日曜研究室

技術的な観点から日常を綴ります

[xv6 #45] Chapter 4 – Scheduling – Code: Context switching

テキストの51〜53ページ

本文

図4-1で示されるように、プロセスを切り替えるために、xv6はローレベルで2種類のコンテキストスイッチを行う。
ひとつは、プロセスのカーネルスレッドから現在のCPUのスケジューラのスレッドへの切り替え、もうひとつは、そのスケジューラのスレッドから他のプロセスのカーネルスレッドへの切り替えである。
xv6は、ユーザ空間のプロセスから他のプロセスへ直接切り替えることは決してしない。
このことは、ユーザカーネルの遷移（システムコールや割り込み）のやり方によって、スケジューラへの切り替え、新しいプロセスのカーネルスレッドへのコンテキストスイッチ、そしてトラップから戻るという事を引き起こす。
この節では、カーネルスレッドとスケジューラスレッドの間の切り替えの仕組みについて説明する。

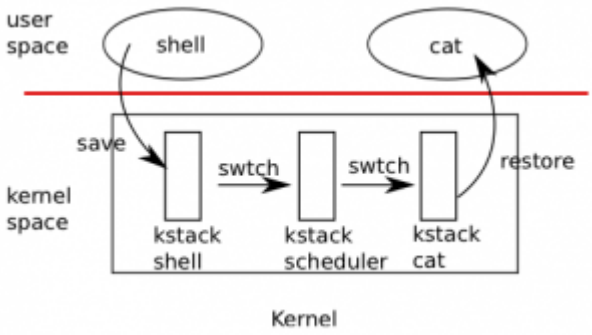

図4–1 あるユーザプロセスから他のプロセスへの切り替え。この例では、xv6は一つのCPUで実行されている。（よってスケジューラのスレッドもひとつである）

xv6のプロセスはどれでも、第1章で見てきたように、それ自身のカーネルスタックとレジスター式を

持つ。
それぞれのCPUは、どこかのプロセスのカーネルスレッドではなく、スケジューラを実行しているときに使うための個別のスケジューラスレッドを持つ。
あるスレッドから他のスレッドへの切り替えは、切り替え元もスレッドのCPUレジスタの保存と、切り替え先のスレッドの以前に保存されたレジスタの復元を伴う。
%espと%eipが保存や復元されるという事実は、CPUがスタックを切り替え、どのコードが実行されているかを切り替えるという事を意味する。

swtchはスレッドについては直接的には知らない。
コンテキストと呼ばれるレジスター式を保存し復元するだけである。
プロセスにCPUを手放させるときが来たとき、そのプロセスのカーネルスレッドは、自身のコンテキストを保存しスケジューラコンテキストに戻るためにswtchを呼ぶだろう。
それぞれのコンテキストは、struct context*として表現され、巻き込まれたカーネルスタック上に保存されたデータ構造を指すポインタである。
swtchは2つの引数を受け取る。
struct context **oldとstruct context *newである。
swtchは、現在のCPUレジスタの値をスタックにプッシュし、そのスタックポインタを*oldに保存する。
そしてswtchは、newから%espにコピーし、以前に保存されたレジスタを取り出し、そして戻る。

proc.hで定義されているcontext構造体

```
01 // Saved registers for kernel context switches.
02 // Don't need to save all the segment registers (%cs, etc),
03 // because they are constant across kernel contexts.
04 // Don't need to save %eax, %ecx, %edx, because the
05 // x86 convention is that the caller has saved them.
06 // Contexts are stored at the bottom of the stack they
07 // describe; the stack pointer is the address of the context.
08 // The layout of the context matches the layout of the stack in
   swtch.S
09 // at the "Switch stacks" comment. Switch doesn't save eip
   explicitly,
10 // but it is on the stack and allocproc() manipulates it.
11 struct context {
12   uint edi;
13   uint esi;
14   uint ebx;
15   uint ebp;
16   uint eip;
17 };
```

swtch.S

```
01 # Context switch
02 #
03 #   void swtch(struct context **old, struct context *new);
04 #
05 # Save current register context in old
06 # and then load register context from new.
```

```
07
08 .globl swtch
09 swtch:
10   movl 4(%esp), %eax
11   movl 8(%esp), %edx
12
13   # Save old callee-save registers
14   pushl %ebp
15   pushl %ebx
16   pushl %esi
17   pushl %edi
18
19   # Switch stacks
20   movl %esp, (%eax)
21   movl %edx, %esp
22
23   # Load new callee-save registers
24   popl %edi
25   popl %esi
26   popl %ebx
27   popl %ebp
28   ret
```

