title: xv6-文件系统 date: 2019-08-31 01:47:50

tags: 学习

文件系统总体设计

文件系统分层

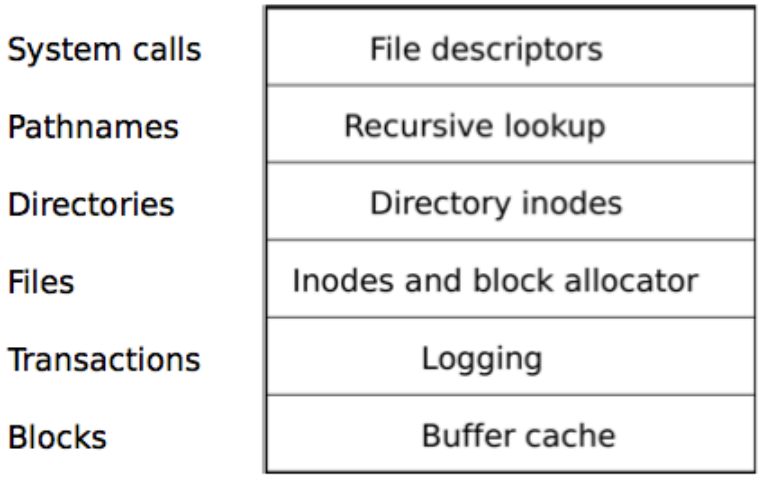

Figure 6-1. Layers of the xv6 file system.

如图所示xv6的文件系统分6层实现：

1. 第一层（自下而上）通过块缓冲读写 IDE 硬盘，同步对磁盘的访问，且通过块的锁保证同一时间只有一个内核进程可以修改磁盘块；
2. 第二层使得更高层的接口可以将对磁盘的更新按会话打包，通过会话的方式来保证这些操作是原子操作。
3. 第三层提供无名文件，每一个文件由一个inode和一连串的数据块组成。
4. 第四层将目录实现为一种特殊的inode，内容是一连串的目录项，每一个目录项包含一个文件名和对应的inode。
5. 第五层提供了层次路经名（如/xx/xxx），这一层通过递归的方式来查询路径对应的文件。
6. 第六层将许多资源（如管道，设备，文件等）抽象为文件系统的接口，极大地简化了程序员的工作。

磁盘分层

磁盘分层为引导块、超级块、日志块、inode块、空闲块位图和数据块等。

文件系统不使用引导块；超级块包含了文件系统的元信息；日志块中的日志维护了文件系统操作的原子性，

防止操作过程中引发的系统崩溃而导致的数据不同步；inode块用于存放inode（一个块可存放多个inode）；空闲块位图便于查找空闲的块；数据块中保存了文件和目录等内容。

块缓冲层

需要实现的目标

1. 同步对磁盘的访问。对于任意一个块，同一时间仅有一份拷贝放在内存中且只允许一个内核线程对该拷贝进行修改。
2. 缓存常用的块以减少磁盘读取次数，提高系统效率。

实现方案

代码参见 `bio.c` 。

根据时间与空间局部性原理，将最近经常访问的磁盘块缓存在内存中，并使用LRU替换策略。

结构定义

xv6将缓冲区分为两种状态：

1. B_VALID 意味着这个缓冲区拥有磁盘块的有效内容；
2. B_DIRTY 意味着缓冲区的内容已经被改变并且需要写回磁盘；

与旧版相比，B_BUSY 状态在最新版代码中被删去。但由于缓存区中的块的锁的存在，只有持有锁才能访问该块，而在另一线程未使用完成前，该锁不会被释放，此时只能循环等待锁的释放，获得锁后方可获得块。故依旧可保持块内容的原子性。

```
// 内存中的磁盘块结构
struct buf {
  int flags; // 标记磁盘状态，valid/dirty
  uint dev; // 磁盘设备号
  uint blockno; // 块编号
  struct sleeplock lock;
  uint refcnt; // 引用计数
  struct buf *prev; // LRU cache list 使用LRU替换
  struct buf *next; // 链式结构连接磁盘块
  struct buf *qnext; // 磁盘队列
  uchar data[BSIZE]; // 块大小为512字节
};

// 块缓冲区结构
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // 通过 prev/next 连接所有缓冲块
  // 其中 head.next 是最近最常使用的块.
  struct buf head;
} bcache;
```

