日曜研究室

技術的な観点から日常を綴ります

[xv6 #15] Chapter 1 – The first process – Code: Process creation

テキストの23〜25ページ

本文

この節では、xv6がどうやって最初のプロセスを生成するかについて述べる。
xv6のカーネルは、プロセスごとに多くの状態を管理する。
それらの状態はproc構造体にまとめられている。
ひとつのプロセスのカーネル状態で一番重要な部品は、それ自身のページテーブルとそれが指す物理メモリ、それ自身のカーネルスタック、それ自身の実行状態（run state）である。
proc構造体の要素を指すための記法として、p->xxxという形を用いる。

proc.hに定義されてるproc構造体

```
01 // Per-process state
02 struct proc {
03   uint sz;                     // Size of process memory (bytes)
04   pde_t* pgdir;                // Page table
05   char *kstack;                // Bottom of kernel stack for this
   process
06   enum procstate state;        // Process state
07   volatile int pid;            // Process ID
08   struct proc *parent;         // Parent process
09   struct trapframe *tf;        // Trap frame for current syscall
10   struct context *context;     // swtch() here to run process
11   void *chan;                  // If non-zero, sleeping on chan
12   int killed;                  // If non-zero, have been killed
13   struct file *ofile[NOFILE];  // Open files
14   struct inode *cwd;           // Current directory
15   char name[16];               // Process name (debugging)
16 };
```

プロセスのカーネル状態を、プロセスの味方上のカーネルの中で実行される、実行スレッド（or thread for short）として見るべきである。
（多分、「プロセスの味方上のカーネル」=「プロセスの仮想アドレス空間上に配置されたカーネル」。「or thread for short」の方はうまい訳が思いつかないどころかイメージも謎）

スレッドは、計算を実行する。
しかしそれは止めれるし再開も出来る。
例えば、プロセスがシステムコールを呼ぶとき、CPUはそのプロセスの実行からそのプロセスのカーネルスレッドの実行へ切り替える。
プロセスのカーネルスレッドは、システムコールの実装（例えばファイルを読んだり）を実行する。
そしてそれからプロセスに戻る。
xv6は、システムコールがカーネルの中でI/Oのために待てる（もしくは”block”できる）ようにするため、そしてI/Oが終わったときに元の実行中の場所に復帰できるようにするために、スレッドとしてシステムコールを実行する。
カーネルスレッドの多くの状態（ローカル変数、関数の戻り先アドレス）は、カーネルスレッドのスタックp->kstackに配置される。
それぞれのプロセスのカーネルスタックは、ユーザスタックを壊さないようにするために、ユーザスタックとは別になっている。
つまりプロセスを、2つの実行スレッドを持っているとみなすことができる。
一つはユーザスレッド、もう一つはカーネルスレッドである。

p->stateは、プロセスが割り当て済みか、実行準備が完了してるか、実行中か、I/O待ちか、終了中かどうかを示す。

p->pgdirは、PTEの配列を保持する。
xv6は、プロセスが実行されるとき、ページングハードウェアにそのプロセスのp->pgdirを使わせるようにする。
プロセスのページテーブルはまた、プロセスのメモリを保持するために割り当て済みの物理ページのアドレスのレコードとして動く。

最初のプロセスを生成する流れは、main関数がuserinit関数を呼ぶところから始まる。
userinit関数の最初のアクションはallocproc関数を呼ぶことである。
allocprocの仕事は、プロセステーブルにスロット（proc構造体）を割り当てることと、カーネルスレッドを実行するために必要なプロセス状態の一部の初期化である。
userinitが一番最初のプロセスのために呼ばれる間、allocprocは、全ての新しいプロセスのために呼ばれる。
allocprocは、UNUSED状態をキーにプロセステーブルをスキャンする。
未使用のプロセスを見つけたとき、allocprocは使用中を示すためEMBRYOをセットし、一意のpidを付与する。
次に、プロセスのカーネルスレッドのためのカーネルスタックを割り当てようとする。
もし割り当てが失敗したら、allocprocは状態をUNUSEDに戻し、失敗を伝えるためにゼロを返す。

