第四次代码阅读

蒲伟良 1600011338

1. 了解 UNIX 文件系统的主要组成部分：超级块（superblock），i 节点（inode），数据块 （data block），目录块（directory block），间接块（indirection block）。分别解释它们的 作用。（以下各题内容按照程序的调用关系自底向上编排）

- superblock用于描述一个文件系统，包括文件系统中的inode的数目，磁盘块数以及空闲链表的起始位置（用于分配空间）
- inode用于描述文件，一个inode对应一个文件，其中包含了文件类型，文件所属，文件大小，文件存储的地址，访问、修改时间等信息
- data block存储所有的文件和目录
- directory block就是包含了许多目录项，用于存储该目录下的文件和目录，目录项包含了4个定长的域和1个变长的文件名域，定长的域分别是inode编号，该目录项的长度（用于寻找下一项），文件的类型（比如文件或目录），文件名的长度
- indirection block使得文件能够扩展，一个inode会存储12个磁盘块地址，如果12个磁盘块还不足以存下文件，就需要更多的磁盘块，为了避免大文件中inode存储大量的地址，使用了三级的间接块，一个间接块就是磁盘块中每一项存储一个磁盘块地址，相当于存储指针。间接块可以有多级，一级的间接块每项直接指向地址，二级间接块中每项指向一个一级间接块，如此类推。inode中存储了12个直接指向的磁盘块地址，还有1个一级间接块，1个二级间接块，1个三级间接块共3个间接块的地址

2. 阅读文件 ide.c。这是一个简单的 ide 硬盘驱动程序，对其内容作大致简述。

- 硬盘在底层被IDE控制器封装，所以我们不需要直接面对一个磁盘，只需要面对一个IDE控制器即可。`ide.c`文件定义了一系列函数控制IDE控制器。

- `idewait`用于等待磁盘可用。通过从IO端口`0x1f7`读取数据，判断磁盘的busy位是0的且ready为是1，表示磁盘可用。同时可以通过传入的参数`checkerr`来设置是否要检查error位和fail位，0为不检查，1即要检查这两位的任意一位有没有置1，有则返回-1。
- `ideinit`用于初始化，设置锁，通知最后一个CPU检测硬盘中断（默认多个CPU），然后等待硬盘准备好后检查有多少个硬盘，因为硬盘0用于启动，已经正在用了，所以必然是存在的，XV6只检测了硬盘1，通过端口`0x1f6`给它发信息，如果有反应说明是存在的。XV6认为要么只有一个硬盘0，要么只有两个硬盘0和1，从很对操作可以看出来，比如说只有一个变量`havedisk1`
- `idestart`将一个buffer的信息给IDE控制器，根据`buf`结构中的flags判断是读或者写，读的话需要等待一段时间，写的话直接将data交给IDE。使用buffer的原因是硬盘比缓存要慢很多，所以在读硬盘或者写硬盘的时候需要中断，读的时候需要等待硬盘，所以`idestart`后会进行退出，交由`ideintr`完成将数据读到缓存中，而写因为和进程没有什么关系，所以不需要中断处理。
- `ideintr`是硬盘中断的处理程序，这里认为中断到来后表示硬盘完成了一个写操作或者读操作，可以进行下一个操作了，所以这个buffer的请求已经完成，一开始就将其从请求队列`idequeue`中移出，前面提到了读需要等待，所以如果buffer请求了读，现在会从IO端口`0x1f0`读取数据到buffer的data中，完成对buffer的操作后设置其flag为`B_VALID`并去除`B_DIRTY`，唤醒可能存在的等待该buffer的进程，然后开始处理下一个请求。
- `iderw`处理对硬盘的写和读的请求，作为一个对上层的接口，接收一个buffer，将其放到请求队列的队尾，如果请求队列空，可以立即开始，否则会等待其他请求执行。因为等待硬盘需要不短的时间，所以只要没有经过`ideintr`设置buffer的flag，就会陷入休眠，由于可能有抢占，所以这里使用了`while`

3. 阅读文件 buf.h，bio.c。简述 XV6 文件系统中 buffer cache 层的内容和实现。描述 buffer 双链表数据结构及其初始化过程。了解 buffer 的状态。结合代码简述对 buffer 的各种操作。

- XV6中IDE层上面试buffer cache层，在`bio.c`中实现
- `bcache`是存储了所有的buffer，并且有一个锁用于确保只有一个进程修改这个表，因为会将所有的buffer组织成双向链表的形式，所以还有一个`head`

```
// in bio.c
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;
```

- buffer在`buf.h`中定义，每个buffer包含了`flags`用于标志有没有被读或者是不是要被写，`dev`表示buffer从哪个磁盘获取数据，`blockno`表示从哪个block获取，`lock`保证只有一个进程操作，`refcnt`为要使用该buffer的进程数，指针`prev`和`next`是在`bcache`组织双向链表时使用，`qnext`是在组织IDE的请求队列`iqueue`时指向下一个buffer，最后的`data`是数据存放的地方，大小为`BSIZE`即512个字节。

```
// in buf.h
struct buf {
  int flags;
  uint dev;
  uint blockno;
```

```
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

- `binit` 初始化 `bcache`，初始化锁，然后将所有的buffer串成一个双向链表，一共有 `NBUF` 个，而 `param.h` 中定义了 `NBUF` 为30。

