日曜研究室

技術的な観点から日常を綴ります

[xv6 #18] Chapter 1 – The first process – Code: exec

テキストの27〜29ページ

本文

システムコールが到達したとき（第2章でどうやってそれが起きるか説明する）、syscallはsyscallテーブルを参照しsys_execを呼び出す。
sys_execはシステムコールの引数を解釈し、execを呼び出す。

syscall.cのsyscall関数とテーブル

```
01 static int (*syscalls[])(void) = {
02 [SYS_fork]    sys_fork,
03 [SYS_exit]    sys_exit,
04 [SYS_wait]    sys_wait,
05 [SYS_pipe]    sys_pipe,
06 [SYS_read]    sys_read,
07 [SYS_kill]    sys_kill,
08 [SYS_exec]    sys_exec,
09 [SYS_fstat]   sys_fstat,
10 [SYS_chdir]   sys_chdir,
11 [SYS_dup]     sys_dup,
12 [SYS_getpid]  sys_getpid,
13 [SYS_sbrk]    sys_sbrk,
14 [SYS_sleep]   sys_sleep,
15 [SYS_uptime]  sys_uptime,
16 [SYS_open]    sys_open,
17 [SYS_write]   sys_write,
18 [SYS_mknod]   sys_mknod,
19 [SYS_unlink]  sys_unlink,
20 [SYS_link]    sys_link,
21 [SYS_mkdir]   sys_mkdir,
22 [SYS_close]   sys_close,
23 };
24
25 void
26 syscall(void)
27 {
28   int num;
29
```

```
30    num = proc->tf->eax;
31    if(num >= 0 && num < SYS_open && syscalls[num]) {
32      proc->tf->eax = syscalls[num]();
33    } else if (num >= SYS_open && num < NELEM(syscalls) &&
    syscalls[num]) {
34      proc->tf->eax = syscalls[num]();
35    } else {
36      cprintf("%d %s: unknown sys call %d\n",
37              proc->pid, proc->name, num);
38      proc->tf->eax = -1;
39    }
40  }
```

sysfile.cのsys_exec関数

```
01  int
02  sys_exec(void)
03  {
04    char *path, *argv[MAXARG];
05    int i;
06    uint uargv, uarg;
07
08    if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
09      return -1;
10    }
11    memset(argv, 0, sizeof(argv));
12    for(i=0;; i++){
13      if(i >= NELEM(argv))
14        return -1;
15      if(fetchint(proc, uargv+4*i, (int*)&uarg) < 0)
16        return -1;
17      if(uarg == 0){
18        argv[i] = 0;
19        break;
20      }
21      if(fetchstr(proc, uarg, &argv[i]) < 0)
22        return -1;
23    }
24    return exec(path, argv);
25  }
```

execはnameiを使ってpathで名付けられたバイナリを開き（その辺の詳細は第5章で説明する）、そしてそのELFヘッダを読む。
xv6のアプリケーションは、広く使われてるELFフォーマットで表現されていて、それはelf.hで定義されている。
ELFバイナリは、ELFヘッダ（elfhdr構造体）から成り、その後にプログラムセクションヘッダの連なり（proghdr構造体）が続く。
どのproghdrも、メモリに読み込まれるべきアプリケーションの一部を記述している。
xv6のプログラムは一つだけしかプログラムセクションヘッダを持たないが、他のシステムでは命令とデータのための分割された一部を持つだろう。

exec.cの全て

```
001  #include "types.h"
002  #include "param.h"
003  #include "memlayout.h"
```

```
004 #include "mmu.h"
005 #include "proc.h"
006 #include "defs.h"
007 #include "x86.h"
008 #include "elf.h"
009
010 int
011 exec(char *path, char **argv)
012 {
013   char *s, *last;
014   int i, off;
015   uint argc, sz, sp, ustack[3+MAXARG+1];
016   struct elfhdr elf;
017   struct inode *ip;
018   struct proghdr ph;
019   pde_t *pgdir, *oldpgdir;
020
021   if((ip = namei(path)) == 0)
022     return -1;
023   ilock(ip);
024   pgdir = 0;
025
026   // Check ELF header
027   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
028     goto bad;
029   if(elf.magic != ELF_MAGIC)
030     goto bad;
031
032   if((pgdir = setupkvm(kalloc)) == 0)
033     goto bad;
034
035   // Load program into memory.
036   sz = 0;
037   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
038     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
039       goto bad;
040     if(ph.type != ELF_PROG_LOAD)
041       continue;
042     if(ph.memsz < ph.filesz)
043       goto bad;
044     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
045       goto bad;
046     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
047       goto bad;
048   }
049   iunlockput(ip);
050   ip = 0;
051
052   // Allocate two pages at the next page boundary.
053   // Make the first inaccessible.  Use the second as the user stack.
054   sz = PGROUNDUP(sz);
055   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
056     goto bad;
057   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
058   sp = sz;
059
060   // Push argument strings, prepare rest of stack in ustack.
061   for(argc = 0; argv[argc]; argc++) {
062     if(argc >= MAXARG)
063       goto bad;
064     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
065     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
066       goto bad;
067     ustack[3+argc] = sp;
```

```
068   }
069   ustack[3+argc] = 0;
070
071   ustack[0] = 0xffffffff;  // fake return PC
072   ustack[1] = argc;
073   ustack[2] = sp - (argc+1)*4;  // argv pointer
074
075   sp -= (3+argc+1) * 4;
076   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
077     goto bad;
078
079   // Save program name for debugging.
080   for(last=s=path; *s; s++)
081     if(*s == '/')
082       last = s+1;
083   safestrcpy(proc->name, last, sizeof(proc->name));
084
085   // Commit to the user image.
086   oldpgdir = proc->pgdir;
087   proc->pgdir = pgdir;
088   proc->sz = sz;
089   proc->tf->eip = elf.entry;  // main
090   proc->tf->esp = sp;
091   switchuvm(proc);
092   freevm(oldpgdir);
093   return 0;
094
095  bad:
096   if(pgdir)
097     freevm(pgdir);
098   if(ip)
099     iunlockput(ip);
100   return -1;
101 }
```

