《 UNIX 高级教程系统技术内幕 》
OCR by djq@SMTH(jqd@263.net)

第 9 章　文件系统实现

9.1 简介

上一章介绍了 vnode/vfs 接口，它提供了支持多种类型文件系统的框架，并且定义了文件系统和内核其他部分的接口。现代 UNIX 系统支持很多不同种类的文件系统，主要可分为本地文件系统和分布式文件系统。本地文件系统只在与系统直接连接的设备上存储和管理数据，而分布式文件系统允许用户访问远程主机上的文件。本章主要讲述几种本地文件系统。第 10 章将讨论网络文件系统，第 11 章介绍了一些具有诸如日志，卷管理等高级特性的新的文件系统。

两种在现代 UNIX 系统中常见的、用于通用目的的本地文件系统是 System V 文件系统(s5fs)和伯克利快速文件系统(FFS)。s5fs 是 UNIX 最初的文件系统，System V 的所有版本和一些商业 UNIX 系统都支持 s5fs。由伯克利 UNIX 4.2BSD 版引进的 FFS 比 sSfs 性能更好，更可靠且功能更强大。它在商业界赢得了广泛的接受，融入 SVR4 系统是 FFS 文件系统的顶峰(SVR4 支持三种用于通用目的文件系统：s5fs，FFS 和 VxFS，Veritas 的日志文件系统)。

当 FFS 第一次被提出时，UNIX 文件系统框架还只能支持一种类型的文件系统。这就迫使厂商们必须在 s5fs 和 FFS 之间作出选择。由 Sun[Kiel 86]提出来的 vnode/vfs 接口可以允许多种类型的文件系统共存于同一台机器上。这样，原先的文件系统实现必须要修改，以便能同 vnode/vfs 集成起来。FFS 集成 vnode/vfs 之后成为现在的 UNIX 文件系统(ufs)①。在[Bach 86]中对 s5fs 做了详细的介绍[Leff 89]则对 FFS 做了详细的分析。本章主要总结和比较了两种不同的实现，既是为了完整性，同时也为理解以后几章描述的高级文件系统打下基础。

在 UNIX 中，文件包括各种 I/O 对象，比如通过套接字和 STREAMS 建立的网络连接，像管道和 FIFO 之类的进程间通信机制，以及块设备和字符设备等等。vnode/vfs 体系结构将文件和文件系统抽象为一种概念，并向内核的其他部分提供模块化接口。这一结构促成了几种专用文件系统的产生。这些文件系统跟文件和 I/O 关系很小，而仅仅利用该接口的抽象特征来实现特殊的功能。本章将介绍几种有趣的实现。

最后，本章描述了 UNIX 的磁盘高速缓存。在早期的像 SVR3 和 4.3BSD 中的 UNIX 版本中，所有的文件 I/O 都使用这个高速缓存。像 SVR4 之类的现代版本将文件 I/O 与内存管理集成在一起，通过将文件映射到内核的地址空间中来访问它们。尽管本章介绍了这种方法的一些细节，大多数的讨论将放在 14 章中介绍(14 章主要讨论 SVR4 中的虚存管理)。传统的磁盘高速缓存机制仍用在元数据块中。这里，元数据指的是一个文件或文件系统的属性和辅助信息。高速缓存是一种全局资源，可以被所有的文件系统所共享，而不仅仅用于某一种特定的文件系统。

我们首先讨论 s5fs，并描述其在磁盘仁的排列方式和在内核中的组织。尽管 FFS 和 s5fs 在很多重要的方面相差很大，但它们还是有一些共同点的，其基本操作的实现方式相同。我们关于 FFS 的讨论将主要集中在它们的区别上面,除了特别指明,所有在 9.3 节中描述的 s5fs 的一般算法也适用于 FFS。

9.2 System V 文件系统(s5fs)

文件系统驻留在某个的逻辑盘或分区(见 8.3.1 小节)内，每个逻辑磁盘可能最多只能有一个文件系统。每一个文件系统是自包含的，有自己的根目录、子目录、文件和所有相关的

数据和元数据。用户可见的文件树是由一个或多个这样的文件系统连接而成的。

图 9-1 显示了一个 s5fs 磁盘分区的排列情况。一个分区在逻辑上可以被视为块的线性数组。一个块的大小为 512 字节乘以 2 的 n 次幂(不同的版本可能为 512,1024 或 2048 个字节)。它代表了文件空间分配和文件 I/O 操作的粒度。物理块号(简称块号)是这个线性数组的索引，它唯一地标识了某个指定磁盘分区上的一个块。这个号必须由磁盘驱动程序转换成柱面号、磁道和扇区号。这种转换依赖于磁盘的物理特性(柱面和磁道、每个磁道中的扇面个数)以及分区在磁盘上的位置。

图 9-1　s5fs 的磁盘布局

在分区的开始部分是包含有操作系统自举(加载和初始)代码的启动区(boot area)。虽然只有一个分区需要包含启动信息，但每一分区都包含一个空的启动区。紧接着启动区的是超级块，其中包含着文件系统属性和元数据。

在超级块后面是 i 节点表，这是 i 节点的一个线性数组。每一个文件都有一个 i 节点。有关 i 节点的内容将在 9.2.2 小节中描述。i 节点由 i 节点号(inode number)唯一地标识，这个号跟该 i 节点在 i 节点表中的索引相同。每个 i 节点有 64 个字节。几个 i 节点放在一个磁盘块中。超级块和 i 节点表的起始偏移在系统的每个分区中都是不变的。因此 i 节点号就可以很容易地转换为一个磁盘块号和从该块起始位置开始的偏移。i 节点表长度固定(其长度是在该分区中创建文件系统时设定)，它限制了一个分区中最多含有的文件数。在 i 节点表之后的空间是数据区。其中有文件和目录的数据块以及间接块，间接块中包含有指向文件数据块的指针，这将在 9.2.2 小节中描述。

9.2.1　目录

s5fs 的目录是包含有文件和子目录列表的特殊文件。它包含有一些固定长度的记录，其中每个记录 16 字节。前两个字节包含的内容是 i 节点号，下面 14 个字节是文件名。这就限制了一个磁盘分区中最多有 65535 个文件(因为 0 不是一个合法的 i 节点号)，并且每个文件名占 14 个字符的长度。如果文件名少于 14 个字符，则会以空(NULL)字符填补。因为目录也是文件，所以它也有一个 i 节点，只是在 i 节点中有一个域将其标识为目录。目录的头两项第一个是“ . ”，代表了目录本身，第二个是“ .. ”，表示父目录。如果一个表项的 i 节点号为零，则表示对应的文件不存在。一个分区的根目录和它的“ . ”表项的 i 节点号都是 2。这也是文件系统能够识别其为根目录的方法。图 9-2 中显示了一个典型的目录结构。

73	.
38	..
9	file1
0	deletedfile
110	subdirectory 1
65	archana

图 9-2　s5fs 目录结构

9.2.2　i 节点

每一个文件都有一个与之相联系的 i 节点。i 节点中包含有文件的管理信息或者称为元数据。它存储在 i 节点表的磁盘空间内。当一个文件被打开，或者一个目录变为活动目录时，内核将磁盘 i 节点中的内容复制到内存中的一个也称为 i 节点的一个数据结构中。但这个结构中包含有许多磁盘 i 节点中所不包含的信息。为了不产生二义性，我们将在磁盘上的数据结构(struct dinode)称为磁盘 i 节点,而将在内存中的结构(struct inode)称为内存 i 节点。表 9-1 描述了磁盘 i 节点的数据内容。

表 9-1　struetd dinode 的域

域	大小(字节)	说明
di_mode	2	文件类型，权限等
di_nlinks	2	对文件的硬链接数
di_uid	2	属主 UID
di_gid	2	属主 GID
di_size	4	字节大小
di_addr	39	块地址数组
di_gen	1	总数(每次 i 节点重用于一个新文件时都递增)
di_atlme	4	上次访问的时间
di_mtime	4	上次修改的时间文件
di_ctime	4	上次改动的时间 i 节点(除了 di_atime 或 di_mtime 的改变)

di_mode 域被分为几个位元(图 9-3)，前四位指定了文件的类型，既可能是 IFREG(常规文件)，也可能是 IFDIR(目录文件)，还有可能是 IFBLK(块设备)，IFCHR(字符设备)等等。9 个低位分别指定厂文件所有者，用户组以及其他人对该文件的读、写、执行权限。

图 9-3　di_mode 位域

关于 di_addr 需要详细地阐述一下。UNIX 文件在磁盘上并不是连续的。当文件长度增加时，内核从磁盘上任何方便的地方分配一个新的块。这样就可以很方便地增大或缩小文件，同时又没有连续分配策略固有的磁盘碎片问题。很显然，这种分配策略仍然没有完全消除磁盘碎片，因为每个文件的最后一个块可能包含有未用的空间。平均下来每个文件浪费了半个磁盘块的空间。

这种方法要求文件系统对文件的每一个磁盘块的位置都维持一个映射。这可以用一个物理块地址数组来实现。每个文件中的逻辑块号便是这个数组的索引。这个数组的大小依赖于文件的大小。大文件可能要用几个磁盘块来存储该数组。然而大多数文件都很小[Saly 81]，如果使用大的数组会浪费很多空间。而且如果将磁盘块数组存放在分离的块中访问文件需要多次读操作，会降低系统性能。

