日曜研究室

技術的な観点から日常を綴ります

[xv6 #52] Chapter 4 – Scheduling – Exercises

テキストの62ページ

本文

1. sleepはデッドロックを回避するために lk != &ptable.lock であるかどうかチェックを行う。

```
1 if(lk != &ptable.lock){
2   acquire(&ptable.lock);
3   release(lk);
4 }
```

を

```
1 release(lk);
2 acquire(&ptable.lock);
```

に置き換えて、その特殊な場合（デッドロックが起きる場合）そのものを除去したとする。
これはsleepを壊すだろう。
どういう流れで問題が起きるか説明しなさい。

2. ほとんどのプロセスのクリーンアップ処理は、exitとwaitのどちらか一方または両方で出来たはずだ。
がしかし、 exitはp->kstackを解放すべきでないということを以前の節で見た。
だったら、exitは開いているファイルを閉じるためのものでなければならないということになる。
何故か？
パイプという言葉をつかって説明しなさい。

3. xv6用のセマフォを実装しなさい。
mutexは使えるが、sleepとwakeupは使えないものとする。
xv6の中でsleepとwakeupを使ってるところをセマフォで置き換えなさい。
その結果を推察しなさい。

作業

1. について
これは”起き損ないの問題”が起きるパターンかと思います。
sleepの呼び出し側のインバリアントの保護のためのlkが解放されているのに、まだsleep/wakeupのための&ptable.lockが確保されていないタイミングが生まれてしまうため、そのタイミング（まだスリープは完了してない段階）で別のプロセスによる同じチャンネル上のwakeupが実行されてしまう危険があります。

proc.cのsleep関数（変更前のソース。参考用）

```
01 // Atomically release lock and sleep on chan.
02 // Reacquires lock when awakened.
03 void
04 sleep(void *chan, struct spinlock *lk)
05 {
06   if(proc == 0)
07     panic("sleep");
08
09   if(lk == 0)
10     panic("sleep without lk");
11
12   // Must acquire ptable.lock in order to
13   // change p->state and then call sched.
14   // Once we hold ptable.lock, we can be
15   // guaranteed that we won't miss any wakeup
16   // (wakeup runs with ptable.lock locked),
17   // so it's okay to release lk.
18   if(lk != &ptable.lock){  //DOC: sleeplock0
19     acquire(&ptable.lock);  //DOC: sleeplock1
20     release(lk);
21   }
22
23   // Go to sleep.
24   proc->chan = chan;
25   proc->state = SLEEPING;
26   sched();
27
28   // Tidy up.
29   proc->chan = 0;
30
31   // Reacquire original lock.
32   if(lk != &ptable.lock){  //DOC: sleeplock2
33     release(&ptable.lock);
34     acquire(lk);
35   }
36 }
```

2. について
ちょっと質問の内容がよく分かりませんが、「クリーンアップ処理についてwaitで子プロセスのproc構造体を、exitで自身のファイルディスクリプタをそれぞれ受け持つような役割になってるのは何故か？」という質問だと仮定してみます。

まずwaitについては、子プロセスを待つために親プロセスで使われるので、その中で子プロセスそれ

ぞれのproc構造体をクリーンアップするのは理にかなってると言えます。
（子プロセスがすべて終了しZOMBIEになった直後にwaitは処理を受け継ぐ事が出来るので。）

もしexitでproc構造体をクリーンアップするとしたら、exit関数の処理中はまだプロセスは生きていて、生きているプロセスのproc構造体をそれ自身でクリーンアップするのは、（ちゃんと追ってませんが）いかにも問題ありそうです。
少なくともp->kstackはexit中では解放出来ません。

exitについては、子プロセスが終了するために子プロセスで使われる（親プロセスでも使われますが、それはそのプロセスのさらに親、例えばシェル、に対して使われるものなので、同じことです）と敢えて考えてみると、それぞれの子プロセスがもつファイルをこのタイミングで閉じるのも理にかなってると言えます。

それぞれのプロセスが開いているファイルは、proc構造体のofileに情報があり、親子関係であろうとそれはプロセス間で全くバラバラになり得るので、それぞれのプロセスが責任をもって自分が開いたファイルを閉じるのがシンプルで良いと思います。

もしこれをwaitで行うとすると、子プロセスのproc構造体をクリーンアップしてる部分にファイルの解放処理を追加することになると思います。
しかしそれだと親がwaitを呼ぶまで子が開いたファイルを閉じれないということになります。

