日曜研究室

技術的な観点から日常を綴ります

[xv6 #46] Chapter 4 – Scheduling – Code: Scheduling

テキストの53〜55ページ

本文

前の節では、swtchのローレベルにおける詳細について見た。
では、swtchが与えられたとして、プロセスからスケジューラへ、スケジューラからプロセスへの切り替えに入り組んだ決まりごとを調べてみよう。
CPUを手放したいプロセスは、プロセステーブルのロックであるptable.lockを獲得し、保持している他のどんなロックをも解放し、自身の状態（proc->state）を更新し、そしてschedを呼ばなければならない。
yieldはこのsleepしexitするような決まりごとに従うが、それについては後で説明する。
schedは、それらの状態を2重にチェック、すなわちそれらの状態の組み合わせをチェックする。
ロックは保持されているので、そのCPUは割り込み無効な状態で実行されている。
最終的にschedは、現在のコンテキストをproc->contextに保存し、cpu->schedulerに格納されているコンテキストに切り替えるためにswtchを呼ぶ。
swtchは、まるでスケジューラがswtchを呼んで返ってきた直後のように、スケジューラのスタック上に戻る。（swtch(&cpu->scheduler, proc->context);のところ）
スケジューラはforループを続行し、実行すべきプロセスを見つけ、それに切り替え、それが繰り返される。

proc.cのscheduler, sched, yield関数

```
01 // Per-CPU process scheduler.
02 // Each CPU calls scheduler() after setting itself up.
03 // Scheduler never returns.  It loops, doing:
04 //  - choose a process to run
05 //  - swtch to start running that process
06 //  - eventually that process transfers control
07 //      via swtch back to the scheduler.
08 void
09 scheduler(void)
10 {
```

```
11    struct proc *p;
12
13    for(;;){
14      // Enable interrupts on this processor.
15      sti();
16
17      // Loop over process table looking for process to run.
18      acquire(&ptable.lock);
19      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
20        if(p->state != RUNNABLE)
21          continue;
22
23        // Switch to chosen process.  It is the process's job
24        // to release ptable.lock and then reacquire it
25        // before jumping back to us.
26        proc = p;
27        switchuvm(p);
28        p->state = RUNNING;
29        swtch(&cpu->scheduler, proc->context);
30        switchkvm();
31
32        // Process is done running for now.
33        // It should have changed its p->state before coming back.
34        proc = 0;
35      }
36      release(&ptable.lock);
37
38    }
39  }
40
41  // Enter scheduler.  Must hold only ptable.lock
42  // and have changed proc->state.
43  void
44  sched(void)
45  {
46    int intena;
47
48    if(!holding(&ptable.lock))
49      panic("sched ptable.lock");
50    if(cpu->ncli != 1)
51      panic("sched locks");
52    if(proc->state == RUNNING)
53      panic("sched running");
54    if(readeflags()&FL_IF)
55      panic("sched interruptible");
56    intena = cpu->intena;
57    swtch(&proc->context, cpu->scheduler);
58    cpu->intena = intena;
59  }
60
61  // Give up the CPU for one scheduling round.
62  void
63  yield(void)
64  {
65    acquire(&ptable.lock);  //DOC: yieldlock
66    proc->state = RUNNABLE;
67    sched();
68    release(&ptable.lock);
69  }
```

我々は、xv6がptable.lockがswtchの呼び出しにまたがって保持されているのをたった今見た。

swtchの呼び出し元は、すでにロックを保持してなければならないが、そのロックの管理は切り替え

先のコードに渡される。
この決まりごとは、ロックの例外的な使い方である。
一般的な決まりごとは、ロックを獲得したスレッドにそのロックを解放する責任もあるということであり、それは正しさのためという単純明快な理由による。
コンテキストスイッチの場合、ptable.lockはプロセスのstateとcontextフィールド（それらはswtchの実行中は本物ではない）のインバリアントを保護するので、その一般的な決まりごとを破る必要がある。
ptable.lockがswtchを実行する間保持されてなかった場合に起きる問題の例をひとつ挙げる。
yieldがその状態をRUNNABLEにセットした後、しかしswtchがそのプロセスが自身のカーネルスタックの使用をやめる前に、別のCPUはそのプロセスを実行することを決定するかもしれない。
これは2つのCPUが同じスタック上で実行される結果を引き起こす可能性があり、それは正しくない状況である。

カーネルスレッドは常にsched中で自身のプロセッサを手放し、常にスケジューラの同じ場所に切り替え、そしてスケジューラは（ほぼ）常にsched中のプロセスへ切り替える。
従って、xv6がどのスレッドに切り替えるかその行番号を印字できたとしたら、次のシンプルなパターンが観測できるだろう。
schedulerのswtchの行番号→schedのswtchの行番号→…繰り返し…
ふたつのスレッド間で起こるこの様式化された切り替えにおける手続きは、ときどきコルーチンの例として言及される。
この例では、schedとschedulerは、それぞれお互いのコルーチンである。