UNIX 的解决方法是在 i 节点中用一个很小的表存储 i 节点，如果文件是够大，则使用附加的结构存储 i 节点。这种方法对小文件来说效率很高，同时也能足够灵活地处理大文件。图 9-4 说明了这种策略。29 个字节的 di_attr 域由 13 个元素表项组成，每个表项用三个字节来存放物理块号。数组元素 0 到 9 包含着文件的从 0 到 9 的数据块的 i 节点号。因此，如果一个文件的数据块个数少于等于 10 个，则这些数据块的地址都在 i 节点表中找到。元素 10 是间接块的 i 节点号，即块中包含块号数组。表项 11 指向一个二级间接块，在二级间接块中有其他间接块的块号。最后表项 12 指向一个三级间接块，其中包含有二级间接块的 i 节点号。

图 9-4　s5fs i 节点的磁盘块数组

对于有 1024 个字节的块来说，这种策略使用直接索引可以寻址 10 个块，使用一次间接块可以寻址 256 多块，使用二级间接块可以寻址 65536 多块，使用三级间接块可以寻址 16，777，216 多块。

UNIX 文件中可以包含有空洞(holes)。如果一个用户创建了一个文件，然后将文件指针调整到一个很大的偏移(通过调用 lseek 可在打开文件对象中设置偏移指针，详见 8.2.4 小节)，再往里写入数据。这样，该偏移前的空间中就没有数据因此使形成了一个“空洞”。如果一个进程试图从文件“空洞”处读取数据，它将得到全零字节。

文件“空洞”有时候会很大，甚至整个磁盘块都是“空洞”。为这样的磁盘块分配空间，无疑是很浪费的。解决方法是内核将 di_addr 数组的相应表项(既可能是直接块，也可能是间接块)置为零。当用户试图读这样一个块时，内核将充满 0 的块返回给用户。只有当有人试图往这个块里写数据时内核才分配磁盘空间。

拒绝为空洞分配空间有很重要的意义。一个进程在试图往空洞里写数据时可能意外地超出了磁盘空间。如果复制一个包含空洞的文件，新文件在磁盘上可能会有充满零的页，而不是预期的空洞。这是因为复制文件要先从源文件中读出内容，然后写到目的文件中。当内核读取一个空洞时，它创建了一个充满零的页，然后该页会被原样复制到目的文件中去。这样，一些像 tar 或 cpio 之类的在文件级而非原始磁盘级上操作的备份和归档工具便会出现问题。系统管理员可能要为一个文件系统作备份，可是他会发现复制的数据在相同磁盘上却没有足够的空间来恢复。

9.2.3　超级块

超级块中包含有文件系统本身的元数据。每一个文件系统都有一个超级块，它在磁盘文件系统的开始处。当内核安装一个文件系统时读取超级块，将其放在内存中，直到该文件系统被卸载。超级块包含有如下的信息：

- 文件系统中块的大小
- i 节点表中块大小
- 空闲块和空闲 i 节点的数目
- 空闲块表
- 空闲 i 节点表

因为文件系统可能有很多空闲 i 节点或空闲磁盘块，因此无论是将空闲块表还是将空闲 i 节点表完全放在超级块中都是不现实的。对于 i 节点，超级块维持一个局部表。当空闲 i 节点表为空时，内核扫描磁盘寻找空闲 i 节点(di_mode==0)，将其充实到表中。

这种方法对于空闲块表是不可能的，因为不可能通过检查一个块的内容而判别它是否空闲。因此无论在任何时候，文件系统都必须维持一个在磁盘上的所有空闲块的完整列表。如图 9-5 所示，这个表占据了几个磁盘块。超级块包含了表的第一部分并且在表的尾部增加或删除数据。表的第一个元素指向包含该表下一部分的磁盘块等等。

图 9-5　s5fs 中的空闲块

有时候，磁盘块分配程序会发现超级块中的空闲块表仅包含一个元素。存储在该元素中的值是包含空闲表下一部分的块号(图 9-5 中的块 a)。内核将表从那个块中复制到超级块，这样那个块便成为空闲的了。这样的好处是存储空闲块表占用的空间直接依赖于分区上的空闲空间大小。对于一个几乎全满的磁盘来说，根本不需要浪费任何空间去存储空闲块表。

9.3　s5fs 内核组织

i 节点是 s5fs 中的基本的文件系统相关对象。它是与一个 s5fs v 节点相关的私有数据

结构，正如前面所说，内存 i 节点同磁盘 i 节点是不同的。本节主要讲述内存 i 节点，我们将会看到 s5fs 是如何操纵它们来实现各种文件系统操作的。

9.3.1 内存 i 节点

struct inode 代表一个内存 i 节点。其中包含所有磁盘 i 节点的内容，另外，它还有其他一些数据，如：

- v 节点——i 节点的 i_vnode 数据域包含有文件的 v 节点。
- 文件所在分区的设备号
- 文件的 i 节点号
- 用于同步和高速缓存管理的标志位
- i 节点在空闲 i 节点表的指针(指向其前后节点的指针)
- i 节点在哈希队列中的指针(指向其在哈希队列中的前后节点的指针)。内核根据 i 节点的 i 节点号由哈希函数求得索引值，从而在需要时可快速得到数据。
- 上一次读取块的块号。

图 9-6 是一个使用了四个哈希队列的简单例子，它展示了 s5fs 是怎样组织 i 节点的。

对磁盘块数组的处理方法与 i 节点不同。在磁盘 i 节点的 di_addr[]数组使用了三个字节表示 i 节点号，而内存 i 节点使用了四个字节存放 i 节点号。这是在空间和性能之间的折衷。对于磁盘 i 节点来说空间节省更重要，而对于内存 i 节点来说性能更加重要。

图 9-6 内存中 i 节点的组织

9.3.2 i 节点查找

文件系统无关层中的 lookuppn()函数用于路径名解析。就像在 8.10.1 小节中所述，它调用 VOP_LOOKUP 操作，每次解析一个分量。当解析到一个 s5fs 目录的时候，则要调用 s5lookup()函数。s5lookup 首先检查日录名查询高速缓存(见 8.10.2 小节)。如果找不到，它每次读取目录中的一个块，查找指定的文件名表项。

如果目录中包含有该文件的一个合法表项，s5lookup()得到该表项的 i 节点号。接着它调用 iget()函数去定位 i 节点。iget()使用哈希函数处理 i 节点号，然后到相应的哈希队列中查找 i 节点。如果 i 节点不在这个哈希队列中，iget()分配一个 i 节点(在 9.3.4 小节中阐述)，并且从磁盘 i 节点中读取数据将其初始化。当将磁盘 i 节点复制到内核 i 节点中去时，iget()将 di_addr[]中的每个元素扩展到四个字节。接着将该新 i 节点加入到相应的哈希队列中。它同时也初始化 v 节点，将其 v_op 域指向 s5vnodeops 向量，v_data 指向其本身，v_vfsp 指向它所属的文件系统。最后它向 s5lookup()返回一个指向 i 节点的指针。s5lookup()将该指针返回给 lookuppn()。

注意，iget()是 s5fs 中唯一用来分配和初始化 i 节点和 v 节点的函数。例如，当创建一个新文件时，s5create()分配一个未使用的 i 节点号(从超级块的空闲 i 节点表中得到)并且调用 iget()将其调入到内存中。

9.3.3 文件 I/O

系统调用 read()和 write()都接受一个文件描述符(open 返回的索引)、数据源或目的用户缓冲区地址，以及要传送的字节数作为参数。文件中的偏移是从与描述符相联系的打开文件对象中获得的。在 I/O 操作的最后，将此次读写操作传输的字节数加到偏移量上去，这意味着下一次读写操作要从本次读写操作结束的地方开始。如果想进行随机 I/O，用户必须首先调用 lseek 函数将文件指针移到希望的偏移处。

文件系统无关代码(见 8.6 节)利用文件描述符从描述符表中索引得到指向打开文件对象(struct file)的指针，然后验证文件是否按所需要的方式被打开。如果是，内核从 file 结构中得到 v 节点指针。在每次 I/O 前内核调用 VOP_RWLOCK 操作将对文件的访问串行化。在

s5fs 中，这是通过给 i 节点加上排斥性锁来实现的②。这保证了在一次系统调用中读写的数据是一致的，并且对文件的所有写是单线程的。内核接着便可以调用 VOP_READ 或 VOP_WRITE 操作，这将分别导致对 s5read()和 s5write()的调用。

在早期的实现中，文件 I/O 例程使用了磁盘块高速缓存(block buffer cache)，它是一块专为文件系统块预留的内存区域。SVR4 把文件 I/O 和虚拟内存结合了起来，它仅为元数据块使用磁盘块高速缓存。9.12 节中将描述原有的高速缓存，本节只是总结 SVR4 的操作，第 14.8 节将详细地讨论虚存管理。

我们使用 read 操作作为例子详细地讲述一下 I/O 过程。s5read()将文件 I/O 操作的起始偏移转换成逻辑块号以及块内偏移量。接着它每次读取一页数据③，方法是将块映射到内核虚地址空间并且调用 uiomove()将数据复制到用户空间，uiomove()调用 copyout()例程来进行实际数据传输。如果页没有在物理内存中，或者内核没有将其合理地进行地址转换，eopyout()将产生一个页面失效。失效处理函数将标识出该页所属的文件，然后在文件 v 节点上进行 VOP_GETPAGE 操作。

