xv6 文件系统分析

按照从底向上的顺序进行分析。

磁盘布局

磁盘布局如下图：

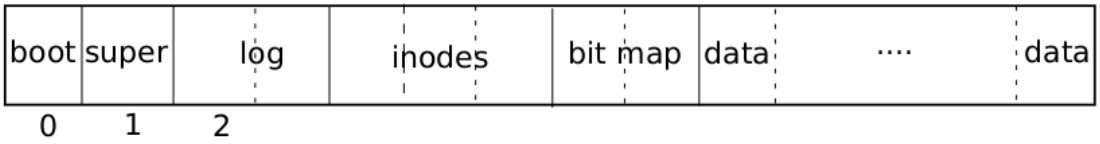

Figure 7.2: Structure of the xv6 file system.

`mkfs/mkfs.c` 将各用户程序打包成文件系统可以识别的磁盘文件 `fs.img` 。

此外，`fs.c` 定义了两个有用的函数 `balloc, bfree` ，用来分配和回收磁盘块。

磁盘驱动层 Disk

主要提供的函数是

```
// virtio_disk.c
void virtio_rw(int n, struct buf *b, int write);
```

`buf` 的定义可以在`buf.h` 中找到：

```
// buf.h
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock; // buf 被一个 sleeplock 保护
  uint refcnt; // 引用计数，有多少进程在使用该 buf
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE]; // BSIZE = 1024, in fs.h
};
```

这里的 buf 指的是在内存中的 buffer cache ，也不知道怎么翻译比较好，姑且叫块缓存吧。

因此，驱动层只需要实现从磁盘将内存读入到块缓存中，以及将块缓存写回到磁盘中两个功能即可。

块缓存层 Buffer Cache

主要提供的函数是：

```
// bio.c
struct buf* bread(uint dev, uint blockno);
void bwrite(struct buf *b);
void brelse(struct buf *b);
```

主要是用结构体 `bcache` 实现：

```
// bio.c
struct {
  struct spinlock lock; // bcache 被一个 spinlock 保护
  struct buf buf[NBUF]; // NBUF = 30, in param.h

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

在函数 `binit` 中，将 `buf` 池中的各元素依次加入双向链表中。同时，初始化 `spinlock`。

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

`bget` 的功能就是从双向链表中找对应的缓存并返回。如果没有的话就在池中找一个空位，并修改它的元数据。在返回之前已经获得 `b->lock`，这是一个 sleeplock 。

```
// Return a locked buf with the contents of the indicated block.
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno); // 获得 buf 指针
  if(!b->valid) { // 如果需要重新从磁盘读取
    virtio_disk_rw(b->dev, b, 0); // 从磁盘读取
    b->valid = 1;
  }
  return b;
}

// Write b's contents to disk.  Must be locked.
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock)) // 同时只有一个进程将该 buf 写入磁盘
    panic("bwrite");
  virtio_disk_rw(b->dev, b, 1); // 向磁盘写入
}

// Release a locked buffer.
// Move to the head of the MRU list.
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock)) // 同时只有一个进程释放该 buf
    panic("brelse");
```

```
  releasesleep(&b->lock); // 释放锁

  // 将刚刚释放的 buf 放在双向链表表头，方便 LRU 使用
  // 这同时也会给 bget 的扫描过程提供局部性支持
  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;
  }

  release(&bcache.lock);
}
```

日志层 Logging

一个文件系统相关系统调用可能会执行若干次磁盘写入操作。如果这些磁盘写入没有全部完成，系统就因为某些原因崩溃，那么此时磁盘中的内容不满足**一致性**，也即它并不能描述一个合法的文件系统。

因此，我们需要保证一次文件系统操作(称为一个事务)的**原子性**，这里是通过日志来实现的。

这一层向上提供的接口就是保证原子性、可恢复的磁盘 I/O 块。

相关数据结构

Log 在磁盘中的位置记录在超级块 Superblock 中。它包含以下内容：一个 header block ，以及一系列被修改的块的 copy 组成的序列。header block 里面记录的是被修改的块的数量，以及它们的扇区号。

