAT 选项。如果有，调用在父目录上的 VOP_CREATE 操作来创建文件。否则，将返回错误代码 ENOENT。

7.调动那个 v 节点的 VOP_OPEN 操作，进行文件系统相关处理。一般来说，这个例程并不干任何事情，但是有些类型的文件系统希望使用此函数在该点上执行一些附加任务。例如，处理所有设备文件的 specfs 文件系统，可能想在此函数中调用设备驱动器的 open 例程。

8.如果指定了 O_TRUNC 选项，调用 VOP_SETATTR 将文件大小设为 0。文件系统相关代码将进行必要的清除工作，比如释放文件的数据块。

9.初始化打开文件对象。将指向 v 节点的指针和打开方式标志存放在里面，将其引用计数设为 0，将偏移指针设置为 0。

10.最后，将文件描述符在描述符表中的索引返回给用户。

注意 lookuppn()增加 v 节点的引用计数，同时将其 V_OP 指针初始化。这保证了随后的系统调用可以用文件描述符来访问文件(v 节点将保留在内存中)，并且将会正确地调用文件系统相关的函数。

8.10.5 文件 I/O

为了对文件进行 I/O 操作，必须首先打开文件，然后再在 open 调用返回的文件描述符上调用 read 和 write 操作(对于分散-聚集 I/O 调用 readv 和 writev 操作)。文件系统无关代码把这些参数全放到一个称为 uio 的结构中(见 8.2.5 小节)。在 read 和 write 系统调用中,uio

将指向一个单独元素的 iovec[]数组。内核使用文件描述符来找到打开文件对象，检测文件是否按所需的访问类型被打开。如果是内核间接引用打开文件对象中的 v 节点指针来找到 v 节点。

UNIX 语义要求对一个文件的 I/O 操作串行化。如果两个用户同时对一个文件调用 read()和 write()，内核必须先完成一个操作，然后再去完成另外一个。因此内核在 read 和 write 之前首先锁定 v 节点，I/O 操作完成后再将其解锁。SVR4 使用新的 VOP_RWLOCK 和 VOP_RWUNLCR2K 操作来做到上述这一点。最后，内核调用 VOP_READ 和 VOP_WRITE 来进行文件系统相关操作。系统调用 read 和 write 的大多数操作都是在文件系统相关层来完成的。在 9.3.3 和 10.6.3 小节中，我们将在 s5fs 和 NFS 中更加的详细地说明这一点。在 SVR4 中，文件 I/O 与虚拟内存子系统密切相关。它们两者的关系将在 14.8 节中更详细地加以讨论。

8.10.6 文件属性

一些系统调用可以修改或者查询像文件的所有者 ID 权限之类的特性(见 8.2.2 小节)。在早期的 UNIX 版本中，这些系统调用直接读写内存 i 节点，或者有可能的话，将它们复制到磁盘 i 节点中去，所有这些都是以实现的方式进行的。因为 v 节点接口可以处理任意-种类型的文件系统，而这些文件系统在内存中和在磁盘上的元数据存储类型的结构是大相径庭的，所以 v 节点接口实际上提供了一个通用的接口。

VOP_GETATTR 和 VOP_SETATTR 分别读写文件的属性。这两个操作都使用了名为 struct vattr 的文件系统无关对象。尽管这个结构中的数据大部分可以在 s5fs 或者 ufs 的 i 节点中找到，其格式是很通用的，而不局限于哪一种特殊的文件系统类型。每一种特定的实现要将该一般结构信息转换为它自己的元数据结构。

8.10.7 用户凭证

有些文件系统操作需要检查调用者是否有权限进行所需的文件访问操作。像这样的一些访问控制一般是由用户 ID 和调用者的组 ID 决定,在传统 UNIX 中这些一般是存放在调用进程的 u 区中，在现代 UNIX 系统中，这些信息被封装在一个称为凭证的对象中(structcred)，该对象一般以显式的形式被传到(通过指针)大多数的文件操作中去。