在 s5fs 中，该操作是由 s5getpage()实现的。s5getpage()首先调用 bmap()函数将逻辑块号转换成磁盘上的物理块号。接着它搜索 v 节点的页表(由 v_page 所指的)查看该页是否已经在内存中了，如果不是，s5getpage()分配一空闲页，调用磁盘驱动程序从磁盘上读取数据。

在 I/O 操作进行过程中调用进程一直处于睡眠状态。当块读取完毕之后，磁盘驱动程序唤醒调用进程，恢复 copyout()中的数据复制过程。在将数据复制到用户空间之前 copyout()首先要确认一下用户是否对其所指定的要将数据复制于其中的缓冲区有写权限。否则，用户可能无意或有意地指定错误的地址，导致很多问题。例如，如果用户指定一内核地址，内核将会错误地重写内核的正文或数据结构。

当所有的数据都已读完或者发生错误时 s5read()返回，系统无关代码将 v 节点解锁(使用 VOP_RWUNLOCK)，将 file 结构中的指针偏移加上读取的字节数，然后返回用户。read 的返回值是读取字节总数。一般来说它等于要求的字节数，除非到达了文件尾或者发生了其他的错误。

write 系统调用过程类似，但有一些差别。write 修改过的数据并不立即写回磁盘，而是留在内存中，根据高速缓存替换算法在以后的某一时刻写回磁盘。除此外，write 可能会增加文件尺寸因而需要分配数据块，也有可能是间接块。最后，如果只有磁盘块的一部分数据被修改，内核必须要从磁盘上将整个数据块读入内存，修改相关的部分，再写回磁盘。

9.3.4 i 节点的分配与回收

一个 i 节点只要它的 v 节点的引用计数不为零，它就会一直保持活跃，当引用计数降至零时，文件系统无关代码便会调用 VOP_INACTIVE 操作释放 i 节点。在 SVR2 中，空闲 i 节点被标明是无效的，因此在需要的时候还要从磁盘重新读取。这会导致效率低下，因此新的 UNIX 系统将 i 节点尽可能地放在高速缓存中。当 i 节点变为不活跃时，内核将其放到空闲表中，但并不使之无效。iget()函数可以在需要的时候找到它，因为它在被重用之前一直被放在某一哈希队列中。

每一个文件系统都有一个长度固定的 i 节点表，这样就限制了内核中同时活动的 i 节点数。在 SVR3 中，i 节点高速缓存机制使用一种最近最少使用的替换算法。内核释放 i 节点时将其放到该表的尾部，当分配 i 节点时则从其头部获得。尽管这是一种很通用的启发式高速缓存替换算法，但把它用到 i 节点的高速缓存中却并不是很合适。这是因为有些不活动的 i 节点大概要比别的不活动 i 节点更有用。

如果一个文件当前正被使用，其 i 节点便被钉(pinned)在 i 节点表中，当文件被访问时，它的页被放到高速缓存中。当文件变为不活动时，它的一些页可能仍在内存中。这些页可以通过访问 v 节点的页表的 v_page 域而被找到。分页系统(paging system)也按照 v 节点指针

和页面在文件中的偏移将它们放到哈希队列中，如果内核重用 i 节点(以及 v 节点)，这些页便丢掉了其标识。如果进程需要某一页，尽管该页已经在内存中，内核也必须重新到磁盘上读取。

注意： 如果一个对象既不能被释放也不能被删除，我们便说该对象被钉在内存中。有引用计数的对象会被钉在内存中，直到其最后一个引用被释放。另一个例子是，一个进程可以用 mlock 系统调用将其地址空间的一部分钉在内存中。因此，重用那些没有内存缓冲页的 i 节点比较好。当 v 节点引用计数到达零后，内核调用 VOP_INACTIVE 操作释放 v 节点以及其私有数据(在这里便是 i 节点)，当释放 i 节点时，内核检查 v 节点的页表。如果页表为空，内核便将 i 节点放到空闲表的头部。如果页表不为空，内核则将 i 节点放到空闲表的尾部。如果 i 节点保持不活跃，分页系统便及时地将它的页释放掉。

[Bark 90]讨论了一种新的 i 节点分配和回收方法，该方法根据系统的负载调整内存 i 节点的数目。文件系统使用内核内存分配程序动态地分配 i 节点，而不使用固定长度的 i 节点表。这样，内存中的 i 节点数便可以根据需要增力喊减少。系统管理员也没有必要在预先配置系统时猜测系统中该有多少个 i 节点。

当 iget()在 i 节点哈希队列中找不到某一 i 节点时，它从空闲表中删除第一个节点。如果这个 i 节点在内存中仍然有页面，iget()将它放回到空闲表的尾部，然后调用内核内存分配程序分配一个新的内存 i 节点结构。当然，也可以检索整个空闲表找到一个在内存中没有页面的 i 节点，但是我们上面描述的算法更简单有效。它唯一的缺点是有可能分配的 i 节点数目比需要的多。

在一个多用户分时系统上的负载基准测试[Gaed 82]实验显示新算法对系统时间(内核态所占的 CPU 的时间)的占有减少了 12%到 16%。这种方法最初是在 s5fs 中实现的，但是也可用在其他的文件系统中，比如说 FFS。

9.4 对 s5fs 的分析

s5fs 以其设计简单而著名，可是设计的简单性却给它的可靠性、性能、功能等带来很多问题。本节我们便讨论一下这些问题，正是这些问题才导致了 BSD 快速文件系统的出现。

最主要的可靠性考虑是超级块。超级块中有很多关于整个文件系统的重要信息，比如空闲块表和空闲 i 节点表等。每个文件系统仅包含超级块的一个副本，如果副本崩溃，整个文件系统便不再可用。

有几个原因导致性能的不佳。s5fs 把所有的 i 节点组织到文件系统的开始位置，而用剩余的磁盘空间容纳文件数据块。访问一个文件需要首先读取 i 节点，然后再读取文件数据.这种分隔导致在两次操作间有一段很长的搜索时间，这使得 I/O 时间大增，i 节点是随机组织的，而没有将相关文件的 i 节点分组，比如将同一个目录下的文件组织起来。这样对一个目录底下所有文件的访问操作(比如，ls -l)就会导致磁盘的随机访问。

磁盘块分配方案也不是最佳的。当文件系统第一次被创建时(使用 mkfs 程序)，s5fs 以最佳方式组织空闲块表，这样磁盘块分配就会以旋转连续的方法进行了。然而，随着文件的创建和删除，返回空闲表的磁盘块的顺序被完全打乱。这样文件的顺序访问就会变慢，这是因为逻辑上连续的磁盘块在物理磁盘上可能相距非常远。

磁盘块大小也是影响性能的一个因素，SVR2 的磁盘块大小 512 字节，SVR3 则使用了 1024 字节。提高磁盘块大小之后，一次磁盘访问可以读取更多的数据，从而提高了性能。然而，这会浪费更多的磁盘空间，因为平均每个文件就要浪费半个磁盘块的空间。这就需要有更加灵活的磁盘中间分配方案。

最后，该文件系统在功能上仍然有很多限制。将文件名字长度限制在 14 个字符之内可能在早期 UNIX 使用的环境中不会有很多麻烦，然而对一个功能强大的商用大系统来说，这和限

制是不能接受的。有些程序往往通过在已有文件名上加上一些扩展来自动产生文件名，可是有了这个限制，他们不得不小心翼翼地避免使得文件名长度超过 14 个字符。另外，每个文件系统只允许有 65535 个 i 节点也是很大的限制。

所有上述这些问题促使伯克利 UNIX 开发一种新的文件系统，这就是快速文件系统(FFS)，它首次出现于 4.2BSD 中④。下面一节讨论其主要特征。

9.5 伯克利快速文件系统(FFS)

FFS[McKu 84]解决于 s5fs 的很多限制。下面主要描述了其设计，我们将看到它是怎样提高可靠性、性能以及功能的。FFS 提供了 s5fs 的所有功能，其大多数的系统调用处理算法和内核数据结构保持不变。两者的主要差别在于磁盘安排，磁盘结构以及空闲块分配方法上。FFS 也提供了一些新的系统调用来支持这些新特性。

9.6 硬盘结构

要想了解影响磁盘性能的因素，首先要看一下数据是怎样在磁盘上放置的。图 9-7 显示了在 20 世纪 80 年代中早期生产的磁盘的格式。磁盘由很多盘片(platters)组成，每个盘片都对应与一个磁头⑤。每个盘片(platter)包含好多磁道，这些磁道形成一组以盘片中心为圆心的同心圆；最外面的是 0 磁道，依此类推。每个磁道又被细分成为扇面(sector)，扇面也是顺序排号的。扇面的尺寸一般是 512 字节，这也是每次磁盘 I/O 操作的最小单位。柱面(cylinder)是由一组磁道组成，这些磁道分布在不同的盘片上，每个盘片上一个磁道，并且这些磁道与磁盘中轴的距离也都相同。所以柱面零由所有盘片的零磁道组成，依此类推。在很多磁盘中，所有的磁盘头(head)同时移动。因此在任一给定时间，所有的磁头都在每个盘片的相同磁道和相同扇面上。

