XV6 文件系统
阅读报告

姓名 陈肯 袁子栩 郭一臣 王旭升 张颜

1.了解 UNIX 文件系统的主要组成部分和作用

①超级块

文件系统的第一块（引导块之后的第一块），这个块中保存了文件系统本身的结构信息，包括每个区域的大小以及未被使用的磁盘块的信息

②inode

inode 中记录了文件的基本信息，如文件大小，创建时间等。在 unix 下，所有的 inode 被统一管理在索引结点区，索引节点区在超级块之后

③数据块

数据块是真正保存数据的块，统一存放在数据区，数据区在索引结点区后，磁盘上所有块的大小都相同，如果文件的内容超过了一个块的大小，则文件会存放于多个磁盘块。

④目录块

在数据块中保存一系列索引结点到文件名的映射，这样的数据块就是目录块。目录块中存放了目录，目录的前两个链接是“.”和“..”，分别表示当前目录与上级目录，根目录的上级目录指向自己。

⑤间接块

由于一个 inode 只包含一个最多含有 13 个项的分配链表，如果分配的数据块超过 13 个块，就需要引入间接块。间接块是用来存放 inode 里放不下的数据块编号的块，间接块位于数据区。例如一个文件被分配了 15 个块，那可以把前 10 个块存放于 inode 中，第 11 到 15 个块的编号存放于间接块中，此间接块的编号作为 inode 存放的第 11 个块编号。

二、阅读 ide.c（ide 硬盘驱动程序），对其内容作大致了解

1.xv6 维护了一个用于请求磁盘操作的队列 idequeue：

```
static struct spinlock idelock;
static struct buf *idequeue;
```

2.等待磁盘进入空闲状态：

```
static int
idewait(int checkerr)
{
  int r;

  while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
  if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}
```

3.初始化 ide 磁盘 io：

```
void
ideinit(void)
```

包含对队列的锁和 io 的初始化

```
  initlock(&idelock, "ide");
  picenable(IRQ_IDE);
  ioapicenable(IRQ_IDE, ncpu - 1);
  idewait(0);
```

4.开始一个磁盘读写请求

```
static void
idestart(struct buf *b)
```

对于脏的 buffer，要写回磁盘

```
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
```

5.当磁盘请求完成后，中断处理程序调用以下函数：

```
void
ideintr(void)
```

如果要请求的是队首 buffer：

```
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);

    return;
```

唤醒等待该 buffer 的进程：

```
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);
```

使用 idestart 开始处理下一个请求

```
  if(idequeue != 0)
    idestart(idequeue);
```

6.上层文件系统调用的磁盘 io 接口：

```
void
iderw(struct buf *b)
```

当使用 void iderw(struct buf *b)请求读写磁盘时，该请求被加入等待队列 idequeue，同时调用此函数的进程进入睡眠状态，当磁盘完成一此读写操作时，会触发中断，并唤醒 idequeue 队头进程

```
 acquire(&idelock);  //DOC:acquire-lock

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
```

```
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }
  release(&idelock);
```

三、阅读 buf.h，bio.c

buf.h 中对 xv6 的磁盘块数据结构进行了定义，并规定一块的大小为 512 字节。

1. bio.c 是 buffer cache 的具体实现

实现了读写函数 struct buf* bread(uint dev , unit sector)，void bwrite(struct buf* b)，读函数 bread()会首先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。写函数 bwrite()会直接将缓存中的数据写入磁盘。

2.buffer 链表的数据结构：

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

包含一个数组 buf，一个头结点 head，以及一个实现互斥访问 bcache 的锁 lock

3.初始化 buffer 的链表：（在 binit()函数中）

```
  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;
    b->dev = -1;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
```

将头尾指针初始化为头结点，然后用头插法把 buf 中的元素依次连接到头节点后

4.了解 buffer 的状态

```
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

有三个状态：忙碌（被某进程锁住了），有效（被磁盘读走了，也就是可用），脏（被改过了，需写回磁盘）

5.了解对 buffer 的各种操作

（1）释放一个忙碌中的 buffer：

```
void
brelse(struct buf *b)
```

释放一个忙碌状态的 buffer，并将等待该 buffer 的进程唤醒

```
 acquire(&bcache.lock);
```

```
  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;
  b->flags &= ~B_BUSY;
  wakeup(b);
  release(&bcache.lock);
```

（2）将 buffer 的内容写入磁盘（buffer 的状态必须是忙碌）

