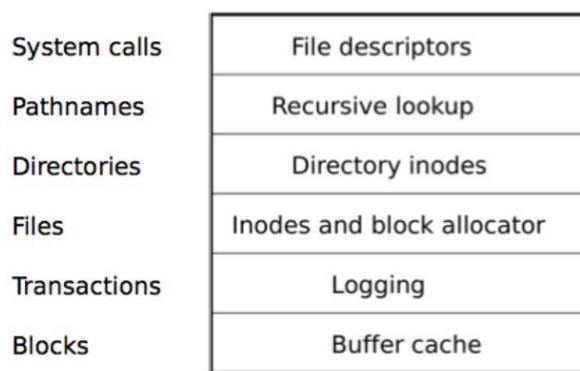


# XV6 文件系统 阅读报告

本次代码阅读主要对 XV6 中的进程调度机制进行调研。主要涉及的代码有：buf.h, fcntl.h, stat.h, fs.h, file.h, ide.c, bio.c, log.c, fs.c, file.c, sysfile.c。除此之外，还涉及到了 traps.c, ioapic.c, picirq.c, param.h 等代码中的功能函数和宏。下面我将逐个剖析，以阐释 XV6 的虚拟存储机制。

在整体性地调研了 XV6 中的文件系统代码后，我们可以发现，XV6 中的文件系统可以大致分为以下 6 层，由低层到高层依次为：缓冲区层、日志层、文件层、目录层、路径层、系统调用层，如下图所示。



下面，我将按从低层到高层的顺序对 XV6 中的文件系统进行分析

## 一、缓冲区层

这一层通过块缓冲机制读写 IDE 硬盘，并把磁盘的访问同步化，保证了只有一个进程可以修改磁盘块。下面让我们来对这一层的实现进行分析。

### (零) IDE 磁盘

在分析最底层的块缓冲之前，我们不妨先来看一下 XV6 中对磁盘的抽象。这部分代码位于 ide.c 中，包含以下内容。

#### 1. 磁盘驱动初始化——ideinit

这个函数用于初始化 XV6 中的磁盘驱动，主要进行以下几方面的工作：

(1) 初始化磁盘操作的互斥锁，初始化中断的特殊位标记，并等待磁盘硬件的初始化完成。

```
void  
ideinit(void)  
{  
    int i;  
  
    initlock(&idelock, "ide");  
    picenable(IRQ_IDE);  
    ioapicenable(IRQ_IDE, ncpu - 1);  
    idewait(0);
```

其中的 picenable 用于设置 IRQ\_IDE 的磁盘中断请求。这个函数调用 picsetmask，调用 outb 函数封装的汇编语句，向对应的硬件 I/O 端口写入修改后（加入 IRQ\_IDE 后）的 irq 中断请求掩码。IRQ\_IDE 在 traps.h 中定义，picenable 在 picirq.c 中实现。

```
void  
picenable(int irq)  
{  
    picsetmask(irqmask & ~(1<<irq));  
}
```

```
static void  
picsetmask(ushort mask)  
{  
    irqmask = mask;  
    outb(IO_PIC1+1, mask);  
    outb(IO_PIC2+1, mask >> 8);  
}
```

```
// I/O Addresses of the two programmable interrupt controllers  
#define IO_PIC1          0x20      // Master (IRQs 0-7)  
#define IO_PIC2          0xA0      // Slave (IRQs 8-15)  
  
#define IRQ_SLAVE        2         // IRQ at which slave connects to master  
  
// Current IRQ mask.  
// Initial IRQ mask has interrupt 2 enabled (for slave 8259A).  
static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
```

接下来，函数开启 I/O APIC 机制，XV6 中的实现参考了 Intel 手册中的说明，限于时间和篇幅，这里不再赘述。

```

void
ioapicenable(int irq, int cpunum)
{
    if(!ismp)
        return;

    // Mark interrupt edge-triggered, active high,
    // enabled, and routed to the given cpunum,
    // which happens to be that cpu's APIC ID.
    ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
    ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
}

```

完成上述硬件初始化后，函数调用 idewait 函数，等待磁盘彻底准备好。这个函数使用 while 循环，不断读取相应 I/O 端口的内容，检查现在是否满足 IDE\_DRDY 的就绪状态。满足后，检查最后一次读取的状态值中是否存在错误，如果存在则返回 -1，否则返回 0，表示已正常初始化。

```

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;
}

```

整个初始化结束后，函数使用循环查询的方式，检查是否存在 disk 1，如果存在，则把全局变量 havedisk1 设置为 1。检查完毕后，函数切换回 disk0。

```

// Check if disk 1 is present
outb(0x1f6, 0xe0 | (1<<4));
for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
        havedisk1 = 1;
        break;
    }
}

// Switch back to disk 0.
outb(0x1f6, 0xe0 | (0<<4));
}

```

## 2.开始（处理）请求——idestart

这个函数用于处理磁盘请求。函数首先等待磁盘恢复到就绪状态，然后设置中断、扇区数（这里默认为 1），并输出缓冲区中记录的扇区号和设备号，这些工作用于确定磁盘的

读/写区域。然后，函数根据缓冲区提供的操作标记进行相应的读/写操作。注意，传入的缓冲区不应为空，否则会调用 panic 报错。

```
static void
idestart(struct buf *b)
{
    if(b == 0)
        panic("idestart");

    idewait(0);
    outb(0x3f6, 0); // generate interrupt
    outb(0x1f2, 1); // number of sectors
    outb(0x1f3, b->sector & 0xff);
    outb(0x1f4, (b->sector >> 8) & 0xff);
    outb(0x1f5, (b->sector >> 16) & 0xff);
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if(b->flags & B_DIRTY){
        outb(0x1f7, IDE_CMD_WRITE);
        outsl(0x1f0, b->data, 512/4);
    } else {
        outb(0x1f7, IDE_CMD_READ);
    }
}
```