图 9-7 磁盘的概念图

UNIX 把磁盘看成是磁盘块的线性数组。一个块内包含有 2 的整次幂个磁道，在本节，我们假设一个块内正好有一个扇面。当一个 UNIX 进程试图读取一特定块号时，设备驱动程序将该块号转化成一逻辑扇面号，并且由此计算出物理磁道号，磁盘头号以及扇面号。这个过程中扇面号首先增加，其次是磁盘道号，最后是柱面号。因此，每个柱面包含一组序号连续的磁盘块。在计算出块号的位置后，磁盘驱动器首先把磁盘头移到相应的柱面。这个磁盘头定位过程是磁盘 I/O 中最耗时的部分，其延迟也直接与磁盘头必须移动的距离有关。这个延迟叫做旋转延迟。一旦正确的扇区转到磁盘头下面，数据传输便可以开始。实际的传输时间也正好是一个扇区移动经过磁盘头的时间。要想使 I/O 带宽最大，必须要使磁盘头定位的次数和磁盘头移动的距离最小，同时通过合理地安排磁盘上的磁盘块以减小旋转延迟。

9.7 磁盘组织

一个磁盘分区由磁盘上的一纽连续柱面组成。格式化过的磁盘分区包含有一个自我包含的文件系统。FFS 将分区进一步划分成一个或多个柱面组，每个柱面组中有一小组连续的柱面。这样 UNIX 就可以将相关的数据存在同一柱面组中，从而使磁头移动距离最小。9.7.2 小节对此做了详细的讨论。

传统的超级块包含的信息分为两个结构。FFS 超级块包含有整个文件系统的有关信息——柱面组的数目、大小和位置，磁盘块大小，磁盘块和 i 节点的总数目，等等。除非重建文件系统，否则超级块中的信息不会发生变化。而且，每个磁盘组都有一个描述有关本组数据结构的概括性信息，其中包括空闲 i 节点表和空闲块表。超级块在分区的开始位置(在启动块区的后面)，然而这还不够。超级块中的数据是很关键的，必须要保证不能发生磁盘错误。因此，每个柱面组都包含有超级块的一个复制，FFS 将这些超级块复制放在每个柱面组的不同

偏移处以保证没有一个单独的磁道、柱面或盘片包含有所有这些超级块的复制。除了第一个柱面组外，在柱面组的开始位置和超级块复制之间的一段空间用作数据块。

9.7.1 块和碎片

正如在前面讨论的那样，磁盘块越大，一次 I/O 操作传输的数据就越多，这样就可以提高性能，但是这样就会浪费更多的磁盘空间(一般一个文件浪费半个磁盘块)。FFS 致力于在传输率和避免磁盘空间浪费两方面性能都达到最优，它采用的方法是将磁盘块细分成片段(fragment)。在 FFS 中，每个文件系统内部磁盘块大小是固定不变的，但是在同一机器上的不同文件系统却可以有不同的磁盘块大小。磁盘块尺寸是 2 的整数次幂，且大于或等于 4096 字节。大多数实现都加上了 8192 字节的上限。这远大于 s5fs 中的 512 字节或 1024 字节的磁盘块，除了增加吞吐量外，还允许通过二级间接块访问大至 2^32 字节的文件，尽管许多变体使用三级间接块来访问大于 4G 的文件，FFS 不支持三级间接块。

典型的 UNIX 系统有大量的小文件，必须要对它们进行高效的存储[Saty 81]。4K 字节的磁盘块会浪费很多空间，FFS 的解决方法是将每一个磁盘块细分为一个或多个片段(Fragment)。每个片段的大小在文件系统创建之初便确定下来了，并且在整个文件系统中保持恒定。每个磁盘可以分成 1，2，4 或者 8 个片段，但最小尺寸和一个扇面的尺寸相等，即 512 字节。每个片段都是可单独寻址和可分配的。这就需有一个能跟踪每一个片段的位映像取代空闲块表。

每个 FFS 文件是由许多完整的磁盘块组成，但在最后一个块中，可能包含几个连续的片段。一个文件块必须在一个磁盘块中，即使两个连续的磁盘块中有是够的连续空闲段容纳一个文件块，它们也不能被合并。而且，如果文件的最后一个块包含多于一个的片段，这些片段必须连续并且只能包含在同一个块中。

这种策略减少了空间的浪费，但时常需要重新复制文件数据。下面考虑一个文件，其最后一个磁盘块只包含一个片段。该磁盘块中的其余片段可能分配给其他文件了。如果该文件长度增加，需要另外一个片段，我们就必须找到另外一个至少有两个连续空闲片段的磁盘块。第一个空闲片段的内容由原来的那个片段中复制而来，而第二个片段用新数据填充。如果一个文件大小经常增长，并且每次增长量都很小，那么其片段可能需要被复制好几次，这将会影响系统性能。FFS 对此加了控制，其方法是只允许用直接块容纳碎片。

因此要想获得最好的性能，应用程序一次应往文件里写满一个磁盘块。在同一台机器上的不同文件系统可能有不能的磁盘块尺寸。应用程序可以使用系统调用 stat 得到文件的属性，这些属性是以文件系统无关格式表示的。stat 返回的一个属性就是最适合 I/O 操作的最佳尺寸，在 FFS 中也就是磁盘块大小。这些信息可以被标准 I/O 库所使用，也可以被其他管理它们自己 I/O 的应用程序所使用。

9.7.2 分配策略

在 s5fs 的超级块中包含有空闲块表和空闲 i 节点表；增加、删除空闲块和空闲 i 节点只在相应的表末尾进行。也就是进行这些操作块时是不考虑磁盘块顺序的。实际上，只有在创建文件系统时对这些表才要求有顺序，此时空闲块表是按旋转最佳的顺序(rotationally optimal order)被创建的，而空闲 i 节点表则是顺序的。经过很多次的使用后，这些表变得相当随机，根本没有方法控制在什么地方分配空闲块和空闲 i 节点。

相反地，FFS 的日标是将磁盘上的各种相关信息搭配起来使顺序访问性能最佳。FFS 提供很多有关于磁盘块，i 节点以及目录分配等的控制信息。这种分配策略使用柱面组的概念，并且需要文件系统知道与磁盘有关的各种参数。下面的一些规则便是这种分配策略的一些概括：

· 试图将一个目录的所有 i 节点放到同一个柱面组中。很多命令(ls -l 是最好的例子)都希望能快速连续地访问一个目录中的所有 i 节点。用户的访问显示出很强的局部性，他们

通常在一个目录下工作很长的时间之后才移向其他的目录。

·在创建子目录时，将其放在和父目录不同的柱面组下，这样可以使数据均匀地分布在磁盘上。分配例程从空闲 i 节点数大于平均数的新的柱面组中选择一组柱面，从中选出目录最少的一个。

·尽量将数据和 i 节点放在相同的柱面上，因为一般来说 i 节点和数据是同时被访问的。

·尽量避免用一个大文件将整个柱面组充满，如果文件尺寸达到 48K 并以每 1M 的倍数增加时，要改变柱面组。之所以选择 48K 是因为对于 4096 字节的磁盘块来说，i 节点直接块能记录前 48K 字节的数据⑥。要选择哪一个新的柱面组取决于其空闲块的数目。

·尽量在旋转最佳位置(rotationally optimal positions)分配顺序磁盘块。当顺序读文件时，在读完一个块后和内核完成 I/O 处理后再进行下一次读取初始化之间有一个延迟。因为此时磁盘是旋转的，所以在这段延时期间可能有一个或多个扇面从磁头底下经过。旋转最佳算法试图决定要跳过几个块，使得在下一次读取初始化完成时，所需要读取的块正好在磁头底下。要跳过的块的数目称为旋转延迟因子，或磁盘问隔。

FFS 实现必须对局部化的努力和将数据均匀地分布在磁盘上的需求间进行平衡。如果过分局部化，所有的数据都将挤在同一个柱面组中，极端情况下，就如同 s5fs 一样也会出现单个大柱面组。为了避免这种情况，可以将子目录分布在不同的柱面组中，同时将大文件分隔开来。

当磁盘上有很多空闲磁盘块时，这种方法非常有效。然而，如果磁盘大约 90%是满的，这种方法使得性能急剧下降。当几乎没有空闲磁盘块时，要想在最佳位置找到空闲块是很困难的。因此文件系统维护了一个空闲空间预留参数，这个参数一般设为 10%。只有超级用户才能从这个预留空间中分配空间。

9.8 FFS 的增强功能

因为 FFS 跟 s5fs 的磁盘安排不同，因此迁移到 FFS 需要将磁盘上信息转储，并且要重建整个磁盘。因为这两个文件系统在根本上是不兼容的，FFS 的设计者在 FFS 中引入了一些与 s5fs 不兼容的功能性变化。

长文件名

FFS 改变了目录结构，从而允许文件名可以超过 14 个字符。如图 9.8 所示，FFS 的目录项是可变的。项中的固定部分包括 i 节点号，分配的尺寸，以及项中文件名字的长度。其后是以 NULL 字符结尾的文件名字，这个名字放到一系列 4 字节的域中，当前支持的文件名最长可达 255 个字符。当删除一个文件时，FFS 将释放的空间与前项合并(图 9-8b)。因此分配尺寸域中记录着项中可变部分所占的总空间。目录本身是在 512 字节的块中分配的。项不允许跨越多个组块(chunks)。最后，为了方便移植，标准库加了一组目录访问函数，可以允许对目录信息做文件系统无关的访问(见图 8.2.1 小节)。