```
// log.c
// Contents of the header block, used for both the on-disk header block
// and to keep track in memory of logged block# before commit.
struct logheader {
  int n;
  int block[LOGSIZE]; // LOGSIZE = 30, in param.h , 单个事务最多允许修改的块数量
};
```

如果 $n = 0$ ，表示在日志中没有记录任何事务；

否则表示日志记录了一个完整的 commited 的事务，一共修改了 n 块。单次 commit 可能会包括多个完整的文件系统操作的写入，即 group commit ，由于多个进程可能并发进行文件系统操作。这里的实现中，是在当前没有任何文件操作的情况下进行 commit ，如果真的是超高并发的场景下可能会有问题。

Log 占用一块大小固定的磁盘空间，因此：

1. 单个系统调用不允许写入比这块磁盘空间更多的块；因此对于 `write` 系统调用会被拆分成多个操作。
2. 如果目前磁盘空间不足，那么新的系统调用会被阻塞。

```
// log.c
// 内存里面的 log
struct log {
  struct spinlock lock; // spinlock
  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[NDISK]; // NDISK = 2, in param.h
// 每个磁盘保存一个 log ，一共有两个磁盘
```

原子操作块流程

第一步：使用 `begin_op` 。

这里主要是为了避免日志空间不足。

```
// called at the start of each FS system call.
void
begin_op(int dev)
{
  acquire(&log[dev].lock);
  while(1){
    if(log[dev].committing){ // 如果该 log 正在 commiting ，阻塞
      sleep(&log, &log[dev].lock);
    } else if(log[dev].lh.n + (log[dev].outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // 感觉这个判定标准定的太宽了
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log[dev].lock);
    } else {
      // 使用该 log 进行日志记录的进程数 + 1
      log[dev].outstanding += 1;
      release(&log[dev].lock);
      break;
    }
  }
}
```

第二步：读取、修改块缓存，并将该块缓存的修改记录到内存中的 log 中，最后释放块缓存：

```
// Caller has modified b->data and is done with the buffer.
// Record the block number and pin in the cache by increasing refcnt.
// commit()/write_log() will do the disk write.
//
// log_write() replaces bwrite(); a typical use is:
//   bp = bread(...)
//   modify bp->data[]
//   log_write(bp)
//   brelse(bp)
void
log_write(struct buf *b)
```

```
{
  int i;

  // 获取设备号
  int dev = b->dev;
  // 这里的 log[dev].size 是由 superblock.nlog 设置，维护磁盘中最多可以有多少 log block
  if (log[dev].lh.n >= LOGSIZE || log[dev].lh.n >= log[dev].size - 1)
    panic("too big a transaction");
  if (log[dev].outstanding < 1)
    panic("log_write outside of trans");

  acquire(&log[dev].lock);
  // 进入 log[dev] 的临界区

  // 在内存中的 log header 记录下当前块缓存的编号，并试图合并
  // 这保证 lh.block 数组的扇区号是互异的
  for (i = 0; i < log[dev].lh.n; i++) {
    if (log[dev].lh.block[i] == b->blockno)   // log absorbtion
      break;
  }
  log[dev].lh.block[i] = b->blockno;
  if (i == log[dev].lh.n) {  // Add new block to log?
    // 并非合并，而是新写入的
    // 在 bread 中，该块缓存已被加入 bcache 中，因此在 bpin 之中只需 refcnt++
    //? 虽说是 log_write 代替 bwrite ，但二者处理 refcnt 方式不同：+1 ; +0
    //: 在 commit 中已使用 bunpin 清除了多出来的 refcnt
    bpin(b);
    log[dev].lh.n++;
  }

  // 推出 log[dev] 的临界区
  release(&log[dev].lock);
}
```

第三步：使用 `end_op` 标志该 operation 的结束。

```
// called at the end of each FS system call.
// commits if this was the last outstanding operation.
void
end_op(int dev)
{
  int do_commit = 0;

  acquire(&log[dev].lock);
  // 进入 log[dev] 的临界区

  // 正在使用该 log 的进程减少一个
  log[dev].outstanding -= 1;
  if(log[dev].committing)
    panic("log[dev].committing");
  if(log[dev].outstanding == 0){
    // 判断是否可以进行 commit
    do_commit = 1;
    log[dev].committing = 1;
  } else {
    // begin_op() may be waiting for log space,
    // and decrementing log[dev].outstanding has decreased
```

```
    // the amount of reserved space.