每一个进程都有一个静态分配的凭证对象，此对象一般存在进程的 u 区或 proc 结构中，对本地文件的操作，我们将一个指针传向这个对象。这看来跟早期的直接从 U 区中获取信息没有区别。但是新方法的优点在于处理远程文件的操作，这些操作一般是由服务器代表远程客户执行的。在这里，权限是由用户凭证对象决定的，而不是由服务器进程的凭证对象决定的。因此服务器可以为每一个客户请求动态分配一个凭证结构，并且用客户 UID 和 GID 将其初始化。

因为这些凭证对象是从一个操作传向另一个操作，并且一直保留到操作结束，所以内核给每一个对象增加一个引用次数的属性，当引用次数降至零时，内核将结构释放掉。

8.11 分　析

vnode/vfs 接口提供了一个功能强大的编程范例。它允许多种不同类型的文件系统并存于同一台机器上。供应商可以以模块方式向内核添加文件系统。面向对象的框架结构可以有效地将文件系统同内核的其余部分分离开。这导致了几种有趣的文件系统的实现。在一个典型的 SVR4 系统中经常安装的文件系统有：

s5fs　最初 System 5 文件系统

ufs　Berkeley 的快速文件系统，被改编以适应于 vnode/vfs 接口的文件系统

vxfs　Veritas 日志文件系统，该文件系统有几个高级特性

specfs　用于特殊设备文件的文件系统

NFS　网络文件共享文件系统

RFS　远程文件系统

fifofs　先入先出文件系统

/proc　该文件系统将每一个进程当作一个文件

bfs　启动文件系统

有些变体，比如说 Solaris，也支持 MS-DOS 的 FAT 文件系统。这在用软盘从 DOS 机移入移出文件时非常有用，下面几章将更加详细地描述几种文件系统。

现在已集成到 SVR4 中的 SUNOS 的 vnode/vfs 接口已经获得了广泛的接受。然而，必须要看到其缺点以及其他 UNIX 变体是如何改进这些缺点的。其缺点主要是路径名查找算法。本节剩下的内容将讲述一下这些缺点及解决了这些问题的其他一些变体。

8.11.1 SVR4 实现的缺点

一个主要的性能问题是 lookuppn()每次只转换路径名的一个分量。因此为路径中的每个分量都要调用文件系统相关的 VOP_LOOKUP 函数。这不仅会导致过多的函数调用开销，而且对远程文件系统来说，这需要服务器和客户要进行大量的交互。之所以选择这样一种方案是因为最初的主要目标是支持 SUN 的网络文件系统(NFS)，而 NFS 要求查找操作必须一次一个分量。然而因为这并不是对所有的远程文件系统都是必须的，所以如果由文件系统自己来决定一次操作要解析名字的哪一部分会更好。

第二个问题是由路径名查找操作的无状态引起的，这也是由于 NFS 协议的无状态特性导致的。因为路径查找操作不锁定父目录，它并不能在任意一段时间内保证结果的有效性。下面的例子说明由此引起的问题。

让我们假设用户要求创建文件/a/b，这由两个步骤完成：内核首先查找该文件是否已经存在。如果文件不存在，内核使用/a 的 v 节点的 VOP_CREAT 项来创建文件。问题出在查找返回和创建调用期间，由丁目录/R 没有被锁定，其他的进程可能在此时将文件创建。因此，为了保证正确性，VOP_CREAT 操作必须重新扫描目录，这会导致不必要的开销。

如果当查找操作之后紧接着 create 和 delete 操作时查找程序不转换最后一个分量，就可以避免上面说的问题。然而这危害了接口的模块性，各个操作就不是相互独立的了。

