日曜研究室

技術的な観点から日常を綴ります

[xv6 #37] Chapter 3 – Locking – Code: Using locks

テキストの46〜47ページ

本文

xv6は、競合状態を避けるためにロックを利用して慎重にプログラムされている。
単純な例は、IDEドライバである。
この章のはじめで言及したように、iderw関数は、ディスクへのリクエストのキューを持ち、プロセッサは同時にそのリストへ新しいリクエストを追加するだろう。（iderw関数のDOC:insert–queueとコメントが付いてる部分）
このリストとその他のインバリアントを保護するために、iderwはidelockを獲得し（acquireし）、関数の終わりでそれを解放する。
キューの操作の後にacquireの呼び出しを移動する事によって、この章のはじめのほうで見たような競合状態を引き起こす方法を調べるという練習問題1がある。
その練習問題は試す価値がある。
なぜなら、競合を引き起こすことは簡単ではないということがハッキリするからである。
つまりそれは、競合状態を引き起こすバグを探すのは難しいということを意味する。
xv6が、そんないくつかの競合を持つ可能性はなくもない。

ide.c

```
001 // Simple PIO-based (non-DMA) IDE driver code.
002
003 #include "types.h"
004 #include "defs.h"
005 #include "param.h"
006 #include "memlayout.h"
007 #include "mmu.h"
008 #include "proc.h"
009 #include "x86.h"
010 #include "traps.h"
011 #include "spinlock.h"
012 #include "buf.h"
013
```

```
014 #define IDE_BSY       0x80
015 #define IDE_DRDY      0x40
016 #define IDE_DF        0x20
017 #define IDE_ERR       0x01
018
019 #define IDE_CMD_READ  0x20
020 #define IDE_CMD_WRITE 0x30
021
022 // idequeue points to the buf now being read/written to the disk.
023 // idequeue->qnext points to the next buf to be processed.
024 // You must hold idelock while manipulating queue.
025
026 static struct spinlock idelock;
027 static struct buf *idequeue;
028
029 static int havedisk1;
030 static void idestart(struct buf*);
031
032 // Wait for IDE disk to become ready.
033 static int
034 idewait(int checkerr)
035 {
036   int r;
037
038   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
039     ;
040   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
041     return -1;
042   return 0;
043 }
044
045 void
046 ideinit(void)
047 {
048   int i;
049
050   initlock(&idelock, "ide");
051   picenable(IRQ_IDE);
052   ioapicenable(IRQ_IDE, ncpu - 1);
053   idewait(0);
054
055   // Check if disk 1 is present
056   outb(0x1f6, 0xe0 | (1<<4));
057   for(i=0; i<1000; i++){
058     if(inb(0x1f7) != 0){
059       havedisk1 = 1;
060       break;
061     }
062   }
063
064   // Switch back to disk 0.
065   outb(0x1f6, 0xe0 | (0<<4));
066 }
067
068 // Start the request for b.  Caller must hold idelock.
069 static void
070 idestart(struct buf *b)
071 {
072   if(b == 0)
073     panic("idestart");
074
075   idewait(0);
076   outb(0x3f6, 0);  // generate interrupt
077   outb(0x1f2, 1);  // number of sectors
```

```
078   outb(0x1f3, b->sector & 0xff);
079   outb(0x1f4, (b->sector >> 8) & 0xff);
080   outb(0x1f5, (b->sector >> 16) & 0xff);
081   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
082   if(b->flags & B_DIRTY){
083     outb(0x1f7, IDE_CMD_WRITE);
084     outsl(0x1f0, b->data, 512/4);
085   } else {
086     outb(0x1f7, IDE_CMD_READ);
087   }
088 }
089
090 // Interrupt handler.
091 void
092 ideintr(void)
093 {
094   struct buf *b;
095
096   // Take first buffer off queue.
097   acquire(&idelock);
098   if((b = idequeue) == 0){
099     release(&idelock);
100     // cprintf("spurious IDE interrupt\n");
101     return;
102   }
103   idequeue = b->qnext;
104
105   // Read data if needed.
106   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
107     insl(0x1f0, b->data, 512/4);
108
109   // Wake process waiting for this buf.
110   b->flags |= B_VALID;
111   b->flags &= ~B_DIRTY;
112   wakeup(b);
113
114   // Start disk on next buf in queue.
115   if(idequeue != 0)
116     idestart(idequeue);
117
118   release(&idelock);
119 }
120
121 //PAGEBREAK!
122 // Sync buf with disk.
123 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
124 // Else if B_VALID is not set, read buf from disk, set B_VALID.
125 void
126 iderw(struct buf *b)
127 {
128   struct buf **pp;
129
130   if(!(b->flags & B_BUSY))
131     panic("iderw: buf not busy");
132   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
133     panic("iderw: nothing to do");
134   if(b->dev != 0 && !havedisk1)
135     panic("iderw: ide disk 1 not present");
136
137   acquire(&idelock);  // DOC:acquire-lock
138
139   // Append b to idequeue.
140   b->qnext = 0;
141   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  // DOC:insert-queue
```

```
142     ;
143   *pp = b;
144
145   // Start disk if necessary.
146   if(idequeue == b)
147     idestart(b);
148
149   // Wait for request to finish.
150   // Assuming will not sleep too long: ignore proc->killed.
151   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
152     sleep(b, &idelock);
153   }
154
155   release(&idelock);
156 }
```