    // 如果不能进行 commit ，则试图唤醒被阻塞在 begin_op 阶段的进程
    wakeup(&log);
  }

  // 退出 log[dev] 的临界区
  release(&log[dev].lock);

  if(do_commit){
    // call commit w/o holding locks, since not allowed
    // to sleep with locks.
    // 进行 commit 操作
    commit(dev);
    acquire(&log[dev].lock);

    // 在临界区内标志 commit 结束
    log[dev].committing = 0;
    wakeup(&log);
    release(&log[dev].lock);
  }
}
```

可见，其关键就在于在合适的时候调用 `commit` 进行 commit 。

```
static void
commit(int dev)
{
  if (log[dev].lh.n > 0) {
    write_log(dev);     // Write modified blocks from cache to log
    write_head(dev);    // Write header to disk -- the real commit
    install_trans(dev); // Now install writes to home locations
    log[dev].lh.n = 0;
    write_head(dev);    // Erase the transaction from the log
  }
}
```

它又分成 4 步：

```
// Copy modified blocks from cache to log.
// commit 第一步：将修改的各块的内容复制到磁盘的 log 区域中
static void
write_log(int dev)
{
  int tail;
  // 遍历内存中的 log header ，获取那些在事务中被修改的块
  for (tail = 0; tail < log[dev].lh.n; tail++) {
    // 获取一个 log 区域中的块，被修改的块的内容要被复制到这里
    struct buf *to = bread(dev, log[dev].start+tail+1); // log block
    // 获取被修改的块，该块只在内存中的块缓存中被修改，尚未被写回到磁盘
    struct buf *from = bread(dev, log[dev].lh.block[tail]); // cache block
    // 现在它们都在块缓存中，将 log 之外的块修改复制到 log 块中
    // 这里的 memmove 在 string.c 中
    memmove(to->data, from->data, BSIZE);
    // 将 log 块写回到磁盘
    bwrite(to);  // write the log
```

```
    // 释放两个 block
    brelse(from);
    brelse(to);
  }
}

// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
// commit 第二步：将内存中的 log header 内容写回到磁盘中的 log header 中
static void
write_head(int dev)
{
  // 获取磁盘中的 log header 块保存在块缓存中
  struct buf *buf = bread(dev, log[dev].start);
  // 复制内存中 log header 的内容
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  hb->n = log[dev].lh.n;
  for (i = 0; i < log[dev].lh.n; i++) {
    hb->block[i] = log[dev].lh.block[i];
  }
  // 将 log header 写回
  bwrite(buf);
  // 释放磁盘 log header 块
  brelse(buf);
}

// 第二步是 commit 的一个分界线
// 如果完成 write_head 后系统崩溃，则这次事务会在重启时被 replay ，从而事务可以被完成
// 即使会重复进行一些写入操作也是没有关系的，因为这里的磁盘写入是覆盖操作
// 这是因为这次事务的全部修改已经被记录在日志中了
//? 但是如果正好在 bwrite->virtio_rw 的过程中崩溃，导致 n > 0 但是 log header 其他信息错误，不知道能不能处理

// Copy committed blocks from log to their home location
// commit 第三步：将磁盘中被修改的块从块缓存真正写回磁盘
static void
install_trans(int dev)
{
  int tail;

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

// commit 第四步：将内存中的 log header 里面的 n 变为 0 ，再次将内存中的 log header 写回磁盘
```

漫长的 commit 流程终于结束了。这也标志着可能含有多个文件操作的事务被成功写回到磁盘。

我们所作的日志清理仅限于将 lh 中的 n 清零，而磁盘中的 log blocks 中仍有内容。

目前似乎仍存在隐患：即调用 `virtio_rw` 向磁盘中写入单个 log header 块崩溃，导致 log header 内容违背一致性，也许会有一点小问题。

索引节点层 Inode

磁盘

磁盘上的 Inode 是固定大小的数据结构，均存放在 Inode blocks 上：

```
// fs.h
// On-disk inode structure
struct dinode {
  short type;           // File type ，区分文件、目录、还是设备
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
  // NDIRECT = 12, in fs.h
  //? 文件最大大小只有 12KiB ？
};
```

内核

```
// in-memory copy of an inode
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];
};
```

可见，除了下半部分用来保存磁盘上 `dinode` 的拷贝之外，还有一些运行时信息。

如引用计数 `ref` ，就可被函数 `iget, iput` 修改。

Inode 典型使用方法

注意，这里已经开始使用日志层提供的接口了。

下面有几个重要函数：

在磁盘上分配新 Inode 函数：`ialloc`

