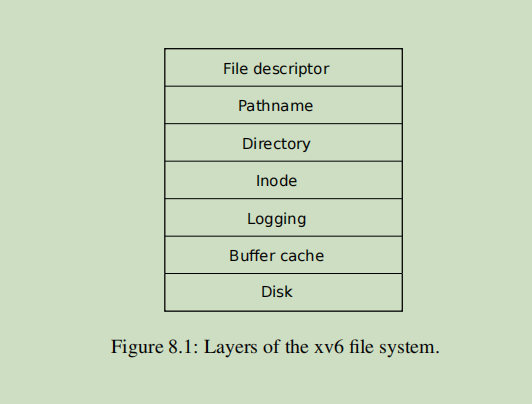
**xv6 note - File System 篇**

**File system**



**一：Buffer cache**

初始化分配1000个节点的双向链表，每一个块的 data数据是1024 个byte(uchar)

**bread 函数**

|  |
| --- |
| C // Return a locked buf with the contents of the indicated block. struct buf\* bread(uint dev, uint blockno) {  struct buf \*b;   // bget 只是获取到一个目前使用，或者 least recently used（LRU，最新最少使用）的未使用的缓存节点  b = bget(dev, blockno);  if(!b->valid) {  // 真正的从磁盘中填充buf, 是在这里  virtio\_disk\_rw(b, 0);  b->valid = 1;  }  return b; } |

**brelse 函数**

将申请到的buffer缓存节点清理掉，和 bread是逆操作

**二：Logging**

一般的logging调用，都是先调用 begin\_op, 再调用 end\_op.

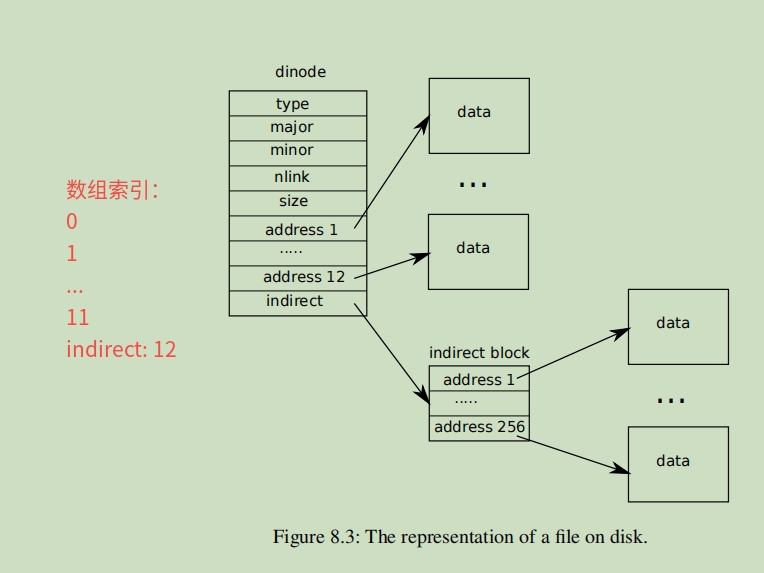
**begin\_op 函数**

**end\_op 函数**

**log\_write 函数**

用于当使用者修改了 buffer里面的数据，然后需要在logging中记录这个被修改的block的number信息，并将buffer里面的 **refcnt** 字段加1。

**三：Inode**



**bmap 函数**

按照上面内存布局的顺序来读取和访问里面的元素信息

|  |
| --- |
| C // Return the disk block address of the nth block in inode ip. // If there is no such block, bmap allocates one. // returns 0 if out of disk space. static uint bmap(struct inode \*ip, uint bn) {  uint addr, \*a;  struct buf \*bp;   if(bn < NDIRECT){  if((addr = ip->addrs[bn]) == 0){  addr = balloc(ip->dev);  if(addr == 0)  return 0;  ip->addrs[bn] = addr;  }  return addr;  }  bn -= NDIRECT;   if(bn < NINDIRECT){  // Load indirect block, allocating if necessary.  if((addr = ip->addrs[NDIRECT]) == 0){  addr = balloc(ip->dev);  if(addr == 0)  return 0;  ip->addrs[NDIRECT] = addr;  }  bp = bread(ip->dev, addr);  a = (uint\*)bp->data;  if((addr = a[bn]) == 0){  addr = balloc(ip->dev);  if(addr){  a[bn] = addr;  log\_write(bp);  }  }  brelse(bp);  return addr;  }   panic("bmap: out of range"); } |