```
//in param.h
#define MAXOPBLOCKS  10     // max # of blocks any FS op writes
#define NBUF         (MAXOPBLOCKS*3)     // size of disk block cache
```

```
// in bio.c
void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");

//PAGEBREAK!
  // Create linked list of buffers
  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;
    initsleeplock(&b->lock, "buffer");
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}
```

- `bget` 通过设备号 `dev` 和 `blockno` 读取相应的buffer，首先在 `bcache` 的链表中从头到尾寻找有没有正在使用的buffer满足要求，有的话 `refcnt` 加1并返回该buffer，没有的话需要从头重新找一个空闲的buffer（因为buffer释放后会被放到前面，所以效率不会很低），一个buffer空闲的标准时 `refcnt` 为零且 `flags` 的 `B_DIRTY` 位为零。找到后会将 `refcnt` 置为1，返回buffer。注意这里会在返回前获取该buffer的锁。

```
// in bio.c
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

```

```
  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached; recycle an unused buffer.
  // Even if refcnt==0, B_DIRTY indicates a buffer is in use
  // because log.c has modified it but not yet committed it.
  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;
    }
  }
  panic("bget: no buffers");
}
```

- `bread`通过输入的参数`dev`和`blockno`调用`bget`得到相应的buffer，然后调用`iderw`读取相应的数据，这里会对`B_VALID`为进行检查，确保式正确的(`B_VALId`置1表示已经读好了)，同时`iderw`要求buffer获得锁，所以`bget`在获得锁之前返回，在这里进行配合。

```
// in bio.c
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b;
}
```

- `bwrite`操作累次`bread`，输入参数`dev`和`blockno`，调用`bget`获得相应的buffer，设置flags为`B_DIRTY`告诉`iderw`这是写的块，最后调用下层的接口`iderw`进行写操作。

```
// in bio.c
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

- `brelse`释放不再需要的buffer，首先释放buffer自带的锁，获得`bcache`的锁，然后buffer的`refcnt`减1，如果减1后为零，表示没有进程需要这个buffer了，可以释放，修改`bcache`中的双向链表，将释放的buffer放到链表的头部，使得最近使用的块被放到前面，这样如果`bget`寻找的块是最近使用过的，那么即使它已经被释放也能被迅速找到，并且由于寻找空闲块的时候是从尾部开始的，所以最近使用的块更不容易被重新分配，即释放后可能会被保持数据不被修改。

```
// in bio.c
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    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;
  }
```

4. 阅读文件 log.c。简述 XV6 文件系统中的 logging 和 transaction 机制。

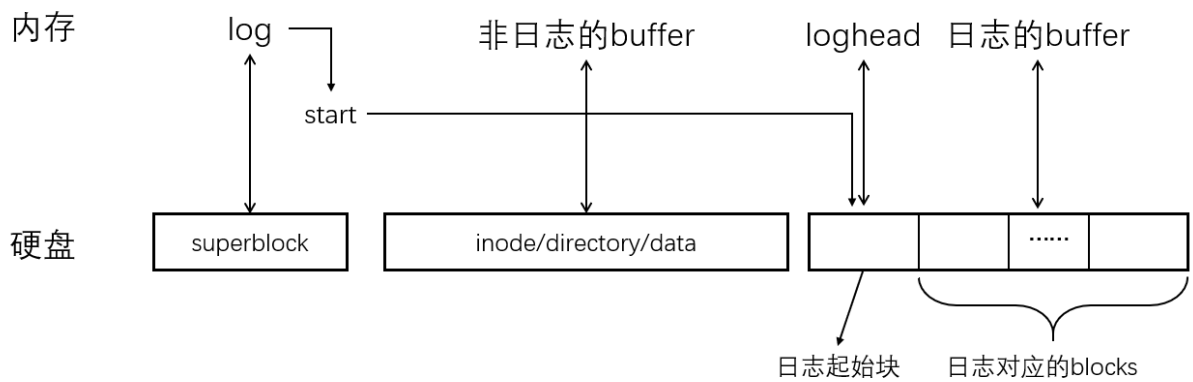
- 文件系统一个重要的功能是错误恢复，保证在断电的情况下不破坏文件，从而能够恢复断电前的数据。XV6这里使用了一个日志层来支持，一个普通的buffer并不直接写入到磁盘中，而是首先在日志log中备份（写入磁盘的log），然后再写入到磁盘中，最后清除磁盘中log的标记。如果buffer在备份到log前崩溃了，则程序会无视对buffer的新的修改，从log中写入磁盘的过程也有可能崩溃，但只要`loghead`没有错误就能够恢复，`loghead`可以看成是log中的一块存有log的信息的块，但是如果恰好在写入`loghead`的时候崩溃了，系统也不会崩溃，这将在后面讨论。相当于把读写任何块出错的风险变成了只有读写特定某一块出错会有风险，其他块读写时断电了也可恢复，依然是降低了风险，但并没有消除。同时增加了日志也降低了效率，本来buffer直接写进硬盘只需要一次，现在需要先写进日志，再从日志读出，最后才写进硬盘，安全性更好的同时效率更低。

