日曜研究室

技術的な観点から日常を綴ります

[xv6 #16] Chapter 1 – The first process – Code: Running a process

テキストの25〜27ページ

本文

最初のプロセスの状態が準備できたら、実行することができる。
main関数がuserinit関数を呼んだ後、プロセスを開始するためにmpmain関数はshceduler関数を呼ぶ。
schedulerは、p->stateがRUNNABLEにセットされたプロセスを探す。
しかしこの時点では一つのプロセスだけが見つかる。
それはinitprocである。
（initprocは前回userinitの中で出てきました。元はproc.cに定義されてるstatic変数（proc構造体）です。）
cpuごとの変数procへ、見つかったプロセスをセットし、対象のプロセスのページテーブルを使うことをハードウェアに伝えるためにswitchuvm関数を呼ぶ。
switchkvm関数は、カーネルのコードとデータのための同一のマッピングを全てのプロセスのページテーブルに持たせるようにするので、カーネルを実行中にページテーブルを切り替える事が出来る。
switchuvm関数もまた、新しいタスク状態セグメント（task state segment）SEG_TSSを生成する。
それは、%ssへSEG_KDATA<<3を、%espへ(uint)proc->kstack+KSTACKSIZE（そのプロセスのカーネルスタックの一番上）をセットすることによってカーネルモードへの移行による割り込みを制御するためハードウェアと対話する。
タスク状態セグメントについては第2章でまた説明する。

main.cのmpmain関数

```
1 // Common CPU setup code.
2 static void
3 mpmain(void)
4 {
5   cprintf("cpu%d: starting\n", cpu->id);
6   idtinit();       // load idt register
7   xchg(&cpu->started, 1); // tell startothers() we're up
```

```
  scheduler();     // start running processes
}
```

proc.cのscheduler関数

```
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

vm.cのswitchuvm関数

```
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  pushcli();
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1,
0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(v2p(p->pgdir));  // switch to new address space
```

```
  14    popcli();
  15  }
```

そしたらschedulerはp->stateをRUNNINGにセットし、切替先のプロセスのカーネルスレッドへコンテキストスイッチさせるためにswtchを呼ぶ。
（switchじゃなくてswtchなので注意。以前switchって書いてる部分があったかも。Cの予約語と被らないようにするためですかね）
swtchは現在のレジスタを保存し、対象のカーネルスレッドの保存されたレジスタ（proc->context）をx86のハードウェアのレジスタにロードする。
それは、スタックポインタと命令ポインタを含む。
現在のコンテキストは、プロセスではなく、CPUごとの特別なスケジューラコンテキストであるので、schedulerはどこかのプロセスのカーネルスレッドコンテキストの中ではなく、CPUごとのストレージ（cpu->scheduler）の中に現在のハードウェアレジスタを保存するようswtchに伝える。
第4章でのその切替についてのより詳細な説明を行う。
最後のret命令（swtchの最後）はスタックから新しい%eipを取り出し、コンテキストスイッチを完了させる。
そうやって、プロセッサは、プロセスpのカーネルスレッドを実行している状態になる。

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
```

allocprocはinitprocのp->context->eipをforkretに設定するので、retはforkretから再開する。
forkretはptable.lockを解放する。（詳しくは第3章）
最初の呼び出しでは（これがそれだが）、forkret関数は、main関数からは実行出来ない初期化処理を

実行する。
なぜmain関数から実行できないかというと、そのプロセスのカーネルスタックとともに通常のプロセスのコンテキストで実行されなければならないからである。
そして、forkretから返る。
allocprocはスタック上でp->contextの後に配置したので、取り出された後はtrapretがトップになる。
それでtrapretは%espへp->tfを設定しつつ実行を開始する。
trapretは、カーネルコンテキストと共にswtchがしたように、トラップフレームを登っていくためにpop命令を使う。
（謎）
そして、popal命令は、普通のレジスタを復元し、popl命令で%gs, %fs, %es, %dsを復元する。
trapnoとerrcodeの二つのフィールドを飛ばすためにaddlを使ってる。
最後に、iret命令で、スタックから%cs, %eip, %flagsを取り除く。
トラップフレームの内容は、CPUに転送された。
だからプロセッサはトラップフレームで指定された%eipから続行する。
initprocについては、これは仮想アドレス0、initcode.Sの最初の命令を意味する。
（initcode.Sは以前出てきましたね。）

proc.cのallocproc関数（前回も載せてます）

```
01 //PAGEBREAK: 32
02 // Look in the process table for an UNUSED proc.
03 // If found, change state to EMBRYO and initialize
04 // state required to run in the kernel.
05 // Otherwise return 0.
06 static struct proc*
07 allocproc(void)
08 {
09   struct proc *p;
10   char *sp;
11
12   acquire(&ptable.lock);
13   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
14     if(p->state == UNUSED)
15       goto found;
16   release(&ptable.lock);
17   return 0;
18
19 found:
20   p->state = EMBRYO;
21   p->pid = nextpid++;
22   release(&ptable.lock);
23
24   // Allocate kernel stack.
25   if((p->kstack = kalloc()) == 0){
26     p->state = UNUSED;
27     return 0;
28   }
29   sp = p->kstack + KSTACKSIZE;
30
31   // Leave room for trap frame.
32   sp -= sizeof *p->tf;
33   p->tf = (struct trapframe*)sp;
34
```

```
35    // Set up new context to start executing at forkret,
36    // which returns to trapret.
37    sp -= 4;
38    *(uint*)sp = (uint)trapret;
39
40    sp -= sizeof *p->context;
41    p->context = (struct context*)sp;
42    memset(p->context, 0, sizeof *p->context);
43    p->context->eip = (uint)forkret;
44
45    return p;
46  }
```