### 3.磁盘中断处理——ideintr

这个函数是一个中断处理函数。函数首先尝试获取磁盘互斥锁，然后检查是否存在缓冲区请求，如果不存在，则释放锁并返回。当存在请求时，如果是读请求，函数调用 insl 的 asm 汇编内嵌代码读取数据。然后，函数设置缓冲区的有效位，并清除修改位，表示这是一个刚刚读取到数据的扇区。设置完毕后，函数试图唤醒这个扇区对应的进程。最后，如果磁盘请求队列仍然不为空，那么函数调用 idestart 函数进行对下一块扇区的预处理。这样，我们只需要在这个函数中处理读操作了。上述流程全部结束后，函数释放磁盘锁。

```

void
ideintr(void)
{
    struct buf *b;

    // First queued buffer is the active request.
    acquire(&idelock);
    if((b = idequeue) == 0){
        release(&idelock);
        // cprintf("spurious IDE interrupt\n");
        return;
    }
    idequeue = b->qnext;

    // Read data if needed.
    if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
        insl(0x1f0, b->data, 512/4);

    // Wake process waiting for this buf.
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b);

    // Start disk on next buf in queue.
    if(idequeue != 0)
        idestart(idequeue);

    release(&idelock);
}

```

可以看到，中断处理时，XV6 使用了互斥锁来保证一次只能有一个进程试图操作磁盘。在调用 idestart 时，虽然 idestart 内部没有互斥机制，但 ideintr 中的实现也保证了互斥。

#### 4. 磁盘与缓冲区的内容同步——iderw

简言之，这个函数用于申请一个磁盘的读写请求。函数首先检查传入的缓冲区状态，传入的缓冲区应满足是 BUSY 的、不是 VALID 的、且操作的磁盘应该存在。检查完毕后，函数尝试获取磁盘互斥锁。获取后，函数把当前缓冲块插入请求队列，当第一个块就是当前待处理块时，函数启动磁盘。接下来，函数使用 while 循环，等待磁盘终端处理完毕（我们已经看到了，完毕时，中断处理函数会主动唤醒进程，且设置好了当前缓冲区的状态）。最后，处理完毕，函数释放磁盘互斥锁。

```

void
iderw(struct buf *b)
{
    struct buf **pp;

    if(!(b->flags & B_BUSY))
        panic("iderw: buf not busy");
    if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
        panic("iderw: nothing to do");
    if(b->dev != 0 && !havedisk1)
        panic("iderw: ide disk 1 not present");

    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);
}

```

## (二) 块缓冲的实现

上面我们分析了 XV6 系统与磁盘的交互，在上面的实现中，反复使用了 buf 缓冲区，下面我们来看一下 XV6 对缓冲区的组织方式。

### 1. 缓冲区结构——buf

buf 结构体定义了单个缓冲块的结构，内容如下。可以看到，一个缓冲区的大小为 512 字节，除此之外，它还包含了缓冲区状态(flag)、设备号（磁盘号）、缓存列表中的前驱和后继、待请求队列中的后继指针。

```

struct buf {
    int flags;
    uint dev;
    uint sector;
    struct buf *prev; // LRU cache list
    struct buf *next;
    struct buf *qnext; // disk queue
    uchar data[512];
};

```

在定义了这个模块的 huf.h 中，还包含了 flags 状态中的三个宏，分别为三位，从第 0 位到第 2 位依次为：占用位 (BUSY)、有效位 (已读取到数据)、修改位 (标记需要写回)。

```
#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
```

以上描述了单个缓冲块的结构。整个缓冲区组织为一个链表，且包含了一个互斥锁，保证只有一个进程可以操作缓冲区。

```
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;
```

下面让我们看一下对缓冲区的各种操作。

## 1. 初始化——binit

对缓冲区的初始化只需要初始化互斥锁，初始化缓存块链表。可以看到，bcache 中的链表元素全部存储在 buf 数组中，缓存块个数为 NBUF 个 (params.h 中定义为 10)。

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

## 2.获取磁盘块——bget

这个函数用于尝试获取缓存的磁盘扇区。函数首先获取缓存互斥锁，然后函数遍历缓存列表，寻找是否有所请求磁盘和扇区的缓存，如果有，且没被占用，那么当前进程设置 BUSY 位，占用这个缓存块，并返回块指针；如果被占用了，那么当前进程睡眠，并定期跳转回循环起始处，重新检查。

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

    acquire(&bcache.lock);

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;
        }
    }
}
```

如果这个扇区没被缓存，那么函数寻找一个没被占用且没被修改过的缓存块，进行占用，返回给当前进程。

```
// 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");
}
```

## 3.读与写——bread、bwrite

这两个函数实现了对缓冲块的读写操作。读取时，函数调用 bget 试图获取一个缓冲块。获取后，如果无效，函数调用 iderw 进行读操作，操作完毕后，返回这个块。这里涉

及到的功能性函数上面已经分析过。

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

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

写操作也类似。函数首先检查 BUSY 情况，然后设置 DIRTY 位，最后调用 iderw 对传入的写过的块进行操作。

```
void
bwrite(struct buf *b)
{
    if((b->flags & B_BUSY) == 0)
        panic("bwrite");
    b->flags |= B_DIRTY;
    iderw(b);
}
```

#### 4. 释放缓冲块——brelse

这个函数用于释放传入的缓冲块。函数获取缓冲区互斥锁后，把当前块放至 head 处，清除 BUSY 位后，唤醒当前进程，并释放缓冲区互斥锁。从这里，我们可以看到，head 存放的是最近使用的那个块，可以推断，缓存的组织方式是 MRU 的。也正因此，回顾 bget 中的替换，我们可以发现，它从 head 的前驱开始，也就相当于从最近最后一次被使用的块开始，因此，替换策略是 LRU 的。

### (三) 小结

块缓冲层通过缓存的互斥锁，加之磁盘访问的互斥锁，提供了对磁盘和缓冲区的互斥访问，并且通过功能函数实现了缓冲区和磁盘内容的及时同步。

块缓冲仅允许最多一个进程引用它，以此来同步对磁盘的访问，如果一个内核线程引用了一个缓冲块，但还没有释放它，那么其他调用 bread 的进程就会阻塞。文件系统的更高几层正是依赖块缓冲层的同步机制来保证其正确性。

块缓冲有固定数量的缓冲区，这意味着如果文件系统请求一个不在缓冲中的块，必须

换出一个已经使用的缓冲区。这里的[置换策略是 LRU（用 MRU 的循环链表组织来实现）](#)，因为我们假设最近未使用的块近期内最不可能再被使用。

## 二、日志层

日志系统的引入目标是，恢复操作文件时出现崩溃导致的问题。下面我们来分析一下 XV6 中日志的实现。

### (一) 相关数据结构

XV6 中以日志相关的数据结构主要有两个：logheader、log。

#### 1.logheader

这个结构中包含了计数变量和扇区存储变量。param.h 中定义的日志大小为 10 个扇区。