- 日志层的管理依靠一个结构体`log`，其数据来源在磁盘中是`superblock`，其中包含了另一个结构体`loghead`，对应磁盘中blockno为`log.start`的块，是日志区的起始块，后面跟着连续`LOGSIZE`即30块用于存储日志buffer的block。

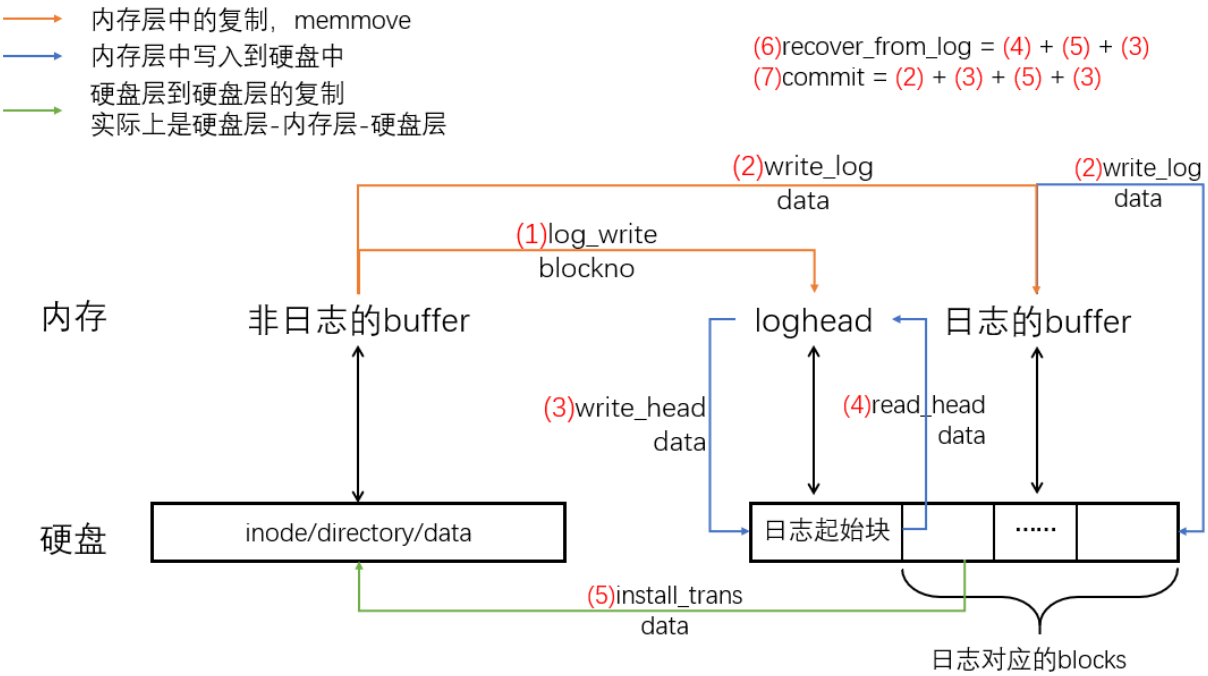
```
// in log.c
struct logheader {
  int n;
  int block[LOGSIZE];
};

struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev;
  struct logheader lh;
};
struct log log;
```

- 这里`log`带有一个锁，`start`指示了日志起始块，`size`是日志的buffer的最大数量，`outstanding`是`begin_op`的调用次数，即现在需要往日志里写的总数，`committing`是一个标志位，表示是否在commit，即是否在从日志的buffer中写到硬盘中，`dev`日志区所在的设备号，`lh`是结构`loghead`的实例。`loghead`是`log.start`指示的日志起始块中的数据，其中记录的`n`表示已经在内存中写入到日志中的buffer数量，或者说commit时需要提交的buffer的数量，`block[]`记录了这`n`个blockno，是这个buffer写入到日志中的唯一标识，即通过这个标识来找到相应的需要写入硬盘的buffer。下面的图给出了日志层和硬盘层的结构和`log.c`中函数的作用示意。



图片来源，我自己画的



图片来源，我自己画的

- 为了方便理解，上面的图从硬盘层和内存层来看这几个函数。不过在此之前先看图中没有标示的`initlog`，该函数在`main`中调用，作用是从`superblock`中获取相应的日志层的信息，包括日志起始块的blockno，日志块的最大数目，设备号，最后调用`recover_form_log`从日志恢复。

```
// in log.c
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

- 下面按照上图(1)(2)(3)的顺序来看
- $\color{red}{(1)}是`log_write`，在检查log的buffer够用并且确实有东西要写后，寻找与输入的buffer相应的block（即log中的buffer），如果已经在log中说明是再次修改了，不在的话是初次修改，需要分配一个block并记录其blockno，两种情况下都需要修改`B_DIRTY`告诉buffer层要写这个buffer。从图中也可看到这一步是将一个平凡的buffer和log中一个buffer关联，就是在日志中登记一下。