图 9-8 FFS 目录

符号链接

符号链接(见 8.4 小节)解决了硬链接中存在的很多限制。符号链接是一个指向一个称为链接目标(target of link)的文件，通过 i 节点中的 type 域可以判断一个文件是否为符号链接。符号链接文件中的内容是目标文件的路径名。路径名既可以是绝对路径名，又可以是相对路径名。路径名遍历例程负责识别和解释符号链接。如果路径名是相对的，路径名遍历例程将其解释为包含这个链接的相对目录。尽管符号链接的处理对大多数程序是透明的，一些应用程序需要识别及处理符号链接。他们可以使用系统调用 lstat，该调用不转化路径名中的最后一个符号链接；也可以使用 readlink 返回链接的内容。符号链接可以由系统调用 symlink 创建。

其余的增强功能

4.2BSD 增加了系统调用 rename 以便能原子地重命名文件和目录，而以前这项工作需要先调用 link,再调用 unlink。4.2BSD 为了限制用户使用的文件系统资源,还增加了配额(quota)机制。配额既适用于 i 节点，也适用于磁盘块。同时还有可以触发一个警告的软限制和内核施加的硬限制。

上述功能中的一部分已被集成到 s5fs 中去丁。在 SVR4 中，s5fs 支持符号链接和原子重命名。然而 SVR4 并不支持长文件名和磁盘配额。

9.9 分析

FFS 极大地增强了各种性能。在有 UNIBUS 接口卡[Krid 83]的 VAX/750 上的测试表明在 s5fs 中(磁盘块大小为 1K)吞吐量为 29KB/s，而在 FFS 中(磁盘块大小为 4KB，片段为 1KB)吞吐量增加到 221KB/s,CPU 的利用率从 11%提高到 43%,在同样的配置下,写吞吐量从 48KB/s 提高到 142KB/s，CPU 的利用率从 29%提高到 42%。

再来看一下磁盘空间浪费。在 s5fs 中，每个文件平均浪费半个磁盘块空间，在 FFS 中平均浪费半个片段的空间。如果 FFS 的片段空间跟 s5fs 中的磁盘块空间大小相同，那么两者的浪费率是相等的。增大磁盘块的好处在于用于映射一个大文件所需的空间减少了，因此文件系统几平不需要间接块。相反，如果磁盘块不足够大，会需要更多的空间来监视空闲块和空闲片段。这两个因素也趋于扯平。当 FFS 的片段尺寸和 s5fs 的块尺寸相同时，其净结果是磁盘利用率相同。

然而，空闲空间预留必须被算作浪费的空间，因为用户访问不到这些空间。考虑到这个因素，磁盘块为 1K 的 s5fs 和磁盘块为 4K，片段为 512 字节的 FFS 的浪费率大致相同，其空闲空间预留大约在 5%左右。

在图 9-7 中描述的磁盘结构对新磁盘来说早已过时。现代的 SCSI(小型计算机系统接口)磁盘[ANSl 92]没有长度固定的柱面。它们利用了磁盘外部磁道比内部磁道可以容纳更多数据的特点，将磁盘分成几个区(zones)，在每个区内部每个磁道有相同数目的扇区。

FFS 没有意识到这种特性，因此用户不得不使用虚构的(completely fictional)的柱面尺寸，为了支持 FFS 的固定尺寸磁道的概念，供应尚们得到磁盘上所有长度的 512 字节扇区的总数，将它们分解到一些磁道以及每个磁道的扇区。这种分解可能纯粹考虑了方便性，同磁盘的物理特性完全不一样。结果 FFS 的旋转最佳方法(rotational placement optimizoting)几乎没起什么作用，在很多情况下可能还会损害系统性能，将柱面分组依然是很有用的方法，因为正如 FFS 中看到的，在相邻柱面上的各个块同样也处在相近的磁道内。

总之，相对 s5fs 来说，FFS 提供了很多优点，这也说明了它广泛流行的原因。System V 在 SVR4 中也将 FFS 做为其支持的文件系统之一。而且，SVR4 将 FFS 的很多特点融进了 s5fs 中。固此在 SVR4 中 s5fs 也支持符号链接、共享和独占性文件加锁，以及系统调用 rename。

尽管 FFS 相对于 s5fs 来说有了很大的改进，它远不是完美的文件系统。仍然有很多方法可以进一步提高性能。一个方法便是将内核缓冲区联成一串，这样一次磁盘读写便可以读写好几个缓冲区。这需要修改所有的磁盘驱动程序。另一种可能性是对某些快速增长的文件事先分配几个磁盘块，如果用不着这些磁盘块时，则关闭文件时将其释放。其他一些重要方法，包括日志结构和基于内容的文件系统，这些将在第 11 章中讲述。

FFS 有很多增强的功能。4.3BSD 增加了两种缓冲机制来加速名字查找[McKu 85]。首先它使用了基于线索(hint-based)的目录命名查找高速缓存。这个高速缓存机制在所有的名字转化中命中率大约是 70%。当在 SVR4 中实现 FFS 时，实现者将这个高速缓存从文件系统相关代码中移了出来，将其变为所有文件系统都可以使用的全局资源。他们还改变了这个高速缓存的实现，以便它能够保持对被缓存文件的引用。8.10.2 小节更加详细地讨论了目录名字查找

高速缓存。

其次，每个进程都缓存了大多数最近转化的路径名的最后一个分量的目录的偏移量。如果下一次转换是针对同一个目录中的文件，便可以从此点开始，而不用从目录顶部开始。当一个进程顺序地搜索目录时(10 一 15%的名字查找都是如此)这种缓冲机制非常有用。SVR4 在实现中将这个缓冲的偏移移到内存 i 节点中，这样 FFS 就可以为每一个目录都缓存一个偏移，而不是每个进程一个目录。但是另一方面，如果多个进程并发地使用同一个目录，这种缓存机制就会失去其作用。

9.10 临时文件系统

很多实用程序和应用程序，比如说编译器和窗口管理器，都使用临时文件来保存执行的中间状态信息。当程序退出时，这些临时文件应该被删除。这意味着它们的生存时间很短，没有必要持久保存(在系统崩溃时没有必要保存)。当往临时文件里写数据时，内核并不直接写到磁盘上，而是使用块缓存来保存数据。数据最后刷新到磁盘之前，临时文件已经被删掉了。因此，对此类文件的 I/O 速度很快，根本不涉及磁盘操作。但是创律以及删除这类文件速度却极慢，因为为了更新目录和元数据块要涉及到很多磁盘同步访问。实际上同步更新对临时文件来说完全没有必要，因为它们并不需要持久，因此就迫切需要一个特殊文件系统来高速创建和访问临时文件。

有一种方法可以解决这个问题，该方法是使用 RAM(随机访问存储器)盘，RAM 盘提供一个完全驻留在物理内存中的文件系统。内核通过使用一个设备驱动程序来仿真一个磁盘，只不过数据是存储在物理内存中，并且访问这些数据仅为访问磁盘时间的很小的一部分、这并不需创建特殊类型的文件系统。一旦 RAM 盘安装上之后(分配一块连续的物理内存)，可以通过使用像 newfs 之类的工具在其上建立一个本地文件系统，诸如 s5fs 或 FFS。RAM 盘和普通磁盘的区别仅在设备驱动层才是可见的。

这种方法的主要缺点是要为一个 RAM 盘分配一块很大的内存，浪费了系统资源。临时文件所需的内存动态地随着系统使用情况变化而变化。而且因为 RAM 盘所占的内存跟踪内核内存是分开管理的，所以元数据更新就需要附加的内存到内存的复制。很显然，我们需要一个能更有效地使用内存的特殊文件系统来支持临时文件。下面介绍两种实现。

9.10.1 内存文件系统

内存文件系统(mfs)是由加州大学伯克利分校开发的[McKu 90]。整个文件系统是建立在执行 mount 命令的进程的虚拟地址空间上。该进程并不从 mount 调用中返回，而是一直在内核中等待对文件系统的 I/O 请求。现在该安装进程(mount process)充当了 I/O 服务器，其 PID 包含在 v 节点的文件系统相关部分 msfsnode 中。为了执行 I/O 操作，调用进程将请求放入 mfsdata 结构(每个文件系统的私有数据)所维护的请求队列中，然后唤醒安装进程，由安装进程执行请求，在此期间调用进程进入睡眠状态。安装进程将数据从其地址空间读出或写到其地址空间中，从而完成请求，最后将调用进程唤醒。

因为文件系统是在调用进程的虚拟地址空间中，它有可能像其他数据一样被标准内存管理机制通过换页换出内存。这样，mfs 文件的页要同其他进程竞争物理内存。当前没有被引用的页会被写到交换区中，以后需要时会被再调入内存。因此系统可以支持比实际物理内存还要大的临时文件系统。

尽管这种文件系统要比磁盘文件系统快许多，它也有许多缺点。这主要由 BSD 的内存结构造成的。使用一个进程来处理所有的 I/O 操作会导致在每次操作中都需要两次上下文切换。并且文件系统仍然驻留在分离的地址空间中，这就意味着我们必须要进行一些附加的内存复制操作。尽管像一些柱面组之类的概念对于一个基于内存的系统来说毫无意义，设计者还是把内存文件系统的格式设计为跟 FFS 相同。