```
struct logheader {
    int n;
    int sector[LOGSIZE];
};
```

#### 2.log

这个结构包含了单个日志的内容。其中包括：控制互斥访问的自旋锁，起始位置，大小，是否为 BUSY 状态，设备号（磁盘号），和一个 logheader。

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

### (二) 日志操作

#### 1. 初始化——initlog

这个函数首先检查日志头的大小是否超过了缓存块大小，然后对日志中的自旋锁初始化。接下来，函数读取文件系统的超级块，并从中获取日志的起始位置、大小、设备位置，然后调用 recover\_from\_log 进行数据恢复。

```

void
initlog(void)
{
    if (sizeof(struct logheader) >= BSIZE)
        panic("initlog: too big logheader");

    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();
}

```

超级块定义在 fs.h 中，包含了文件系统的总大小、数据块数、i 节点数、日志块数。

```

// File system super block
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
};

```

读取超级块的操作定义在 fs.c 中，就是调用缓存层提供的 bread 函数读取对应磁盘扇区，进行数据拷贝后，再释放这个缓存块。

```

void
readsb(int dev, struct superblock *sb)
{
    struct buf *bp;

    bp = bread(dev, 1);
    memmove(sb, bp->data, sizeof(*sb));
    brelse(bp);
}

```

## 2. 数据恢复——recover\_from\_log

上面的初始化中调用了 recover\_from\_log 函数，这个函数的流程为：读取日志头——数据迁移——清空日志大小——写回日志头的修改。

```

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
}

```

read\_head 函数实现了读取日志头的操作。函数利用初始化后得到的磁盘号和起始扇

区，读取日志内容，包括：日志文件的大小（扇区个数）、日志文件内容被放在的扇区号。

读取完毕后，释放读取时申请的缓存块，

```
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.sector[i] = lh->sector[i];
    }
    brelse(buf);
}
```

install\_trans 包含了对日志数据的迁移。函数遍历每个占用了的扇区，读取日志中的内容，并写回应该存放的目标扇区中。写回时，先把数据拷贝到缓存块中，再调用 bwrite 写回磁盘。

```
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.sector[tail]); // read dst
        memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
        bwrite(dbuf); // write dst to disk
        brelse(lbuf);
        brelse(dbuf);
    }
}
```

write\_head 函数则用于更新日志头中的扇区个数和扇区标号，并写回磁盘，与 read\_head 流程类似，只不过读取后要更新数据并写回

```
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->sector[i] = log.lh.sector[i];
    }
    bwrite(buf);
    brelse(buf);
}
```

### 3. 日志占用与修改——begin\_trans、commit\_trans、log\_write

当一个进程想修改日志时，首先要获取日志的占用权，这一过程由 begin\_trans 实现。

函数首先获取互斥锁，然后等待 log.busy 为 0，等待时进程睡眠以提高 CPU 效率。然后，进程设置 log.busy，并释放互斥锁。这样，后续的进程要么等待在获取锁的位置，要么等待 busy 变为 0，这样就实现了对日志的互斥。

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

进程想要修改日志内容时，调用 log\_write 函数。这个函数相当于对 bwrite 的一个封装。当进程修改文件内容时，首先修改到日志中，再由日志来进行统一的数据更新，这样便可以防止缓存崩溃时的不同步问题。这个函数首先检查日志情况（日志容量、日志占用），然后找到一个可用的日志扇区，再把对应的修改后的扇区写到这个被选中的日志扇区中。

```
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.busy)
        panic("write outside of trans");

    for (i = 0; i < log.lh.n; i++) {
        if (log.lh.sector[i] == b->sector) // log absorbtion?
            break;
    }
    log.lh.sector[i] = b->sector;
    struct buf *lbuf = bread(b->dev, log.start+i+1);
    memmove(lbuf->data, b->data, BSIZE);
    bwrite(lbuf);
    brelse(lbuf);
    if (i == log.lh.n)
        log.lh.n++;
    b->flags |= B_DIRTY; // XXX prevent eviction
}
```

当进程完成修改，想要结束对日志的占用时，进程可以调用 commit\_trans。这个函数首先把更新了的数据从日志中迁移到对应的目标扇区（流程与 recover\_from\_log 相同），然

后清除 busy 位，唤醒等待的进程，并释放锁。

```
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);
}
```

### (三) 小结

日志存在于磁盘中已知的固定区域。它包含了一个起始块，紧接着一连串的数据块。

起始块包含了一个扇区号的数组，每一个对应于日志中的数据块，起始块还包含了对日志数据块的计数。

XV6 在提交后修改日志的起始块，而不是之前，并且在将日志中的数据块都拷贝到文件系统之后将数据块计数清 0。提交之后，清 0 之前的崩溃就会导致一个非 0 的计数值。

当进程需要写时，需要包含一个完整的原子写操作序列，任何时刻只有一个进程再执行这个序列。但 XV6 允许读操作并发执行。

XV6 中的日志空间是固定大小的，所以对于那些需要大量写的操作，我们可能需要把它进行拆分。

## 三、文件层

这一层的主要内容是 i 节点的实现。i 节点记录了每个文件的基本信息，是文件系统中管理文件的重要工具，下面我们一起来看一下 XV6 中的 i 节点设计。

### (零) 块操作

在分析 i 节点之前，我们首先来看一下 XV6 中对块的操作，这里主要包含以下几方面的内容：

#### 1.bzero

这个函数用于清空一个缓存块，并把它写回磁盘。函数首先根据参数获取对应磁盘块缓存，然后刷新缓存，写回磁盘，再释放这个缓存块。

```
static void
bzero(int dev, int bno)
{
    struct buf *bp;

    bp = bread(dev, bno);
    memset(bp->data, 0, BSIZE);
    log_write(bp);
    brelse(bp);
}
```

## 2.balloc

这个函数用于选择一个磁盘块，分配给相应的文件。函数首先读取超级块的细腻，然后遍历所有的数据块，检查数据块是否空闲，如果空闲，那么占用这个块，更新日志，清空对应的磁盘块后，返回磁盘块的标号。可以看到，XV6 中使用类似于位图的方式来管理块的空闲情况，所以只要通过简单的位操作即可判断当前块是否被占用。

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

    bp = 0;
    readsb(dev, &sb);
    for(b = 0; b < sb.size; b += BPB){
        bp = bread(dev, BBLOCK(b, sb.ninodes));
        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");
}
```

