文件系统

概述

目标

（1）了解文件系统相关的概念
（2）掌握文件系统的数据结构和管理方法
（3）学习文件系统与虚拟内存结合使用
（4）实现mmap内存映射到文件

简述

文件系统是计算机操作系统中用于管理和组织文件和目录的一种机制。它提供了对存储设备上数据的组织、访问和操作，使用户能够方便地创建、读取、写入和删除文件，以及组织和管理文件的层次结构。

文件系统的主要目标是提供数据的持久性存储和可靠性保护。它通过将文件和目录组织成层次结构，通常采用树状结构，使用户能够按照目录路径访问文件。文件系统还提供了对文件的命名、权限管理和保护机制，以控制对文件的访问权限和数据的完整性。

文件系统还支持对文件进行扩展和截断，以及对文件进行随机访问。它使用文件描述符来表示打开的文件，并提供了文件指针来跟踪读写操作的位置。文件系统还提供了缓存机制，将常用的数据缓存到内存中，以提高文件访问的性能。

在现代操作系统中，文件系统通常支持各种类型的文件，如普通文件、目录、符号链接等。此外，文件系统还提供了一些特殊文件和设备文件，用于与硬件设备进行交互和通信。

总之，文件系统是计算机操作系统中用于管理和组织文件和目录的一种机制，它提供了数据的持久性存储和可靠性保护，并支持文件的创建、读取、写入和删除，以及文件的权限管理和保护机制。文件系统在操作系统中起着重要的作用，为用户和应用程序提供了方便的文件操作和数据存储功能。

- 实验 1 - 软链接
- 实验 2 - large file
- 实验 3 - mmap

讲解

xv6文件系统实现分为七层，如图所示。磁盘层读取和写入 virtio 硬盘上的块。缓冲区高速缓存层缓存磁盘块并同步对它们的访问，确保每次只有一个内核进程可以修改存储在任何特定块中的数据。日志记录层允许更高层在一次事务（transaction）中将更新包装到多个块，并确保在遇到崩溃时自动更新这些块（即，所有块都已更新或无更新）。索引结点层提供单独的文件，每个文件表示为一个索引结点，其中包含唯一的索引号（i-number）和一些保存文件数据的块。目录层将每个目录实现为一种特殊的索引结点，其内容是一系列目录项，每个目录项包含一个文件名和索引号。路径名层提供了分层路径名，如 ***/usr/rtm/xv6/fs.c*** ，并通过递归查找来解析它们。文件描述符层使用文件系统接口抽象了许多 Unix 资源（例如，管道、设备、文件等），简化了应用程序员的工作。

文件描述符（File descriptor）
路径名（Pathname）
目录（Directory）
索引结点（Inode）
日志（Logging）
缓冲区高速缓存（Buffer cache）
磁盘（Disk）

图8.1 XV6文件系统的层级

文件系统必须有将索引节点和内容块存储在磁盘上哪些位置的方案。为此，xv6将磁盘划分为几个部分，如图8.2所示。文件系统不使用块0（它保存引导扇区）。块1称为超级块：它包含有关文件系统的元数据（文件系统大小（以块为单位）、数据块数、索引节点数和日志中的块数）。从2开始的块保存日志。日志之后是索引节点，每个块有多个索引节点。然后是位图块，跟踪正在使用的数据块。其余的块是数据块：每个都要么在位图块中标记为空闲，要么保存文件或目录的内容。超级块由一个名为 `mkfs` 的单独的程序填充，该程序构建初始文件系统。

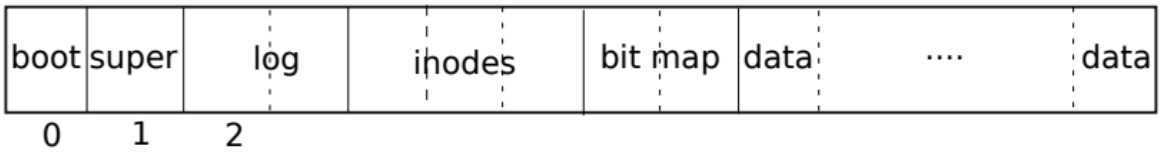

Figure 8.2: Structure of the xv6 file system.