```
// in log.c
void
```

```
log_write(struct buf *b)
{
  int i;

  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");

  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)   // log absorbtion
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // prevent eviction
  release(&log.lock);
}
```

- (2)是函数`write_log`，有意思的是和上一个函数名字恰好相反。这个函数的作用是根据`loghead`中保存的blockno将这些已经在日志中登记的平凡的buffer写到日志区对应的硬盘中，因为只有对应的buffer才能写入对应的block，所以需要将这些平凡的buffer先写到日志区相应的buffer（图中橙色线），再将buffer调用buffer层的`bwrite`写入硬盘（图中蓝色线）

```
// in log.c
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev, log.start+tail+1); // log block
    struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to);  // write the log
    brelse(from);
    brelse(to);
  }
}
```

- (3)(4)是两个相似的函数，对`loghead`进行读写，前面也提到了，`loghead`实际上就是一个buffer，单独占据了硬盘的一块，这两个函数的作用就是将内存中的`loghead`写入到硬盘中或者将硬盘中的`loghead`读取到内存中。从`loghead`中可以得知日志中有多少个buffer，同时这n个buffer对应的blockno是什么。

```
// in log.c
static void
read_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
```

```
  struct logheader *lh = (struct logheader *) (buf->data);
  int i;
  log.lh.n = lh->n;
  for (i = 0; i < log.lh.n; i++) {
    log.lh.block[i] = lh->block[i];
  }
  brelse(buf);
}

static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

- (5) `install_trans` 函数的作用是将已经写进日志区硬盘的内容写到原来的位置（home location），使得平凡的buffer的内容终于可以写入硬盘。因为这些buffer都已经在 `loghead` 中登记，并且已经写入硬盘，所以需要从硬盘中读取到buffer，再在内存中复制一下，最后再写入到硬盘中对应的位置。从这里可以看到为了可靠性牺牲了不少效率。

```
// in log.c
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}s
```

- (6) `recover_from_log`，这里就是初始化时从日志中读取信息，恢复硬盘。读取头部，可以得知有多少个block需要从日志中写会原来的位置，调用 `install_trans` 写回，最后设置 `log.lh.n` 为0，重新写入硬盘中，设置为0的目的就是表示没有需要恢复的日志了。

```
// in log.c
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
}
```

- 图中最后一个(7)`commit`，作用是将所有已经登记在日志中的block写入到对应的硬盘区。这里需要前置`log_write`，不然`n=0`就没有需要写的了，然后`write_log`将平凡的block对应的buffer的内容写入到日志区的硬盘中，接着写入`loghead`，再将日志区中的内容写回到需要的硬盘地址，最后擦除日志区记录。

```
// in log.c
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    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
  }
}
```

- 图中没有的两个函数`begin_op`和`end_op`可以看成是`acquire`和`release`的升级版，两个函数之间即为一次transaction，`being_op`在这基础上增加了判断如果日志正在提交或者日志的块数不够用会sleep，而`end_op`会设置日志的`committing`标志位，唤醒sleep的进程等。另外，`end_op`是唯一调用`commit`的函数，即会在进行完一次transaction时将buffer写回硬盘。

```
// in log.c
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}

```

```
20 void
21 end_op(void)
22 {
23   int do_commit = 0;
24
25   acquire(&log.lock);
26   log.outstanding -= 1;
27   if(log.committing)
28     panic("log.committing");
29   if(log.outstanding == 0){
30     do_commit = 1;
31     log.committing = 1;
32   } else {
33     wakeup(&log);
34   }
35   release(&log.lock);
36
37   if(do_commit){
38     // call commit w/o holding locks, since not allowed
39     // to sleep with locks.
40     commit();
41     acquire(&log.lock);
42     log.committing = 0;
43     wakeup(&log);
44     release(&log.lock);
45   }
46 }
```