9.10.2 tmpfs 文件系统

tmpfs 是由 Sun 公司开发的[Snyd 90],它将 vnode/vfs 接口的强有力的功能和新的 VM(虚拟内存)体系结构结合起来，从而为临时文件提供了一套有效的机制。在这里提到的 VM 将在第 14 章中作详细的讨论。

tmpfs 完全在内核中实现，并不需要一个单独的服务器进程。所有的文件元数据都存放在由系统内核堆中动态分配的不分页内存中。文件的数据块都在分页内存中，在 VM 子系统中的匿名页程序(anonymous page facility)表示这些数据块。一个匿名对象(anonymous object：struct anon)可以将这样一个页映射，该对象中包含着在内存中和交换设备中的每一个页的地址。tmpnode 是每个文件的文件系统相关对象，它包含有一个指向文件的匿名映射(anonymous map，struct anon_map)的指针。该映射中包含着指向文件的每个页的匿名对象的指针数组。它们之间的联系如图 9-9 所示。因为页本身在可换页的内存中，所以分页子系统可以把它们换出内存，因此它们必须要同其他的进程竞争物理内存，就像在 mfs 中那样。

VM 子系统也使用这些匿名对象和映射来描述一个进程的地址空间中的页。这样进程就可以使用系统调用接口 mmap 来将一个 tmpfs 文件直接映射到其地址空间上。这种映射是通过使文件和进程共享 anon_map 来实现的。因此，一个进程不用将文件的数据复制到它的地址空间内就可以直接访问文件的页。

tmpfs 的实现解决了 mfs 的一些缺点。它不使用分离的 I/O 服务器，因此避免了上下文

图 9-9 定位 tmpfs 页面切换的时间。将元数据放在不分页的内存中避免了内存到内存的复制和磁盘 I/O 的操作。内存映射可以使得对文件数据的访问变的更加快速和直接。

实现临时文件系统的另外一种方法在[Ohta 90]中做了描述。它给系统调用 mount 增加了一个 delay 选项，该选项可以在 ufs 的私有对象 vfs_data 上设置一个相应的标志。在实现该文件系统的时候修改了很多同步更新磁盘(一般为更新元数据)的 ufs 程序，使它们能够先检查这个标志。如果已经设置了这个标志，那么这些例程通过将缓冲区标志为 dirty 的方法延迟写操作。这种方法有几个优点。它不使用分离的 RAM 磁盘，因此就避免了维护两个内存块复制的时间和空间上开销。它带来的性能上的提高是引人注目的。其主要缺点是需要对 ufs 的一些应用程序做一些修改，使得要想不改变 ufs 源码而向一个已存的内核中加入 tmpfs 变得非常困难。对于支持多种类型文件系统的系统来说，需要修改每一个实现以便使用安装延迟选项。

9.11 特殊目的文件系统

设计 mode/vfs 接口的初衷是使得像 s5fs 和 FFS 这样的本地文件系统和像 RFS、NFS(见第 10 章))这样的远程文件系统能够共存于一台机器上。在这个接口得到接受后，人们便利用该接口的强大功能和通用性开发了一些特殊目的的文件系统。其中有些已经成为 SVR4 版的一部分，而其他的则是可选的。下面我们就讲述几种这样的文件系统。

9.11.1 specfs 文件系统

specfs 文件系统为所有的设备文件提供了一个统一的接口。它对用户来说是不可见的，也不能被用户安装。它向所有支持特殊设备的文件系统开放了一个标准的接口。其主要的目的是拦截对设备文件的 I/O 调用并且据此来调用相应的设备驱动程序。表面上看这很简单，因为可以通过 vnode 的 v_type 域来断定该文件是设备文件，vnode 的 v_rdev 域提供了主设备号和次设备号。文件系统无关代码应该可以直接使用块设备和字符设备转换来调用设备驱动程序。

然而当多个设备文件引用同一个底层设备时，我们会碰到很多问题。当不同的用户使用不同的文件名访问同一个设备时，内核必须同步对该设备的访问。对于块设备，我们还必须保证它在块缓存中各个复制能够保持一致，很明显，内核必须意识到不同的 v 节点实际上代

表同一设备。

specfs 层为每一个设备文件创建一个影子 v 节点(shadow vnode)。其文件系统相关数据结构被称为一个 snode。对设备文件的查找操作返回一个指针，该指针指向影子 v 节点而不是真实 v 节点。如果需要的话，可以使用 vop_realvp 操作得到真实 v 节点。snode 中有一个域 s_commonvp，它指向一个该设备的普通 v 节点(同另外一个 snode 相联系)。每一个设备只有一个普通 v 节点，可能有多个影子 v 节点指向它。所有需要同步的操作，包括块设备读写，都通过这个普通 v 节点进行操作。第 16.4 节描述了实现的详细细节。

9.11.2 /proc 文件系统

/proc 文件系统[Faul 91]提供了一个功能强大的访问任意进程地址空间的接口。设计的最初目的是为了能够取代 ptrace 来支持调试，然而它已经发展成为一个进程模型的通用接口。它允许用户使用标准文件系统接口和系统调用来读取和修改其他进程的地址空间并且对其进行若干控制操作，结果，访问控制是使用读-写-执行权限夹实现的。缺省情况下，只有其所有者才能读写/proc 文件。

在早期的实现中，每个进程由/proc 目录底下的一个文件代表。文件的名字是进程 ID 的十进制数，其大小跟进程的用户地址空间相同。通过打开相应的/proc 文件以及使用 lseek，read 和 write 系统调用可以访问进程的任何地址的数据。对该文件的一组 ioctl 命令可以对进程实行各种控制操作。

不过发展到现在，/proc 的实现发生了极大的变化，本小节讲述 SVR4.2 接口。在 SVR4.2 中，每个进程由在/proc 的一个目录代表，目录的名字是进程 ID 的十进制表达。每一个目录包含下面的文件和子目录：

status　这是一个只读文件，其中包含有关子进程状态的信息。其格式由结构 struct pstatus 定义，里面的信息包括进程 ID，进程组和会话 ID，堆栈和堆的尺寸和位置，以及其他信息。

psinfo　这也是一个只读文件，其中包含着命令 ps(1)所需的信息。其格式由结构 struct psinfo 定义，其中包含着状态文件的某些域，还有其他一些信息，比如控制终端的映像尺寸和设备 ID。

ctl　这是一个只写文件，使用该文件用户可以通过向一个进程的文件写入数据而对进程进行控制操作。本节的后面描述了一些控制操作。

map　这是一个只读文件，描述进程的虚拟地址空间。它包含一个 prmap 结构的数组，其中每一个元素描述进程的一段连续地址空间。进程地址映射在 14.4.3 小节中描述。

as　这是一个可读写文件，用来映射进程的地址空间。任何试图对进程的某一个地址空间的访问都可以使用 lseek 来定位到相应的地址处，然后进行 read 或者 write 操作。

sigact　这是一个只读文件，其中包含着信号控制信息。该文件包含一个 sigaction 结构的数组(见 4.5 节)，数组的一个元素用于处理一个信号。

cred　这是一个只读文件，其中包含进程的用户凭证(credentials)。其格式由 struct prcred 定义。

object　这是一个目录。每一个被映射到进程地址空间中的对象都有一个文件在该目录下(见 14.2 节)。用户可以通过打开对象的相应的文件来得到对象的文件描述符。

lwp　这是一个目录。每一个进程的 LWP(见第 3 章)都有一个文件在该目录下。每一个目录包含三个文件——lwpstatus，lwpsinfo 和 lwpctl，它们提供了单个 LWP 状态和控制操作，分别类似于 status，psinfo 和 ctl 文件。

必须注意的是这些文件并不是实际存储的物理文件。它们仅仅提供了访问进程的一个接口。/proc 文件系统将用户对这些文件的访问转换为目标进程或者其地址空间的恰当的操作。几个用户可以并发的打开同一个/proc 文件。当用户写打开一个 as，ctl 或者 lwpctl 文

件时，O_EXCL 标志提供建议性加锁。ctl 和 lwpctl 文件提供了几个控制和状态操作，包括以下几种：

PCSTOP　停止进程所有的 LWPS。

PCWSTOP　等待进程所有的 LWP 停止。

PCRUN　恢复一个停止的 LWP。一些可选的标志可能提供附加的操作，比如 PRC-SIG 用来清除当前信号，而 PRSTEP 可以单步执行进程。

PCKILL　向进程发送特定的信号。

PCSENTRY 指示 LWP 在特定的系统调用上停止。

PCSEXIT　指示 LWP 当从特定的系统调用退出时停止。

这里没有显式支持断点。不过这很容易实现，方法是用 write 系统调用在正文段中写入一个断点指令即可。大多数系统提供断点指令。另外，我们可以使用任何合法的导致内核陷阱的指令。

/proc 接口提供了一个处理目标进程的子进程的机制。调试程序可以在目标进程上设置一个 inherit-on-fork 标志，监视从 fork 和 vfork 调用的返回。这样，当从 fork 返回时父进程和子进程都会停止。当父进程终止时，调试程序可以检查从 fork 返回的值，从而决定子进程的 PID，然后为子进程打开/proc 文件。因为子进程在其从 fork 返回之前便停止了，所以调试程序从此时起便完全控制了它。

通过这一接口可以开发若干比较复杂的调试器和性能分析器。例如，dbx 可以通过/proc 依附和脱离任一运行中的进程。这种实现还可以通过 RFS[Rifk 86]访问/proc 文件。这就可以以同样的方法对本地和远程的进程进行控制。系统调用 ptrace 已经过时无用了。还有一些命令，如 ps，也被重新实现，现在它们使用/proc 文件。基于 VM 系统的功能已开发出一种通用数据监察工具，可以动态地修改页面的保护权限。