**readi函数**

**writei函数**

|  |
| --- |
| C // 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. // Returns the number of bytes successfully written. // If the return value is less than the requested n, // there was an error of some kind. int writei(struct inode \*ip, int user\_src/\*bool\*/, 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){  uint addr = bmap(ip, off/BSIZE);  if(addr == 0)  break;  bp = bread(ip->dev, addr);  // m 在内部变更  m = min(n - tot, BSIZE - off%BSIZE);  // 如果user\_src 为 0， 表示 src 为内核地址，内部调用了memmove  // 如果src\_src 为1， 则表示src为用户空间，就需要从用户空间往内核拷贝数据，调用 copyin  if(either\_copyin(bp->data + (off % BSIZE), user\_src, src, m) == -1) {  brelse(bp);  break;  }  log\_write(bp);  brelse(bp);  }   if(off > ip->size)  ip->size = off;   // write the i-node back to disk even if the size didn't change  // because the loop above might have called bmap() and added a new  // block to ip->addrs[].  iupdate(ip);   return tot; } |

**balloc 函数**

|  |
| --- |
| 下面的 balloc 函数和 bfree 函数，都必须使用在一个事务（***transaction***） 中 |

BPB 表示 Bits-Per-Block, 即一个block里面，有多少个bits（1024 \* 8）

主要用在bmap 函数中，这个函数的返回值，表示第 b + bi，表示block\_num\_id，后面bmap函数会通过这个返回值，来调用 bread 函数，并获取到指定扇区的数据

|  |
| --- |
| C // Allocate a zeroed disk block. // returns 0 if out of disk space. static uint balloc(uint dev) {  int b, bi, m;  struct buf \*bp;   bp = 0;  for(b = 0; b < sb.size; b += BPB){  // 这一步，获取的是bit-map的所在block块。是从上面总的 sb.size 中按照 一个BPB来划分的步长  bp = bread(dev, BBLOCK(b, sb));    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){  // 向左移动一位，是为了让bites位和实际的计数可以映射上  // 比如15，按照 bi % 8, 会放在第二个bytes的倒数第一位上（从0开始奇数），  // 实际需要的是让他放在倒数第二位上，因为后面还需要放 16  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); // 如果本轮没有找到空的block， 则清理掉这个bp 缓存  }  printf("balloc: out of blocks\n");  return 0; } |

**bfree 函数**

清理一个在bit-map中的block, 和 balloc 是逆操作

**ialloc 函数**

里面用到了 iget 函数

1. dinode 结构是保存在哪里？

整理下 dinode 结构如何保存在 bp->data 中：

目前看，所有的buf都是以数组的形式来排列的，然后所有的dinode结构也都是以数组的形式保存在buf->data中， 可以用inum来定位到buf, 进而定位到在buf的哪个地方，也就是下面宏定义展开所描述的。

而具体一开始如何排放的，需要看 mkfs里面的内容

|  |
| --- |
| C // Allocate an inode on device dev. // Mark it as allocated by giving it type type. // Returns an unlocked but allocated and referenced inode, // or NULL if there is no free inode. struct inode\* ialloc(uint dev, short type) {  int inum;  struct buf \*bp;  struct dinode \*dip;   for(inum = 1; inum < sb.ninodes; inum++){  // 宏展开  bp = bread(dev, ((inum) / (1024 / sizeof(struct dinode)) + sb.inodestart));  // bp = bread(dev, IBLOCK(inum, sb));  // 宏展开  dip = (struct dinode\*)bp->data + inum%(1024 / sizeof(struct dinode))  //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);  }  printf("ialloc: no inodes\n");  return 0; } |

**iput 函数**

**iget 函数**

iget 函数，仅仅从 **itable** 返回一个内存节点的指针，没有进行磁盘操作。

如果itable 中有对应 inum 节点的inode, 那么增加引用，返回，否则的话，占用一个空的inode，然后返回。

|  |
| --- |
| JSON 使用的时候，基本上要沿用如下的顺序，比如在 sysfile.c 的 create 函数中  Thus a typical sequence is: // 要留意，这里的设计，iget和ilock是分开的，因为里面分别有一个锁的存在 // ip = iget(dev, inum) // ilock(ip) // ... examine and modify ip->xxx ... // iunlock(ip) // iput(ip) |

**四：Directory**

**dirlookup 函数**

|  |
| --- |
| 疑问： |

1. 函数内部，是如何通过 dp的inode指针，以及一个文件名，来找到这个文件对应的信息，比如inode，还有off?

