日曜研究室

技術的な観点から日常を綴ります

[xv6 #2] Chapter 0 – Operating system interfaces – Code: Processes and memory

テキストの8〜10ページ

概要

xv6のプロセスは、命令とデータとスタックを置くためのユーザー空間のメモリ領域と、カーネルで管理されるプロセスの状態から成る。
xv6はタイムシェアリングを提供していて、それは実行を待ってるプロセス群の間で使えるCPUを自動的に切り替える。プロセスが実行されてないときはそのレジスタの内容を保存して、次にプロセスが再開されるときに読み込む。
どのプロセスも一意に識別可能な正の整数がついてて、それはプロセスIDもしくはpidと呼ばれる。

あるプロセスは、forkというシステムコールを使って新しいプロセスを生成することが出来る。
forkは新しいプロセス（子プロセスと呼ばれる）を生成し、そのプロセスは元のプロセス（親プロセス）と全く同じメモリの内容を持つ。
forkは親プロセスでも子プロセスでも返り値を返す。
親プロセスでは、子プロセスのpidを返す。
子プロセスでは、ゼロを返す。
次のプログラムの断片について考えてみよう。

```
01 int pid;
02 pid = fork();
03 if(pid > 0){
04   printf("parent: child=%d\n", pid);
05   pid = wait();
06   printf("child %d is done\n", pid);
07 } else if(pid == 0){
08   printf("child: exiting\n");
09   exit();
10 } else {
11   printf("fork error\n");
12 }
```

exitというシステムコールは、それを呼んだプロセスの実行を止めて、メモリや開いてるファイルな

どのリソースを解放する。
waitというシステムコールは、現在のプロセスの子プロセスが終了したときにそのpidを返す。
まだ子プロセスが終了してなかったら終了するまで待つ。
上記の例はまず次のような結果を印字する。

```
1 parent: child=1234
2 child: exiting
```

親か子、どちらのprintfが先に呼ばれたかによるので順番は上記の限りではない。
で、その二つの印字のあと、子プロセスは終了し、親プロセスのwaitシステムコールが返ってくる。
その結果、親プロセスはさらに下記を印字する。

```
1 parent: child 1234 is done
```

親プロセスと子プロセスは内容は同じなれど違うメモリ領域、違うレジスタで実行されるので、一方の変更がもう一方に影響しないことに注意。

execシステムコールは、呼び出し元のプロセスのメモリを新しいメモリイメージ（ファイルシステムに置かれたファイルから読み込まれたもの）で置き換える。
そのファイルは、どの部分が命令で、どの部分がデータか、さらにどの命令から開始するか等を指定する、特定のフォーマットに従ってなければならない。
xv6はそのフォーマットとしてELFと呼ばれるものを使う。（その詳細についてはChapter 1で）
execが成功したら、呼び出し元のプログラムへは戻らない。
その代わり、読み込まれた命令をELFヘッダに定義されたエントリポイントから実行する。
execは二つの引数（実行可能なファイルの名前とその引数の配列）を受け取る。
例えば、以下のような感じである。

```
1 char *argv[3];
2 argv[0] = "echo";
3 argv[1] = "hello";
4 argv[2] = 0;
5 exec("/bin/echo", argv);
6 printf("exec error\n");
```

このプログラムは、自分自身を、引数 echo hello と共に実行される /bin/echo というプログラムで置き換える。
多くのプログラムは最初の引数を無視する。
習慣的に最初の引数はプログラムの名前である。

xv6のシェルは、以上のような呼び出しを使って、ユーザーの代わりにプログラムを実行する。
シェルの主な構造はシンプルである。（sh.cのmain関数参照。以下にとりあえず載せときます。）

```
01 int
02 main(void)
03 {
04   static char buf[100];
05   int fd;
```

```
06
07    // Assumes three file descriptors open.
08    while((fd = open("console", O_RDWR)) >= 0){
09      if(fd >= 3){
10        close(fd);
11        break;
12      }
13    }
14
15    // Read and run input commands.
16    while(getcmd(buf, sizeof(buf)) >= 0){
17      if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
18        // Clumsy but will have to do for now.
19        // Chdir has no effect on the parent if run in the child.
20        buf[strlen(buf)-1] = 0;  // chop \n
21        if(chdir(buf+3) < 0)
22          printf(2, "cannot cd %s\n", buf+3);
23        continue;
24      }
25      if(fork1() == 0)
26        runcmd(parsecmd(buf));
27      wait();
28    }
29    exit();
30  }
```

