日曜研究室

技術的な観点から日常を綴ります

[xv6 #42] Chapter 3 – Locking – Exercises

テキストの49〜50ページ

本文

1. acquire関数からxchg命令を取り除きなさい。
そしてxv6を実行したとき何が起きるか説明しなさい。

2. iderw関数のacquireの呼び出しをsleep関数の前に移動しなさい。
競合は存在しますか？
xv6を起動してstressfsプログラムを実行し観察してみよう。
今度は、ダミーのループを使って危険な箇所を増やしてみましょう。
そしたら何が起きますか？説明しなさい。

3. 投稿された宿題をやりなさい。（謎）

4. バッファのflagsフィールドの設定は、アトミックな操作ではない。
プロセッサは、flagsのコピーをレジスタに作り、そのレジスタを変更し、それを書き戻す。
なので、2つのプロセッサが同時にflagsを書き換えないことが重要である。
xv6は、B_BUSYフラグを編集するときだけbuflockを保持するが、B_VALIDとB_WRITEを編集するときはどのロックも使わない。
なぜこれは安全なのだろう？

作業

1. について
spinlock.cのacquire関数のxchg命令を実行してるwhile文ごとコメントアウトしてみました。
release関数内で、ロックが保持されてないのに呼ばれるとpanicするようになってるので、そのとおりになりました。
（これって質問の意図と合ってるのかな）

2. について
ide.cのiderw関数のacquire呼び出しは元々sleep関数の前にありますが、間にキューの操作が入ってるので、その直後と解釈して（stressfsのソースに書いてあるし）、キュー操作の直後（*pp = b;の直後）にacquireを移動してみました。
起動やstressfsの実行程度では問題は表面化しませんでした。
ちなみにstressfsは、4つの子プロセスを数珠つなぎ？に生成しながら、最初の親プロセスも含めた計5つのプロセスで、それぞれ同時に個別のファイルに0〜99を出力するというプログラムです。

次に、acquireの直前のキュー操作の最中（*pp = b;の直前、ループの外）に、ダミーで4万回ループするコード（volatile int i;を使って4万回forループ）を挿入しました。
しかし何回かstressfsしても競合は起きず。
ダミーループの回数を増やしたり、開発環境を実機で動かしたり（いままでVMware上で2コア割り当てたFedoraでコンパイル・実行してた）、QEMUの設定でCPUコア数を8コアぐらいまで増やしたり、コンパイルで生成されたディスクイメージをVMwareに割り当てて直接起動してみたり、stressfsのソースも弄って負荷を上げるようにしたり色々やりましたが、競合は観測できず。

競合が起きると、idequeueの最後の要素が上書きされて失われる場合が出てくるはずで、そうなるとstressfsで出力されるファイルのサイズが普段より少し小さくなるはず、と思ってstressfs後に各ファイルのサイズを調べて競合が起きてるかどうか判断してたんですが、もしかしてそれが悪かったんでしょうか。

なんとしてでも競合を観測したかったんですが、後の章もあるので、この問題は一旦ここで置いておきます。
くやしい。

ide.cのiderw関数（変更前）

```
01 // Sync buf with disk.
02 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
03 // Else if B_VALID is not set, read buf from disk, set B_VALID.
04 void
05 iderw(struct buf *b)
06 {
07   struct buf **pp;
08
09   if(!(b->flags & B_BUSY))
10     panic("iderw: buf not busy");
11   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
12     panic("iderw: nothing to do");
13   if(b->dev != 0 && !havedisk1)
14     panic("iderw: ide disk 1 not present");
15
16   acquire(&idelock);  // DOC:acquire-lock
17
18   // Append b to idequeue.
19   b->qnext = 0;
20   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  // DOC:insert-queue
21     ;
22   *pp = b;
```

```
23
24    // Start disk if necessary.
25    if(idequeue == b)
26      idestart(b);
27
28    // Wait for request to finish.
29    // Assuming will not sleep too long: ignore proc->killed.
30    while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
31      sleep(b, &idelock);
32    }
33
34    release(&idelock);
35  }
```

stressfs.c

```
01  // Demonstrate that moving the "acquire" in iderw after the loop
    that
02  // appends to the idequeue results in a race.
03
04  // For this to work, you should also add a spin within iderw's
05  // idequeue traversal loop.  Spinning 40000 times demonstrated the
    bug
06  // after about 5 runs of stressfs in QEMU on a 2.1GHz CPU.
07
08  #include "types.h"
09  #include "stat.h"
10  #include "user.h"
11  #include "fs.h"
12  #include "fcntl.h"
13
14  int
15  main(int argc, char *argv[])
16  {
17    int fd, i;
18    char path[] = "stressfs0";
19
20    printf(1, "stressfs starting\n");
21
22    for(i = 0; i < 4; i++)
23      if(fork() > 0)
24        break;
25
26    printf(1, "%d\n", i);
27
28    path[8] += i;
29    fd = open(path, O_CREATE | O_RDWR);
30    for(i = 0; i < 100; i++)
31      printf(fd, "%d\n", i);
32    close(fd);
33
34    wait();
35
36    exit();
37  }
```

3. について

謎です。

4. について

本文にB_WRITEとありますが、ソースにそんな定義はないので、他の定義からして多分B_DIRTYの間違いでしょう。
あとbuflockというロック変数もないので、状況からしてbcache.lockのことかと思います。

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 }
```

それぞれのフラグの意味をまとまると次のようになります。

B_BUSY
そのバッファはbreadで読み込まれたが、まだbrelseされてない。
breadはディスクからバッファへの読み込みを行う。
brelseはデータをバッファからキャッシュへ移動する。
bio.cのbget関数（breadから呼ばれる）のみでセットされる。

B_VALID
そのバッファは読み込まれまだ変更されてない。
セットされてるのは、ide.cのideintr関数でのみ。

B_DIRTY
バッファの内容が変更されたので書き込みの必要あり。
セットされてるのは、bio.cのbwrite関数でのみ。
ロックはinodeレベルで行われる。

課題の意図としては、bio.cのbget関数でB_BUSYフラグを立てるときはbcache.lockを使ってるのに、他のフラグを立てるときはbcache.lockを使ってないのはなぜ？ということかなと思います。

まずB_VALIDについては、ide.cのideintr関数でフラグを立てますが、ideintrは割り込み中かつ他のロック（idelock）を獲得済みであり、他でロックが必要な操作がされてない事が明らかなのでロックする必要がない、ってところかなと思います。

B_DIRTYについてはちょと追うのが難しかったのですが、要はinodeのレベルでロックを獲得済みだから必要ないのかなと思います。
まだ出てきてないfs.cを読んだところ、システムコールを使って書き込む場合、超大雑把には

sys_write関数→filewrite関数（この中でinode単位でロックを獲得）→writei関数→bwrite関数
という階層関係で関数が呼ばれます。
バッファを内包するinode単位でロックを獲得済みなら、バッファの操作も他と被ることがないから
改めてバッファ単位でロックを獲得する必要はないのかなと思います。

感想

なんだか不完全燃焼です。
しかしいつまでもここで留まってるわけにも行かないので先に進みますが、ちょくちょく試してみて
とにかく競合状態は確認したいなと思ってます。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/17 土曜日 [http://peta.okechan.net/blog/archives/1535] |