ロックを使うことの難しい部分は、どうやって多くのロックを使うか、そしてどのロックがどのデータとインバリアントを保護するのか決めることである。
そこにはいくつかの基本的な原則がある。
最初に、他のCPUが読み書きできるような変数はいつでも、同時にあるCPUから書き込まれる可能性がある。
ロックは、その2つの操作がオーバーラップしないように導くべきである。
次に、ロックはインバリアントを保護するということを忘れるな。
あるインバリアントが複数のデータ構造にまたがる場合、典型的には、インバリアントを保護することを確実にするために、一つのロックによって、その全てのデータ構造は保護されなければならない。

上記のルールは、ロックが必要なときの話であり、ロックが不必要な場合については何も語ってはいない。
そして、効率上重要な点であるが、ロックは並列性を損なうので、ロックは使い過ぎないようにしなければならない。
効率が重要でない場合、ユニプロセッサのコンピュータとして使えば、ロックについては気にする必要はなくなる。
カーネルのデータ構造を保護するためなら、カーネルに入るときに獲得され、カーネルを出るときに解放されるようなロックが一つあればいい。
多くのユニプロセッサ用のOSのは、ときどきジャイアントロック（giant kernel lock）と呼ばれるこのやり方を用いてマルチプロセッサ上で実行出来るよう移行してきた。
しかしこの方法は、本物の同時実行性を犠牲にする。
一度にひとつのCPUだけが、カーネルで実行出来るのみである。
カーネルが何か重い計算を行う場合、より細かい粒度のロックの大きなセットを扱うのが、より効率的である。
もしそうなら、カーネルは複数のCPU上で同時に実行出来る。

究極的には、ロック粒度の選択は、並列プログラミングにおける課題である。
xv6は、少なく粒度が粗いデータ構造特有のロックを使う。
例えばxv6は、プロセステーブルとそのインバリアントを保護するために一つのロックを使う。
（プロセステーブルについては第4章で説明する。）

より細かい粒度の方法では、プロセステーブルの個別のエントリ上で動いているスレッドが、並行して続行出来るようにするために、プロセステーブル内のひとつのエントリ毎にロックを持つだろう。
しかしながら、プロセステーブル全体に渡るインバリアントを持つ操作は、いくつかのロックを持ち出さなきゃならないので、複雑になる。
xv6の例が、ロックの使い方を伝える手助けになることを願う。

感想

おもむろに練習問題1が出てきました。
もし試すなら、決まったパターンをファイルに書き込むプログラムを作って、それを複数同時実行するのが手っ取り早いかなと思います。
もし競合状態が発生したら、ファイルの内容に欠落が起きるはずです。
ただ、確率的になかなり低そうなのでどのくらいやったら観測可能かは分かりません。
iderwではキューの最後まで辿って、すかさずそのキューの最後に新しいリクエストを追加してます（*pp = b;の行）が、あるCPUにおけるその行の処理が、他のCPUがキューの最後まで辿った直後かつ追加の直前に行われたら競合状態になるはずです。
多分panicしたりとかデッドロックしたりとかにはならないはずなので、表面上は何も問題ないように見えるかもしれませんね。

プログラマ界隈の笑い話として、「謎のprint文に、この行をコメントアウトしたら何故か他の箇所で問題が起きるので注意。なんていうコメントが付いてた」なんてのがありますが、ここまで分かるようになると、（その原因を放置してるという意味では異常ですが）コンピュータの仕組み的には特に不思議でもなんでもないですね。
そういう場合メモリ管理に問題がある可能性もありますが、ちょっとしたタイミングの違いによって潜在的な競合状態が表面化するかしないかの差が出てるだけってのがしっくり来ます。

並列性を上げるには、ロック粒度を細かくして数を増やすのが有効とも書いてありますね。
似たような例として、DBのテーブルロックと行ロックの並列性についての話がありますね。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/12 月曜日 [http://peta.okechan.net/blog/archives/1433] |