除了 SVR4 之外，UNIX 中还有几种其他的方法支持多种文件系统。每一种是在通用接口基础之上用自己的方法实现该接口。它们之间区别可以追溯到最初的设计目标的差异，主要基于的操作系统的差别以及主要支持的特定文件系统的区别。最早的两种是 AT&T 的 SVR3 UNIX 中文件系统开关表[Rifk 86]以及 Ultrix 的通用文件系统(GFS)[Rodr 86]。这两种方法把 i 节点的概念当作表达文件的基本对象，但是都把 i 节点分为文件系统相关和文件系统无关两部分。在现代 UNIX 变体中，4.4BSD 和 OSF/1 提供了 v 节点接口的替换模型。下面描述这些接口。

在 vnode/vfs 接口中有其他的更为基本的问题。尽管它的目标是允许用户以模块方式扩充，但实际效果并不太好。不使用操作系统源代码而想实现文件系统简直是不可能的事情。在文件系统和内存管理子系统之间有很多复杂的依赖关系，我们将在 14.8 节中讲述该问题。而且，该接口不但在 UNIX 的不同变体间不一致，而且在同一类型的不同版本之间也不一致。结果，尽管又出现了一些新的基于该接口的文件系统(Episode 和 BSD-LFS，将在 11 章中阐述)，文件系统的开发仍然是一件困难的事情。1L1 节中更加详细地分析了这些问题，描述了一种新的基于可堆叠式 v 节点接口的文件系统，使用这种文件系统可以更加方便地构造文件系统。

8.11.2 4.4BSD 模型

4.4BSD v 节点接口[Kare 86]试图解决 SunOS/SVR4 方法的缺点，它使用了状态模型，同时使用了增强的，集成了 4.3BSD(和 GFS)namei 接口特性的改进的查找操作。它允许跨越多个操作锁定 v 节点，也允许在一个多阶段系统调用的相关操作之间传递状态信息。

遍历路径名是由 namei()例程驱动的。该例程首先对当前工作目录虚拟接点调用查找例程，传人的参数是要转换的整个路径名。文件系统相关的 loopup 程序可以在一次调用中转换一个或多个分量，但是不经过安装点。像 NFS 之类的实现一次调用只转换一个分量，而 s5fs 或 uts 的杏找函数可以转换整个路径名(除非在路径上有安装点)。在对安装点进行处理之后，namei()将剩下路径名传递到下一次查找操作。

传递给查找函数的参数被封装一个 nameidata 结构中。这个结构中也包含为了传递状态信息和返回附加信息所需要的数据域。于是这个结构被传递给其他像 create 和 symlink 之类的操作，从而允许相关操作之间可以在不使用堆栈传递变量的情况下共享状态信息。

nameidata 的一个数据域说明了路径转换的原因。如果是为了创建或删除而作的搜索，那么最后的杳找操作将文件目录的 v 节点加锁。同时在 nameidata 中返回一些信息，比如，文件在目录中的位置(如果找到的话)和目录中的第一个空闲块(可以为随后的创建操作所使用)，接着，这个 nameidata 被传递到 create 和 delete 中。因为父目录被查找函数锁定，所以其内容不会发生变化，因此 create 和 delete 便没有必要再次搜索目录。结束后，该操作将父目录解锁。

有时候，在查找操作完毕后，内核可能决定不再往下进行创建或删除操作了(可能因为调用者没有相应的权限)。这种情况下，内核必须调用一个 abortop 操作，将父目录解锁。

即使接口是有状态的(因为可以在多个操作之间进行保持锁定状态)，加锁和解锁操作都是在文件系统相关层进行的。因此在尽量避免冗余操作的情况下，该接口即可以容纳无状态文件系统，也可以容纳有状态文件系统。比如，像 NFS 之类的无状态文件系统可能根本不对父目录进行锁定,而实际上将最后一个分量的查找工作放到稍后的 create 和 delete 操作中。

