实例分析

舒波文
李永康
张裕

Fileread & Filewrite 三个参数

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

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

1.文件描述符（代表一个文件）

2.Addr表示用于保存从file文件中读取的数据的内存地址

3.读取/写入的数据大小

F-type的判断

```
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;
};
```

F->type 的作用是用于表示文件类型

FD_NONE

FD_PIPE 管道类型：管道是一个小的内核缓冲区，以文件描述符对的形式提供给进程

FD_INODE inode类型

文件的权限管理如何实现？

```c
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;
};
```

File结构体中存在名为readable和writeable的域，分别表示可读和可写。

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

  if(f->readable == 0)
    return -1;
```

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

  if(f->writable == 0)
    return -1;
```

在fileread和filewrite函数中通过判断readable和writeable是否为0来实现读和写的权限管理

F->off变量代表什么？如何变化

- F->off变量代表相当于文件头的偏移量

```
if((r = readi(f->ip, addr, f->off, n)) > 0)
  f->off += r;
```

Readi成功后，使得off增加r，r表示的是从文件中读取数据的长度。

Readi&writei函数的参数ip类型是什么？包含的成员变量都是什么含义

```c
readi(struct inode *ip, char *dst, uint off, uint n)
```

```c
writei(struct inode *ip, char *src, uint off, uint n)
```

```c
// 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];
};
```

Inode表示内核中的inode

相应的存在磁盘中的disk inode

Inode成员变量

```c
// 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];
};
```

```c
// On-disk inode structure
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
};
```

1.Dev 设备号
2.Inum inode号
3.Ref 引用该inode的数量
4.Lock 锁，用于保护该inode
5.Valid 是否有效（从磁盘中读取）
//与disk inode中的成员变量相同
6.Type 文件类型
7.Major 最大设备数目
8.Minor 最小设备数目
9.Nlink 文件系统中该inode的链接数量
10. Size 文件大小
11. Addrs 数据块的地址

Readi函数中文件操作实际执行到的内容

```c
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;
}
```

```c
#define NDEV         10  // maximum major device number
```

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

处理类型为device类型的inode
如果出错，则返回-1
如果正确读出，返回devsw中读取的read值

当偏移量大于size或者n小于0时报错

当要读取的数据大小和偏移量的和大于size时重置n

从磁盘中读数据，存储到dst位置的内存中

```c
#define BSIZE 512  // block size
```

```c
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;
}
```

Log_write写日志

更新内存中的inode（更新size）
更新磁盘中的inode（将内存的inode内容更新到磁盘中对应的inode）

Bget函数的功能

```c
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");
}
```

```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;
```

```c
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];
};
```

功能：输入设备号，块号。
先查找整个bcache，如果该block已经有cache与其对应，则返回对应的cache
否则找到一块未被使用的cache返回。

结构体bcache用于管理所有的cache块，通过next和prev指针形成了环状的结构。Head.next代表最近常用的cache块。
算法：
使用的算法为LRU
通过head.prev和prev指针来寻找最近最不常用的cache块。

Buffer cache中的dirty块是由哪个函数写入磁盘的？

```c
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");
}
```

```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;
}
```

```c
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!holdingsleep(&b->lock))
    panic("iderw: buf not locked");
  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);
}
```

linux 内核代码中 buffer cache 的替换是如何实现的？替换算法和 xv6 有何不同

Linux内核中文件Cache替换的具体过程是这样的：刚刚分配的Cache项链入到active_list头部，并将其状态设置为active，当内存不够需要回收Cache时，系统首先从尾部开始反向扫描active_list并将状态不是referenced的项链入到inactive_list的头部，然后系统反向扫描inactive_list，如果所扫描的项的处于合适的状态就回收该项，直到回收了足够数目的Cache项。

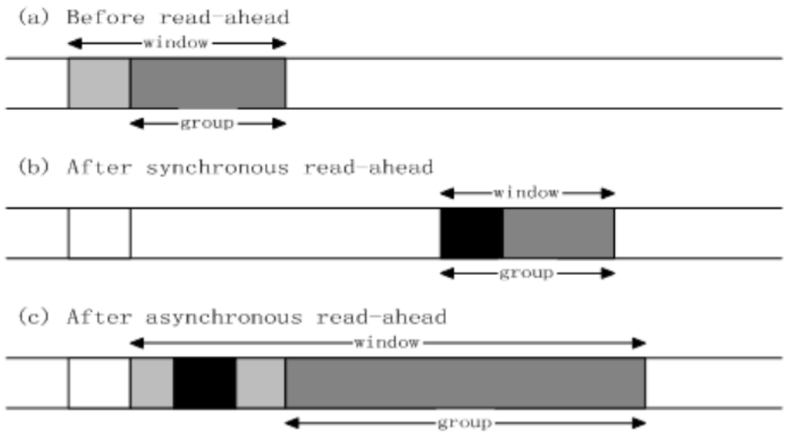

图 5　Linux 文件预读机制示意图

Linux中存在两个队列。替换算法也为LRU
与xv6不同的地方在于
Xv6为环状的形式，通过反向查找的方式来实现替换最长时间没有使用的页