## 3.bfree

这个函数用于释放一个已分配的磁盘块。函数首先读取超级块内容，然后更新超级快的内容，写回日志中，最后释放缓存。

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

    readsb(dev, &sb);
    bp = bread(dev, BBLOCK(b, sb.ninodes));
    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);
}
```

## (二) i 节点相关的数据结构

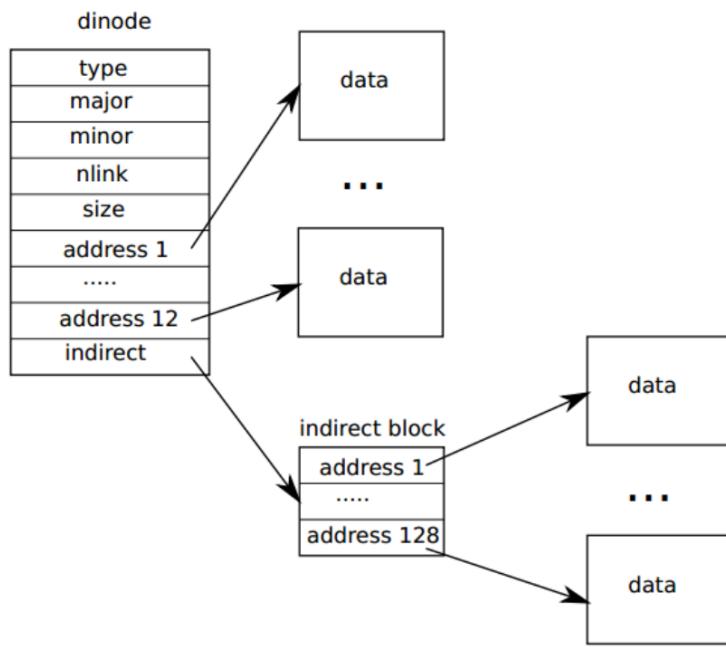
XV6 中抽象了 2 种 i 节点，一种是磁盘中的 i 节点，另一种是内存中的 i 节点。对于内存中的 i 节点，XV6 还提供了一个 i 节点缓存，来组织它们。

### 1.dinode

这个结构描述了磁盘中的 i 节点内容，包括：文件类型、最大设备号、最小设备号、链接的个数、文件大小、块的地址。

```
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
};
```

dinode 组织文件的格式大致如下图所示：



## 2.inode

这个结构描述了磁盘中的 i 节点内容，包括：设备号、i 节点号、引用计数、标记、以及 dinode 中的各个内容。

```
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];
};
```

## 3.icache

这个结构是组织 inode 的缓存结构，包括 inode 数组和一个互斥用的锁。param.h 中定义的最大 inode 个数为 50。

```
struct {
    struct spinlock lock;
    struct inode inode[NINODE];
} icache;
```

## 4.功能性宏

除了上述基本数据结构，XV6 还提供了一些用于简化计算的宏操作，包括：每个块中的 i 节点个数、包含第 i 个 i 节点的块的块号、每个块中的位图位数、块 b 的位图位所在的块。

```
// Inodes per block.  
#define IPB          (BSIZE / sizeof(struct dinode))  
  
// Block containing inode i  
#define IBLOCK(i)    ((i) / IPB + 2)  
  
// Bitmap bits per block  
#define BPB          (BSIZE*8)  
  
// Block containing bit for block b  
#define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
```

## (三) i 节点相关的操作

XV6 中提供的对 i 节点的操作大致如下。

### 1.缓存初始化——iinit

这个函数用于初始化 i 节点缓存。我们只需要初始化缓存结构中的互斥锁。

```
void  
iinit(void)  
{  
    initlock(&icache.lock, "icache");  
}
```

### 2.i 节点分配——ialloc

这个函数用于分配一个 i 节点。函数首先读取超级快，然后利用与 balloc 函数类似的逻辑来分配 i 节点块。不同的是，这里使用的是对 i 节点块的管理数据。找到以后，xindei 节点仍被初始化为全 0。

```

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

    readsb(dev, &sb);

    for(inum = 1; inum < sb.ninodes; inum++){
        bp = bread(dev, IBLOCK(inum));
        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 igrab(dev, inum);
        }
        brelse(bp);
    }
    panic("ialloc: no inodes");
}

```

这个函数中还调用了 igrab 函数，用于寻找对应的 dinode，并把它封装为内存中的 inode 来返回。当想要获取 i 节点时，这个函数首先检查 icache 里是否已经缓存，如果缓存，直接返回对于的 i 节点，并增加对这个 i 节点的引用计数。否则，函数寻找一个未使用的缓存位置，并对这个 i 节点中的设备、块号、引用计数、标记进行初始化，然后返回这个初始化好的 i 节点指针。整个过程也受到了 icache 中的互斥锁的保护。

```

static struct inode*
igrab(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("igrab: no inodes");

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

    return ip;
}

```

### 3.i 节点更新——iupdate

这个函数用于更新 i 节点信息。函数首先读取磁盘中对应存放 i 节点的块，然后更新磁盘 i 节点的信息，先写回日志，再释放磁盘缓存块。

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

    bp = bread(ip->dev, IBLOCK(ip->inum));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    dip->type = ip->type;
    dip->major = ip->major;
    dip->minor = ip->minor;
    dip->nlink = ip->nlink;
    dip->size = ip->size;
    memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
    log_write(bp);
    brelse(bp);
}
```

### 4.i 节点复制——idup

类似于 linux 中 dup 函数的机制，增加 i 节点的引用计数，让更高层的结构指向相同的 i 节点。增加计数也受到互斥锁的同步。

```
struct inode*
idup(struct inode *ip)
{
    acquire(&icache.lock);
    ip->ref++;
    release(&icache.lock);
    return ip;
}
```

### 4.i 节点的锁定与解锁——ilock、iunlock

在使用 i 节点之前，我们要先把 i 节点锁定住，这个过程通过 ilock 实现。函数首先检查 i 节点指针是否为空，以及引用计数是否为 0，如果二者之中有一个成立则报错。接下来，函数获取 icache 的锁，利用 while-sleep 结构循环地睡眠等待当前 i 节点不处在 BUSY 状态，然后自己为 i 节点设置 BUSY 状态后，再释放 icache 的锁。如果目前的 i 节点是无效的，那么函数还会先从磁盘中对应的为 u 在读取磁盘 i 节点信息，并重新设置有效位。此时，如果 i 节点中的文件种类位 0，会报“无文件类型”的错误。

```

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

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

    acquire(&icache.lock);
    while(ip->flags & I_BUSY)
        sleep(ip, &icache.lock);
    ip->flags |= I_BUSY;
    release(&icache.lock);

    if(!(ip->flags & I_VALID)){
        bp = bread(ip->dev, IBLOCK(ip->inum));
        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->flags |= I_VALID;
        if(ip->type == 0)
            panic("ilock: no type");
    }
}