初始化 binit

从一个静态数组 `buf` 中构建出一个有 `NBUF` 个元素的双向链表。所有对块缓冲的访问都通过链表而非静态数组。

```
void
binit(void)
{
  struct buf *b;

  // 初始化块缓冲锁
  initlock(&bcache.lock, "bcache");

//PAGEBREAK!
  // 构建缓冲双向链表
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    // 初始化缓冲区中的块的锁
    initsleeplock(&b->lock, "buffer");
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}
```

查找 bget

扫描缓冲区链表，通过给定的设备号和扇区号找到对应的缓冲区。如果未找到，则分配一个缓冲区，否则返回一个持有锁的缓冲区。代码中在双向链表的搜索中向最近最常使用方向查找，若未找到则向另一方向查找空闲缓冲区以分配。

其中若找到缓冲区中指定的块，返回的结果也不一定是指定的块。因为在请求该块的锁前已释放缓冲区锁，若该块未持有锁且正在被用来缓冲另外一个块，则重新获得锁时已是另一个块。

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock); // 请求块缓冲区锁

  // 不能保证 b 仍然是可用的缓冲区：它有可能被用来缓冲另外一个块。
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++; // 引用计数加一
      release(&bcache.lock); // 释放块缓冲区锁
      acquiresleep(&b->lock); // 请求该块的锁
      return b;
    }
  }

  // Not cached; recycle an unused buffer. 未找到则重新查找
  // Even if refcnt==0, B_DIRTY indicates a buffer is in use
  // because log.c has modified it but not yet committed it.
  // 即使refcnt等于0，也可通过B_DIRTY知道缓冲区被使用，此时日志已修改但未提交
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->flags = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // 未找到块且未能分配块，引发内核错误
  panic("bget: no buffers");
}
```

读 bread

`bread` 会首先调用 `bget` 从缓存中去寻找块是否存在。如果存在直接返回；如果不存在则请求磁盘读操作，读到缓存中后再返回结果。

```
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  // 获取缓冲区
  b = bget(dev, blockno);
  // 如果缓冲区中不存在指定的块，则从磁盘中读出
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b;
}
```

写 bwrite

设置 B_DIRTY 位并且调用 iderw 将缓冲区的内容写到磁盘。

```
void
bwrite(struct buf *b)
{
  // 该块未锁，可能已被释放，无法写入，引发内核错误
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

释放 brelse

将一块缓冲区移动到链表的头部，唤醒睡眠在这块缓冲区上的进程。

```
// 唤醒睡眠在这块缓冲区上的进程；将一块缓冲区移动到链表的头部。
void
brelse(struct buf *b)
{
  // 缓冲区未持有块锁，引发内核错误
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock); // 释放块锁

  acquire(&bcache.lock); // 请求  缓冲区锁
  b->refcnt--; //引用计数减一
  if (b->refcnt == 0) {
    // 无进程等待使用，移动到链表头部
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }

  release(&bcache.lock); //释放缓冲区锁
}
```

日志层

每一个系统调用都可能包含一个必须从头到尾原子完成的写操作序列，称这样的一个序列为一个会话。任何时候只能有一个进程在一个会话之中，其他进程必须等待当前会话中的进程结束。因此同一时刻日志最多只记录一次会话。

该层用于维护系统崩溃后到原子性，即会话要么完成，要么系统恢复至会话开始前的状态。

块分配器

本部分涉及的函数位于 `fs.c` 。

由于 `bread` 与 `brelse` 已有锁进行互斥，块分配器无需再加锁。

分配新的磁盘块

```
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){ // 读位图每一块
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){ // 读块内每一位
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // 判断块是否空闲
        bp->data[bi/8] |= m;  // 将块标记为使用中
        log_write(bp);
        brelse(bp); // 释放块
        bzero(dev, b + bi); // 清零块
        return b + bi;
      }
    }
    brelse(bp);
  }
  // 超出块范围，引发错误
  panic("balloc: out of blocks");
}
```