Instead of following the scheduler into swtch, let's instead follow our user process back in.（ここ訳が分からない）
それぞれの割り込みの最後にtrap関数がyield関数を呼ぶ可能性について、我々は第2章で見た。
同様にyield関数は、現在のコンテキストをproc->contextへ保存し、以前にcpu->schedulerに保存されたスケジューラのコンテキストへ切り替えるためのswtchを呼ぶsched関数、を呼ぶ。

proc.cのsched関数とyield関数

```
01 // Enter scheduler.  Must hold only ptable.lock
02 // and have changed proc->state.
03 void
04 sched(void)
05 {
06   int intena;
07
08   if(!holding(&ptable.lock))
09     panic("sched ptable.lock");
10   if(cpu->ncli != 1)
11     panic("sched locks");
12   if(proc->state == RUNNING)
13     panic("sched running");
14   if(readeflags()&FL_IF)
15     panic("sched interruptible");
16   intena = cpu->intena;
17   swtch(&proc->context, cpu->scheduler);
18   cpu->intena = intena;
19 }
20
21 // Give up the CPU for one scheduling round.
22 void
23 yield(void)
24 {
25   acquire(&ptable.lock);  //DOC: yieldlock
26   proc->state = RUNNABLE;
27   sched();
28   release(&ptable.lock);
```

```
29 }
```

swtchは、%eaxレジスタと%edxレジスタへ自身の引数を読み込む事から開始する。
これは必要な処理である。
スタックポインタを変更すると、もはや%esp経由で引数にアクセスすることが出来なくなるからである。
そしてswtchは、現在のスタック上にcontext構造体を作るため、レジスタの状態をプッシュする。
呼び出される側で保存されるレジスタだけ保存される必要がある。
x86の慣習では、それらは%ebp, %ebx, %esi, %ebp, %espである。
swtchは、最初の4つを直接プッシュする。（# Save old callee–save registersのところ）
間接的にstruct context*として*oldに書き込むために最後のものを保存する。（movl %esp, (%eax)のところ）
もうひとつ重要なレジスタがある。
プログラムカウンタ%eipは、swtchを呼び出したcall命令によって保存され、%ebpのちょうどすぐ上位のスタックにある。
古いコンテキストが保存されたら、swtchは新しいコンテキストを復元する準備をする。
新しいコンテキストのポインタをスタックポインタに移動する。（movl %edx, %espのところ）
新しいスタックは、swtchがちょうどいま処理したばかりの古いスタックと同じ形式であり、（前回のswtchの呼び出しではその新しいスタックは古いスタックだった）swtchは逆の手順で新しいコンテキストを復元する。
%edi, %esi, %ebx, %ebpをポップし、そして戻る。（# Load new callee–save registersのところ）
swtchはスタックポインタを変更するので、復元された値と戻り先の命令アドレスは、新しいコンテキストのものとなる。

我々の例では、schedはCPUごとのスケジューラコンテキストであるcpu->schedulerへ切り替えるためにswtchを呼んだ。
そのスケジューラコンテキストは、scheduler関数によるswtchの呼び出しによってすでに保存されたものである。
我々が追跡しているswtchから戻ったとき、schedではなくschedulerへ戻り、そしてスタックポインタは、initprocのカーネルスタックではなく、現在のCPUのスケジューラのスタックを指す状態になっている。

感想

濃い節でした。
まとめると、
プロセスを切り替える際は、スケジューラに一旦切り替えてから、目的のプロセスに切り替える。
切り替え処理のコアはswtchである。
swtchでは、切り替え元のコンテキストを保存して、切り替え先のコンテキストを復元する。
ってところですかね。

yieldの説明のところでは訳がよく分からない文がありますが、ソースを見るとtrap関数の最後にタイマ割り込みだったらyieldを実行するというコードがあり、そこが時分割のキモとなる部分かと思います。
そこでスケジューラスレッドへ切り替わり、スケジューラはRUNNABLEなプロセスを探してそれに切り替える、という処理が行われるはずです。

最後のinitprocなんちゃらのところは、第1章のCode: Running a processで説明があった部分です。
swtchは起動中の特別な処理でも使われるのですが、単に今回は（というかこちらが通常の使い方だけど）その場合とは違うよということを言ってるんだと思います。

カテゴリー: 技術 l タグ: xv6 l 投稿日: 2012/3/19 月曜日 [http://peta.okechan.net/blog/archives/1547] l