また、「親が何かの処理を繰り返すためにメインループを実行し、そのメインループ中に子の生成を繰り返し、子は各々勝手に処理を行い、親が子のexitを待つ必要なく、親がwaitを呼ぶのは親が終了するときだけ」というアプリケーションもあり得ます。（サーバプログラムによくあるパターンです）

その場合、waitで子のファイルを閉じるようになってると、無数の子が開いた無数のファイルが親が終了するまで全く閉じられないことになります。
もちろんこの場合も、waitを最後だけじゃなくて適宜呼ぶことでこの問題を回避することは出来るはずですが、パフォーマンスは落ちるはずです。
それが無視出来る程度だったとしても、ユーザサイドのプログラミングにおける決まりごとを増やすのはあまり良いやり方とは思えません。

proc.cのexit, wait関数（参考用）

```
01 // Exit the current process.  Does not return.
02 // An exited process remains in the zombie state
03 // until its parent calls wait() to find out it exited.
04 void
05 exit(void)
06 {
07   struct proc *p;
08   int fd;
09
```

```
10    if(proc == initproc)
11      panic("init exiting");
12
13    // Close all open files.
14    for(fd = 0; fd < NOFILE; fd++){
15      if(proc->ofile[fd]){
16        fileclose(proc->ofile[fd]);
17        proc->ofile[fd] = 0;
18      }
19    }
20
21    iput(proc->cwd);
22    proc->cwd = 0;
23
24    acquire(&ptable.lock);
25
26    // Parent might be sleeping in wait().
27    wakeup1(proc->parent);
28
29    // Pass abandoned children to init.
30    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
31      if(p->parent == proc){
32        p->parent = initproc;
33        if(p->state == ZOMBIE)
34          wakeup1(initproc);
35      }
36    }
37
38    // Jump into the scheduler, never to return.
39    proc->state = ZOMBIE;
40    sched();
41    panic("zombie exit");
42  }
43
44  // Wait for a child process to exit and return its pid.
45  // Return -1 if this process has no children.
46  int
47  wait(void)
48  {
49    struct proc *p;
50    int havekids, pid;
51
52    acquire(&ptable.lock);
53    for(;;){
54      // Scan through table looking for zombie children.
55      havekids = 0;
56      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
57        if(p->parent != proc)
58          continue;
59        havekids = 1;
60        if(p->state == ZOMBIE){
61          // Found one.
62          pid = p->pid;
63          kfree(p->kstack);
64          p->kstack = 0;
65          freevm(p->pgdir);
66          p->state = UNUSED;
67          p->pid = 0;
68          p->parent = 0;
69          p->name[0] = 0;
70          p->killed = 0;
71          release(&ptable.lock);
72          return pid;
73        }
```

```
74      }
75
76      // No point waiting if we don't have any children.
77      if(!havekids || proc->killed){
78        release(&ptable.lock);
79        return -1;
80      }
81
82      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
83      sleep(proc, &ptable.lock);  //DOC: wait-sleep
84    }
85  }
```

3. について
まずはxv6で使われてるスピンロックとmutexの違いを。
どちらも処理の同期化（直列化）のために使われるものですが、

スピンロック
ロックが獲得できるようになるまでビジーウェイトで待つ方式。
ビジーウェイトと聞くと重そうではあるが、典型的にはCのコードにして数行分待つ間だけ（もちろんデッドロックを除いた最悪の場合、かなり長い間CPUを浪費する場合も有りうる）で、他でロックが解放されてから獲得できるようになるまでのタイムラグが究極的に小さい（もちろん他との獲得競争があってそれに破れた側の視点からのみ見るとその限りではないが、全体としてはロックが必要とされてるのに誰も獲得していないという空白期間を最小に出来る）。

mutex
mutexは色んなOSで色んな実装があって、仕様的にもそれぞれ微妙に違ってたりするので、ここはxv6的に以下のように定義されるとします。

ロックが確保出来た場合はすぐに呼び出し元に戻る。
ロックが確保できなかった場合はsleepして確保できるようになるまで待つ。
再帰ロックは出来ない。
ロックを解放するときは解放した後wakeupを呼び、他のロック解放待ちのプロセスを起こす。
インターフェイスは、
void acquiremutex(struct mutex *m);
void releasemutex(struct mutex *m);
とする。

と、ここまで考えたものの、そのようなmutexを使い、直接sleepとwakeupを使わないセマフォの実装は考えつかず。
前提がおかしいのか頭がおかしいのか…

感想

3が手も足も出なくてくやしい

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/27 火曜日 [http://peta.okechan.net/blog/archives/1577] |