释放磁盘块

```
static void
bfree(int dev, uint b)
{
  struct buf *bp;
  int bi, m;

  readsb(dev, &sb);
  bp = bread(dev, BBLOCK(b, sb));
  bi = b % BPB;
  m = 1 << (bi % 8);
  if((bp->data[bi/8] & m) == 0) // 释放空闲块，错误
    panic("freeing free block");
  bp->data[bi/8] &= ~m;
  log_write(bp);
  brelse(bp); // 释放块
}
```

inode

inode有两种类别：

- 磁盘上的记录文件大小、数据块扇区号的数据结构

```
struct dinode {
  short type;           // 区分文件、目录和特殊文件的 i 节点，0表示为空闲节点
  short major;          // 主设备号（仅限T_DEV）
  short minor;          // 辅设备号（仅限T_DEV）
  short nlink;          // 文件系统中的i节点连接数
  uint size;            // 文件的字节数
  uint addrs[NDIRECT+1]; // 用于这个文件的数据块的块号
};
```

- 内存中的一个 i 节点，包含一个磁盘上 i 节点的拷贝，以及一些内核需要的附加信息

```
struct inode {
  uint dev;           // Device number 设备号
  uint inum;          // Inode number inode号
  int ref;            // Reference count 引用数
  struct sleeplock lock; // protects everything below here 保护以下所有数据
  int valid;          // inode has been read from disk? i节点是否已从磁盘中读取

  // 以下均为磁盘上的i节点，即dinode，的拷贝
  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

xv6中有inode的缓冲区，作用与块缓冲区相似。以下函数分析仅包含进行特殊操作或较为复杂的函数，其余函数仅写出作用和使用条件（如 `iunlock` ）。其中函数位于 `fs.c` 。

获取inode iget

iget 返回的 i 节点的指针将保证这个 i 节点会留在缓存中，不会被删掉且不会被用于缓存另一个文件（即使未读入磁盘dinode内容）。若磁盘内容未读入，则以后在调用 `ilock` 时读入。

```
static struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // 在inode的cache中寻找目标inode
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      ip->ref++;
      release(&icache.lock);
      return ip; // 此时返回的i节点内容不一定已从磁盘读入cache
    }
    if(empty == 0 && ip->ref == 0)    // 记录扫描到的第一个空槽
      empty = ip;
  }

  // 保持inode入口，以下未从磁盘读入内容
  if(empty == 0) // cache中无i节点空闲，出错
    panic("iget: no inodes");

  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->valid = 0;
  release(&icache.lock);

  return ip;
}
```

分配inode ialloc

在设备dev上分配inode：通过给它类型类型将其标记为已分配，返回未锁定但已分配和引用的inode。这里与 `balloc` 相似，可参照前文。

```
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;

  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // inode未分配
      memset(dip, 0, sizeof(*dip)); // 分配空间
      dip->type = type;
      log_write(bp);
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

锁inode ilock

锁定给定的inode。如有必要，从磁盘读取inode。

```
void
ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquiresleep(&ip->lock); // 请求inode锁

  if(ip->valid == 0){ // inode内容未从磁盘读取
    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
    brelse(bp);
    ip->valid = 1;
    if(ip->type == 0) // inode未被分配，出错，引发恐慌
      panic("ilock: no type");
  }
}
```

释放引用 iput

释放对内存中的某一inode的引用，即引用数减一。对iput（）的所有调用必须在会话内，因为它可能释放inode。

```
void
iput(struct inode *ip)
{
  acquiresleep(&ip->lock); // 请求inode锁
  // 如果这是最后一个引用，则可以回收inode缓存条目。
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
      // inode has no links and no other references: truncate and free.
      // 如果那是最后一个引用并且inode没有连接，释放磁盘上的inode（及其内容）。
      itrunc(ip);
      ip->type = 0;
      iupdate(ip);
      ip->valid = 0;
    }
  }
  releasesleep(&ip->lock); // 释放 inode 锁

  acquire(&icache.lock);
  ip->ref--; // 引用计数减一
  release(&icache.lock);
}
```