这种方法的主要缺点是它将对目录的所有操作都串行化了，因为在整个操作期间锁都是被锁住的，即使调用者因 I/O 而阻塞也是这样。这种实现使用了排斥性加锁，这样即使两个只读操作也不能并行的执行了。如果在分时系统中，对于那些访问高度共享的诸如“ / ”或者“ /etc"的文件的操作，会造成非常大的延时。

另外有一种优化的方法，那就是把上次成功的名字查找结果的目录和偏移量放到为每个进程级设置的高速缓存中。对于那些需要对一个目录中的文件迭代的操作来说，这有助于操作性能的改善。当查找路径名的最后一个分量时，namei()使用这个高速缓存。如果父目录跟上次调用 namei()的一样，那么搜索从缓存的偏移量开始，而不是从目录的头部开始(如果需要的话，达到尾部后再绕回)。

4.4BSD 的文件接口提供了很多有趣的特性，比如可堆叠式 v 节点以及联合安装。我们将在 11.12 节中讲述这些特性。

8.11.3 OSF/1 方法

OSF/1 致力于找到一种方法，能够消除由于多余的目录操作带来的问题，同时又能保持 v 节点接口的无状态的特性。而且，这种方法必须要同时在单处理器和多处理器的机器上能正确而有效的工作。这个目标已经实现了[LoVe 91]。方法是将在相关的操作之间传递的信息看作是线索(hint)，线索同一个时间戳相联系，以后的操作可以检查该时间戳来确认操作的合法性。内核使用这些线索，当在两个操作之间相关的信息没有发生改变时就不要浪费时间去检查。

文件元数据由互斥(mutex)锁来保护。互斥锁一般也就是自旋锁(spinlock)。通常利用多处理机安全的互斥(mutex)锁保护文件元数据。这个互斥锁可简单的实现为自旋锁，并保持一小段时间。特别值得一提的是，只有文件系统相关代码才对其加锁，并且不会跨越多个操作，这就是说在操作间元数据可能发生变化。可以通过为每个同步对象关联一个时间戳来监视这种变化。时间戳包括一个简单的单调递增计数器，每次修改相关的对象都要对其计数。4BSD 中使用的基于线索的目录查找高速缓存(见 8.10.2 小节)，也使用了类似的机制。

作为一个例子，让我们考虑一个创建文件的操作。当查找到最后一个部分时，lookup 操作锁住父目录，并对其扫描查看是否文件已存在。如果不存在，lookup 就计算在父目录中放置新表项的偏移，然后它释放父目录并将信息返回给调用者，同时也将时间戳返回给调用者。这之后，内核以上面获得那个偏移和时间戳为参数调用 direnter()在父目录中创建新表项。如果两个时间戳相同，也就是说这期间目录没有变化，此时无需再检查目录就可以将名字插入到那个相应的偏移上。如果时间戳不同，也就是目录在这期间被修改了，此时要重复查找过程。

这种修改主要是由于考虑到多处理机平台的性能，其在单机平台上的优点也是很明显的。这一修改也导致了一些新的竞争条件，但这些找到的错误都通过[LoVe 91]中介绍的方法一一得以解决。OSF/1 模型将无状态和有状态模型的优点结合起来，支持诸如 AFS(见 10.15 节)和 Episode(见 11.8 节)等文件系统。

8.12 小　结

vnode/vfs 接口提供一种功能强大的机制，可以允许用户自己开发文件系统，然后将其加到 UNIX 内核中去。它允许内核对文件的抽象也就是 v 节点进行操作，将文件系统相关代码归类到单独的层，而使用一个严格定义的接口访问它们。供应商可以构造任何实现该接口的文件系统。其过程和写一个设备驱动程序差不多。

不过我们必须注意到符合这个接口的各个实现之间的巨大差别。尽管 SVR4，BSD 和 OSF/1 等都基于类似的原理，它们之间却有着极大的差别，这些差别不仅表现在他们在接口的很多细节描述上(比如各个操作和它们的参数，以及 vnode 和 vfs 的具体格式)，而且表现在它们如何看待状态，同步等等。这就意味着文件系统开发商要想使他们的文件系统同不同的 vfs 接口兼容，必须要进行很大的改动。

8.13 练　习

1.将文件看做字节流有什么好处?这个模型的缺点是什么?