|  |
| --- |
| Plain Text inode 函数是通过iget 来返回的，也就是先通过name，找到对应的dirent结构信息，然后才能获取里面保存到的inum信息，最后再将实际的inode指针返回回来，off就是 在buff中寻找dirent数组中的对应元素 |

2. dirent 目录结构是保存在哪里的？缺少一个整体的布局

保存在最底层磁盘中的buf的 data 结构中

3. Inode 结构是保存在哪里？

放在一个全局的itable列表中，保存在内存中

|  |
| --- |
| C // Look for a directory entry in a directory. // If found, set \*poff to byte offset of entry. struct inode\* dirlookup(struct inode \*dp, char \*name, uint \*poff) {  uint off, inum;  struct dirent de;   // 必须是DIR才可以，因为下面会按照目录的方式来遍历磁盘中的数据  if(dp->type != T\_DIR)  panic("dirlookup not DIR");   // 根据里面的遍历方式，我可以推测，buf里面的dirent 结构是以数组的形式排列的，这样才可以  // 按顺序递增  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;  // 所有获取 inode指针的方式，都是要通过inum 的方式才可以  return iget(dp->dev, inum);  }  }   return 0; } |

**五：PathName**

|  |
| --- |
| directory entry ！= directory  Directory entry 也有可能是一个普通的文件名，也有可能是是一个目录 |

**nameiparent 函数**

作用： 返回 path 的最后一个路径元素的 parent 的inode. 然后将最后一个path元素填充到name中。

**namei 函数**

作用： 返回 path 的元素所在的inode节点

**namex 函数**

作用：如果 nameiparent != 0, 那么 就是nameiparent 函数逻辑

内部循环会停止一级返回，返回的inode是

如果 nameiparent == 0, 那么就是 namei 函数逻辑， name不会返回任何东西

|  |
| --- |
| C // 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;   // 从cwd 获取inode, cwd 表示当前目录，cwd的更新在于：chdir, 简称 **cd**  // 注意： ip会在下面被next更新！  if(\*path == '/')  ip = iget(ROOTDEV, ROOTINO);  else  ip = idup(myproc()->cwd);   while((path = skipelem(path, name)) != 0){ // Copy the next path element from path into name.  ilock(ip);  if(ip->type != T\_DIR){  iunlockput(ip);  return 0;  }  if(nameiparent && \*path == '\0'){ // 这里是nameiparent 正常返回的位置  // Stop one level early.  iunlock(ip);  return ip;  }  if((next = dirlookup(ip, name, 0)) == 0){ // 在当前目录中寻找  iunlockput(ip);  return 0;  }  iunlockput(ip);  ip = next;  }  if(nameiparent){  iput(ip);  return 0;  }  return ip; } |