本章的其余部分将从缓冲区高速缓存层开始讨论每一层。注意那些在较低层次上精心选择的抽象可以简化较高层次的设计的情况。

Buffer cache 层

Buffer cache有两个任务：

1. 同步对磁盘块的访问，以确保磁盘块在内存中只有一个副本，并且一次只有一个内核线程使用该副本
2. 缓存常用块，以便不需要从慢速磁盘重新读取它们。代码在***bio.c***中。

Buffer cache层导出的主接口主要是 `bread` 和 `bwrite` ；前者获取一个*buf*，其中包含一个可以在内存中读取或修改的块的副本，后者将修改后的缓冲区写入磁盘上的相应块。内核线程必须通过调用 `brelse` 释放缓冲区。Buffer cache每个缓冲区使用一个睡眠锁，以确保每个缓冲区（因此也是每个磁盘块）每次只被一个线程使用； `bread` 返回一个上锁的缓冲区， `brelse` 释放该锁。

让我们回到Buffer cache。Buffer cache中保存磁盘块的缓冲区数量固定，这意味着如果文件系统请求还未存放在缓存中的块，Buffer cache必须回收当前保存其他块内容的缓冲区。Buffer cache为新块回收最近使用最少的缓冲区。这样做的原因是认为最近使用最少的缓冲区是最不可能近期再次使用的缓冲区。

日志层

文件系统设计中最有趣的问题之一是崩溃恢复。出现此问题的原因是，许多文件系统操作都涉及到对磁盘的多次写入，并且在完成写操作的部分子集后崩溃可能会使磁盘上的文件系统处于不一致的状态。例如，假设在文件截断（将文件长度设置为零并释放其内容块）期间发生崩溃。根据磁盘写入的顺序，崩溃可能会留下对标记为空闲的内容块的引用的inode，也可能留下已分配但未引用的内容块。

后者相对来说是良性的，但引用已释放块的inode在重新启动后可能会导致严重问题。重新启动后，内核可能会将该块分配给另一个文件，现在我们有两个不同的文件无意中指向同一块。如果xv6支持多个用户，这种情况可能是一个安全问题，因为旧文件的所有者将能够读取和写入新文件中的块，而新文件的所有者是另一个用户。

Xv6通过简单的日志记录形式解决了文件系统操作期间的崩溃问题。xv6系统调用不会直接写入磁盘上的文件系统数据结构。相反，它会在磁盘上的*log*（日志）中放置它希望进行的所有磁盘写入的描述。一旦系统调用记录了它的所有写入操作，它就会向磁盘写入一条特殊的*commit*（提交）记录，表明日志包含一个完整的操作。此时，系统调用将写操作复制到磁盘上的文件系统数据结构。完成这些写入后，系统调用将擦除磁盘上的日志。

如果系统崩溃并重新启动，则在运行任何进程之前，文件系统代码将按如下方式从崩溃中恢复。如果日志标记为包含完整操作，则恢复代码会将写操作复制到磁盘文件系统中它们所属的位置。如果日志没有标记为包含完整操作，则恢复代码将忽略该日志。恢复代码通过擦除日志完成。

为什么 xv6的日志解决了文件系统操作期间的崩溃问题？如果崩溃发生在操作提交之前，那么磁盘上的登录将不会被标记为已完成，恢复代码将忽略它，并且磁盘的状态将如同操作尚未启动一样。如果崩溃发生在操作提交之后，则恢复将重播操作的所有写入操作，如果操作已开始将它们写入磁盘数据结构，则可能会重复这些操作。在任何一种情况下，日志都会使操作在崩溃时成为原子操作：恢复后，要么操作的所有写入都显示在磁盘上，要么都不显示。

索引结点层

术语inode（即索引结点）可以具有两种相关含义之一。它可能是指包含文件大小和数据块编号列表的磁盘上的数据结构。或者“inode”可能指内存中的inode，它包含磁盘上inode的副本以及内核中所需的额外信息。

磁盘上的inode都被打包到一个称为inode块的连续磁盘区域中。每个inode的大小都相同，因此在给定数字n的情况下，很容易在磁盘上找到第n个inode。事实上，这个编号n，称为inode number或i-number，是在具体实现中标识inode的方式。

磁盘上的inode由 `struct dinode` （***kernel/fs.h***:32）定义。字段 `type` 区分文件、目录和特殊文件（设备）。 `type` 为零表示磁盘inode是空闲的。字段 `nlink` 统计引用此inode的目录条目数，以便识别何时应释放磁盘上的inode及其数据块。字段 `size` 记录文件中内容的字节数。 `addrs` 数组记录保存文件内容的磁盘块的块号。

