日曜研究室

技術的な観点から日常を綴ります

[xv6 #56] Chapter 5 – File system – Code: Buffer cache

テキストの65〜66ページ

本文

バッファキャッシュは、バッファの2重リンクリスト（次の要素だけでなく前の要素へのリンクも持つリスト）である。
binit関数は、main関数から呼ばれ、buf構造体を利用したNBUF個のバッファ（固定長配列）を初期化する。
この他のすべてのバッファキャッシュへアクセスは、bufの配列へ直接ではなく、bcache.head経由で行われる。

bio.c

```
001 // Buffer cache.
002 //
003 // The buffer cache is a linked list of buf structures holding
004 // cached copies of disk block contents.  Caching disk blocks
005 // in memory reduces the number of disk reads and also provides
006 // a synchronization point for disk blocks used by multiple
    processes.
007 //
008 // Interface:
009 // * To get a buffer for a particular disk block, call bread.
010 // * After changing buffer data, call bwrite to flush it to disk.
011 // * When done with the buffer, call brelse.
012 // * Do not use the buffer after calling brelse.
013 // * Only one process at a time can use a buffer,
014 //     so do not keep them longer than necessary.
015 //
016 // The implementation uses three state flags internally:
017 // * B_BUSY: the block has been returned from bread
018 //     and has not been passed back to brelse.
019 // * B_VALID: the buffer data has been initialized
020 //     with the associated disk block contents.
021 // * B_DIRTY: the buffer data has been modified
022 //     and needs to be written to disk.
023
```

```
024 #include "types.h"
025 #include "defs.h"
026 #include "param.h"
027 #include "spinlock.h"
028 #include "buf.h"
029
030 struct {
031   struct spinlock lock;
032   struct buf buf[NBUF];
033
034   // Linked list of all buffers, through prev/next.
035   // head.next is most recently used.
036   struct buf head;
037 } bcache;
038
039 void
040 binit(void)
041 {
042   struct buf *b;
043
044   initlock(&bcache.lock, "bcache");
045
046 //PAGEBREAK!
047   // Create linked list of buffers
048   bcache.head.prev = &bcache.head;
049   bcache.head.next = &bcache.head;
050   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
051     b->next = bcache.head.next;
052     b->prev = &bcache.head;
053     b->dev = -1;
054     bcache.head.next->prev = b;
055     bcache.head.next = b;
056   }
057 }
058
059 // Look through buffer cache for sector on device dev.
060 // If not found, allocate fresh block.
061 // In either case, return locked buffer.
062 static struct buf*
063 bget(uint dev, uint sector)
064 {
065   struct buf *b;
066
067   acquire(&bcache.lock);
068
069  loop:
070   // Try for cached block.
071   for(b = bcache.head.next; b != &bcache.head; b = b->next){
072     if(b->dev == dev && b->sector == sector){
073       if(!(b->flags & B_BUSY)){
074         b->flags |= B_BUSY;
075         release(&bcache.lock);
076         return b;
077       }
078       sleep(b, &bcache.lock);
079       goto loop;
080     }
081   }
082
083   // Allocate fresh block.
084   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
085     if((b->flags & B_BUSY) == 0){
086       b->dev = dev;
087       b->sector = sector;
```

```
088       b->flags = B_BUSY;
089       release(&bcache.lock);
090       return b;
091     }
092   }
093   panic("bget: no buffers");
094 }
095
096 // Return a B_BUSY buf with the contents of the indicated disk
    sector.
097 struct buf*
098 bread(uint dev, uint sector)
099 {
100   struct buf *b;
101
102   b = bget(dev, sector);
103   if(!(b->flags & B_VALID))
104     iderw(b);
105   return b;
106 }
107
108 // Write b's contents to disk.  Must be locked.
109 void
110 bwrite(struct buf *b)
111 {
112   if((b->flags & B_BUSY) == 0)
113     panic("bwrite");
114   b->flags |= B_DIRTY;
115   iderw(b);
116 }
117
118 // Release the buffer b.
119 void
120 brelse(struct buf *b)
121 {
122   if((b->flags & B_BUSY) == 0)
123     panic("brelse");
124
125   acquire(&bcache.lock);
126
127   b->next->prev = b->prev;
128   b->prev->next = b->next;
129   b->next = bcache.head.next;
130   b->prev = &bcache.head;
131   bcache.head.next->prev = b;
132   bcache.head.next = b;
133
134   b->flags &= ~B_BUSY;
135   wakeup(b);
136
137   release(&bcache.lock);
138 }
```