```
void
bwrite(struct buf *b)
```

将这个 buffer 的状态改成脏，然后调用 iderw 函数

```
  b->flags |= B_DIRTY;
  iderw(b);
```

（3）返回一个忙碌状态的 buffer，这个 buffer 包含指定的磁盘内容（用 buffer 读取磁盘内容）

```
struct buf*
bread(uint dev, uint sector)
```

调用 bget 方法获取目标 buffer，并调用 iderw()读取内容

```
 b = bget(dev, sector);
  if(!(b->flags & B_VALID))
    iderw(b);
  return b;
```

（4）在 buffer cache 中寻找某设备的指定扇区，如果没找到，则分配一个空闲 buffer，否则以 busy 状态返回找到的 buffer

```
static struct buf*
bget(uint dev, uint sector)
```

遍历 bcache，找目标 buffer：

```
 loop:
  // Is the sector already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->sector == sector){
      if(!(b->flags & B_BUSY)){
        b->flags |= B_BUSY;
        release(&bcache.lock);
        return b;
      }
      sleep(b, &bcache.lock);
      goto loop;
    }
```

如果没找到，就以 busy 状态返回一个空闲的 buffer

```
  // Not cached; recycle some non-busy and clean buffer.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
```

```
    if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
      b->dev = dev;
      b->sector = sector;
      b->flags = B_BUSY;
      release(&bcache.lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

四、阅读 log.c，了解 logging 和 transaction 机制

log.c 模块主要维护文件系统的一致性。在引入 log（日志）模块后，对于上层文件系统的全部磁盘操作都会被切分为 transaction（事务），每个 transaction 都会首先将数据和其对应磁盘号写入磁盘上的 log 区域，且只有在 log 区域写入成功后，才将 log 区域的数据写入真正存储的数据块。

上层文件系统要操作磁盘，不会直接使用系统调用，而是通过 logging 使用系统调用操作磁盘，log 在系统调用中的典型模式如下：

```
begin_trans();
...
bp = bread();
bp->data[]=...;
log_write(bp);
...
commit_trans();
```

begin_trans()为此次事务申请必要的资源，也就是等待 log 空闲，获取到 log 之后，将 log 的 busy 置 1，表示 log 繁忙，一个事务正在处理。

```
void
begin_trans(void)
{
  acquire(&log.lock);
  while (log.busy) {
    sleep(&log, &log.lock);
  }
  log.busy = 1;
  release(&log.lock);
}
```

log_write()可以视作 bwrite()的代理，它记录了块的扇区号，并为它在 log 磁盘上保留一个位置，然后将 buffer 标记为被修改（脏）状态，块在事务提交之前都会一直存在于 cache 中，cache 中的内容保存的是修改后的磁盘数据，并且只有这个 buffer 会保存修改记录，在 commit 之前，它都不会被写回磁盘。

commit_trans()将 buffer 中的内容写回磁盘以提交事务，并且释放 log，也就是将 log 的 busy 置 0；

```
void
commit_trans(void)
```

```
{
  if (log.lh.n > 0) {
    write_head();    // Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }

  acquire(&log.lock);
  log.busy = 0;
  wakeup(&log);
  release(&log.lock);
}
```

以下是 log 的基本结构：

1. 结构体 log

```
struct log {
  struct spinlock lock;
  int start;
  int size;
  int busy; // a transaction is active
  int dev;
  struct logheader lh;
};
```

2. 初始化 log

```
void
initlog(void)
{  ...
  struct superblock sb;
  initlock(&log.lock, "log");
  readsb(ROOTDEV, &sb);
  log.start = sb.size - sb.nlog;
  log.size = sb.nlog;
  log.dev = ROOTDEV;
  recover_from_log();
}
```

initlog()会在第一个用户进程运行前被调用，它会调用 recover_from_log()将已经提交但未执行完的事务执行完，然后清空 log，install_trans()是将 log 区内容写回磁盘的函数。

```
static void
recover_from_log(void)
{
  read_head();
  install_trans(); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
```

```
}
```