内核将活动的inode集合保存在内存中； `struct inode` （***kernel/file.h***:17）是磁盘上 `struct dinode` 的内存副本。只有当有C指针引用某个inode时，内核才会在内存中存储该inode。 `ref` 字段统计引用内存中inode的C指针的数量，如果引用计数降至零，内核将从内存中丢弃该inode。 `iget` 和 `iput` 函数分别获取和释放指向inode的指针，修改引用计数。指向inode的指针可以来自文件描述符、当前工作目录和如 `exec` 的瞬态内核代码。

xv6的inode代码中有四种锁或类似锁的机制。`icache.lock`保护以下两个不变量：inode最多在缓存中出现一次；缓存inode的`ref`字段记录指向缓存inode的内存指针数量。每个内存中的inode都有一个包含睡眠锁的`lock`字段，它确保以独占方式访问inode的字段（如文件长度）以及inode的文件或目录内容块。如果inode的`ref`大于零，则会导致系统在cache中维护inode，而不会对其他inode重用此缓存项。最后，每个inode都包含一个`nlink`字段（在磁盘上，如果已缓存则复制到内存中），该字段统计引用文件的目录项的数量；如果inode的链接计数大于零，xv6将不会释放inode。

`iget()`返回的`struct inode`指针在相应的`iput()`调用之前保证有效：inode不会被删除，指针引用的内存也不会被其他inode重用。`iget()`提供对inode的非独占访问，因此可以有许多指向同一inode的指针。文件系统代码的许多部分都依赖于`iget()`的这种行为，既可以保存对inode的长期引用（如打开的文件和当前目录），也可以防止争用，同时避免操纵多个inode（如路径名查找）的代码产生死锁。

`iget`返回的`struct inode`可能没有任何有用的内容。为了确保它保存磁盘inode的副本，代码必须调用`ilock`。这将锁定inode（以便没有其他进程可以对其进行`ilock`），并从磁盘读取尚未读取的inode。`iunlock`释放inode上的锁。将inode指针的获取与锁定分离有助于在某些情况下避免死锁，例如在目录查找期间。多个进程可以持有指向`iget`返回的inode的C指针，但一次只能有一个进程锁定inode。

inode 缓存只缓存内核代码或数据结构持有 C 指针的 inode。它的主要工作实际上是同步多个进程的访问；缓存是次要的。如果经常使用 inode，在 inode 缓存不保留它的情况下 buffer cache 可能会将其保留在内存中。inode 缓存是直写的，这意味着修改已缓存 inode 的代码必须立即使用`iupdate`将其写入磁盘。

文件描述符层

Unix界面的一个很酷的方面是，Unix中的大多数资源都表示为文件，包括控制台、管道等设备，当然还有真实文件。文件描述符层是实现这种一致性的层。

正如我们在第1章中看到的，Xv6为每个进程提供了自己的打开文件表或文件描述符。每个打开的文件都由一个`struct file`（***kernel/file.h***:1）表示，它是inode或管道的封装，加上一个I/O偏移量。每次调用`open`都会创建一个新的打开文件（一个新的`struct file`）：如果多个进程独立地打开同一个文件，那么不同的实例将具有不同的I/O偏移量。另一方面，单个打开的文件（同一个`struct file`）可以多次出现在一个进程的文件表中，也可以出现在多个进程的文件表中。如果一个进程使用`open`打开文件，然后使用`dup`创建别名，或使用`fork`与子进程共享，就会发生这种情况。引用计数跟踪对特定打开文件的引用数。可以打开文件进行读取或写入，也可以同时进行读取和写入。`readable`和`writable`字段可跟踪此操作。

系统中所有打开的文件都保存在全局文件表`ftable`中。文件表具有分配文件（`filealloc`）、创建重复引用（`filedup`）、释放引用（`fileclose`）以及读取和写入数据（`fileread`和`filewrite`）的函数。

前三个函数遵循现在熟悉的形式。`Filealloc`（***kernel/file.c***:30）扫描文件表以查找未引用的文件（`f->ref == 0`），并返回一个新的引用；`filedup`（***kernel/file.c***:48）增加引用计数；`fileclose`（***kernel/file.c***:60）将其递减。当文件的引用计数达到零时，`fileclose`会根据`type`释放底层管道或inode。

