日曜研究室

技術的な観点から日常を綴ります

[xv6 #39] Chapter 3 – Locking – Interrupt handlers

テキストの48ページ

本文

xv6は、あるCPUで実行される割り込みハンドラを、他のCPUで実行され同じデータにアクセスする非割り込みコードから保護するために、ロックを使う。
例えば、タイマー割り込みハンドラは、ticks変数をインクリメントするが、他のCPUでその変数を使うsys_sleep関数が同時に実行される可能性がある。
tickslockというロックは、2つのCPUが1つの変数へ同期的にアクセスするためにある。

trap.cのtrap関数のタイマー割り込みを制御する部分（ticksはtrap.cで uint ticks; と定義されている）

```
1  case T_IRQ0 + IRQ_TIMER:
2    if(cpu->id == 0){
3      acquire(&tickslock);
4      ticks++;
5      wakeup(&ticks);
6      release(&tickslock);
7    }
8    lapiceoi();
9    break;
```

sysproc.cのsys_sleep関数（ticksはtrap.cの uint ticks; を参照する）

```
01 int
02 sys_sleep(void)
03 {
04   int n;
05   uint ticks0;
06
07   if(argint(0, &n) < 0)
08     return -1;
09   acquire(&tickslock);
10   ticks0 = ticks;
11   while(ticks - ticks0 < n){
12     if(proc->killed){
```

```
13        release(&tickslock);
14        return -1;
15      }
16      sleep(&ticks, &tickslock);
17    }
18    release(&tickslock);
19    return 0;
20  }
```

割り込みは、シングルプロセッサ上でも同じく同時実行を引き起こす可能性がある。
割り込みが有効な場合、カーネルのコードは、割り込みハンドラを実行しないで、いつでも停止することが出来る。
iderw関数がidelockを保持してるときに、ideintrを実行するために割り込まれたと仮定しよう。
ideintrは、idelockを獲得しようとするだろうが、それは既にiderwによって保持されているので、ideintrはそのロックが解放されるまで待つだろう。
しかしこの場合、idelockは決して解放されず（iderwだけがそれを解放可能なのに、iderwはideintrが終わるまで続行することは出来ないので）、プロセッサ、そして結局はシステム全体が、デッドロックに陥るだろう。

そんな事態を避けるために、割り込みが有効かつ、割り込みハンドラでロックが使われている場合、プロセッサはその割り込みの前にそのロックを決して保持しないようにする必要がある。
xv6はかなり保守的である。
xv6は、割り込みが有効な場合は、どんなロックも決して保持しない。
xv6は、”割り込み無効化”の操作のスタックを管理するために、pushcli関数とpopcli関数を使う。
cliとは、割り込みを無効化するx86の命令である。
acquire関数は、ロックを獲得しようとする前にpushcli関数を呼ぶ。
そしてrelease関数は、ロックを開放した後にpopcli関数を呼ぶ。
pushcliとpopcliは、cli命令とsti命令を単純にラップするだけではない。
それらの呼び出し回数は記録され、2回のpushcliの呼び出しを元に戻すためには、2回のpopcliの呼び出しが必要となる。
以上のことから、コードがそれぞれ違う2つのロックを獲得する場合、両方のロックが解放されるまで、割り込みが再度有効になることは無いだろう。