3. 追踪事务：write_head()&read_head()

wirte_head()将内存中的 log 头部 lh 写入到磁盘中。

read_head()会读取磁盘上保存的 log 头部 lh 到内存，来追踪上次执行的事务。

五、阅读 fs.h，fs.c，了解硬盘布局

fs.h/c 中声明了超级块，dinode，目录项 dirent

1. 超级块：

```
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
};
```

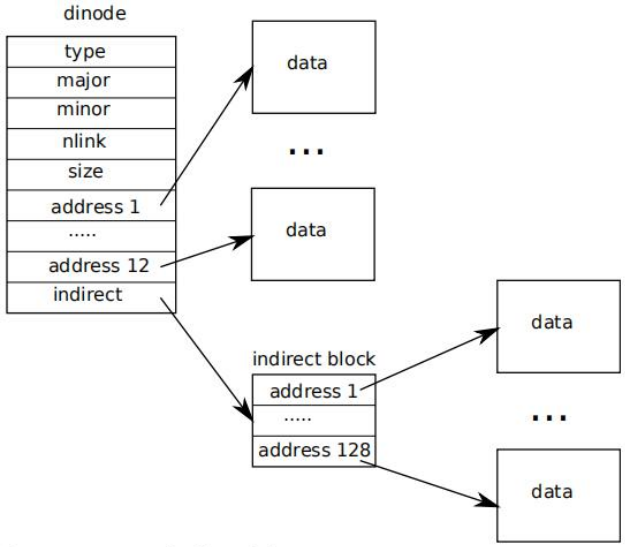
用结构体 superblock 实现超级块，包含了 log 块数量，inode 节点数量，数据块数量以及文件系统块容量

2. dinode：磁盘上的 inode 结构

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

包括文件类型，最大最小设备数量，链接到此设备的文件系统 inode 数。对应的文件大小，数据块地址数组（实际上是存放直接索引的地址）

dinode 结点结构如下图所示：



The representation of a file on disk.

dinode 里有直接索引 12 个：

```
#define NDIRECT 12
  uint addrs[NDIRECT+1];   // Data block addresses
```

间接索引：512/4 = 128 个（32 位系统中，unsigned int 类型占 4 字节）

```
#define NINDIRECT (BSIZE / sizeof(uint))
#define BSIZE 512  // block size
typedef unsigned int   uint;
```

3. Dirent

```
#define DIRSIZ 14
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

xv6 的目录是一个包含了一系列 dirent 结构体的文件，dirent 相当于目录项。

六、阅读 file.h，file.c

①xv6 的“文件”有哪些，以及文件，inode，设备相关的数据结构

文件或目录数据结构本身都是以文件的形式存储到磁盘中的

Xv6 的“文件”有管道文件，设备文件与普通文件。

File.h/c 是 xv6 的 file descriptor 层的实现，该层将管道，设备均抽象为文件。

xv6 给每个进程自己的打开文件表或者文件描述器，每一个打开文件都被呈现为结构体 file 的形式：

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

其中，

type 标记着这个文件是什么类型的文件（普通，管道，inode）

ref 为引用计数，ref 代表了当前打开了该文件的进程数

readable 与 writable 是文件是否可读，可写的标志位，xv6 中的文件可以是可读的，可写的，或者可读写的。

off 为 I/O 偏置，如果多个进程独立地打开了同一个文件，那么不同的实例将会拥有不同的 I/O 偏置；I/O 偏置在文件的读写操作中使用，可理解为当前进程在该文件中的读写指针；管道文件不含 I/O 偏置。

ip 为指向文件 inode 节点的指针。

pipe 为管道指针，当文件是管道文件时，这个指针有效。

结构体 inode 为在内存中 inode 节点的体现形式，包含了文件的一些基本信息，比如状态，类型等

```
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  int flags;          // I_BUSY, I_VALID
```

```
  short type;          // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

设备相关的数据结构是 devsw 结构体定义的

```
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};
extern struct devsw devsw[];
```

Devsw 的定义

②xv6 对文件的基本操作有哪些

当一个进程打开文件时，将会调用 dup 函数让文件的引用计数+1。进程也可以使用 fork，将这个文件共享给自己的子进程。这会导致一个文件可以出现于多个进程的打开文件表中，或者在一个进程的打开文件表中出现多次。

系统中所有的打开文件都会被维护于一张全局打开文件表 ftable 中，以下是 ftable 的初始化与操作。

1. 初始化全局打开文件表 ftable

```
void
fileinit(void)
{
  initlock(&ftable.lock, "ftable");
}
```

2. 在 ftable 中分配位置给新的打开文件

扫描 ftable，来寻找一个 ref==0 的文件，并将它的 ref 置为 1，表示分配一个打开文件位。

```
struct file*
filealloc(void)
{
  struct file *f;
  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
```