メインのループでgetcmdを使ってコマンドラインから入力を読み取っている。
そしてforkを呼んで、別のシェルプログラムを生成する。
子プロセスがコマンドを実行してる間、親のシェル（親プロセス）はwaitを呼んで待つ。
例えば、もしユーザがが”echo hello”と入力したら、runcmdは”echo hello”を引数として呼ばれる。
runcmdは以下のとおり。（sh.cのruncmd関数）

```
01  void
02  runcmd(struct cmd *cmd)
03  {
04    int p[2];
05    struct backcmd *bcmd;
06    struct execcmd *ecmd;
07    struct listcmd *lcmd;
08    struct pipecmd *pcmd;
09    struct redircmd *rcmd;
10
11    if(cmd == 0)
12      exit();
13
14    switch(cmd->type){
15    default:
16      panic("runcmd");
17
18    case EXEC:
19      ecmd = (struct execcmd*)cmd;
20      if(ecmd->argv[0] == 0)
21        exit();
22      exec(ecmd->argv[0], ecmd->argv);
23      printf(2, "exec %s failed\n", ecmd->argv[0]);
24      break;
25
26    case REDIR:
27      rcmd = (struct redircmd*)cmd;
28      close(rcmd->fd);
29      if(open(rcmd->file, rcmd->mode) < 0){
```

```
30        printf(2, "open %s failed\n", rcmd->file);
31        exit();
32      }
33      runcmd(rcmd->cmd);
34      break;
35
36    case LIST:
37      lcmd = (struct listcmd*)cmd;
38      if(fork1() == 0)
39        runcmd(lcmd->left);
40      wait();
41      runcmd(lcmd->right);
42      break;
43
44    case PIPE:
45      pcmd = (struct pipecmd*)cmd;
46      if(pipe(p) < 0)
47        panic("pipe");
48      if(fork1() == 0){
49        close(1);
50        dup(p[1]);
51        close(p[0]);
52        close(p[1]);
53        runcmd(pcmd->left);
54      }
55      if(fork1() == 0){
56        close(0);
57        dup(p[0]);
58        close(p[0]);
59        close(p[1]);
60        runcmd(pcmd->right);
61      }
62      close(p[0]);
63      close(p[1]);
64      wait();
65      wait();
66      break;
67
68    case BACK:
69      bcmd = (struct backcmd*)cmd;
70      if(fork1() == 0)
71        runcmd(bcmd->cmd);
72      break;
73    }
74    exit();
75  }
```

runcmdは実際のコマンドを実行する。
“echo hello”の場合、runcmd関数の中のcase EXEC:の中のexecが呼ばれる。
もしexecが成功したら、その子プロセスはruncmdを途中で投げ出してechoの（実行ファイルから読み込んだ）命令を実行する。
そしてどこかの段階でechoはexitを呼ぶだろう。
そうすることで親プロセスはmain関数のwaitから復帰する事が出来る。
forkとexecがなぜ一度の呼び出しで行えないか不思議に思うかもしれない。
プロセスの生成とプログラムのロードが分かれてる事が賢いデザインだと後で気づくだろう。

xv6は暗黙のうちに最大量のユーザ空間メモリを割り当てる。
forkは親プロセスのメモリの複製のために必要となる子プロセスのメモリを割り当てる。

そしてexecは実行ファイルを保持するのに十分なメモリを割り当てる。
（mallocなどで）実行中にメモリがさらに必要になったプロセスは、データメモリをnバイトまで増やすためにsbrk(n)を呼ぶことができる。
sbrkは新しいメモリの場所を返す。

xv6はユーザの概念は提供しないので、ファイルやプロセスにユーザを結びつけて他のユーザから保護するという事は出来ない。
Unix的に言えば、全てのxv6のプロセスはroot権限で実行される。

感想

意外だなと思ったのがexecの動作です。
呼び出し元のメモリ読み込んだプログラムの実行イメージで”置き換える”ワケですね。
なのでその前にforkしてると。
execでそのまま別のプロセスを開始出来ればforkする必要もないと思うんですが、テキストにもforkとexecが分かれてるのには理由がある。それは後ほど乞うご期待と書いてあるので追々分かってくるのでしょう。

ソースがちょろっと出てきました。
シェルのソースの一部です。
前回、シェルはユーザ空間で実行される通常のプログラムである、と書かれてましたので、まだカーネル部分を読み始めるというところまでは至ってないということになります。

あと、xv6にはユーザの概念がないと書かれています。
ユーザの概念があると、管理やセキュリティのために色々な概念を導入せざるを得ず、ソースが複雑でより理解しづらいものになるのでしょう。
そのへん気にしなくていいということは、とりあえず読むという身にとってはありがたいことです。

sh.cのruncmd関数を見ると、switchの最初にdefaultを置いてあります。
defaultって最初でもいいのかー！ってのが今回一番の驚きです。

ほとんど訳してしまいました。
たぶん今回に限らずですが、翻訳は超適当ですのでご注意ください。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/7 火曜日 [http://peta.okechan.net/blog/archives/1220] |