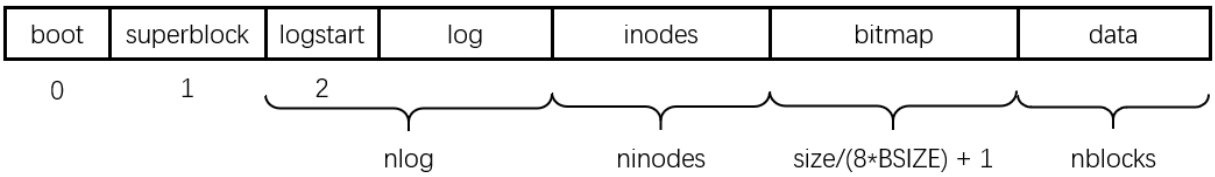
- 最后讨论一下关于日志和崩溃的事情。为什么需要日志呢，因为我们的内存是易失性的，断电就没有了，而不容易丢失数据的硬盘访问、读取速度非常慢。所以断电或者系统崩溃时可能正在进行写硬盘的操作，如果刚好写到一半，下次开机可能会有一些奇怪的bug，所以需要日志来保障。如果读到一半崩溃了，并不影响硬盘中的数据，所以不予考虑。如果没有日志，那么写硬盘的操作只有一个，就是将buffer写回硬盘，调用`bwrite`，有了日志层之后虽然也是调用`bwrite`来写硬盘，但是有好几处会写：写头部（特指$n \neq 0$），擦除头部（即写头部，但$n = 0$），写日志（即`write_log`）,写平凡的block(`install_trans`)
  - 如果在**写平凡的block**时崩溃(只有`recover_form_log`和`commit`会有这一步），由于没有擦除头部，所以调用`recover_from_log`时可以从日志中恢复，重新写一次即可
  - 如果在**写日志**时崩溃（只有`commit`有这一步），由于在写头部之前，所以调用`recover_from_log`时会发现`loghead.n`为0，即当作从来进程调用`log_write`在`loghead`中登记过，没有需要写的，而崩溃之前已修改的文件就当成没有修改过，丢失了。
  - 如果在**擦除头部**时崩溃了（只有`recover_form_log`和`commit`有这一步），此时所有在日志中的内容已经写进相应的硬盘区中，没有丢失任何数据。当然因为正在写头部，所以可能会造成头部数据错乱，但从代码中看，崩溃可能发生在写入`loghead.n`前后
    - 如果是之前，则后果是没来得及擦除头部，需要重新写一遍日志中的内容，否则没有任何后果，因为该块中只修改了`n`，当然这里要求修改`n`是原子的，并假设硬盘不会对没有修改过的内容进行修改（比如说如果硬盘是1位1位修改的，那么修改`n`时断电就会不知道出什么结果，另外如果硬盘会修改没有修改过的内容，比如改动了`block[]`（虽然这个不需要修改，我的意思是万一会有改过去又改回来这种操作），那么即使`n`没有被修改依然会出错）
    - 如果是之后，则结果是唯一的改动`n`已经被置零，相当于已经操作了（只要硬盘不修改不需要修改的位），所以没有任何代价，也不需要做任何额外的事

    - 如果在**写头部**时崩溃了（只有`commit`会有这一步），这时候除了`n`还要写入`block[]`，所以如果是在写入`n`后但没有写完`block[]`时崩溃了，那就真的崩溃了，因为`loghead`告诉我们有n个block需要写，但是我们却没有n个blockno（因为崩溃没来得及写），所以最终系统在意图`recover_from_log`时再次崩溃。

- 从上面的讨论可以发现即使使用了这样的日志层依然有崩溃的风险，但是现在只有写头部时崩溃会造成整个系统不可恢复的崩溃（其实这种时候只要放弃`recover_from_log`，即不再`read_head`就能再次正常运行，不过会丢失数据，但不会造成原来的文件错乱），而不使用日志则会可能在任意一次写硬盘崩溃时造成文件的错乱，系统的崩溃，相当于减小了风险，将原来的任意一次写硬盘都有的风险缩小到只有写头部时有风险，而且头部是十分短的，进一步降低了风险。当然另一面是无可避免的牺牲了至少一半的效率。

5. 阅读文件 fs.h，fs.c。简述 XV6 文件系统的硬盘布局。

下面的图片显示了XV6文件系统的硬盘布局。块0是boot，块1是superblock，存储了整个文件系统的信息，接着是log，log中第一块是loghead头部，接着是inodes，用于存储文件的inode，在XV6中被命名为`dinode`以区别于内存中的`inode`，后面是位图bitmap，其中一位代表一个block，用于表示该块是否已经分配，最后是data，是所有的目录和文件内容存储的地方



图片来源，我自己画的

- `superblock`的定义如下，其中`size`表示总计的block个数，后面三个变量`nblocks`,`ninodes`和`nlog`分别表示data、inodes和log区的block数量，bitmap区域的block数量取决于`size`，所以不需要，`logstart`，`inodestart`和`bmapstart`分别表示log、inode和bitmap区的起始位置，data区域的block通过`inode`定位，不需要开头。

```
// in fs.h
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
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
```

- `readsb`函数通过设备`dev`读取第一个block来读入superblock。
- `fs.h`还定义了`IBLOCK`来通过inum索引相应inode区中相应的block，`BBLOCK`通过blockno索引其在bitmap中的bit对应的block

```
// in fs.h
#define IPB           (BSIZE / sizeof(struct dinode))
#define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
#define BPB           (BSIZE*8)
#define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
```

- 当我们需要一个块的时候，需要块分配器来分配，不需要的时候要释放。`balloc`分配一个空闲的块，这里有两层循环，外层逐个访问bitmap区的block，内层访问逐个bit，是为了提高效率，实际上也可以看成一层循环逐位访问，找到空闲的块后将位图中相应的位置为1，更新位图块，返回blockno

```
// in fs.c
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}
```

- `bfree`用于释放一个块，将位图中相应的位置0，更新位图块

```
// in fs.c
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)
    panic("freeing free block");
  bp->data[bi/8] &= ~m;
```

```
  log_write(bp);
  brelse(bp);
}
```