其他函数

- readsb 读超级块
- bzero 将块内容清零
- iupdate 将修改后的内存中inode复制到磁盘；必须在每次更改ip-> xxx字段后调用；它存在于磁盘上，因为i-node缓存是直写式的；调用者必须持有i节点锁。
- iunlock 对指定inode解锁
- iunlockput iunlock和iput的结合。由于两种常常一起使用，故整合。
- bmap 返回inode中第n个块的磁盘块地址，如果没有这样的块，bmap会分配一个。
- stati 从inode复制属性信息。
- readi 从inode读数据，调用者必须持有ip-> lock。
- writei 给inode写入数据，调用者必须持有ip-> lock。

其中readi和writei均要求给定的偏移和读出的量不超出文件大小。

目录层

数据结构

目录的i节点的类型是T_DIR,.目录本身是以文件的方式存储到磁盘上的，它的数据是一系列的目录条目。

```
struct dirent {
  ushort inum; // i节点号
  char name[DIRSIZ]; // 目录名
};
```

函数

详细注释位于 `fs.c` 。

- dirlookup 查找目录中指定名字的条目
- dirlink 会写入一个新的目录条目到某一目录下

路径

函数

- namex

查找并返回inode以获取路径名。如果parent！= 0，则返回父项的inode并复制最终项。路径元素到名称，必须有DIRSIZ字节的空间。该函数必须在会话内部调用，因为它调用iput（）。

```
static struct inode*
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;

  if(*path == '/') // 路径以反斜杠开始，则解析从根目录开始
    ip = iget(ROOTDEV, ROOTINO);
  else // 其他情况下则从当前目录开始
    ip = idup(myproc()->cwd);

  while((path = skipelem(path, name)) != 0){ // 考虑路径中的每一个部分
    ilock(ip); // 确保ip->type从磁盘中加载出来
    if(ip->type != T_DIR){ // 不是目录，查找失败
      iunlockput(ip);
      return 0;
    }
    // 最后一个路径元素，循环提前结束
    if(nameiparent && *path == '\0'){
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){ // 寻找路径元素失败，查找失败
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}
```

- skipelem 将路径中的下一个路径元素复制到名称中，返回指向复制后的元素的指针。
- namei 解析path并返回对应的inode。
- nameiparent 返回上级目录的i节点并且把最后一个元素拷贝到name中。

函数的详细内容位于 `fs.c`

文件描述符层

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type; // 文件分为管道文件和普通文件等，FD_NONE表示文
  int ref; // reference count 引用计数
  char readable; // 可读
  char writable; // 可写
  struct pipe *pipe; // 管道
  struct inode *ip; // 指向i节点
  uint off;
};
```

xv6中每个进程都有自己的打开文件表。每个打开文件均由上面的结构体表示，它是一个对i节点或者管道和文件偏移的封装。每次调用 `open` 都会创建一个新的file结构体。

如果多个进程相互独立地打开了同一个文件，不同的实例将拥有不同的I/O偏移。而且同一个file结构体可以在一个进程的文件表中多次出现，同时也可以在多个进程的文件表中出现。对每一个打开的文件都有一个引用计数，一个文件可以被打开用于读、写或者二者。

系统中所有的打开文件都存在于一个全局的文件表 `ftable` 中。

函数

该层的函数位于 `file.c` ，详细注释见代码文件。

- filealloc 分配文件，扫描整个文件表寻找没有被引用的文件并返回一个新的引用。
- filedup 重复引用文件，增加引用计数。
- fileclose 释放对文件引用的函数，减少引用计数，当一个文件的引用计数变为0的时候，根据文件类型的不同,释放掉当前的管道或者i节点。
- fileread 读文件，实现对文件的 `read` 操作
- filewrite 写文件，实现对文件的 `write` 操作。
- filestat 获取文件的元信息，实现对文件的 `stat` 操作（只允许作用在inode）。

系统调用 此处为草稿标签!

文件系统中xv6可改进之处

cache

1. 使用的替换策略为LRU，算法较为简单。
2. 链表的使用虽然实现简单，但查找效率低，时间复杂度高。若能使用哈希表等实现查找可提高效率。

目录查找

目录查找为线性查找，在目录数量较多的情况下耗时巨大。若能使用树等结构，便能大大减小时间复杂度。

大小固定

文件系统的大小在xv6中是不改变的，固定在一个磁盘设备上。这一点对于个人PC影响不大，但在需要存储大量文件或超大文件的情况下，便有些捉襟见肘。