```
// Allocate an inode on device dev.
// Mark it as allocated by  giving it type type.
```

```
// Returns an unlocked but allocated and referenced inode.
// ialloc ，在磁盘上的 inode blocks 区域新建一个 inode
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;

  // 在磁盘上找一个空闲的 dinode 结构体
  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb)); // 该宏可以在 fs.h 中找到
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      log_write(bp);   // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

类比块缓存 Buffer Cache ，这里我们也有索引节点缓存 Inode Cache ，在结构体 `icache` 中实现：

```
struct {
  struct spinlock lock;
  struct inode inode[NINODE]; // NINODE = 50, in param.h
} icache;
```

`iget, iput` 与 `icache` 打交道； `icache` 用一个 spinlock 进行保护。

```
// Find the inode with number inum on device dev
// and return the in-memory copy. Does not lock
// the inode and does not read it from disk.
// iget ，从 icache 中获取 inode 指针
static struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // Is the inode already cached?
  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;
    }
    if(empty == 0 && ip->ref == 0)    // Remember empty slot.
      empty = ip;
  }
```

```
  // Recycle an inode cache entry.
  if(empty == 0)
    panic("iget: no inodes");

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

  return ip;
}

// Drop a reference to an in-memory inode.
// If that was the last reference, the inode cache entry can
// be recycled.
// If that was the last reference and the inode has no links
// to it, free the inode (and its content) on disk.
// All calls to iput() must be inside a transaction in
// case it has to free the inode.
// iput ，回收 inode 指针，或只是修改引用计数
void
iput(struct inode *ip)
{
  acquire(&icache.lock);

  if(ip->ref == 1 && ip->valid && ip->nlink == 0){
    // inode has no links and no other references: truncate and free.

    // ip->ref == 1 means no other process can have ip locked,
    // so this acquiresleep() won't block (or deadlock).

    // 最典型的应用场合是删除文件？
    acquiresleep(&ip->lock);

    release(&icache.lock);

    // 回收该 inode 在磁盘上分配的 data block
    itrunc(ip);
    ip->type = 0;
    // 修改过 inode 的信息，写回到磁盘
    iupdate(ip);
    ip->valid = 0;

    releasesleep(&ip->lock);

    acquire(&icache.lock);
  }

  ip->ref--;
  release(&icache.lock);
}
```

在 `iget` 获取到 `inode` 指针后，接下来要做的第一件事就是加锁：

```
// Lock the given inode.
// Reads the inode from disk if necessary.
// ilock , 给 inode 指针加锁
void
ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

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

  acquiresleep(&ip->lock);

  // iget 之后只是在 icache 中占位，有可能 ip->valid = 0，还未从磁盘中获取 dinode
  if(ip->valid == 0){
    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)
      panic("ilock: no type");
  }
}

// Unlock the given inode.
// iunlock , 释放 inode 指针上的锁
void
iunlock(struct inode *ip)
{
  if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
    panic("iunlock");

  releasesleep(&ip->lock);
}
```

讲的一大串锁的正确性没怎么看懂...

总之，修改 Inode 的典型过程如下：

```
ip = iget(dev, inum);
ilock(ip);
ip->xxx = ...;
iunlock(ip);
iput(ip);
```