9.11.3　处理器文件系统

在多处理机中可以通过处理机文件系统[Nadk 92]访问每一个处理器，这个文件系统安装在目录/system/processor 上，每个处理器对应一个文件。文件名是处理器号的十进制数。文件的大小固定，只读，它的数据域包括如下一些信息

- 处理器状态——联机或脱机。
- CPU 类型。
- 以 MHz 计的 CPU 速度。
- 以 KB 计的高速缓存大小。
- 是否有浮点运算部件。
- 绑定在其上的驱动程序。
- 最近一次处理器状态发生变化的时间

此外，文件还包括一个只写的称为 ctl 的文件，只能由超级用户访问。通过向这个文件进行写操作可以触发对某个处理器的操作，如将处理器设置为联机或脱机状态。

处理器文件系统是 SVR4.2 多处理机版本的一部分。今后它可能会被进一步扩充，例如通过在名字空间中为实体对应的文件来增加对处理器集和轻量级进程的标识。

9.11.4　半透明文件系统

半透明文件系统(TFS)[Hend 90]是白 Sun Microsystems 公司开发的支持大型软件开发的文件系统。它的目标是为先进的版本控制和软件构造控制提供必要的机制，并且为 Sun 公司的配置管理工具、网格软件环境和提供支持。Sun 公司将其作为 SunOS 的一个标准部件出售：

在一个大型软件构造环境中，通常有一些非常典型的需求。用户自己通常要维护一个私有的构造树层，他们需要修改其中的某些文件。他们一般不希望保持那些不修改文件的私有副本，同时他们还希望能够与其他开发人员所进行的修改隔离开来。此外，环境还需要提供

版本控制工具，这样用户就可以选择他想访问的那个版本的构建树。

TFS 利用文件系统的 copy—on-write 语义提供上述这些工具。要修改文件时首先要将共享层选中的文件复制到用户的私有层中。为了实现这一功能，TFS 的目录被组织成若干层次，每一层就是一个物理目录。这些层通过一个称为查询链的隐藏文件连在一起，查询链中保存了下一层目录的名字。每一层都类似于这个目录的一个版本，最上一层是最新的版本。

在 TFS 目录中看见的文件是所有这些层文件的组合。缺省情况下总是访问最新版本的文件(层是从前至后访问的)。如果需要较早期的版本，必须显式地顺着查询链进行访问。由于每一层仅是一个简单的目录，这个操作可以在用户层完成。通过除最上一层目录外的其他层次均标记为只读访问权限来实现 copy-on-write 机制；在非顶层的文件要光复制到顶层才能进行修改。

由于每次查询都要搜寻若干个层(在典型系统中，层数会变得很大)，TFS 的性能受到严重影响。TFS 解决这个问题的方法是大量地使用名字查询高速缓存。由于不同机器结构的目标文件彼此不尽相同，TFS 还为此提供了支持不同机器结构对应的不同层的工具。用户程序不用修改就可以访问 TFS 文件。为了利用 TFS 系统管理员必须执行一些初始配置。

尽管 TFS 最初被设计成一个 NFS 服务器，现在它已经变换到直接使用 vnode/vfs 接口了。11.12.1 小节介绍了 4.4BSD 的联合安装文件系统，它也提供了类似的功能，但它是基于 4.4BSD 的堆栈式 v 节点接口实现的。

9.12 以往的磁盘缓存

磁盘 I/O 在任何一个系统中都是最主要的瓶颈。从磁盘上读一个 512 字节的块所需的时间是毫秒级的。而在内存中复制同样的大小的数据所需的时间则是微秒级的。这两者之间的差将近 1000 倍。如果每次文件 I/O 都需要访问磁盘操作，系统就会变得无法忍受的慢。必须采取一切措施来减少磁盘 I/O 操作，UNIX 系统是通过将最近访问的磁盘块在内存中缓存起来达到这个目的的。

传统的 UNIX 系统在内存中使用一块称为磁盘缓存的内存区。虚存系统分别缓存进程的正文和数据页。像 SVR4 和 SunOS(版本 4 及更高的版本)这样的现代 UNIX 系统将磁盘缓存与分页系统集成起来。本节中，我们介绍以往的磁盘缓存。第 14.8 节将介绍新的集成方法。

磁盘缓存由一系列数据缓冲区组成，每一个数据缓冲区的大小都足以保存一个磁盘块。基于 BSD 的系统使用变长的缓冲区，在相同机器上的不同文件系统可能有不同的磁盘块和片段大小，每个缓冲区都对应一个头结构，其中保存命名、同步和缓存管理信息。缓存的大小通常为物理内存的 10%。

缓存的后援存储是数据的持久性存储单元。一个缓存可以管理若干个不同后援存储的数据。对于磁盘缓存来说，后援存储就是磁盘上的文件系统。如果机器是在网络上的，后援存储还包括远程节点上的文件。

通常，缓存可以是写通或写后的。写通缓存将被修改的数据立即写回后援存储中。这有若干优点。后援存储上的数据始终都是当前的(有时可能除了最后一次写操作)，一旦系统崩溃时没有数据丢失及文件系统毁坏等问题。而且缓存管理也是很简单的，这种方法对于硬件实现的缓存是一个非常好的选择，比如某些硬盘上的磁道缓冲区。

由于写通方法有很大的性能代价，它不适合于磁盘缓存。大约有 1/3 的磁盘操作是写操作，其中的许多是临时性的——在写操作执行的若干分钟旦数据被覆盖或是文件被删除。这将会造成许多没必要的写操作，使系统性能明显地慢了下来。

出于这样的原因，UNIX 磁盘缓存基本上是写后的⑦。被修改的块被简单地标记为已修改，然后在以后的某个时刻在写到磁盘上。这就可以使 UNIX 减少了许多写操作，而且还对写操作重新排序来进一步提高磁盘访问的性能。然而将写操作延迟，在系统一旦崩溃时会破坏

文件系统。这个问题将在第 9.12.5 节中讨论。

9.12.1 基本操作

无论何时进程要读写一个数据块时，它都首先在磁盘缓存中查找数据块。为了更高效地完成这个操作，缓存依据数据块的设备号和块号组织成一组哈希队列。如果数据块不在缓存中，就必须从磁盘上读取(除了当整个磁盘块都要被覆盖时)。内核在缓存中分配一个缓冲区，将其对应到这个数据块上，然后启动所需的磁盘读操作。如果数据块已被修改了，内核就相应地修改磁盘缓存中的副本，并在其头部标记为已修改。当已修改的数据块必须释放以便重用时，首先要将其回写磁盘。

当缓冲区正在被使用时，必须首先对其加锁。这一操作发生在启动磁盘 I/O 或之前当进程要对缓冲区进行读写时。如果缓冲区已被锁住，试图对其访问的进程必须睡眠，等待它被解锁。由于磁盘中断处理函数可能也要访问缓冲区，内核在试图获取缓冲区时要屏蔽磁盘中断。

当缓冲区没有被加锁时，它保留在空闲链表上。空闲链表按最近最少使用次序维护。无论何时内核需要一个空闲缓冲区，它都会选择一个未被访问时间最长的缓冲区。这条规则呈根据这样一个事实，即一般的系统使用都有非常好的访问局域性：最近访问的数据很可能要比那些在很长时间没使用的数据先访问。当缓冲区被访问然后又被释放后，将其放置在空闲链表的末尾(从这时刻算它是最近访问的)。随着时间的推移，它逐渐向着链表的表头前进。如果在某个时刻它又被再一次访问，它又返回链表的末尾。当它到达链表的表头时，它就称为了最近最少使用的缓冲区，而且将被某个需要空闲缓冲区的进程重新分配。

这种情形也有例外。首先包括变为无效的缓冲区，或者是由于 I/O 错误，或者是因为它们所属的文件被删除或被截短。此时，由于这些缓冲区可以保证将不会再被访问，它们将被立即放置在队头。另一种情况就是那些到达链表表头的已修改页面，它们将从链表中删除并放置在磁盘驱动程序的写队列中。当写操作完成时，缓冲区被标记为已更新并返回到空闲链表中。因为它已经到达了链表头并且没有再被访问过，它将返回到链表头而不是链表的末尾。

9.12.2 缓冲区头结构

每个缓冲区都用一个缓冲区头结构来表示。内核使用头结构来标识和定位缓冲区，同步对其进行的访问，完成缓存管理。头结构还用做与磁盘驱动程序间的接口，当内核需要从磁盘上读写数据块时，它从头结构中调出相应的 I/O 操作的参数，并且将它传递给磁盘驱动程序。头结构包含磁盘操作所需的所有信息。表 9-2 列出了 struct buf 表示的头结构中的重要数据域。

b_flags 域是一个若干标志的位图。内核使用 B_BUSY 和 B_WANTED 来同步对缓冲区的操作。B_DELWRI 标记缓冲区为已修改的。由磁盘驱动程序使用的标志包括 B_READ，B_WRITE，B_ASYNC，B_DONE 和 B_ERROR。过时的缓冲区用 B_AGE 标志标记其时可重新使用的缓冲区。