elf.hの全て

```
01 // Format of an ELF executable file
02
03 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
04
05 // File header
06 struct elfhdr {
07   uint magic;  // must equal ELF_MAGIC
08   uchar elf[12];
09   ushort type;
10   ushort machine;
11   uint version;
12   uint entry;
13   uint phoff;
14   uint shoff;
15   uint flags;
16   ushort ehsize;
17   ushort phentsize;
18   ushort phnum;
19   ushort shentsize;
20   ushort shnum;
21   ushort shstrndx;
22 };
23
24 // Program section header
25 struct proghdr {
26   uint type;
```

```
27    uint off;
28    uint vaddr;
29    uint paddr;
30    uint filesz;
31    uint memsz;
32    uint flags;
33    uint align;
34  };
35
36  // Values for Proghdr type
37  #define ELF_PROG_LOAD           1
38
39  // Flag bits for Proghdr flags
40  #define ELF_PROG_FLAG_EXEC      1
41  #define ELF_PROG_FLAG_WRITE     2
42  #define ELF_PROG_FLAG_READ      4
```

最初の一歩は、ファイルがELFバイナリを含むかどうかの簡易的なチェックである。
ELFバイナリは、4バイトのマジックナンバー0x7F, 'E', 'L', 'F'（elf.hに定義されてるELF_MAGIC）で始まる。
もしELFヘッダが正しいマジックナンバーを持つなら、execはそのバイナリを正しいという事にする。

execが、setupkvmでユーザ用のマッピング無しで新しいページテーブルを割り当てると、allocuvmでそれぞれのELFセグメントのためのメモリを割り当て、そしてloaduvmを使ってそれぞれのセグメントをメモリに読み込む。
/initのためのプログラムセクションヘッダは次のようになる。

```
1  # objdump -p _init
2  _init:     file format elf32-i386
3  Program Header:
4  LOAD off 0x00000054 vaddr 0x00000000 paddr 0x00000000 align 2**2
5                filesz 0x000008c0 memsz 0x000008cc flags rwx
```

allocuvmは要求された仮想アドレスがKERNBASE未満かどうかをチェックする。
loaduvmはELFセグメントのページを書き込むための割り当て済み物理メモリをwalkpgdirを使って探し、readiを使ってファイルから読み込む。

vm.cのallocuvmとloaduvm

```
01  // Load a program segment into pgdir.  addr must be page-aligned
02  // and the pages from addr to addr+sz must already be mapped.
03  int
04  loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset,
    uint sz)
05  {
06    uint i, pa, n;
07    pte_t *pte;
08
09    if((uint) addr % PGSIZE != 0)
10      panic("loaduvm: addr must be page aligned");
11    for(i = 0; i < sz; i += PGSIZE){
12      if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
13        panic("loaduvm: address should exist");
```

```
14      pa = PTE_ADDR(*pte);
15      if(sz - i < PGSIZE)
16        n = sz - i;
17      else
18        n = PGSIZE;
19      if(readi(ip, p2v(pa), offset+i, n) != n)
20        return -1;
21    }
22    return 0;
23  }
24
25  // Allocate page tables and physical memory to grow process from
    oldsz to
26  // newsz, which need not be page aligned.  Returns new size or 0 on
    error.
27  int
28  allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
29  {
30    char *mem;
31    uint a;
32
33    if(newsz >= KERNBASE)
34      return 0;
35    if(newsz < oldsz)
36      return oldsz;
37
38    a = PGROUNDUP(oldsz);
39    for(; a < newsz; a += PGSIZE){
40      mem = kalloc();
41      if(mem == 0){
42        cprintf("allocuvm out of memory\n");
43        deallocuvm(pgdir, newsz, oldsz);
44        return 0;
45      }
46      memset(mem, 0, PGSIZE);
47      mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U,
    kalloc);
48    }
49    return newsz;
50  }
```