main.cのmain関数（参照コミットが変わったので以前載せたものとは違うはずです。）

```
01 // Bootstrap processor starts running C code here.
02 // Allocate a real stack and switch to it, first
03 // doing some setup required for memory allocator to work.
```

```
04 int
05 main(void)
06 {
07   kvmalloc();       // kernel page table
08   mpinit();         // collect info about this machine
09   lapicinit(mpbcpu());
10   seginit();        // set up segments
11   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
12   picinit();        // interrupt controller
13   ioapicinit();     // another interrupt controller
14   consoleinit();    // I/O devices & their interrupts
15   uartinit();       // serial port
16   pinit();          // process table
17   tvinit();         // trap vectors
18   binit();          // buffer cache
19   fileinit();       // file table
20   iinit();          // inode cache
21   ideinit();        // disk
22   if(!ismp)
23     timerinit();    // uniprocessor timer
24   startothers();    // start other processors (must come before
kinit)
25   kinit();          // initialize memory allocator
26   userinit();       // first user process  (must come after kinit)
27   // Finish setting up this processor in mpmain.
28   mpmain();
29 }
```

proc.cのuserinit関数とallocproc関数

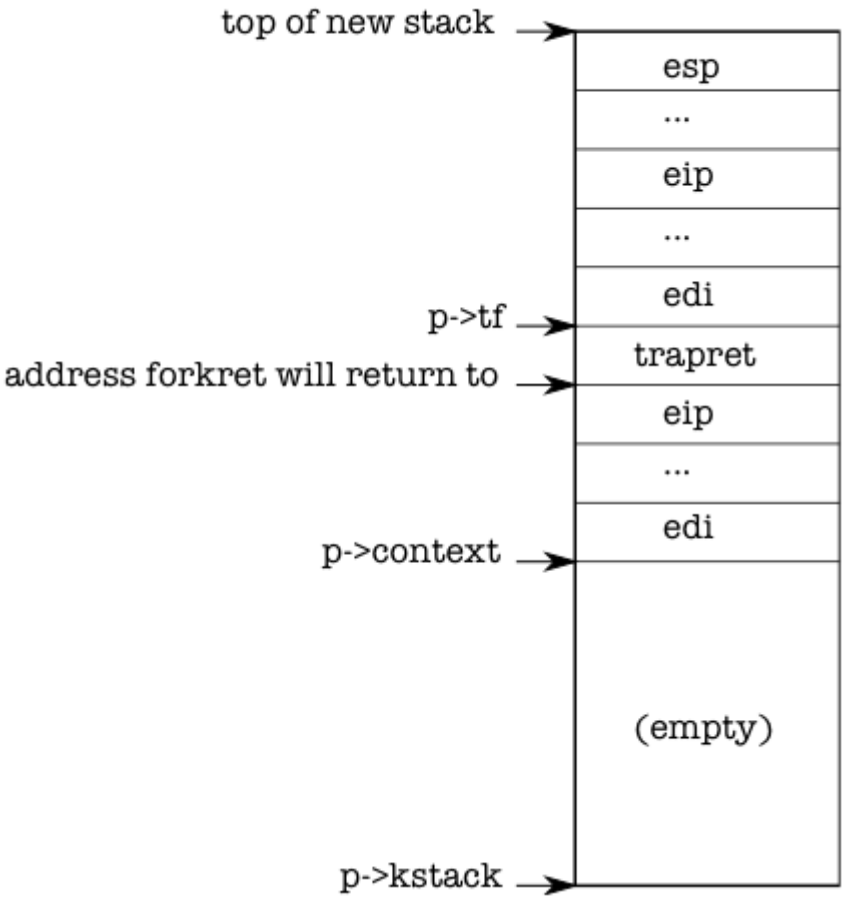
```
01 //PAGEBREAK: 32
02 // Set up first user process.
03 void
04 userinit(void)
05 {
06   struct proc *p;
07   extern char _binary_initcode_start[], _binary_initcode_size[];
08
09   p = allocproc();
10   initproc = p;
11   if((p->pgdir = setupkvm(kalloc)) == 0)
12     panic("userinit: out of memory?");
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 }
29
30 //PAGEBREAK: 32
31 // Look in the process table for an UNUSED proc.
32 // If found, change state to EMBRYO and initialize
```

```
33 // state required to run in the kernel.
34 // Otherwise return 0.
35 static struct proc*
36 allocproc(void)
37 {
38   struct proc *p;
39   char *sp;
40
41   acquire(&ptable.lock);
42   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
43     if(p->state == UNUSED)
44       goto found;
45   release(&ptable.lock);
46   return 0;
47
48 found:
49   p->state = EMBRYO;
50   p->pid = nextpid++;
51   release(&ptable.lock);
52
53   // Allocate kernel stack.
54   if((p->kstack = kalloc()) == 0){
55     p->state = UNUSED;
56     return 0;
57   }
58   sp = p->kstack + KSTACKSIZE;
59
60   // Leave room for trap frame.
61   sp -= sizeof *p->tf;
62   p->tf = (struct trapframe*)sp;
63
64   // Set up new context to start executing at forkret,
65   // which returns to trapret.
66   sp -= 4;
67   *(uint*)sp = (uint)trapret;
68
69   sp -= sizeof *p->context;
70   p->context = (struct context*)sp;
71   memset(p->context, 0, sizeof *p->context);
72   p->context->eip = (uint)forkret;
73
74   return p;
75 }
```