```

在使用完 i 节点以后，我们可以调用 iunlock 进行释放。函数首先检查 i 节点的状态，然后互斥地释放 BUSY 位，并唤醒一个等待的进程。

```

void
iunlock(struct inode *ip)
{
    if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
        panic("iunlock");

    acquire(&icache.lock);
    ip->flags &= ~I_BUSY;
    wakeup(ip);
    release(&icache.lock);
}

```

#### 4.i 节点的释放——iuput、iunlockput

当使用完一个 i 节点以后，我们需要释放它。iuput 中的主要流程为：如果引用计数为 1，且 i 节点有效，nlink 数为 0，那么函数会截断并释放 i 节点，然后再进行引用计数-1 的

操作。否则，函数只是单纯地进行引用计数的递减。

```
void
iput(struct inode *ip)
{
    acquire(&icache.lock);
    if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
        // inode has no links: truncate and free inode.
        if(ip->flags & I_BUSY)
            panic("iput busy");
        ip->flags |= I_BUSY;
        release(&icache.lock);
        itrunc(ip);
        ip->type = 0;
        iupdate(ip);
        acquire(&icache.lock);
        ip->flags = 0;
        wakeup(ip);
    }
    ip->ref--;
    release(&icache.lock);
}
```

截断并释放 i 节点的操作为：首先设置 BUSY 位，然后调用 itrunc' 函数截断，最好清空 type、flags 位，更新 i 节点信息到磁盘中，并唤醒等待使用的进程。

截断时会调用 itrunc 函数。这个函数释放所有的引用的磁盘块。itrunc 首先遍历直接索引，清除索引块；然后判断是否存在间接索引，如果存在，函数读取间接索引块，先释放二级索引，再释放一级索引块。最好，函数更新 i 节点的大小位 0，并把所有释放后的数据写回磁盘。

```

static void
itrunc(struct inode *ip)
{
    int i, j;
    struct buf *bp;
    uint *a;

    for(i = 0; i < NDIRECT; i++){
        if(ip->addrs[i]){
            bfree(ip->dev, ip->addrs[i]);
            ip->addrs[i] = 0;
        }
    }

    if(ip->addrs[NDIRECT]){
        bp = bread(ip->dev, ip->addrs[NDIRECT]);
        a = (uint*)bp->data;
        for(j = 0; j < NINDIRECT; j++){
            if(a[j])
                bfree(ip->dev, a[j]);
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT]);
        ip->addrs[NDIRECT] = 0;
    }

    ip->size = 0;
    iupdate(ip);
}

```

iunlockput 则提供了解锁并释放的封装。

```

void
iunlockput(struct inode *ip)
{
    iunlock(ip);
    iput(ip);
}

```

## 5.内容获取——bmap

这个函数用于获取索引部分的内容，函数根据直接索引区域或简介索引区域，分别给出对应的索引地址。当地址为空时，函数会调用 balloc 先分配一块，然后再返回地址。

```

static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;

    if(bn < NDIRECT){
        if((addr = ip->addrs[bn]) == 0)
            ip->addrs[bn] = addr = balloc(ip->dev);
        return addr;
    }
    bn -= NDIRECT;

    if(bn < NINDIRECT){
        // Load indirect block, allocating if necessary.
        if((addr = ip->addrs[NDIRECT]) == 0)
            ip->addrs[NDIRECT] = addr = balloc(ip->dev);
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
        if((addr = a[bn]) == 0){
            a[bn] = addr = balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);
        return addr;
    }

    panic("bmap: out of range");
}

```

## 6.状态复制——stat

这个函数把 i 节点的状态复制到 stat 结构中。

```

void
stat(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;
}

```

stat 的内容如下，包含了：文件类型、设备号、i 节点号、链接数、文件大小（字节为单位）。

```

struct stat {
    short type; // Type of file
    int dev; // File system's disk device
    uint ino; // Inode number
    short nlink; // Number of links to file
    uint size; // Size of file in bytes
};

```

其中的文件类型定义如下，主要区分的是目录、文件、设备。

```
#define T_DIR 1 // Directory
#define T_FILE 2 // File
#define T_DEV 3 // Device
```

## 7.i 节点读写——readi、writei

读取 i 节点的数据时，使用 readi 函数。

```
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
    uint tot, m;
    struct buf *bp;

    if(ip->type == T_DEV){
        if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
            return -1;
        return devsw[ip->major].read(ip, dst, n);
    }

    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);
        memmove(dst, bp->data + off%BSIZE, m);
        brelse(bp);
    }
    return n;
}
```

函数的流程如下：

如果 i 节点对应的文件类型是设备文件，那么如果 i 节点的主设备号不符合要求，或者设备中没有读函数，那么返回-1，认为读取失败；否则函数调用设备中的 read 函数进行读取。设备文件的定义在 file.h 中，内容为：读操作的函数指针，写操作的函数指针。而 file.c 中声明了一个储存设备文件的数组 devsw，在这里也得到了使用。特别地，控制台设备记作 1。param.h 中，给出的设备数量最大为 10 (NDEV)。

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

extern struct devsw devsw[];

#define CONSOLE 1
```

```
struct devsw devsw[NDEV];
```

对于其他文件，函数首先检查偏移是否越界，以及读取的长度是否非负。然后，如果发现完全按照输入的字节数读取之后越界了，函数还会对读取到的字节数进行调整。

在做好准备工作后，函数利用 bmap 获取对应的块，并读取需要的内容。读取完毕后，函数返回真正读取到的字节数。

```
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
    uint tot, m;
    struct buf *bp;

    if(ip->type == T_DEV){
        if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
            return -1;
        return devsw[ip->major].read(ip, dst, n);
    }

