日曜研究室

技術的な観点から日常を綴ります

[xv6 #62] Chapter 5 – File system – Code: Inodes

テキストの70〜71ページ

本文

新しいinodeを割り当てるために（例えば、ファイルを作るときなど）、xv6はialloc関数を呼ぶ。
ialloc関数は、balloc関数に似ていて、ディスク上のinodeの構造体群を走査し、一度に一つのブロックについて、そのブロックが「空き」とマークされているかどうか調べる。
そのようなブロックが見つかったら、新しいtype変数の内容をディスクに書き込み、それからreturn文で呼ばれるiget関数の呼び出しによって、inodeキャッシュから一つのエントリを返す。
balloc関数での処理のように、ialloc関数は正しく処理を行うために、変数bpに対する参照を保持できるのは一度に一つのプロセスだけという事実に依存している。
ialloc関数は、その取得可能なinodeを他のプロセスが同時に参照してない事を保証でき、そしてその取得可能なinodeの要求を試みる。

fs.cのialloc, iget関数

```
01 // Allocate a new inode with the given type on device dev.
02 struct inode*
03 ialloc(uint dev, short type)
04 {
05   int inum;
06   struct buf *bp;
07   struct dinode *dip;
08   struct superblock sb;
09
10   readsb(dev, &sb);
11   for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode
   blocks
12     bp = bread(dev, IBLOCK(inum));
13     dip = (struct dinode*)bp->data + inum%IPB;
14     if(dip->type == 0){  // a free inode
15       memset(dip, 0, sizeof(*dip));
16       dip->type = type;
17       log_write(bp);   // mark it allocated on the disk
18       brelse(bp);
```

```
19        return iget(dev, inum);
20      }
21      brelse(bp);
22    }
23    panic("ialloc: no inodes");
24  }
25
26  // Find the inode with number inum on device dev
27  // and return the in-memory copy.
28  static struct inode*
29  iget(uint dev, uint inum)
30  {
31    struct inode *ip, *empty;
32
33    acquire(&icache.lock);
34
35    // Try for cached inode.
36    empty = 0;
37    for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
38      if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
39        ip->ref++;
40        release(&icache.lock);
41        return ip;
42      }
43      if(empty == 0 && ip->ref == 0)    // Remember empty slot.
44        empty = ip;
45    }
46
47    // Allocate fresh inode.
48    if(empty == 0)
49      panic("iget: no inodes");
50
51    ip = empty;
52    ip->dev = dev;
53    ip->inum = inum;
54    ip->ref = 1;
55    ip->flags = 0;
56    release(&icache.lock);
57
58    return ip;
59  }
```

iget関数は、渡されたデバイス番号とinode番号を元に、アクティブなエントリ（ip->ref > 0）を、inodeキャッシュの中を走査して探す。
そのようなエントリが見つかったら、そのinodeに対する新しい参照を返す。
そのiget関数の走査によって、最初の空きスロットの位置を記録し（Remember empty slot.というコメントの部分）、その空きスロットは新しいキャッシュエントリを割り当てる必要がある場合に使われる。
どちらの場合も（inodeキャッシュから見つかった場合とそうでない場合）、iget関数は、呼び出し元に一つの参照を返す。
呼び出し側には、iput関数世を呼び、そのinodeを解放する責任がある。
これは、いくつかのプロセスが、複数回のiput関数の呼び出しを配置する場合に便利である。
idup関数は、inodeの参照カウントを増加させるので、そのinodeのキャッシュをやめさせる前に、追加のiput関数の呼び出しが必要である。

fs.cのidup関数

```
01 // Increment reference count for ip.
02 // Returns ip to enable ip = idup(ip1) idiom.
03 struct inode*
04 idup(struct inode *ip)
05 {
06   acquire(&icache.lock);
07   ip->ref++;
08   release(&icache.lock);
09   return ip;
10 }
```

呼び出し側は、inodeのメタデータや内容を読み書きする前に、ilock関数を使ってそのinodeをロックしなければならない。
ilock関数は、今はおなじみのsleepループを使って、ip->flagのI_BUSYビットがクリアされるのを待ち、そしたらip->flagにI_BUSYビットをセットする。
一度ilock感菅、inodeに対する排他的なアクセス権を得たら、必要に応じてディスクから（バッファキャッシュからの可能性も高い）そのinodeのメタデータを読み込むことが出来る。
iunlock関数は、I_BUSYビットをクリアし、ilockでスリープしている他のプロセスのいずれかを起こす。