プログラムセクションヘッダのfileszはmemszより多分小さくなり、その差はファイルから読み込まれる代わりにゼロで埋められていなければならない（Cのグローバル変数のため）。
/initの場合、fileszは2240バイトでmemszは2252バイトであり、allocuvmは2252バイトを保持するのに十分な物理メモリを割り当てるが、/initファイルから読み込まれるのは2240バイトのみである。
（fileszやmemszというのはELFバイナリの中のプログラムセクションヘッダごとに定義されてる値）

そしたら、execは1ページ分だけユーザスタックを割り当て初期化する。
そして、プログラムが1ページより多く使おうとしたら失敗するようにするために、そのスタック用のページの下位にアクセス出来ないページを一つ配置する。
このアクセス出来ないページは、長すぎる引数にexecが対処することもまた可能にする。
そういうとき（長すぎる引数が渡されたとき）は、execが引数をスタックにコピーするために使ってるcopyout関数が、コピー先のページがアクセス不能である事を検知し、-1が返る。

execは、一度に一つずつスタックのトップに引数の文字列をコピーし、そこに対するポインタをustackに記録する。

そして、mainに渡されるargvのリストになるであろうものの最後にヌルポインタを置く。
最初の3つのustackの項目は、argvとargcのポインタとPCへの偽のリターン（fake return PC）である。

新しいメモリイー目地を用意している間、もしexecがプログラムセグメントの異常のようなエラーを検出したら、badラベルにジャンプし、その新しいイメージを解放し、-1を返す。
execは、古いイメージを解放するのをそれが確実になるまで待たなければならない。
もし古いイメージの解放が完了したら、-1を返す事はない。
execにおけるエラーは、イメージの生成をしてる間だけ起きる。
一度イメージが完了設楽、execは新しいイメージをインスールし古いイメージを解放することが出来る。
（exec関数のbadラベルの直前の部分。switchuvmとfreevm）
最後に、execは0を返す。
成功だ！

これで、initcodeは完了した。
execは、それを/initバイナリで置き換え、loaded out of the file system.（ここ分からない）
initは、必要なら新しいコンソールデバイスファイルを生成し、そしてそれをファイルディスクリプタ0,1,2として開く。
そしてループし、コンソールシェルを開始し、シェルが終了するまで親のないゾンビを制御し、繰り返す。
これでシステムは起動を完了した事になる。

init.cの全て

```
01 // init: The initial user-level program
02
03 #include "types.h"
04 #include "stat.h"
05 #include "user.h"
06 #include "fcntl.h"
07
08 char *argv[] = { "sh", 0 };
09
10 int
11 main(void)
12 {
13   int pid, wpid;
14
15   if(open("console", O_RDWR) < 0){
16     mknod("console", 1, 1);
17     open("console", O_RDWR);
18   }
19   dup(0);  // stdout
20   dup(0);  // stderr
21
22   for(;;){
23     printf(1, "init: starting sh\n");
24     pid = fork();
25     if(pid < 0){
```

```
26        printf(1, "init: fork failed\n");
27        exit();
28      }
29      if(pid == 0){
30        exec("sh", argv);
31        printf(1, "init: exec sh failed\n");
32        exit();
33      }
34      while((wpid=wait()) >= 0 && wpid != pid)
35        printf(1, "zombie!\n");
36    }
37  }
```

システムが起動完了したとはいえ、xv6の重要なサブシステムをいくつか飛ばして説明してきた。
次の章では、int $T_SYSCALLによって引き起こされる割り込みシステムコールを制御させるためにx86ハードウェアをどうやってxv6が設定するかについて説明する。
残りの章では、プロセスの管理とファイルシステムについて説明する。

感想

exec中心の話です。
最初のプロセスの起動にもexecを使うんですね。
もちろんこれまでの節で、最初のプロセスといえどなるべく通常のプロセスと同じように標準のシステムコールを利用しているという感じの説明があったのでそう驚くようなことではないかもしれません。

途中の説明で古いイメージの解放の部分に一瞬おや？となりましたが、そもそもexecはファイルディスクリプタ部分などを残して元のイメージを新しいイメージで置き換えるのが通常の動作です。
（なので通常使うときはその前にforkする必要がありましたね）
解放の話が入ってくるのも当然ですね。

この章の説明部分は終わりですが、あとReal worldという節とExercisesという節が残ってます。
一応全部やるつもりですがExercisesは、やろうと思えばいくらでも掘り下げれそうな気がするので、サラッと程度にしとくつもりです。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/23 木曜日 [http://peta.okechan.net/blog/archives/1307] |