スケジューラのswtchが新しいプロセスへ切り替えるとき、schedで終わらない場合がある。
この場合を我々は第1章で見た。
新しいプロセスが最初にスケジュールされるとき、forkretから開始する。
forkretは、ptable.lockを解放するというこの決まりごとを引き受けるためだけに存在する。
そうでなければ、新しいプロセスはtrapretから開始できた。

proc.cのforkret関数

```
01 // A fork child's very first scheduling by scheduler()
02 // will swtch here.  "Return" to user space.
03 void
04 forkret(void)
05 {
06   static int first = 1;
07   // Still holding ptable.lock from scheduler.
08   release(&ptable.lock);
09
10   if (first) {
11     // Some initialization functions must be run in the context
12     // of a regular process (e.g., they call sleep), and thus cannot
13     // be run from main().
14     first = 0;
15     initlog();
16   }
17
```

```
18    // Return to "caller", actually trapret (see allocproc).
19  }
```

schedulerはシンプルなループを実行する。
実行すべきプロセスを見つけ、止まるまで実行し、それを繰り返す。
schedulerは、自身の活動の大部分のためにptable.lockを保持し、外側の各ループで一度ロックを解放する。（そして割り込みを有効にする）
これは、このCPUが暇なとき（RUNNABLEなプロセスが見つからない）という特別な場合に重要である。
もしロックを保持したまま暇なスケジューラがループした場合、プロセスを実行中の他のどのCPUもコンテキストスイッチや、プロセス関連のシステムコールを実行できなくなり、特に暇なCPUをそのスケジューリングのループの外に脱出させるために、プロセスをRUNNABLEとしてマークすることが出来なくなる。
プロセス（例えばシェル）はI/O待ちになったりするので、RUNNABLEなプロセスが全くない状況になる可能性があるということが、暇なCPU上で定期的に割り込みを有効にする理由である。
もしスケジューラが常に割り込みを無効化したままだと、I/Oは決して到着しない。

スケジューラは、p->state == RUNNABLEである実行可能なプロセスを探すためにプロセステーブルをループしながら走査する。
プロセスを見つけたら、CPUごとの現在のプロセスを表すproc変数をセットし、switchuvm関数を使ってそのプロセスのページテーブルへ切り替え、そのプロセスをRUNNINGとしてマークし、そしてそのプロセスを開始するためにswtchを呼ぶ。（proc = p;からswtchの部分）

スケジューリングのコードの構造について考えるひとつの方法は、スケジューリングがそれぞれのプロセスに関するインバリアント一式を強制するため準備し、それらのインバリアントが正しくない間は常にptable.lockを保持するという事である。
もしプロセスがRUNNINGなら、タイマ割り込みによって呼び出されるyieldが、そのプロセスからどこかへ正しく切り替える事が出来るようにするためにセットアップされなければならない、ということがインバリアントのひとつである。
これは、CPUのレジスタはプロセスのレジスタの値（それらは実はコンテキストの中に無い）を保持しなければならず、%cr3はプロセスのページテーブルを指さなければならず、%espはswtchが正しくレジスタをプッシュできるようにするために、そのプロセスのカーネルスタックを指さなければならず、procはproc配列におけるそのプロセスを表すものを指さなければならない、ということを意味する。
もしプロセスがRUNNABLEなら、暇なCPUのスケジューラがそれを実行出来るようにするためにセットアップされなければならない。
これは、p->contextはプロセスのカーネルスレッドの変数を保持しなければならず、そのプロセスのカーネルスタック上ではどのCPUも実行中ではなく、そのプロセスのページテーブルをどのCPUの%cr3も指しておらず、そのプロセスをどのCPUのproc変数も指していない、ということを意味する。

上記のインバリアントを保護することが、なぜxv6があるスレッド（しばしばyield中）でptable.lock

を獲得し、他のスレッド（スケジューラのスレッドや他の次のカーネルスレッド）でそのロックを解放するのか、ということの理由である。
一度、実行中のプロセスの状態をRUNNABLEへ変更するためのコードが開始したら、そのコードはインバリアントの復元が完了するまでロックを保持しなければならない。
開放するタイミングとして早く正しいのは、schedulerがプロセスのページテーブルの使用をやめ、procをクリアした後である。
同様に、一度、スケジューラが実行可能なプロセスをRUNNINGに変更しはじめたら、カーネルスレッドが確実に実行されるまで（yield中のswtchの後）ロックを開放してはいけない。

ptable.lockは、よく他のものも保護する。
プロセステーブルスロットの空きやプロセスIDの割り当て、exitとwait間の相互作用、起床（wakeups。 次の章で説明する）の失敗を避けるための仕掛け、多分他のことも。
ptable.lockの別の関数が分割可能かどうかについて、よく理解するためには確実に、パフォーマンスのためには多分、考える価値がある。

感想

長い。

プロセス切り替え時のコードパスの詳細です。
大きく2パターンあって、ひとつはプロセスからスケジューラへの切り替え、もうひとつはスケジューラからプロセスへの切り替えとなります。

プロセスからスケジューラへの切り替えの場合、頻度が一番多いのがタイマ割り込み起点の切り替えですかね。
その際は、割り込み→trap→yield→shced→swtch→schedulerという流れになると思います。

スケジューラからプロセスへの切り替えの場合は、RUNNABLEなプロセスが見つかったらswtchで切り替えるという感じですね。

途中色々書いてあるところは、プロセスを表すデータ構造のインバリアントを保護するために切り替え前に何を行うかについて書かれてるんだと思います。
あとは、ロックと割り込みの兼ね合い、複数CPU時における問題、を通じてなぜこの流れで処理するのかについての説明ですね。

違うプロセスでptable.lockの獲得と開放が行われる部分があります。
切り替えによって違うプロセスになるといっても、CPUは同じなので一応問題なくロックの開放が出来るわけですが、かなり特殊な例だと書いてあります。

細かいところでは、コルーチンという言葉が出てきました。

コルーチン – Wikipediaに説明が書いてあります。

あと、schedで出てくるcpu->intenaですが、pushcliの前に割り込みが有効だったかどうかを表すフィールドみたいです。
cpu->intenaの保存と復元がないと、それを参照したり書き換えたりしているpushcliとpopcli（それぞれacquireとreleaseから呼ばれる。特にpopcliの方が）の動作がおかしくなるはずです。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/20 火曜日 [http://peta.okechan.net/blog/archives/1551] |