- `bzero`清零一个块
- `inode`在硬盘中存储的数据结构叫`dinode`，同时还有`inode`作为其包装。`dinode`中记录了文件类型，设备编号（用于设备IO，将设备当成文件看待，此时`type`为`T_DEV`），链接数量，文件大小和文件数据的存储地址。`inode`在其之上增加了硬盘号，引用次数`ref`（区别于`nlin`k，若`nlink`为0，说明没有目录指向该文件，inode会在硬盘中被释放，这个文件就被删除了，`ref`为0表示当前没有进程需要这个inode，会在内存中踢出，而不会在硬盘中删除），一个锁以及`valid`表示是否已经从硬盘中读取该inode

```
// in fs.h
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
};

// in file.h
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

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

- `inode`中的`addrs`指向了存储该文件数据的块，有`NDIRECT = 12`个直接块还有1个一级间接块，如下图所示。因为一个块为512字节，同时blockno为4字节，所以inode中直接指向的文件大小为6KB，而通过间接指向

的大小为64KB，则文件最大的大小为70KB。

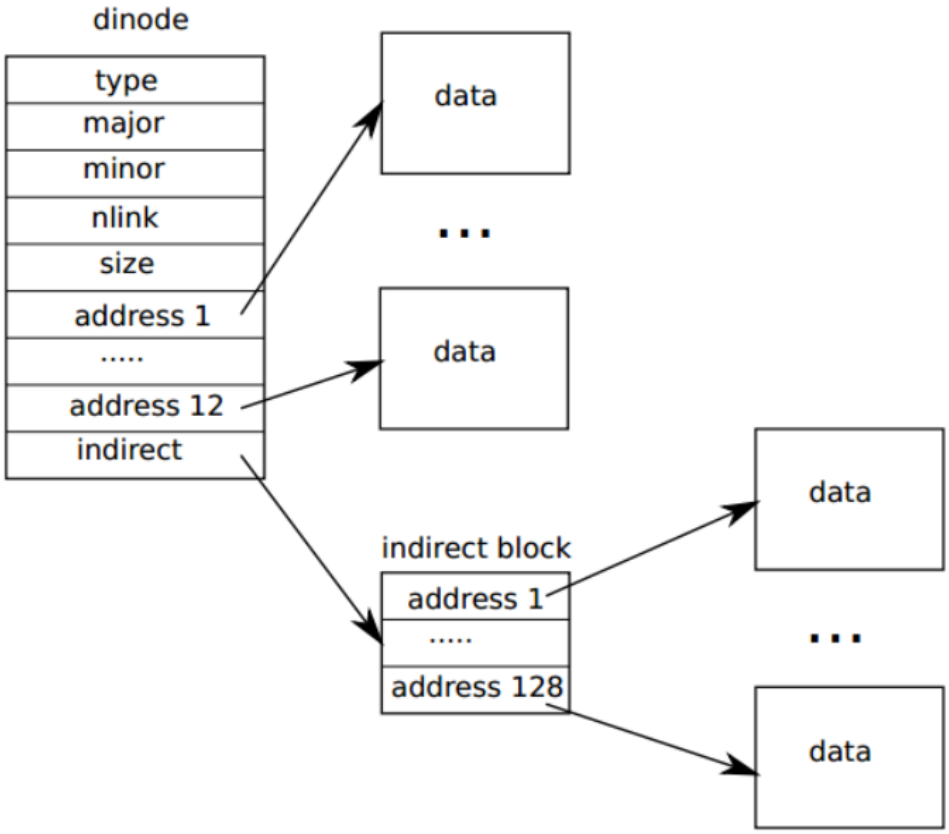


Figure 6-4. The representation of a file on disk.

图片来源，XV6中文文档

- `inode`的类型有三种，文件、目录和设备

```
// in stat.h
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device
```

- `icache`是存储内存中所有的inode的结构，需要一个锁避免多个进程修改inode时发生错误