    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);
        memmove(dst, bp->data + off%BSIZE, m);
        brelse(bp);
    }
    return n;
}
```

向 i 节点对应内容写入时，调用 writei 函数。函数的流程与 readi 总体相似。对于设备文件，函数检查 i 节点信息后调用设备文件的 write 函数进行写入。对于其他文件，函数则也是进行检查后，进行写操作。检查时，还需要检查写入后的长度不超过允许的最大总长度 (MAXFILE\*BSIZE,) 这个函数在成功时，返回写入的字节数，失败时，返回-1。

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
```

```

int
writei(struct inode *ip, char *src, uint off, uint n)
{
    uint tot, m;
    struct buf *bp;

    if(ip->type == T_DEV){
        if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
            return -1;
        return devsw[ip->major].write(ip, src, n);
    }

    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);
        memmove(bp->data + off%BSIZE, src, m);
        log_write(bp);
        brelse(bp);
    }

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

```

## (四) 小结

文件和目录的内容存在磁盘块中，磁盘块都从一个空闲块池中分配出来。xv6 的块分配器包含一个磁盘上的空闲块位图，每个块占一个位。引导区，超级块，i 节点块和位图块的位永远都被置为有效。

块分配器提供三个功能：alloc 分配一个新的磁盘块，free 释放一个块。bzero 清零一个块。

XV6 的 i 节点有两种语义：它可以指的是磁盘上的记录文件大小、数据块扇区号的数据结构；也可以指内存中的一个 i 节点，它包含了一个磁盘上 i 节点的拷贝，以及一些内核需要的附加信息。

所有的磁盘上的 i 节点都被打包在一个称为 i 节点块的连续区域中。每一个 i 节点的大小都是一样的，所以对于一个给定的数字 n，很容易找到磁盘上对应的 i 节点。事实上这个给定的数字就是操作系统中 i 节点的编号。

内核中维护了一个 i 节点缓存，它只会缓存被 C 指针指向的 i 节点。它主要的工作是

同步多个进程对 i 节点的访问而非缓存。如果一个 i 节点频繁被使用，块缓冲可能会把它保留在内存中，即使 i 节点缓存没有缓存它。

为了更好地使用 i 节点，以及提供互斥的抽象，XV6 中使用 `iget`、`iput`、`iloch`、`iunlock` 等一系列机制来实现对 i 节点缓存的互斥操作。

## 四、目录层

XV6 中对目录的管理与普通文件的管理有些类似。正如上面所展示，目录文件的类型定义为 `T_DIR`，也受 i 节点的管理。那么下面我们来分析一下目录文件的实现。

### (一) 目录文件涉及到的其他数据结构

除了 i 节点以外，XV6 还为目录射击了一个 `dirent` 的目录项数据结构，里面包含目录名和 i 节点号。

```
// Directory is a file containing a sequence of dirent structures.
#define DIRSIZ 14

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

### (二) 与目录相关的操作

XV6 中提供了以下用来操作目录的函数。

#### 1. 目录名拷贝——`namecmp`

这个函数用于拷贝目录名称，添加了对目录名长度的限制。

```
int
namecmp(const char *s, const char *t)
{
    return strncmp(s, t, DIRSIZ);
```

#### 2. 目录查找——`dirlookup`

这个函数用于查找目录项。函数根据传入的内容，遍历目录文件，查找其中是否有相应的目录项。

```

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, (char*)&de, off, sizeof(de)) != sizeof(de))
            panic("dirlink read");
        if(de.inum == 0)
            continue;
        if(namecmp(name, de.name) == 0){
            // entry matches path element
            if(poff)
                *poff = off;
            inum = de.inum;
            return igit(dp->dev, inum);
        }
    }

    return 0;
}

```

### 3. 目录项添加——dirlookup

这个函数根据传入的目录文件 i 节点，名称和 i 节点号，在目录文件中添加新的目录项。在添加之前，首先要检查是否已存在重名项，然后寻找一个新的项，再更新目录文件并写回。

```

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, (char*)&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, (char*)&de, off, sizeof(de)) != sizeof(de))
        panic("dirlink");

    return 0;
}

```

## 五、路径层

这一层相当于是对目录层的扩展。XV6 提供了几个处理函数，允许进程把路径逐层添加上去。让我们来一起分析一下。

### (一) 路径相关的操作

#### 1.路径分解——skipelem

这是一个字符串处理函数，函数把传入的完整路径分解为当前最高级目录名和后续路径，最高级路径名放入 name 指针中，余下路径作为返回值传回。操作的思路就是对路径分隔符'/'进行搜索和前后字符串的分隔。

```
static char*
skipelem(char *path, char *name)
{
    char *s;
    int len;

    while(*path == '/')
        path++;
    if(*path == 0)
        return 0;
    s = path;
    while(*path != '/' && *path != 0)
        path++;
    len = path - s;
    if(len >= DIRSIZ)
        memmove(name, s, DIRSIZ);
    else {
        memmove(name, s, len);
        name[len] = 0;
    }
    while(*path == '/')
        path++;
    return path;
}
```

#### 2.寻找多级目录下的文件 i 节点——namex

这个函数用于寻找输入的路径中，最后一级文件对应的 i 节点。函数从最高级目录开始（如果以'/'开头，从根目录开始，否则从进程中记录的当前目录开始），通过 while 循环，反复调用 skipelem 来分隔余下路径和当前目录，每次循环中，函数在对文件类型进行检查（过程中应该是目录文件）后，调用 dirlookup 函数查询当前路径中是否存在对应地目录项。如果存在，函数更新当前的 i 节点变量，直到最后一级。

退出 while 循环以后，函数再对 nameiparent 做检查以后，返回最后一级的 i 节点，

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

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

    while((path = skipel(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;
}
```

### 3.namex 的封装——namei、nameiparent

XV6 提供了两个简便的封装。namei 解析的是最后一级下的 i 节点，而 nameiparent 给出的的倒数第二级下的 i 节点。

```
struct inode*
namei(char *path)
{
    char name[DIRSIZ];
    return namex(path, 0, name);
}

struct inode*
nameiparent(char *path, char *name)
{
    return namex(path, 1, name);
}
```

## 六、系统调用层

以上我们分析的都是 XV6 系统内部的实现和抽象，而对于程序员，XV6 给出了更简化

的抽象——文件、文件描述符和文件的系统调用，这两部分构成了文件系统的最高层——系统调用层，它们是直接面向程序员的接口。下面让我们来分析一下这一层的具体实现。

## (一) 与文件（描述符）相关的数据结构

XV6 提供了一个文件类，它标识着一个文件所容纳的信息。内容为：文件类型、引用计数、可读性、可写性、管道指针、i 节点指针、文件偏移（文件位置）。

```
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;
};
```