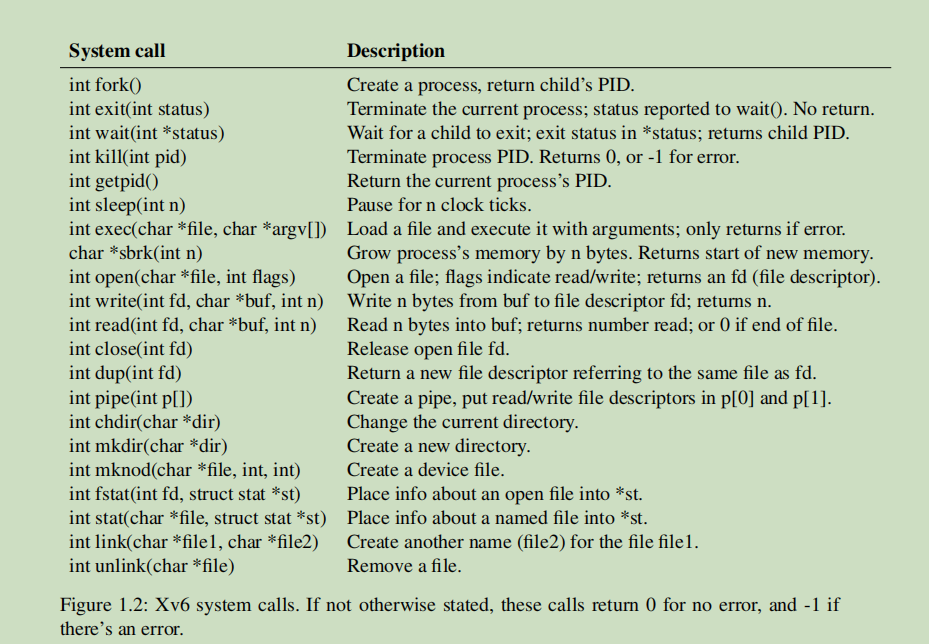
**六：File descriptor**

**filestat 函数**

|  |
| --- |
| C // Get metadata about file f. // addr is a user virtual address, pointing to a struct stat. int filestat(struct file \*f, uint64 addr) {  struct proc \*p = myproc();  struct stat st;    if(f->type == FD\_INODE || f->type == FD\_DEVICE){  ilock(f->ip);  stati(f->ip, &st);  iunlock(f->ip);  // Copy from kernel to user.  // Copy len bytes from src to virtual address dstva in a given page table.  // Return 0 on success, -1 on error.  //// int copyout(pagetable\_t pagetable, uint64 dstva, char \*src, uint64 len)  if(copyout(p->pagetable, addr, (char \*)&st, sizeof(st)) < 0)  return -1;  return 0;  }  return -1; } |

**七：system call**

**列表：**



|  |
| --- |
| C // System call numbers #define SYS\_fork 1 #define SYS\_exit 2 #define SYS\_wait 3 #define SYS\_pipe 4 #define SYS\_read 5 #define SYS\_kill 6 #define SYS\_exec 7 #define SYS\_fstat 8 #define SYS\_chdir 9 #define SYS\_dup 10 #define SYS\_getpid 11 #define SYS\_sbrk 12 #define SYS\_sleep 13 #define SYS\_uptime 14 #define SYS\_open 15 #define SYS\_write 16 #define SYS\_mknod 17 #define SYS\_unlink 18 #define SYS\_link 19 #define SYS\_mkdir 20 #define SYS\_close 21 |

**sys\_link 函数**

它的第一个参数（旧文件名），第二个参数（新文件名）

|  |
| --- |
| C // Create the path new as a link to the same inode as old.( 只能在同一个device上创建 link? ok! dp->dev != ip->dev) uint64 sys\_link(void) {  char name[DIRSIZ], new[MAXPATH], old[MAXPATH];  struct inode \*dp, \*ip;   if(argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)  return -1;   begin\_op();  if((ip = namei(old)) == 0){  end\_op();  return -1;  }   ilock(ip);  if(ip->type == T\_DIR){  iunlockput(ip);  end\_op();  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){ // new directory entry 指向 旧的inode(ip->inum)  iunlockput(dp);  goto bad;  }  iunlockput(dp);  iput(ip);   end\_op();   return 0;  bad:  ilock(ip);  ip->nlink--;  iupdate(ip);  iunlockput(ip);  end\_op();  return -1; } |

**sys\_unlink 函数**

|  |
| --- |
| C uint64 sys\_unlink(void) {  struct inode \*ip, \*dp;  struct dirent de;  char name[DIRSIZ], path[MAXPATH];  uint off;   if(argstr(0, path, MAXPATH) < 0)  return -1;   begin\_op();  if((dp = nameiparent(path, name)) == 0){ // dp -> 父目录  end\_op();  return -1;  }   ilock(dp);   // Cannot unlink "." or "..".  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)  goto bad;   // **获取到的off信息很关键！**  //   if((ip = dirlookup(dp, name, &off)) == 0) // 寻找name对应的inode(ip)  goto bad;  ilock(ip);   if(ip->nlink < 1)  panic("unlink: nlink < 1");  if(ip->type == T\_DIR && !isdirempty(ip)){ // 如果最终是目录，且不为空，是不能unlink的  iunlockput(ip);  goto bad;  }   memset(&de, 0, sizeof(de));  // 写入的是什么？  // 只是将一个空的de结构体，填充到 父目录 dp指针所对应的 buf->data中，表示删除对应的信息  if(writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))  panic("unlink: writei");  if(ip->type == T\_DIR){  dp->nlink--; // ? 是unlink的目录中的 ..  iupdate(dp);  }  iunlockput(dp);   ip->nlink--;  iupdate(ip);  iunlockput(ip);   end\_op();   return 0;  bad:  iunlockput(dp);  end\_op();  return -1; } |