```
// in fs.c
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;
```

下面是关于`inode`的操作

函数	描述
`iinit`	在`main`中被调用，作用是初始化`icache`并从硬盘中读取`superblock`
`iget`	根据输入`dev`和`inum`，在`icache`中寻找相应的inode，如果这个inode已经存在，则增加其`ref`并返回，否则会定位到一个未使用的inode，将其初始化，设置`inode.dev`和`inode.inum`，设置`inode.ref`为1，`inode.valid`为0。这个函数的总体作用即将硬盘中的一个`dinode`绑定一个`icache`中的`inode`
`ialloc`	在硬盘中从1开始顺序查找inode，直到找到一个`type`为0（表示空闲inode）的inode，初始化其`type`，写回硬盘并调用`iget`返回内存中对应的inode。该函数分配硬盘中的`dinode`，区别于`iget`分配（返回引用）`icache`中的`inode`
`iupdate`	根据输入的inode来更新硬盘中相应的dinode
`idup`	增加某个inode的引用次数1
`ilock`	获取相应inode的锁，如果该inode还没有从硬盘中读取信息，即`valid`为0，则从硬盘中读取dinode的信息
`iunlock`	释放相应inode的锁
`iput`	`ref`减1,如果减到0同时`nlink`为0，表示不再需要该inode，或者说该文件，调用`itrunc`将其删除，删除后记`type`和`valid`为0，表示该inode空闲可以重新被`ialloc`分配，并用`iupdate`将更新的信息写回硬盘
`iunlockput`	`iunlock`+`iput`
`itrunc`	调用`bfree`释放`inode.addrs`指向的块，相当于从硬盘中删除文件的内容。该函数会先释放直接指向的块，若间接块也有用到再释放间接块指向的块，最后释放间接块
`bmap`	函数接收两个参数`ip`为要操作的inode的指针，`bn`为要获取的inode的block的序号，若`bn < NDIRECT`则直接从直接指向中返回，否则需要先载入间接块，再返回间接块指向的地址。如果inode中第`bn`块不存在，或者间接块块不存在，`bmap`会通过`balloc`申请新的块
`stati`	从参数`ip`指向的inode中读取状态信息到参数`st`中
`readi`	从参数`ip`指向的inode代表的文件中`off`字节开始读取`n`字节到`dst`中，如果inode代表的是设备，会通过`inode.major`调用设备的`read`函数，返回读取的字节数（若读到文件尾则还不足`n`会返回更少的字节）
`writei`	和`readi`类似，不过从读变成写，将`n`字节长的`src`中的内容写到文件中从`off`字节开始处，写完文件后需要硬盘，若是写设备也会调用设备的`write`函数，返回最终写入的字节

`writei`、`readi`和`stati`是inode层给上层的接口

以下是关于目录的操作

- 首先要明确一个目录文件时怎样的，目录文件的`inode`和其他文件都一样，所有的目录会记录在`inode.addrs`指向的块中，在块中，目录会以`dirnet`结构排列，形成一个目录表，空的表项`inum = 0`

```
// in fs.h
#define DIRSIZ 14

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

函数	描述
`namecmp`	在`DIRSIZ`字节内比较字符串
`dirlookup`	第一个参数`dp`指向一个inode，表示的是一个目录，目录块中目录的结构是`dirent`，第二个参数`name`是文件名，该函数从目录中根据名字返回相应的`inode`，若设置了第三个参数，还需要返回该`dirnet`条目在块中的偏移`poff`
`dirlink`	第一个参数`dp`指向了一个目录的inode,第二个参数`name`是文件名，第三个参数是inode的编号，该函数在`dp`指向的目录中创建一个新的目录，名字为`name`。若该名字已存在，返回-1，否则函数会在目录中寻找一个空的目录槽，写入`inum`和`name`,返回0
`skipelem`	解析输入的第一个参数`path`即路径，返回删除了路径中第一个元素及其后面的'/'的路径，相当于进入了最外层的目录，返回在最外层目录视角下的相对路径，第二个参数`name`为最外层目录的名字(即被删掉的部分，不包含'/')，如果最外层为空返回0
`namex`	第一个参数`path`为路径，根据第二个参数`nameiparent`是否为1决定是返回路径指向的文件（或目录）的父目录的inode还是路径指向的文件（或目录）的inode，第三个参数`name`需要返回该函数所返回的inode代表的文件的名字。
`namei`	输入路径，返回路径指向的文件或目录的inode，通过`namex`实现（设置第二个参数为0）
`nameiparent`	输入路径，返回路径指向的文件或目录所在的目录（即不加载到路径的最后一个元素），返回路径指向的文件或目录的名字，通过`namex`实现(设置第二个参数为1)

其中`namecmp`，`dirlookup`，`dirlink`，`namei`和`nameiparent`都是给上一层的接口

- 下面是`namex`的实现
  - 根据`path`第一个字符是否为'/'判断是绝对路径还是相对路径
    - 绝对路径则调用`iget`获取根目录的inode
    - 相对路径则调用`idup`获取当前工作目录的inode（因为当前目录已经在`icache`中，不需要调用`iget`）
  - 将获得的inode赋值给`ip`（`ip`表示的是当前函数中打开的目录）。
  - 通过一个while循环和`skipelem`不断打开外层目录，通过`ilock`锁住inode同时保证inode已经从硬盘中读取
    - 若刚刚打开的“目录”不是目录，说明路径是错误的，返回0；
    - 若第二个参数为1且剩下的路径为空，说明刚刚打开的就是路径指向的文件，直接返回现在`ip`，同时返回的还有最内层文件的名字；
    - 若上面两点都不为真，说明还没有到头或者需要的是最里层的文件，调用`dirlookup`在现有目录`ip`下打开目录，在目录中根据`name`寻找相应的文件（或下一级目录）的inode，更新`ip`（这一次是真

的打开目录，更新了现有的目录（即`ip`），`skipelem`只是解析路径，在路径上看起来像是打开了一层目录）
        - 若`dirlookup`打开失败会返回0
        - 若成功则进入下一层while循环
    - 直到最内层文件也被`skipelem`“打开”，退出while循环
    - 若要返回的是父目录，则出错（因为父目录不会打开到最里一层，所以必然会在while循环中返回），返回0
    - 否则返回`ip`，即路径指向的inode

```
// in fs.c
static struct inode*
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;

  if(*path == '/')
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(myproc()->cwd);

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
    if(nameiparent && *path == '\0'){
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}
```