バッファは3つの状態フラグを持つ。
B_VALIDは、そのブロックの正しいコピーをそのバッファが含むことを示す。
B_DIRTYは、バッファの内容が変更され、ディスクへ書き出さなければならないことを示す。
B_BUSYは、どこかのカーネルスレッドがそのバッファを参照していて、まだ解放してないことを示す。

bread関数は、与えられたセクタのためのバッファを得るためにbget関数を呼ぶ。

そのバッファの内容をディスクから読み込む必要がある場合、breadはそのバッファを返す前に、iderw関数を呼ぶ。

bget関数は、与えられたデバイスとセクタに対応するバッファを捜すためにバッファのリストを走査する。
合致するバッファがあり、かつそのバッファがB_BUSYではない場合、bgetはそのバッファにB_BUSYフラグをセットし返す。
そのバッファが利用中の場合、bgetはそのバッファが解放されるのを待つために、そのバッファをチャンネルとしてsleepする。
sleepが返ってきたときでも、bgetはそのバッファが利用可能であると仮定することは出来ない。
sleepは、bcache.lockを解放し再獲得するので、そのバッファbがまだ利用可能であるという保証はない。
そのバッファは、別のセクタのために再利用されているかもしれない。
bgetは、今回とは違った結末になるであろう事を希望して、やり直すしかない。（goto loop;のところ）

（この段落の意味が分からなかったので直訳風味になってます。詳細は感想に書きます。）
bgetに、もしgoto文がなかったとしたら、図5-3のような競合状態が起きる可能性がある。
最初のプロセスが、あるバッファを持ち、そしてその中にセクタ3を読み込みこんだ。
今2つの他のプロセスがついてくる。
最初のそれは、バッファ3のために取得を行い、そしてキャッシュされたブロックのためのループの中でスリープする。
2番目のそれは、バッファ4のために取得を行い、新たに割り当てられたブロックのためのループの中以外の同じバッファ上でスリープ出来た。
なぜなら、空きバッファがなく、そして3を保持しているバッファが、リストの最前面であり、再利用のために選択されたから。
最初のプロセスがそのバッファを開放し、wakeupはプロセス3をまずスケジュールする事態を引き起こし、それはバッファをつかみ、それにセクタ4を読み込むだろう。
それが完了するとき、それはそのバッファ（セクタ4を含む）を解放し、プロセス2を起こすだろう。
goto文なしのプロセス2は、そのバッファをBUSYとマークし、そしてbgetから戻る。
しかしそのバッファはセクタ3ではなく、セクタ4を含む。
セクタ3とセクタ4は違う内容を持つので、このエラーは様々な大破壊を起こす可能性がある。
xv6はそれらをinodeの保管に使っている。

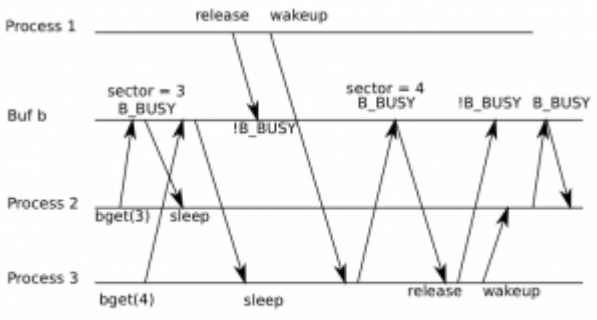

図5-3 process 2はブロック3について問い合わせたのに、結果的にブロック4を含むバッファを受け取ってしまう競合の例
（原文では、process 3は～となってるけど多分間違い）