**create 函数**

|  |
| --- |
| C static struct inode\* create(char \*path, short type, short major, short minor) {  struct inode \*ip, \*dp;  char name[DIRSIZ];   if((dp = nameiparent(path, name)) == 0)  return 0;   ilock(dp);   if((ip = dirlookup(dp, name, 0)) != 0){  iunlockput(dp);  ilock(ip);  if(type == T\_FILE && (ip->type == T\_FILE || ip->type == T\_DEVICE))  return ip;  iunlockput(ip);  return 0;  }   if((ip = ialloc(dp->dev, type)) == 0){  iunlockput(dp);  return 0;  }   // ip 是新创建的inode节点，所以这里加锁之后，后面都没有进行iunlockput的解锁操作？  // 是否会造成死锁？？  // 目前看是不会的，因为在返回之后，会有地方进行解锁操作  ilock(ip);   ip->major = major;  ip->minor = minor;  ip->nlink = 1;  iupdate(ip);   if(type == T\_DIR){ // Create . and .. entries.  // No ip->nlink++ for ".": avoid cyclic ref count.  if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)  goto fail;  }   if(dirlink(dp, name, ip->inum) < 0)  goto fail;   if(type == T\_DIR){  // now that success is guaranteed:  dp->nlink++; // for ".."  iupdate(dp);  }   iunlockput(dp);   return ip;   fail:  // something went wrong. de-allocate ip.  ip->nlink = 0;  iupdate(ip);  iunlockput(ip);  iunlockput(dp);  return 0; } |

**八：mkfs**

一开始的文件系统，都是在这部分完成的：

|  |
| --- |
| C // mkfs computes the super block and builds an initial file system. The // super block describes the disk layout: struct superblock {  uint magic; // Must be FSMAGIC  uint size; // Size of file system image (blocks) 2000  uint nblocks; // Number of data blocks 1954  // ninodes 的总数是固定的，后面暂时都不会修改（**限制的地方！**） 200  uint ninodes; // Number of inodes. 会提前记录是有inodes的总数  uint nlog; // Number of log blocks 30  uint logstart; // Block number of first log block 2  uint inodestart; // Block number of first inode block 32  uint bmapstart; // Block number of first free map block 45 }; |

|  |
| --- |
| Plain Text  Disk layout:  total: 2000  [ boot block | **super block** | log | inode blocks | free bit map | data blocks ]    1 1 30 13 1 1954 |

|  |
| --- |
| 虽然data\_blocks区域的节点索引，从1954开始的，不过 inum的计数，是从1开始的 |

**iappend 函数**

填充数据到data block区域

|  |
| --- |
| **文件大小不超过 256kb** |

|  |
| --- |
| C // 向inum对应的实际data\_block中按data\_block的索引顺序写入数据，这个数据目前都是目录dirent信息， // 或者是user里面工具程序的二进制文件信息 void iappend(uint inum, void \*xp, int n) {  char \*p = (char\*)xp;  uint fbn, off, n1;  struct dinode din;  char buf[BSIZE];  uint indirect[NINDIRECT]; // 256  uint x;   rinode(inum, &din);  off = xint(din.size);  // printf("append inum %d at off %d sz %d\n", inum, off, n);  while(n > 0){  fbn = off / BSIZE; // 映射到dinode 里面addrs的数组索引  assert(fbn < MAXFILE); // **文件大小不超过 256kb**  if(fbn < NDIRECT){  if(xint(din.addrs[fbn]) == 0){  din.addrs[fbn] = xint(freeblock++);  }  x = xint(din.addrs[fbn]); // 可以相关转化  } else {  if(xint(din.addrs[NDIRECT]) == 0){  din.addrs[NDIRECT] = xint(freeblock++);  }  rsect(xint(din.addrs[NDIRECT]), (char\*)indirect);  if(indirect[fbn - NDIRECT] == 0){  indirect[fbn - NDIRECT] = xint(freeblock++);  wsect(xint(din.addrs[NDIRECT]), (char\*)indirect);  }  x = xint(indirect[fbn-NDIRECT]);  }  n1 = min(n, (fbn + 1) \* BSIZE - off);  rsect(x, buf);  bcopy(p, buf + off - (fbn \* BSIZE), n1); // void bcopy(const void \*src, void \*dest, size\_t n);  wsect(x, buf);  n -= n1;  off += n1;  p += n1;  }  din.size = xint(off); // 更新了文件大小后，再写回去  winode(inum, &din); } |

执行逻辑：

**填充 data\_blocks：**

一开始先创建一个root的节点，inode: 为1，类型为目录，表示一个根目录，后面的一些工具程序文件名都会添加到后面，以dinode 这种结构

同时，这些二进制的程序文件，如 rm, mkdir, cat 等，也会添加到data blocks区域，每一个文件都会申请一个inode, 从2开始，类型为

文件。

最后，会对 user目录的数据做一次block的对齐，然后将已经使用的data blocks 信息同步到bitmap中。