fs.cのilock, iunlock関数

```
01 // Lock the given inode.
02 void
03 ilock(struct inode *ip)
04 {
05   struct buf *bp;
06   struct dinode *dip;
07
08   if(ip == 0 || ip->ref < 1)
09     panic("ilock");
10
11   acquire(&icache.lock);
12   while(ip->flags & I_BUSY)
13     sleep(ip, &icache.lock);
14   ip->flags |= I_BUSY;
15   release(&icache.lock);
16
17   if(!(ip->flags & I_VALID)){
18     bp = bread(ip->dev, IBLOCK(ip->inum));
19     dip = (struct dinode*)bp->data + ip->inum%IPB;
20     ip->type = dip->type;
21     ip->major = dip->major;
22     ip->minor = dip->minor;
23     ip->nlink = dip->nlink;
24     ip->size = dip->size;
25     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
26     brelse(bp);
27     ip->flags |= I_VALID;
28     if(ip->type == 0)
29       panic("ilock: no type");
30   }
31 }
32
33 // Unlock the given inode.
34 void
35 iunlock(struct inode *ip)
36 {
```

```
37    if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
38      panic("iunlock");
39
40    acquire(&icache.lock);
41    ip->flags &= ~I_BUSY;
42    wakeup(ip);
43    release(&icache.lock);
44  }
```

iput関数は、参照カウントをデクリメントする事によって、あるinodeに対するCポインタを解放する。
もし最後の参照だった場合、inodeキャッシュにおけるそのinodeのスロットは空きとなり、他のinodeによって再利用可能となる。

fs.cのiput関数

```
01  // Caller holds reference to unlocked ip.  Drop reference.
02  void
03  iput(struct inode *ip)
04  {
05    acquire(&icache.lock);
06    if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
07      // inode is no longer used: truncate and free inode.
08      if(ip->flags & I_BUSY)
09        panic("iput busy");
10      ip->flags |= I_BUSY;
11      release(&icache.lock);
12      itrunc(ip);
13      ip->type = 0;
14      iupdate(ip);
15      acquire(&icache.lock);
16      ip->flags = 0;
17      wakeup(ip);
18    }
19    ip->ref--;
20    release(&icache.lock);
21  }
```

もしinodeに対するCポインタが一つも無く、リンクも持たない（どのディレクトリにも属さない）ということをiput関数が発見した場合、そのinodeとそのデータブロックは解放されなければならない。
iput関数は、そのinodeを再ロックし（ip->flags |= I_BUSY;）、ファイルを0バイトに切り詰めるためにitrunc関数を呼び、データブロックを解放し、そのinodeのタイプを0（未割り当て）にセットし、この変更をディスクに書き込み、そして最後にそのinodeのロックを解放（ip->flags = 0;）する。（この一連の処理は、iput関数のif文のブロックの中の話。）

iput関数におけるロック手法は注目に値する。
価値ある例のまず一つめの部分は、ipをロックするとき、iput関数は、sleepループを使わずに、単純にそれがロックされてないだろうと仮定する。
これは、呼び出し側がiput関数を呼び出す前にipをロックしてない事を要求され、そして呼び出し側がそれに対する唯一の参照を持つ（ip->ref == 1）場合でなければならない。
価値ある例の二つめの部分は、iput関数が一時的に解放し（ifブロック中のrelease(&icache.lock);のところ）、再度キャッシュのロックを獲得する（ifブロック中のacquire(&icache.lock);のところ）事で

ある。
これは、itrunc関数とiupdate関数がディスクI/O待ちでスリープする可能性があるので、必要とされるのだが、我々は、ロックを保持してない間に何が起きうるかについて熟考しなければならない。
具体的に言うと、一度iupdate関数が完了したら、ディスク上の構造体は、利用可能である（空きである）とマークされ、そしてiput関数が終わる前に、同時実行されるiallocの呼び出しが、それを見つけ再割当てしてしまうかもしれない。
ialloc関数で、iget関数が呼び出され、iget関数がキャッシュの中からipを見つけ、最終的にialloc関数はそのブロックへの参照を返す可能性がある。
その場合、そのinodeはI_BUSYフラグがセットされている状態であり、ialloc関数の呼び出し側がそのinodeを読み書きしようとしてilock関数を呼び出すと、スリープする。
そうなるとメモリ上のinodeがディスク上のそれと比べて一致してない状態になる。
ialloc関数は、ディスクの側を再初期化したが、ilock関数の間に呼び出し側がそれをメモリに読み込むことを当てにしている。
これが起きることを確実にするために、iput関数は、inodeのロックを解放する前に、I_BUSYフラグだけでなくI_VALIDフラグもクリアしなければならない。
これは、flagsに0をセットすることで行われる。（iput関数のifブロック中のip->flags = 0;のところ）

感想

inodeの管理についてのコードの説明です。
ちょっと最後の段落が難しいです。
重要なのは多分、iputにおける消去処理中のinodeを、偶然他のプロセスがiallocによってされてしまった場合、そのinodeを何も気づかず使ってしまうのを防ぐために、最後にI_VALIDもクリアしてる部分だと思います。
他のプロセスがすでにilockまで到達しsleep中かもしれないので、さらに直後にwakeupを呼ぶことも何気に重要かもしれません。

今まではなるべくソースはそのまま掲載するように、部分的に掲載する場合も関数の順番はなるべく元のままにするようにしてましたが、ちょっと今回は参照する関数が多い上に、本文と元のソースの関数の順番が違いすぎるので、本文に合わせて関数を抜き出しつつ順番を変えて掲載しています。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/4/10 火曜日 [http://peta.okechan.net/blog/archives/1634] |