日曜研究室

技術的な観点から日常を綴ります

[xv6 #50] Chapter 4 – Scheduling – Code: Wait and exit

テキストの60ページ

本文

sleepとwakeupは、条件が整うまで待つ事が必要な、様々な状況で使用され得る。
第0章で見たように、親プロセスは、子プロセスが終了するのを待つためにwait関数を呼ぶことが出来る。
xv6では、子プロセスが終了（exit）したとき、すぐには終了（die）しない。
その代わり、親プロセスが終了出来る状態になるためにwaitを呼ぶまで、プロセスの状態をZOMBIEに変更する。
親プロセスには、そのプロセスに割り当てられたメモリを解放し、再利用のためにproc構造体を準備する責任がある。
それぞれのプロセスのproc構造体は、p->parentとして親プロセスを指すポインタを保持する。
子プロセスより先に親プロセスが終了した場合、一番最初のプロセスであるinitがその子プロセスを受け継ぎ、その終了を待つ。
このステップは、いくつかのプロセスが子プロセスが終了した後にクリーンアップする事を確実にするために必要である。
すべてのプロセスのproc構造体は、ptable.lockによって保護される。

waitは、ptable.lockの獲得から始める。
そしたら、そのプロセスの子プロセスを捜すためにプロセステーブルを走査する。
もしwaitが、現在のプロセスがまだ終了してない子プロセスを持つということを見つけた場合、その子プロセスが終了するのを待つためにsleepを呼び、それを繰り返す。
そして、sleep中にptable.lockが解放される。
それは以前の節で見た特別な場合に相当する。（特別 = 同じロックの獲得と解放を別のプロセスで行うパターンということ？）

proc.cのwait関数

```
01 // Wait for a child process to exit and return its pid.
02 // Return -1 if this process has no children.
03 int
04 wait(void)
05 {
06   struct proc *p;
07   int havekids, pid;
08
09   acquire(&ptable.lock);
10   for(;;){
11     // Scan through table looking for zombie children.
12     havekids = 0;
13     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
14       if(p->parent != proc)
15         continue;
16       havekids = 1;
17       if(p->state == ZOMBIE){
18         // Found one.
19         pid = p->pid;
20         kfree(p->kstack);
21         p->kstack = 0;
22         freevm(p->pgdir);
23         p->state = UNUSED;
24         p->pid = 0;
25         p->parent = 0;
26         p->name[0] = 0;
27         p->killed = 0;
28         release(&ptable.lock);
29         return pid;
30       }
31     }
32
33     // No point waiting if we don't have any children.
34     if(!havekids || proc->killed){
35       release(&ptable.lock);
36       return -1;
37     }
38
39     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
40     sleep(proc, &ptable.lock);  //DOC: wait-sleep
41   }
42 }
```

exitは、ptable.lockを獲得し、それから現在のプロセスの親プロセスを起こす。
exit関数が現在のプロセスをZOMBIEとしてマークする前の段階で、親プロセスを起こすのは時期尚早に見えるだろう。
しかしこれは安全である。
親プロセスはその時点でRUNNABLEとしてマークされてるとは言え、exitがスケジューラに処理を移すためにschedを呼ぶ事によってptable.lockを解放するまで、waitの中のループは実行できないので、ZOMBIE状態になる前に終了中のプロセスをwaitで操作してしまうような事態は起きない。
exitはスケジューラを呼ぶ前に、終了中の現在のプロセスの子プロセスのすべての親プロセスをinitprocへ変更する。
最後にexitはCPUを手放すためにschedを呼ぶ。

proc.cのexit関数

```
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 }
```

そしたらスケジューラは、終了中のプロセスの親プロセス（waitのsleepの呼び出しによってスリープしている）を実行するために選択することが出来る。
sleepの呼び出しから戻るとき、ptable.lockを保持した状態になってるので、waitはプロセステーブルを再捜査し、state == ZOMBIEとなっている終了した子プロセスを捜すことが出来る。
そしたらその子プロセスのpidを記録し、そしてそのproc構造体をクリーンアップし、そのプロセスに割り当てられたメモリを解放する。（ここはwait関数のif(p–>state == ZOMBIE){の中の話）

これで子プロセスはexitにおけるクリーンアップ処理の大部分を完了したことになるが、子プロセスのp->kstackやp->pgdirは親プロセスが解放しなければならないというのが重要な点である。
子プロセスがexitを実行するとき、そのスタックはp->kstackとして割り当てられたメモリに置かれ、そのプロセス自身のページテーブルとして使う。
それらは、子プロセスが終了した後（exit内でsched経由でswtchが呼ばれた後）でないと解放できない。
これはスケジューラの手続きが、schedと呼ばれるスレッドのスタック上ではなく、それ自身のス

タック上で動作するひとつの理由である。

感想

waitとexitのコードの説明ですが、ちゃんと理解するためにはプロセスの仕組みについての総合的な知識が必要です。

自分の知識の一つ上ぐらいの事ならまだあーだこーだ言える余地があるんですが、この節は二・三段ぐらい上な感じで、なんとか理解は出来るけど受け身で「へぇそうなんだ」と思うことしかできませんでした。

親プロセスのp->parentはどうなってるのかというと、それはシェルから起動されたプログラムの場合は、親はシェルになってるはずです。（シェルはforkを使ってプログラムを実行するので）
ではシェルはどうなのかというと、ソースを全部読んだわけではないので、間に何かはさまってるかもしれませんが、initprocの子になってるはずです。
initprocがすべてのプロセスの最終的な親になるので、exit関数はinitprocで実行されるとpanicするようになってますね。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/26 月曜日 [http://peta.okechan.net/blog/archives/1567] |