此外，XV6 中还提供了一个文件表，它包含了一个文件数组和一个互斥锁，记录了目前系统中所有的打开文件。param.h 中给出的最大文件数量 NFILE 为 100。

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

## (二) 文件操作

XV6 在文件这个抽象层面上提供了以下几种操作

### 1.文件表初始化—fileinit

这个函数初始化了打开文件表。主要操作为初始化表中的互斥锁。

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

### 1.分配文件—filealloc

这个函数用于在文件表中寻找一个没被引用过的文件结构。函数遍历打开文件表，寻找一个引用计数为 0 的表项，增加引用计数，返回其指针。如果寻找失败，函数便会返回空指针。

```
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;
}
```

## 2. 复制文件—filedup

这个函数为对应的打开文件表表项增加一个引用计数，但要求这个文件得是已经有引用计数的。它提供了类似于 dup 的抽象。

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

## 2. 关闭文件—fclose

当试图关闭一个文件时，我们首先对引用计数-1.如果发现关闭以后引用计数变为 0，那么函数把这个表项的类型设置为 FD\_NONE。如果原来打开的是管道文件，那么函数调用 pipeclose 函数关闭管道；如果原来打开的是普通文件，有 i 节点对应，那么函数也会对相应的 i 节点进行更新。

```
void
fclose(struct file *f)
{
    struct file ff;

    acquire(&ftable.lock);
    if(f->ref < 1)
        panic("fclose");
    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();
    }
}
```

### 3.获取文件状态—filestat

这个函数通过调用 stat 函数向传入的 stat 指针中写入文件的状态。在操作时，如果文件类型是有 i 节点的文件，那么函数通过 ilock 和 iunlock 保证了读取状态时不会被干扰，读取完毕后返回 0；对于其他类型的文件，函数直接返回 -1。

```
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;
}
```

## 4. 读文件—fileread

在真正开始读操作之前，函数首先检查是否可读，不可读则返回-1。接下来，对于管道文件，函数直接调用 piperead 函数进行读操作；对于普通文件，这个函数调用 readi 函数，基于当前的偏移 off，读取一定长度的文件内容。当读取成功后，函数返回读取到的长度。如果出现了其他情况，应当认为系统出现了问题，函数会调用 panic 进行报错。

```
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");
}
```

## 5. 写文件—filewrite

写文件的流程与读操作类似。首先检查可写性，并对管道文件进行调用 pipewrite 函数的特殊处理。接下来，对于一般文件，为了避免前面提到过的日志溢出的问题，这里的实现是，每次只写一部分内容，写完以后就从日志刷新到目标扇区。函数循环这个流程，直到彻底完成所有操作。此时，如果全部写入，函数返回总长度，否则函数返回-1。

```
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){
        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");
}
```

### (三) 向用户提供的系统调用

最终，文件系统提供的最高层抽象即为 sysfile.c 中的 17 个系统调用。对于大部分系统调用来说，它们只是把（二）中的函数进行了封装，这里我挑选几个实现起来稍有难度的调用来进行分析。

#### 1.syslink、sysunlink

sys\_link 和 sys\_unlink 用于修改目录文件，它们可能创建或者移除对 i 节点的引用。

sys\_link 最开始获取自己的参数 old 和 new 两个字符串。这里需要假设 old 是存在的并且不是一个目录文件，那么 sys\_link 会增加它的 ip->nlink 计数。然后 sys\_link 调用 nameiparent(new) 来寻找上级目录和最终的目录元素，并且创建一个目录项指向 old 的 i 节点。new 的上级目录必须和已有 old 的 i 节点在同一个设备上；i 节点号只在同一个磁盘上有意义。如果发生了错误，sys\_link 必须回溯并且回原引用计数。

```
int
sys_link(void)
{
    char name[DIRSIZ], *new, *old;
    struct inode *dp, *ip;

    if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
        return -1;
    if((ip = namei(old)) == 0)
        return -1;

    begin_trans();

    ilock(ip);
    if(ip->type == T_DIR){
        iunlockput(ip);
        commit_trans();
        return -1;
    }

    ip->nlink++;
    iupdate(ip);
    iunlock(ip);

    if((dp = nameiparent(new, name)) == 0)
        goto bad;
    ilock(dp);
    if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
        iunlockput(dp);
        goto bad;
    }
    iunlockput(dp);
    iput(ip);

    commit_trans();

    return 0;

bad:
    ilock(ip);
    ip->nlink--;
    iupdate(ip);
    iunlockput(ip);
    commit_trans();
    return -1;
}
```

sysulink 的流程与 syslink 大致相似，只不过是删除目录中的某个目录项。

```
int
sys_unlink(void)
{
    struct inode *ip, *dp;
    struct dirent de;
    char name[DIRSIZ], *path;
    uint off;

    if(argstr(0, &path) < 0)
        return -1;
    if((dp = nameiparent(path, name)) == 0)
        return -1;

    begin_trans();

    ilock(dp);

    // Cannot unlink ".." or ...
    if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
        goto bad;

    if((ip = dirlookup(dp, name, &off)) == 0)
        goto bad;
    ilock(ip);

    if(ip->nlink < 1)
        panic("unlink: nlink < 1");
    if(ip->type == T_DIR && !isdirempty(ip)){
        iunlockput(ip);
        goto bad;
    }

    memset(&de, 0, sizeof(de));
    if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
        panic("unlink: writei");
    if(ip->type == T_DIR){
        dp->nlink--;
        iupdate(dp);
    }
    iunlockput(dp);

    ip->nlink--;
    iupdate(ip);
    iunlockput(ip);

    commit_trans();

    return 0;

bad:
    iunlockput(dp);
    commit_trans();
    return -1;
}
```

## 2.create

sys\_link 为一个已有的 i 节点创建新的名字，而 create 为一个新的 i 节点创建新名字。

实际上，它是三个文件创建系统调用的综合：用 O\_CREATE 方式 open 一个文件创建一个新的普通文件，用 mkdir 创建一个新的目录文件，或者用 mkdev 创建一个新的设备文件。就像 sys\_link 一样，create 调用 nameiparent 获取上级目录的 i 节点。然后调用 dirlookup 来检查同名文件是否已经存在。如果的确存在，create 的行为就由它服务的系统调用所决定。