6. 阅读文件 file.h，file.c。简述 XV6 的“文件”有哪些，以及文件，i 节点，设备相关的数 据结构。简述 XV6 对文件的基本操作有哪些。XV6 最多支持多少个文件？每个进程最多 能打开多少个文件？

- 文件的类型有`FD_NONE`，`FD_PIPE`， `FD_INODE`，代表空，管道和`inode`，而`inode`又可以分为三类，文件、目录和设备。文件的结构在`file.h`中定义，各项的意义非常清晰。`file`就是我们常说的文件描述符。

```
// in file.h
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;
};
```

- inode的结构在上一题已经提到，这里不重复
- 设备的结构同样在`file.h`中定义，因为设备隐藏了更多细节，我们只需要关注其输入输出，所以只有两个函数指针，输入和输出

```
// in file.h
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};
extern struct devsw devsw[];
```

- `file.c`中实例化了`devsw`和`file`

```
// in file.c
struct devsw devsw[NDEV];   // NDEV = 10
struct {
  struct spinlock lock;
  struct file file[NFILE];  // NFILE = 100
} ftable;
```

`file.c`中定义一些对`file`的操作

函数	描述
`fileinit`	初始化 `ftable` 的锁
`filealloc`	在 `ftable` 中分配一个空闲的槽位并返回，失败则返回0
`filedup`	增加输入的 `f` 的引用次数1
`fileclose`	减小输入的 `f` 的引用次数1，如果 `ref` 减为零，设置文件类型为 `FD_NONE` 表示空文件，如果该文件类型是管道，需要调用 `pipeclose` 关闭管道，若是inode则调用 `iput` 减小对inode的引用
`filestat`	调用 `stati` 修改文件 `f` 的状态并返回0，若文建类型不是 `FD_INODE` 则返回-1
`fileread`	从文件 `f` 中读取 `n` 字节到地址 `addr` 处，函数会区分管道和inode以调用 `piperead` 或 `readi`，同时会修改 `f->off`，这是当前文件的偏移（就是读写头的位置）
`filewrite`	在文件 `f` 中写入来自于 `addr` 地址处的 `n` 字节，同样会根据类型调用 `pipewrite` 或 `writei`，但是调用 `writei` 时为了避免一次性过多的写入造成日志层的溢出，会分成好几次调用 `begin_op` 和 `end_op`

- XV6最多支持打开100个文件，而对于硬盘中文件的个数没有限制，而硬件的个数最多是10个，一个进程最多只能打开16个文件，这些都在 `param.h` 中定义

```
// in param.h
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
```

7. 阅读文件 sysfile.c。了解与文件系统相关的系统调用，简述各个系统调用的作用。

`sysfile.c` 中的函数及其描述，其中以 `sys_` 开头的函数是系统调用

函数	描述
`argfd`	把当前栈中第`n`个参数当作文件描述符，返回描述符的在进程打开文件表中的编号及其对应的`file`
`fdalloc`	在当前进程`ofile`中分配一个空的槽位，即新建一个文件描述符，并加载输入的`file`
`sysydup`	调用了`argfd`和`fdalloc`，假设栈中第一个参数是文件描述符，并且是已经打开的，但我们重复引用它，给它该文件新增一个描述符
`sys_read`	在栈中读取三个参数调用`fileread`
`sys_write`	在栈中读取三个参数调用`filewrite`
`sys_close`	关闭文件，并在`ofile`释放槽位
`sys_fstat`	从栈中读取两个参数调用`filestat`读取状态
`sys_link`	在栈中读取两个参数`old`和`new`，`old`是旧文件（非目录）的路径，`new`是一个新文件的路径，指向一个文件，该文件是在其父目录下的新文件，链接到`old`指向的文件（新文件可以有自己的别名）
`isdirempty`	返回一个目录是不是空的（除了有'.'和'..'）
`sys_unlink`	与`sys_link`相反，取消一个链接，要求链接不能是'.'或'..'，`nlink`大于1，如果取消链接的是目录则目录必须为空，取消链接后减小`nlink`
`create`	创建一个文件，该文件可以是普通的文件、目录甚至设备，如果普通的文件已经存在，会返回已存在的文件，已存在的目录和设备则会返回0
`sys_open`	打开一个文件，同时可以选择不同的打开模式，调用的`create`
`sys_mkdir`	创建一个文件夹，调用的`create`
`sys_mknod`	创建设备文件，调用的`create`
`sys_chdir`	更改当前目录，`proc`中的`cwd`是inode类型的，修改当前工作目录只需要用新的inode覆盖它即可
`sys_exec`	从栈中读取参数并执行一个文件
`sys_pipe`	创建一个管道

- 一个需要注意的地方时一个目录创建时就会有两个链接，一个是`.`，指向自己，另一个是`..`，指向父目录。所以`sys_unlink`中，若取消链接的是一个目录，则其父目录的`nlink`也需要减一，以为少了子目录的`..`。另外创建新目录的时候也要注意新建三个链接，一是父目录到新目录，二是新目录到新目录，三是新目录到父目录，不过目录对自己的链接并不会算在`nlink`中，所以一个目录新建时仍然有`nlink = 1`。