2.假设一个用户反复的调用 readdir 来读取目录的内容，那么如果其他的用户在两个 readdir 之间创建或者删除文件会发生什么事情?

3.为什么不允许用户直接写目录?

4.为什么文件的属性不存储在该文件的目录项中?

5.为什么每一个进程都有一个缺省的文件创建掩码?这个掩码存放在什么地方?为什么内核不直接使用提供给 open 或者 creat 的模式?

6.为什么即使一个用户对文件有写权限，而当文件以只读方式打开时，却不允许该用户对文件操作?

7.下面的 shell 脚本叫做 myscript：

```
date
cat/etc/motd
```

那么执行下面的命令将会有什么样的效果?文件描述符是怎样被共享的?

```
myscript>result.log
```

8.将 lseek 单独作为一个系统调用，而不是每次 read 或者 write 时都将偏移量传入。这有什么优点?其缺点是什么?

9.什么时候 read 调用会返回比要求少的字节?

10.分散-聚集 I/O 操作的优点是什么?什么样的应用程序最可能用到它?

11.建议性加锁和强制性加锁的区别是什么?什么样的应用程序有可能用到字节范围内的加锁?

12.假设一个用户的当前工作目录是/usr/mnt/kaumu。如果系统管理员在/usr/mnt 目录上安装一个新的文件系统，这对用户有什么影响?用户能继续看到在/kaumu 下的文件吗?此时的 pwd 命令的执行结果是什么?其他的什么样的命令将遇到意想不到的结果?

13.使用符号链接取代硬链接的缺点是什么?

14.为什么硬链接不能跨越不同的文件系统?

15.如果错误的使用硬链接将会出现什么样的问题?

16.当一个 v 节点的引用记数到 0 时内核会怎么做?

17.讨论基于线索的和基于引用的的目录名查找缓存的优缺点。

18.[Bark 90]和[John 95]描述了动态分配和释放 v 节点的两个文件系统的实现。这样的系统能否使用基于线索的名字查找缓存?

19.给出一个导致名字查找无限循环的符号链接。lookuppn()怎么处理这种情况?

20.为什么 VOP_LOOKUP 每次只能解析路径名的一个分量?

21.4.4BSD 允许一个进程可以在一个系统调用中跨越多个 v 节点操作锁定一个 v 节点。如果持有该锁的进程被一个信号杀掉了，将会发生什么情况了系统将怎样处理这样的情况?

① 硬链接的存在意味着正确抽象应该是有向无环图(忽略指同父目录的“..”项)，但就大多数实际情况而言，用树来表示更简单更合适。

② 也称为用户，国为 chmod 命令使用 u，g 和 o 分别代表用户(属主)，组和其他。

③ 大部分安全 UNIX 版本是依据国防部出版的评测可信任计算机系统的一套标准制定需求的。这份被称为桔皮书的文件定义了一个系统可以提供的不同安全级。

④ 或者当进程打开文件两次。

⑤ 大部分程序员并不直接使用 read 和 write 系统调用。通常他们都使用库函数 fread 和 fwrite，这两个函数提供数据缓冲等额外的功能。

⑥ 只有在其创建管道后派生的子进程才能共享访问该管道。

⑦ 还受限于 RLIMIT_NOFILE。

⑧ 这两个 i 节点结构在许多方面都是不同的。

⑨ 进程可以调用 chroot 改变自己的根目录。这一操作会影响内核如何解释进程使用的绝对路径。通常某些登录 shell 都会调用 chroot。系统管理员可以利用这一功能来限制某些用户只能访问整个文件树的一部分。为了达到这一目的，lookuppn()首先检查 u 区的 u.u_rdir 域，如果它为 NULL，再检查变量 rootdir。