如果是 open (type==T\_FILE) 调用的 create 并且按指定文件名找到的文件是一个普通

文件，那么就认为打开成功，因此 create 中也认为是成功。在其他情况下，这就是一个错误。如果文件名并不存在，create 就会用 ialloc 分配一个新的 i 节点。如果新的 i 节点是一个目录，create 就会初始化 . 和 .. 两个目录项。最后所有的数据都初始化妥当了，create 就可以把它连接到它的上级目录。create，正如 sys\_link 一样，函数同时拥有两个 i 节点锁：ip 和 dp。这不可能导致死锁，因为 i 节点 ip 是刚被分配的：系统中没有其他进程会持有 ip 的锁并且尝试锁 dp。

```
static struct inode*
create(char *path, short type, short major, short minor)
{
    uint off;
    struct inode *ip, *dp;
    char name[DIRSIZ];

    if((dp = nameiparent(path, name)) == 0)
        return 0;
    ilock(dp);

    if((ip = dirlookup(dp, name, &off)) != 0){
        iunlockput(dp);
        ilock(ip);
        if(type == T_FILE && ip->type == T_FILE)
            return ip;
        iunlockput(ip);
        return 0;
    }

    if((ip = ialloc(dp->dev, type)) == 0)
        panic("create: ialloc");

    ilock(ip);
    ip->major = major;
    ip->minor = minor;
    ip->nlink = 1;
    iupdate(ip);

    if(type == T_DIR){ // Create . and .. entries.
        dp->nlink++; // for ".."
        iupdate(dp);
        // No ip->nlink++ for ".": avoid cyclic ref count.
        if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
            panic("create dots");
    }

    if(dirlink(dp, name, ip->inum) < 0)
        panic("create: dirlink");

    iunlockput(dp);

    return ip;
}
```

### 3. sys\_open、sys\_mkdir、sys\_mknod

基于 sys\_create，这三个系统调用的实现便变得容易了许多。sys\_makedir 和 sys\_mknod 本质上就是对 create 的特定形式的调用，以及不同条件下错误检查的封装。

```

int
sys_mkdir(void)
{
    char *path;
    struct inode *ip;

    begin_trans();
    if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
        commit_trans();
        return -1;
    }
    iunlockput(ip);
    commit_trans();
    return 0;
}

```

```

int
sys_mknod(void)
{
    struct inode *ip;
    char *path;
    int len;
    int major, minor;

    begin_trans();
    if((len=argstr(0, &path)) < 0 ||
       argint(1, &major) < 0 ||
       argint(2, &minor) < 0 ||
       (ip = create(path, T_DEV, major, minor)) == 0){
        commit_trans();
        return -1;
    }
    iunlockput(ip);
    commit_trans();
    return 0;
}

```

而 sys\_open 是最复杂的，创建一个新文件只是它能做的很少一部分事。如果 open 以 O\_CREATE 调用，它就会调用 create。否则，它就会调用 namei。create 会返回一个带锁的 i 节点，但是 namei 并不会，所以 sys\_open 必须要自己锁上这个 i 节点。这提供了一个合适的地方来检查目录只被打开用于读，而不是写。

现在，我们获得了一个 i 节点（不管是用 create 还是用 namei），sys\_open 分配了一个文件和文件描述符（用 filealloc 的返回值来充当描述符），接着填充了这个文件。注意到，没有其他进程能够访问初始化尚未完成的文件，因为这样的文件只存在于当前进程的文件表中。

```

int
sys_open(void)
{
    char *path;
    int fd, omode;
    struct file *f;
    struct inode *ip;

    if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
        return -1;
    if(omode & O_CREATE){
        begin_trans();
        ip = create(path, T_FILE, 0, 0);
        commit_trans();
        if(ip == 0)
            return -1;
    } else {
        if((ip = namei(path)) == 0)
            return -1;
        ilock(ip);
        if(ip->type == T_DIR && omode != O_RDONLY){
            iunlockput(ip);
            return -1;
        }
    }

    if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
        if(f)
            fileclose(f);
        iunlockput(ip);
        return -1;
    }
    iunlock(ip);

    f->type = FD_INODE;
    f->ip = ip;
    f->off = 0;
    f->readable = !(omode & O_WRONLY);
    f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
    return fd;
}

```

### 3. sys\_pipe

这个函数通过管道对的方式把管道的实现和文件系统连接来。它的参数是一个指向可装入两个整数的数组指针，这个数组将用于记录两个新的文件描述符。然后它分配管道，将新的文件描述符存入这个数组中。

```

int
sys_pipe(void)
{
    int *fd;
    struct file *rf, *wf;
    int fd0, fd1;

    if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
        return -1;
    if(pipealloc(&rf, &wf) < 0)
        return -1;
    fd0 = -1;
    if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
        if(fd0 >= 0)
            proc->ofile[fd0] = 0;
        fileclose(rf);
        fileclose(wf);
        return -1;
    }
    fd[0] = fd0;
    fd[1] = fd1;
    return 0;
}

```

## 七、小结

本次代码阅读中，我调研了 XV6 的文件系统机制。文件系统的目的是组织和存储数据，典型的文件系统支持用户和程序间的数据共享，并提供数据持久化的支持（即重启之后数据仍然可用）。

xv6 的文件系统中使用了类似 Unix 的文件，文件描述符，目录和路径名，并且把数据存储到一块 IDE 磁盘上。从上面的分析中，我们可以看到，这个文件系统解决了以下几个问题：

1. 访问磁盘比访问内存要慢几个数量级，所以文件系统必须要维护一个内存内的 cache 用于缓存常被访问的块。（磁盘缓存）
2. 不同的进程可能同时操作文件系统，要保证这种并行不会破坏文件系统的正常工作。（同步互斥）
3. 该文件系统必须支持崩溃恢复，也就是说，如果系统崩溃了（比如掉电了），文件系统必须保证在重启后仍能正常工作。问题在于一次系统崩溃可能打断一连串的更新操作，从而使得磁盘上的数据结构变得不一致（例如：有的块同时被标记为使用中和空闲）。（日志）
4. 该文件系统需要磁盘上数据结构来表示目录树和文件，记录每个文件用于存储数据的块，以及磁盘上哪些区域是空闲的。

从总体抽象上来说，XV6 中的文件系统也类似于三级结构——文件描述符、打开文件、i 节点，提供给程序员的接口和作用方式也与 UNIX 系列的操作系统相当。