さてここで、allocprocは新しいプロセスのカーネルスタックをセットアップしなければならない。
通常は、プロセスはforkによってのみ作られる。
なので、新しいプロセスは、親からコピーされて始まる。
forkの結果、内容は親と同じだが独立なメモリを持つ子プロセスが生まれる。
allocprocは、特別に用意されたカーネルスタックと、親と同じユーザ空間上の場所（forkシステムコールから戻る場所）に戻るためのカーネルレジスタの組で、その子プロセスのカーネルスレッドでの実行を開始するため子プロセスをセットアップする。
その用意されたカーネルスタックのレイアウトは図1-3に示す。
allocprocはこの仕事の一部を、リターンプログラムカウンタ値（return program counter values）をセットアップすることでこなす。
リターンプログラムカウンタ値は、新しいプロセスのカーネルスレッドをまずforkretで実行しそしてtrapretで実行するようにする。（allocproc関数の最後のあたりを参照。コード上ではまず戻り先のtrapretを設定してから飛び先のforkretを設定してる。）

カーネルスレッドは、p->contextからコピーされたレジスタの内容で実行を開始する。
要するに、p->context->eipをforkretに設定することは、カーネルスレッドをforkretのはじめから実行することを引き起こす。（後に載せますが、forkretは関数です。つまり関数ポインタをセットするということになります。）
この関数は、スタックの下にあるアドレスにとにかく戻る。
コンテキストスイッチのコード（以下のswitch.S参照）は、p->contextの終わりのすぐ次を指すようにスタックをセットする。
allocprocは、スタック上にp->contextを置き、そしてtrapretへのポインタをすぐその上に置く。
それがforkretの戻り先となる。
trapretは、カーネルスタックのトップに保持された値からユーザレジスタを復元し、プロセスにジャンプする。（以下のtrapasm.S参照）
このセットアップは、通常のforkや最初のプロセスの生成と同じである。
ただ、最初のプロセスの生成の場合は、forkから返ってきたところからというよりむしろ、位置ゼロから実行を始める。

図1-3 新しいカーネルスタックのセットアップ



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 }
```

switch.Sのswitch（これでswitch.Sの全てです。）

```
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
29
30 trapasm.Sのtrapret
31 .globl trapret
32 trapret:
33   popal
34   popl %gs
35   popl %fs
36   popl %es
37   popl %ds
38   addl $0x8, %esp  # trapno and errcode
39   iret
```

第2章に載ってるが、ユーザソフトウェアからカーネルへの制御を移動する方法は、割り込みの仕組みを元にしている。
その仕組みは、システムコールや中断や例外で使われる。
プロセスが実行されてる間、制御をカーネルに移すときはいつでも、ハードウェアとxv6は、そのプロセスのカーネルスタックのトップにユーザレジスタを保存し、そのときとのコードをトラップする。
userinit関数は、もしプロセスが割り込み経由（userinit関数の中でp->tfの一連のメンバに値をセットしてる部分）でカーネルに入ったら、新しいスタックのトップに、そこにあるであろう彼らにそっくりな値を書き込む。
だから、カーネルからプロセスのユーザコードに戻る通常のコードはうまく動作する。
それらの値は、ユーザレジスタに保持されるtrapframe構造体である。
そうやってプロセスのカーネルスタックは、図1-3で示されるような準備完了状態になる。

最初のプロセスは、小さいプログラム（initcode.S）を実行する。
そのプロセスは、このプログラムを保持するための物理メモリが必要である。
そのプログラムは、そのメモリにコピーされる必要がある。
そしてそのプロセスは、そのメモリを指すページテーブルを必要とする。

initcode.S（これで全てです。）