后面进行了一长串的说明：

即使是只读的系统调用如 `read` 在 `iput` 的时候，也有可能因为 `ip->ref = 0` 产生磁盘写入，因此所有文件操作都必须基于事务完成。

? 但是如果是 `read` 的话，应该不会满足 `ip->nlink = 0` 啊？这地方挺疑惑的。

此外，后面还提到某时刻 `ip->nlink = 0` 但 `ip->ref > 0` 时崩溃，会占用额外的磁盘空间。

Inode 内容

无论是 `inode, dinode` 里面都有 `addrs` 数组，前 `NDIRECT = 12` 个指向一个磁盘上 data block 的扇区号，最后一个指向一个 indirect data block 的扇区号，这个块只存储该 inode 文件的其他 data blocks 的扇区号，每个 4 字节，能存 $1024/4 = 256$ 个。因此文件最大 $12 + 256 = 268\mathrm{KiB}$，前 $12\mathrm{KiB}$ 直接寻址，后 $256\mathrm{KiB}$ 需要间接寻址。

```
static uint bmap(struct inode *ip, uint bn);
// 负责找到第 bn 个块的扇区号，若不存在，则分配一个
static void itrunc(struct inode *ip);
// 回收该 ip 的所有 data block

// 有了 bmap ，readi 和 writei 就容易实现了
// 可以将 inode 在某个 offset 处的内容与用户/内核态虚拟内存进行 I/O

// Read data from inode.
// Caller must hold ip->lock.
// If user_dst==1, then dst is a user virtual address;
// otherwise, dst is a kernel address.
int
readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    if(either_copyout(user_dst, dst, bp->data + (off % BSIZE), m) == -1) {
      brelse(bp);
      break;
    }
    brelse(bp);
  }
  return n;
}

// Write data to inode.
// Caller must hold ip->lock.
// If user_src==1, then src is a user virtual address;
// otherwise, src is a kernel address.
int
writei(struct inode *ip, int user_src, uint64 src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(off > ip->size || off + n < off)
```

```
    return -1;
  if(off + n > MAXFILE*BSIZE)
    return -1;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    if(either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1) {
      brelse(bp);
      break;
    }
    log_write(bp);
    brelse(bp);
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    iupdate(ip);
  }
  return n;
}

// 将 ip 的信息复制到 stat 结构体中
void
stati(struct inode *ip, struct stat *st);
```

目录层 Directory

目录也是文件，只不过其类型为 `T_DIR` ；同时，它的内容为一个 `dirent` 序列。

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

// Look for a directory entry in a directory.
// If found, set *poff to byte offset of entry.
// 在 inode dp 中找名为 name 的 dirent，如果找到返回 dp 中的偏移量
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
```

```
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}

// Write a new directory entry (name, inum) into the directory dp.
// 在父目录下新增一个 dirent，前提是子目录的 inode 已存在
int
dirlink(struct inode *dp, char *name, uint inum)
{
  int off;
  struct dirent de;
  struct inode *ip;

  // Check that name is not present.
  if((ip = dirlookup(dp, name, 0)) != 0){
    iput(ip);
    return -1;
  }

  // Look for an empty dirent.
  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      break;
  }

  strncpy(de.name, name, DIRSIZ);
  de.inum = inum;
  if(writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
    panic("dirlink");

  return 0;
}
```

路径层 Path Name

相关函数：

```
struct inode* namei(char *path); // 返回路径对应的 inode
struct inode* nameiparent(char *path, char *name); // 返回父目录的 inode，并将文件名复制到 name
```

二者都是用下面的函数实现的

```
// Look up and return the inode for a path name.
// If parent != 0, return the inode for the parent and copy the final
// path element into name, which must have room for DIRSIZ bytes.
// Must be called inside a transaction since it calls iput().
static struct inode*
namex(char *path, int nameiparent, char *name)
{
```

```
  struct inode *ip, *next;

  // 获取根 inode
  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;
}
```

在这个过程中，`iget, ilock` 的分离起到了很大的作用，虽然我现在看不懂。

文件描述符层 File Descriptor

对于进程而言，每个被打开的文件可以用 `file` 来描述：

```
// file.h
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE and FD_DEVICE
  short major;       // FD_DEVICE
  short minor;       // FD_DEVICE
};
```

单个 `struct file` 可以多次出现在一个进程的 file table 中，如使用了 `dup` ；也可以同时出现在多个进程的 file table 中，如通过 `fork` 。

系统中打开的所有文件被保存在 `ftable` 中：

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

在 `ftable` 中分配、复制、关闭文件函数：

```
// Allocate a file structure.
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;
}

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

// Close file f.  (Decrement ref count, close when reaches 0.)
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);

  // 关闭管道
```

```
  if(ff.type == FD_PIPE){
    pipeclose(ff.pipe, ff.writable);
  } else if(ff.type == FD_INODE || ff.type == FD_DEVICE){
    // 对于文件或者设备，通过 iput 减少其引用计数
    begin_op(ff.ip->dev);
    iput(ff.ip);
    end_op(ff.ip->dev);
  }
}
```

实现文件的 `stat, read, write` 操作提供了用户态/内核态虚拟内存与文件的交互，分别使用 `filestat, fileread, filewrite` 实现。

系统调用层 System Call