9.12.3 优点

设置磁盘缓冲区的主要动机是减少磁盘操作，去除不必要的磁盘 I/O，它确实很好地达成了这个目的。据报告，精心调整的缓存可以达到 90%的命中率[Oust 85]。此外它还有其他几项优点。磁盘缓存通过加锁和请求标志来同步对磁盘数据块的访问。如果两个进程试图访问同一块，只有其中的一个能够锁住它。磁盘缓存在磁盘驱动程序和内核其他部分间提供了一个模块化的接口。内核的其他部分不能访问磁盘驱动程序，整个接口被封装在缓冲区头结构的数据域中。此外，由于缓冲区本身是页面对齐的，磁盘缓存将磁盘 I/O 所需的对齐操作与内核的其他部分隔离开宋。对可能没对齐的内核地址所进行的任意磁盘 I/O 请求就不再是问题了。

表 9-2 struct buf 中的域

域	描述

int b_flags　　状态标志
struct buf * b_forw，* b_back　　指向哈希队列的指针
struct buf * av_forw，* av_back　　指向空闲链表的指针
caddr_t b_addr　　指向数据本身的指针
dev_t b_edev　　设备号
daddr_t b_blkno　　数据在设备上的块号
int b_error　　I/O 错误状态
unsigned b_resid　还需传送的字节数

9.12.4　缺点

除了这么多的优点外，磁盘缓存还有一些非常致命的缺点。首先，缓存的写后这个性质意味着，如果系统崩溃将造成数据丢失。而且还可能会导致磁盘处于不一致状态。这个问题将在 9.12.5 节中进一步探讨。其次，尽管减少磁盘访问可以明显地提高性能，但数据必须要复制两次——一次是从磁盘到缓存，另一次是从缓存到用户地址空间。后者比前者要快几个数量级，正常情况下减少的磁盘访问足以弥补额外造成的内存间复制。但当串行读写一个大文件时，这就会成为一个重要因素。事实上，这种串行操作会引起一个称为缓存清洗的相关问题。如果从大文件的一头读到另一头，之后就不再进行访问了，这会造成刷新缓存的效果。由于在很短的时间内读取了文件所有的数据块，它们消耗掉了缓存中的所有缓冲区，刷新其中所有的数据。于是造成在一段时间内的大量的缓存失效，降低了系统的性能，这样的情况直到缓存重新被恢复到一个较稳定的数据块集后才结束。如果用户能够预见到这种情况，就可以避免缓存清洗(wiping)。例如，Veritas 文件系统(VxFS)允许用户提供一些线索来说明文件将如何被访问，利用这一特性，用户可以关闭大文件的缓存并要求系统直接将数据从磁盘传递到用户空间。

9.12.5　保证文件系统的一致性

磁盘缓存的主要问题是磁盘上的数据并不总是最新的。当系统启动和运行时这不是什么问题，此时内核使用磁盘数据块的缓存副本，它总是最新的。由于若干修改操作可能丢失，当系统崩溃时会出现问题，丢失的数据可能会影响文件的数据块或元数据。UNIX 对这两种情况的处理是不同的。

如果某些文件数据最终没有写到磁盘上，尽管这可能对于用户看来是灾难性的，但从操作系统的角度来看，这样的丢失并不是致命的。这是因为它不会危及文件系统的一致性。要求所有的写操作都要同步的代价是非常高的，因此缺省情况下文件数据的写操作都是写后。但有若干方法可以强制内核将数据写到磁盘上。sync 系统调用将对所有已修改的页面启动写操作。然而它并不等待这些写操作完成，所以 sync 调用完成后并不能保证数据块一定被写到了磁盘上。用户还可以以同步方式打开一个文件，强制对文件的所有写操作都必须是同步的。最后，如果 UNIX 实现中还有一个 update 守护进程(在 SVR4 中称为 fsflush)，它周期性地(典型情况下，30 秒一次)调用 sync 刷新磁盘缓存。

如果某些元数据修改丢失了，文件系统就会变成不一致了。许多文件操作修改一个以上的元数据，如果仅仅这些修改中的某些最终达到了磁盘，就会造成文件系统的不一致。例如，为文件增加一个连接要求为新名字在合适的目录中写一个表项并累加 i 节点中的连接计数。假设系统在对目录的更改达到磁盘后，但在对 i 节点的更改到达前崩溃了，当系统重启后，它将有两个目录项引用同一个文件，但连接计数为 1。如果某个用户使用其中任一个名字删除了文件，由于连接计数被减为 0，i 节点和磁盘数据块就被释放掉。另一个目录项将指向一个未分配的 i 节点(或者又被重新分配给另一个文件)。这种对文件系统的破坏必须加以避免。

在 UNIX 系统中有两种方法来防止这样的毁坏。其一是，内核选择一种元数据更新的顺序来减少系统崩溃的影响。在前面那个例子中，考虑将写操作的顺序倒过来会怎么样呢?现在假

设系统崩溃了，i 节点被更新了，但目录没有。当系统重启后，这个文件有了一个额外的连接，但原始的目录项是有效的且文件可以毫无问题地进行访问。如果某人要删除这个文件，目录项将被删掉，但 i 节点和数据由于连接计数尚为 1 而不能被释放掉。尽管这样不能防止毁坏，但它引起的破坏比起前者来要小得多。

因此更新元数据的顺序必须仔细地选择。由于磁盘驱动程序并不按接收到的请求的顺序处理请求，强制次序的问题依然存在。内核可以对写操作排序的唯一方法就是将这些操作同步进行。因此，在上面那个例子中，内核将 i 节点写回磁盘并等待其写操作结束，然后再启动目录写操作。内核在许多需要修改一个以上相关对象的操作中都使用这种同步元数据写操作。

另一个对付文件系统毁坏的方法是 fsck(文件系统检查)工具[Kowa 78，Bina 89]。这个程序检查一个文件系统，查找其中的不一致之处，并尽可能修复它们。当修改的方法不是很明确时，它就提示用户给出相应的指令。缺省情况下，系统管理员在每次系统重启时都要运行 fsck，并且还要手工运行这个工具。fsck 使用磁盘驱动程序的原始接口访问文件系统这些将在第 11.2.4 节中进一步介绍。

9.13 小结

vnode/vfs 接口可以使多个文件系统同时共存于一台机器中。本章中，我们介绍了几种文件系统的实现。我们首先介绍了两种非常普遍的本地文件系统——s5fs 和 FFS。然后我们介绍了几种特殊用途的文件系统，它们利用 vhode/vfs 接口的特殊性质提供了一些非常有用的功能。最后，介绍了磁盘缓存，它是所有文件共享的全局资源。

在下面的章节中我们将继续介绍其他文件系统。下一章将讨论分布式文件系统——特别是 NFS，RFS，AFS 和 DFS。第 11 章将介绍先进的和实验性的文件系统，它们使用了诸如日志等技术，有更好功能和性能。

9.14 练习

1.为什么在 s5fs 和 FFS 中每个文件系统的磁盘 i 节点数目固定?

2.为什么文件的 i 节点和文件的目录项是分离的?

3.将一个文件的所有数据块都分配在一块连续的磁盘空间上有什么优点?又有什么缺点?什么样的应用程序可能需要这样的文件系统?

4.如果一个磁盘错误毁掉了 s5fs 的超级块将会发生什么情况?

5.动态的分配和释放 i 节点有什么好处?

6.使用基于引用目录名查找高速缓存的系统有时会耗尽了空闲 i 节点，而这仅仅是因为查找缓存引用了太多的 i 节点造成的，否则这些 i 节点就都是空闲的了。文件系统该如何处理这种情况?

7.对于像 s5fs 和 FFS 这样的传统文件系统来说,在一个大目录中进行名字查找是非常低效的。试探讨将目录组织成一个哈希表的可能性。哈希表是仅仅在内存中呢?还是应该有一部分在持久性介质上?这样做是否和名字查找缓存功能重复？

8.在 4.4BSD 中，名字查找缓存还为不成功的查找维护表项。缓存这些信息的好处是什么?实现中要注意的问题有哪些?

9.为什么 write 系统调用有时要首先从磁盘上读数据块?

10.为什么 FFS 要在与父目录不同的柱面组上分配新目录?

11.在什么情况下旋转延时因子会降低系统的性能?

12.为什么使用现代 SCSI 硬盘降低了使用旋转算法的 FFS 的性能?

13.FFS 中预留的空间的用途是什么?

14.假设某个文件系统要将小文件的数据部分保留在它的 i 节点而不是一个独立的数据块上。这种方法的优点和问题各是什么?

15.临时文件使用一种特殊文件系统的优点有哪些?

16.操作系统可以做哪些工作来降低缓存的清除操作?

17.将高速缓存系统从虚存系统中独立出来有哪些好处?缺点有哪些?

9.15 参考文献

① 最初，ufs 在不同的变体中有不同的含义。基于 System V 的版本是指它们自身的文件系统，现在这个文件系统称为 s5fs。基于 BSD 的系统与本节使用 ufs 和 s5fs 的方式一样。直到 SVR4 接受了这种约定后，这一混淆才得以消除。

② 许多 UNIX 变体使用单写音、多读者锁，可以获得更好的并发性。

③ 一个页面是一个内存抽象。它可以包含一个块，多个块或者部分块。

④ FFS 最早出现在 4.1bBSD，这是伯克刊内部最初的测试版本。FFS 也是 4.1cBSD 的一部分，这也是一个测试版本，共发送给了大约 100 个站点[Salu 94]。

⑤ 事实上，盘片的两边都使用了，每一面都有一个独立的磁头。

⑥ 在 FFS 中磁盘阵列的直接块数目从 10 增加到 12。

⑦ 某些元数据的更新要进行同步回写，9.12.5 节将讨沦这个内容。