```
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
```

userinit関数は、（最初は）カーネル用のメモリだけを対応付けるそのプロセスのためのページテーブ

ルを作成するためにsetupkvm関数を呼ぶ。
この関数はカーネルがそれ自身のページテーブルをセットアップするために使うのと同じである。
（以前、その11, 12, 14で出てきましたね。）

vm.cのsetupkvm関数

```
01 // Set up kernel part of a page table.
02 pde_t*
03 setupkvm(char* (*alloc)(void))
04 {
05   pde_t *pgdir;
06   struct kmap *k;
07
08   if((pgdir = (pde_t*)alloc()) == 0)
09     return 0;
10   memset(pgdir, 0, PGSIZE);
11   if (p2v(PHYSTOP) > (void*)DEVSPACE)
12     panic("PHYSTOP too high");
13   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
14     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
15                 (uint)k->phys_start, k->perm, alloc) < 0)
16       return 0;
17   return pgdir;
18 }
```

最初のプロセスのメモリの初期の内容は、initcode.Sのコンパイルされたものである。
カーネルがプロセスを作る過程において、リンカはそのバイナリをカーネルに埋め込み、その位置とサイズを伝えるために、二つの特別なシンボル_binary_initcode_startと_binary_initcode_sizeを定義する。
userinit関数は、inituvm関数を使ってそのバイナリを新しいプロセスのメモリにコピーする。
inituvm関数は物理メモリの1ページを割り当て、仮想アドレス0にマップし、そのページにそのバイナリをコピーする。

vm.cのinituvm関数

```
01 // Load the initcode into address 0 of pgdir.
02 // sz must be less than a page.
03 void
04 inituvm(pde_t *pgdir, char *init, uint sz)
05 {
06   char *mem;
07
08   if(sz >= PGSIZE)
09     panic("inituvm: more than a page");
10   mem = kalloc();
11   memset(mem, 0, PGSIZE);
12   mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U, kalloc);
13   memmove(mem, init, sz);
14 }
```

そういうわけで、userinit関数は初期のユーザモードの状態とともにトラップフレームをセットアップする。
%csレジスタは、特権レベルDPL_USER（例えばユーザモードはカーネルモードではない）で実行

されてるセグメントSEG_UCODEのためのセグメントセレクタを含む。
そして同様に、%ds, %es, %ssは、特権レベルDPL_USERと共にSEG_UDATAを使う。
%eflagsにFL_IFをセットしてハードウェア割り込みを許可する。
このあたりは第2章でまた説明する。
（この段落は、userinit関数の中でp->tfの一連のメンバに値をセットしてる部分の事を説明してるみたいです。）

スタックポインタ%espは、一番重要で有効な仮想アドレスp->szである。
その命令ポインタは、initcodeの開始点であるアドレス0である。

userinit関数は、主にデバッグのためにinitcodeにp->nameをセットする。
p->cwdが、そのプロセスの現在の作業ディレクトリに対応する。
namei（userinit関数内のp->cwd = namei("/");の部分）については第5章で説明する。

一度プロセスが初期化されたら、userinitは、p->stateをRUNABLEに設定することでスケジューリング可能である印を付ける。

感想

長かった…

プロセスの生成についての話ですが、主にproc構造体とuserinit関数を中心に説明されてます。

途中やたら指示代名詞が多くて訳どころかイメージを思い浮かべることさえ難儀した部分があります。
そもそも言葉で説明するのが難しいんでしょうね。
通常は、説明を頭に入れた上でソースを読むと理解が早かったりしますが、そういうところは一旦謎の説明は忘れて頑張ってソースをじっくり読み解くほうが理解への近道でしょう。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/19 日曜日 [http://peta.okechan.net/blog/archives/1285] |