spinlock.c

```
001 // Mutual exclusion spin locks.
002
003 #include "types.h"
004 #include "defs.h"
005 #include "param.h"
006 #include "x86.h"
007 #include "memlayout.h"
008 #include "mmu.h"
009 #include "proc.h"
010 #include "spinlock.h"
011
012 void
013 initlock(struct spinlock *lk, char *name)
014 {
```

```
015   lk->name = name;
016   lk->locked = 0;
017   lk->cpu = 0;
018 }
019
020 // Acquire the lock.
021 // Loops (spins) until the lock is acquired.
022 // Holding a lock for a long time may cause
023 // other CPUs to waste time spinning to acquire it.
024 void
025 acquire(struct spinlock *lk)
026 {
027   pushcli(); // disable interrupts to avoid deadlock.
028   if(holding(lk))
029     panic("acquire");
030
031   // The xchg is atomic.
032   // It also serializes, so that reads after acquire are not
033   // reordered before it.
034   while(xchg(&lk->locked, 1) != 0)
035     ;
036
037   // Record info about lock acquisition for debugging.
038   lk->cpu = cpu;
039   getcallerpcs(&lk, lk->pcs);
040 }
041
042 // Release the lock.
043 void
044 release(struct spinlock *lk)
045 {
046   if(!holding(lk))
047     panic("release");
048
049   lk->pcs[0] = 0;
050   lk->cpu = 0;
051
052   // The xchg serializes, so that reads before release are
053   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
054   // 7.2) says reads can be carried out speculatively and in
055   // any order, which implies we need to serialize here.
056   // But the 2007 Intel 64 Architecture Memory Ordering White
057   // Paper says that Intel 64 and IA-32 will not move a load
058   // after a store. So lock->locked = 0 would work here.
059   // The xchg being asm volatile ensures gcc emits it after
060   // the above assignments (and after the critical section).
061   xchg(&lk->locked, 0);
062
063   popcli();
064 }
065
066 // Record the current call stack in pcs[] by following the %ebp
    chain.
067 void
068 getcallerpcs(void *v, uint pcs[])
069 {
070   uint *ebp;
071   int i;
072
073   ebp = (uint*)v - 2;
074   for(i = 0; i < 10; i++){
075     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp ==
    (uint*)0xffffffff)
076       break;
```

```
077     pcs[i] = ebp[1];     // saved %eip
078     ebp = (uint*)ebp[0]; // saved %ebp
079   }
080   for(; i < 10; i++)
081     pcs[i] = 0;
082 }
083
084 // Check whether this cpu is holding the lock.
085 int
086 holding(struct spinlock *lock)
087 {
088   return lock->locked && lock->cpu == cpu;
089 }
090
091
092 // Pushcli/popcli are like cli/sti except that they are matched:
093 // it takes two popcli to undo two pushcli.  Also, if interrupts
094 // are off, then pushcli, popcli leaves them off.
095
096 void
097 pushcli(void)
098 {
099   int eflags;
100
101   eflags = readeflags();
102   cli();
103   if(cpu->ncli++ == 0)
104     cpu->intena = eflags & FL_IF;
105 }
106
107 void
108 popcli(void)
109 {
110   if(readeflags()&FL_IF)
111     panic("popcli - interruptible");
112   if(--cpu->ncli < 0)
113     panic("popcli");
114   if(cpu->ncli == 0 && cpu->intena)
115     sti();
116 }
```

acquireが、実際にロックを獲得するためのxchg命令を実行する前にpushcliを呼ぶ事は重要である。
それら2つの順番が逆だと、割り込みが有効な場合にロックが保持されたとき、いくつかの命令サイクルののち、残念ながら定期的に区切られる割り込みは、システムをデッドロックに陥れるだろう。（謎）
同様にreleaseが、実際にロックを解放するためのxchg命令を実行する前にpopcliを呼ぶ事は重要である。

割り込みハンドラと割り込みじゃないコードの間の相互作用は、再帰ロックに何故問題があるのかということの良い例である。
xv6が再帰ロックを使った場合（最初のロックの獲得の時と同じCPUでロックを獲得した場合、ひとつのCPU上で分割してロックを獲得することが可能となる。）、割り込みじゃないコードが重要な部分を実行してる時に、割り込みハンドラが実行出来ることとなる。
割り込みハンドラの実行中、そのハンドラのインバリアントは一時的な違反に依存するので、これは混乱をもたらす。
例えばideintrは、リンクリストが正しい形式の未実行リクエストで構成されていると仮定する。

もしxv6が再帰ロックを使ってた場合、ideintrはiderwがリンクリストを操作してる最中に実行でき、そしてリンクリストは最終的におかしな状態になる。

感想

割り込みハンドラとロックの関係についてです。

本文では「xv6は、割り込みが有効な場合は、どんなロックも決して保持しない。」と書かれてますが、処理の説明としては、割り込みが有効なときロックを保持しないのではなく、ロックを保持したら割り込みを無効にする、と言ったほうが正しいかもです。
まぁどっも事実の説明としては、主眼をどこに置くかが違うだけで同じような意味ではあるのですが。

謎と書いた部分についてですが、割り込み無効化→ロック獲得、の順番ではなく、ロック獲得→割り込み無効化、の順番だと、非割り込みコードがロックを保持したまま、割り込みハンドラが実行される可能性があり、そのハンドラが同じロックを保持しようとしてデッドロックに陥る可能性があり、だからダメだということかなと思います。

再帰ロックを採用した場合、非割り込みコードでロックを獲得してるにもかかわらず割り込みが無効にならない、というその理由がよく分かりません。
再帰ロックの場合、何回acquireされたかは、ロック変数毎に保持しとけばいいだけですし、ロックがゼロか否かなんてのは、今のacquireとreleaseの仕組みで十分判定出来るはずです。
だから再帰ロックを使ってても、どっかで1回でもacquireされてたら割り込みを無効にし、全てreleaseされたら再度割り込みを有効にする、なんて事は普通に出来る気がするんですが。

割り込み非割り込みに関わらず、同じCPUなら同じロックを何回でも獲得”出来なければならない”（単に出来るというのとは違って）というのが、ここで言う再帰ロックの定義だというのなら、本文の通りだと思います。

う〜ん、何か思い違いをしてる気がする。

カテゴリー: 技術 l タグ: xv6 l 投稿日: 2012/3/14 水曜日 [http://peta.okechan.net/blog/archives/1438] l