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  }
```

trapasm.Sのtrapret

```
01    # Return falls through to trapret...
02  .globl trapret
03  trapret:
04    popal
05    popl %gs
06    popl %fs
07    popl %es
08    popl %ds
09    addl $0x8, %esp  # trapno and errcode
10    iret
```

この点では、%eipはゼロを保持し、%espは4096を保持する。
それらは、プロセスのアドレス空間における仮想アドレスである。
プロセッサのページングハードウェアは、それらを物理アドレスに翻訳する。
allocuvm関数は、このプロセスに割り当てられた物理メモリを指すための仮想アドレスゼロのページのPTEをセットアップし、そのPTEをそのプロセスが使えるようにPTE_Uでマークする。
PTE_Uがセットされ たPTEはそのプロセスのページテーブルでは他には存在しない。
userinitが、プロセスのユーザコードをCPL=3で実行するために%csの下位ビットを設定するという事実は、ユーザコードが、PTE_Uがセットされ たPTEだけを使えて、そして%cr3のような機密扱いのハードウェアレジスタを変更出来ないという事になる。

（CPL: Current Privilege level 現在の特権レベルの事かと）
なので、プロセスは自分自身のメモリを使うことを強制される。

proc.cのuserinit関数

```
01 //PAGEBREAK: 32
02 // Set up first user process.
03 void
04 userinit(void)
05 {
06   struct proc *p;
07   extern char _binary_initcode_start[], _binary_initcode_size[];
08
09   p = allocproc();
10   initproc = p;
11   if((p->pgdir = setupkvm(kalloc)) == 0)
12     panic("userinit: out of memory?");
13   inituvm(p->pgdir, _binary_initcode_start,
   (int)_binary_initcode_size);
14   p->sz = PGSIZE;
15   memset(p->tf, 0, sizeof(*p->tf));
16   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
17   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
18   p->tf->es = p->tf->ds;
19   p->tf->ss = p->tf->ds;
20   p->tf->eflags = FL_IF;
21   p->tf->esp = PGSIZE;
22   p->tf->eip = 0;  // beginning of initcode.S
23
24   safestrcpy(p->name, "initcode", sizeof(p->name));
25   p->cwd = namei("/");
26
27   p->state = RUNNABLE;
28 }
```

initcode.Sは、$argv, $init, $0の3つの値をスタックにプッシュすることからはじめ、%eaxにSYS_execをセットし、int T_SYSCALLを実行する。
それは、execシステムコールを実行するためにカーネルに伝える。
もしすべてうまく行ったら、execは決して戻らない。
$initという名前が付けられたプログラムの実行を開始する。
$initは、”/init”というNUL終端文字列へのポインタである。
もしexecが失敗し戻ってきたときは、確実に戻らないようにするためにinitcodeはexitシステムコールを呼びつづける。

initcode.S

```
01 # Initial process execs /init.
02
03 #include "syscall.h"
04 #include "traps.h"
05
06
07 # exec(init, argv)
08 .globl start
09 start:
```

```
10    pushl $argv
11    pushl $init
12    pushl $0  // where caller pc would be
13    movl $SYS_exec, %eax
14    int $T_SYSCALL
15
16  # for(;;) exit();
17  exit:
18    movl $SYS_exit, %eax
19    int $T_SYSCALL
20    jmp exit
21
22  # char init[] = "/init\0";
23  init:
24    .string "/init\0"
25
26  # char *argv[] = { init, 0 };
27  .p2align 2
28  argv:
29    .long init
30    .long 0
```

execシステムコールへの引数は、$initと$argvである。
最後の$0は、この手書きのシステムコールを通常のシステムコールに見せかけるためにある。
その辺の詳しいことは第2章で。
前の通り、このセットアップは、最初のプロセスの特別なやり方（その場合、それが最初のシステムコールである）は避けている。
その代わり、基本的な操作のためにxv6が提供すべきコードを再利用する。

感想

これでやっと最初のプロセスinitcodeが実行できたってところですかね。

mpmainは論理CPUごとに実行されるみたいなので、スケジューラも論理CPUごとに実行されるみたいですね。
マルチプロセッサ関連の話は後で出てくるんでしょう。

正直半分も理解できてない感じです。
理解しているつもりの部分に関しても自信はないです。
ただまだ最初のプロセスが起動するまでについての章なので、それ以外の部分がまだ概要レベルの説明にとどまってるというのもあるかと思います。
とりあえずこのまま読み続けて全体が見えるようになれば理解が深まるはずだと思っています。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/21 火曜日 [http://peta.okechan.net/blog/archives/1293] |