3. **增加文件的引用计数**

```
struct file*
filedup(struct file *f)
{
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  release(&ftable.lock);
  return f;
}
```

4. 释放文件

减少文件的 ref 计数，如果减少后的 ref 计数等于 1，则关闭文件。如果关闭的是普通文件，则会调用 begin_trans()与 commit_trans()，使用 logging 与 transaction 机制来完成 ionde 结点的回收操作。input()为减少 inode 结点的引用计数，当计数==0 时，该 inode 的 cache 单元便可以被回收。如果是管道文件，则调用 pipeclose()来回收资源。

```
void
fileclose(struct file *f)
{
  struct file ff;
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);
  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_trans();
    iput(ff.ip);
    commit_trans();
  }
}
```

5. 获取文件的 inode 结点信息

filestat()只允许作用于 inode 结点

```
int
filestat(struct file *f, struct stat *st)
{
```

```
  if(f->type == FD_INODE){
    ilock(f->ip);
    stati(f->ip, st);
    iunlock(f->ip);
    return 0;
  }
  return -1;
}
```

该函数调用 stati 函数，来获取 inode 中的信息

```
// Copy stat information from inode.
void
stati(struct inode *ip, struct stat *st)
{
  st->dev = ip->dev;
  st->ino = ip->inum;
  st->type = ip->type;
  st->nlink = ip->nlink;
  st->size = ip->size;
}
```

6. 读文件

首先检查 readable 位是否为 1（文件是否可读），然后根据文件类型来决定执行的操作。对于管道文件，调用 piperead 方法，来完成此次读取。如果是 inode 文件，则会使用 I/O 偏置来完成读取操作，读取操作完成后，I/O 偏置前进，I/O 偏置的前进操作由 fileread 的调用者完成（也就是执行读取操作的进程），并且对 inode 的操作是互斥的，因此多个进程同时对同一个文件进行读取操作时，不会出现读取到别人所需数据的情况发生，这一点对写操作 filewrite()同样适用。

```
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

7. 写文件

逻辑与 fileread()基本相同，只是写操作要通过 logging 与 transaction 机制来实现

```
int
filewrite(struct file *f, char *addr, int n)
{
  int r;

  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return pipewrite(f->pipe, addr, n);
  if(f->type == FD_INODE){
    // write a few blocks at a time to avoid exceeding
    // the maximum log transaction size, including
    // i-node, indirect block, allocation blocks,
    // and 2 blocks of slop for non-aligned writes.
    // this really belongs lower down, since writei()
    // might be writing a device like the console.
    int max = ((LOGSIZE-1-1-2) / 2) * 512;
    int i = 0;
    while(i < n){
      int n1 = n - i;
      if(n1 > max)
        n1 = max;
      begin_trans();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      commit_trans();
      if(r < 0)
        break;
      if(r != n1)
        panic("short filewrite");
      i += r;
    }
    return i == n ? n : -1;
  }
  panic("filewrite");
}
```

③xv6 最多支持多少个文件

```
#define NFILE       100  // open files per system
```

查 param.h，可以看到 xv6 最多支持 100 个文件

④每个进程最多能打开多少个文件

```
#define NOFILE       16  // open files per process
```

查 param.h，可以看到一个进程最多可以打开 16 个文件

七、阅读 sysfile.c，了解文件系统相关的系统调用，简述各个系统调用的作用

1.申请一个打开文件位

```
static int
fdalloc(struct file *f)
```

2.对文件的引用计数+1

```
int
sys_dup(void)
```

3.读取文件数据

```
int
sys_read(void)
```

4.向文件写数据

```
int
sys_write(void)
```

5.关闭文件

```
int
sys_close(void)
```

6.修改文件统计信息

```
int
sys_fstat(void)
```

7.为已有的 inode 创建一个新的连接

```
int
sys_link(void)
```

8.为 inode 释放一个连接，若全部连接被释放，则 inode 被释放

```
int
sys_unlink(void)
```

9.创建一个 inode

```
static struct inode*
create(char *path, short type, short major, short minor)
```

10.打开一个文件

```
int
sys_open(void)
```

11.创建一个目录

```
int
sys_mkdir(void)
```

12.创建一个文件

```
int
sys_mknod(void)
```

13.切换目录

```
int
sys_chdir(void)
```

14.文件重定向

```
int
sys_exec(void)
```

15.创建一个管道文件

```
int
sys_pipe(void)
```