与えられたセクタに対するバッファがない場合、bgetはそれを作らなければならず、別のセクタを保持してたバッファを出来る限り再利用する。
bgetは、もう一度バッファのリストを走査し、B_BUSYではないブロック（使えるブロック）を捜す。
bgetは、呼び出し元にバッファを返す前に、新しいデバイス番号とセクタ番号を記録し、利用中であるということをマークするためにブロックのメタデータを編集する。
このフラグへの代入は、B_BUSYフラグを追加するだけではなく、B_VALIDやB_DIRTYフラグもクリアすることに注意。
これによって、breadが以前のブロックの内容を使う事無く、ディスクからそのバッファのデータをリフレッシュすることを保証する。

同期化のためにバッファキャッシュは使われるので、ディスクの各セクタについて、多くてもたった一つのバッファしか存在しないということが重要である。
bgetの最初のループは、そのセクタに対するバッファが既に存在していないと決心しているので、代入（b->dev = dev; b->sector = sector; b->flags = B_BUSY;）は、唯一の安全策であり、そしてbgetはそれ以来bcache.lockを手放していない。（謎）

もしすべてのバッファがB_BUSYな場合、何かおかしな事態になっており、bgetはpanicを呼ぶ。
もっと親切な反応としては、バッファに空きが出来るまでスリープすべきかもしれない。
もっとも、デッドロックの可能性はあるが。

一度breadが、呼び出し側にバッファを返したら、呼び出し側はそのバッファの独占権を持ち、バイトデータを読み書きできる。
呼び出し側が、データを書き込んだ場合、そのバッファを解放する前に、bwriteを呼んで、変更されたデータをディスクに書き込まなければならない。
bwriteは、B_DIRTYフラグをセットし、そしてディスクへバッファの内容を書き込むためにiderwを呼ぶ。

呼び出し側がバッファを使い終わったら、バッファを解放するためにbrelseを呼ぶべきである。
（brelseという名前は、b-releaseの短縮で、隠語ではあるが覚えておく価値がある。Unixを起源とし、BSDやLinuxやSolarisなどで使われている。）
brelseは、そのバッファをリンクリストの元の位置からリストの最前に移動し、B_BUSYフラグをクリアし、そしてそのバッファを待ってスリープしてるどこかのプロセスを起こす。
バッファの移動は、最近使われた（解放されたという意味で）順番にバッファを並び替える効果があ

る。
リストの最初のバッファは、一番最近使われたものであり、リストの最後のバッファは、使われてから一番時間が経っているものである。
bgetの2つのループは、この優位性を利用している。
存在しているバッファの走査は、最悪の場合リストのすべてを処理しなければならないが、最近使われたバッファを先に調べる事（bcache.headからはじめてnextポインタをたどる事）は、良い参照の局所性がある場合に、走査に掛かる時間を節約するだろう。
再利用するためのバッファを選ぶための走査は、逆順に走査（prevポインタをたどる）することによって使われてから一番時間が経っているものから調べる。

感想

バッファキャッシュの実装の説明です。

途中意味が分からないと書いた段落について。
goto loop;がない場合に発生する競合状態の機序についての説明かと思いますが、どう考えても書かれてるような事は起きない気がします。

他のプロセスによってbcache.headの位置が変わる可能性があるので、取りこぼさないよう初めから再ループするためにgoto loop;が必要な事は分かりますが、b->dev == dev && b->sector == sector のチェックはループの中で行ってるので、問題ないというか、そもそもセクタがチェックされてるので、別のセクタを必要とする（図でいうところの）プロセス2とプロセス3が同じバッファを待ってスリープすることはないはずです。

もしプロセス2がプロセス3で利用中のバッファを待ってスリープするということがあり得たとしても、プロセス3でwakeupされてプロセス2のsleepが返ってきた段階で、またb->dev == dev && b->sector == sector のチェックが行われ、それに合致しないので（セクタが違うので）単純にそのループは飛ばされるだけじゃないかと思います。

何か重大な勘違いをしてるんだろうか。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/31 土曜日 [http://peta.okechan.net/blog/archives/1590] |