函数`filestat`、`fileread`和`filewrite`实现对文件的`stat`、`read`和`write`操作。`Filestat`（***kernel/file.c***:88）只允许在inode上操作并且调用了`stati`。`Fileread`和`filewrite`检查打开模式是否允许该操作，然后将调用传递给管道或inode的实现。如果文件表示inode，`fileread`和`filewrite`使用I/O偏移量作为操作的偏移量，然后将文件指针前进该偏移量（***kernel/file.c***:122-123）（***kernel/file.c***:153-154）。管道没有偏移的概念。回想一下，inode的函数要求调用方处理锁（***kernel/file.c***:94-96）（***kernel/file.c***:121-124）（***kernel/file.c***:163-

166）。inode锁定有一个方便的副作用，即读取和写入偏移量以原子方式更新，因此，对同一文件的同时多次写入不能覆盖彼此的数据，尽管他们的写入最终可能是交错的。

实验

实验介绍

实验内容 1 - symlink

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab8-1-symlink    #切换到本实验的分支
```

任务：

- 添加系统调用 `symlink(char *target, char *path)`,向 xv6添加符号链接。
- 符号链接（或软链接）是指按路径名链接的文件；当一个符号链接打开时，内核跟随该链接指向引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。尽管 xv6不支持多个设备，但实现此系统调用是了解路径名查找工作原理的一个很好的练习。

Tips：

- https://stackoverflow.com/questions/16912997/what-is-there-behind-a-symbolic-link
- https://www.serveradminz.com/blog/essential-differences-implementation-symbolic-links-windows-linux/
- https://www.ibm.com/support/pages/explanation-symbolic-link-and-how-create-it

实验内容 2 - larger file

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab8-2-largefile    #切换到本实验的分支
```

任务：

- 在本作业中，您将增加 xv6文件的最大大小。
- 目前，xv6文件限制为268个块或 `268*BSIZE` 字节（在 xv6中 `BSIZE` 为1024）。此限制来自以下事实：一个 xv6 inode 包含12个“直接”块号和一个“间接”块号，“一级间接”块指一个最多可容纳256个块号的块，总共12+256=268个块。
- 您将更改 xv6文件系统代码，以支持每个 inode 中可包含256个一级间接块地址的“二级间接”块，每个一级间接块最多可以包含256个数据块地址。结果将是一个文件将能够包含多达65803个块，或256*256+256+11个块（11而不是12，因为我们将为二级间接块牺牲一个直接块号）。

Tips：

- https://en.wikipedia.org/wiki/Inode_pointer_structure
- https://www.sans.org/blog/understanding-indirect-blocks-in-unix-file-systems/

实验内容 3 - mmap

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab8-3-mmap    #切换到本实验的分支
```

任务：

- 添加系统调用 `mmap` 和 `munmap`.
- `mmap` 和 `munmap` 系统调用允许 UNIX 程序对其地址空间进行详细控制。它们可用于在进程之间共享内存，将文件映射到进程地址空间.
- 本实验只需要与内存映射文件相关的功能子集。您可以假设 `addr` 始终为零，这意味着内核应该决定映射文件的虚拟地址。 `mmap` 返回该地址，如果失败则返回 `0xffffffffffffffff`。 `length` 是要映射的字节数；它可能与文件的长度不同。 `prot` 指示内存是否应映射为可读、可写，以及/或者可执行的；您可以认为 `prot` 是 `PROT_READ` 或 `PROT_WRITE` 或两者兼有。 `flags` 要么是 `MAP_SHARED` （映射内存的修改应写回文件），要么是 `MAP_PRIVATE` （映射内存的修改不应写回文件）。您不必在 `flags` 中实现任何其他位。 `fd` 是要映射的文件的打开文件描述符。可以假定 `offset` 为零（它是要映射的文件的起点）。
- 允许进程映射同一个 `MAP_SHARED` 文件而不共享物理页面。
- `munmap(addr, length)` 应删除指定地址范围内的 `mmap` 映射。如果进程修改了内存并将其映射为 `MAP_SHARED` ，则应首先将修改写入文件。

Tips：

- https://man7.org/linux/man-pages/man2/mmap.2.html
- https://www.sobyte.net/post/2022-03/mmap/

拓展实验内容

> 上文是实验中 `mmap` 仅仅支持了将文件映射到内存，请你拓展 `mmap` 的实现，使其更加接近真实 Linux 上实现，可以用于进程间共享内存等。

Tips：

- https://man7.org/linux/man-pages/man2/mmap.2.html
- https://www.sobyte.net/post/2022-03/mmap/