プログラミング環境 第12回

久野 靖 *

1992.1.21

18 プロセスを操作するシステムコール

18.1 新しいプロセスを生成する

あけましておめでとう (だいぶ遅いかな?)。で、今回はようやく Unix らしくプロセスをいじったりするのだが、その前にプロセスとは何でしたっけ?(忘れているでしょ?)

で、Unix では新しいプロセスのでき方は常に、既にあるプロセスが 2 つに分裂する、という方法によっている (ちょうど細胞分裂のようなものですね)。皆様の書いたプログラムは既によくご存じのように、1 つのプロセスとして走り出すわけだから、プロセスを増やしたいときは「ここで分裂する」というシステムコール (`fork()` という–フォークって先が分裂しているでしょ?) を呼び出す。次のような具合である。

```
/* t12.c -- create process. */
main() {
  int pid = fork();
  printf("Hello.\n");
}
```

さて、これで何が起きると思うか?

```
% cc t12.c
% a.out
Hello.
Hello.
%
```

予想は当たりましたか? もちろん、細胞分裂した 2 つのプロセスがそれぞれ "Hello" と打つのだから、2 回打てるわけである。

18.2 親と子の区別

にしても、このままではいくら分裂してあたらしいプロセスができても、全部がおんなじことをやるだけでちっとも面白くない。じゃあどうしたらいいだろう? 実は、分裂したあとの 2 つのプロセスは全く同じようだが同じでないところがある (何か?)。もちろん PID である。分裂したあとの 2 つのプロセスのうち片方はもとのプロセスと同じ PID、もう一つは... 新しい PID で、そしてそのプロセスの親を見るともとのプロセスということになっている。つまり図 1 に示すように、`fork()` というのは「分裂する」というよりも、「それを呼んだ時点での親のそっくりコピーである子プロセスを作り出す」という方が正確である。その様子はどうやったら観察できるか?まあ、次のを見てみよう。

```
/* t13.c -- create process, and sleep. */
main() {
  int pid = fork();
  sleep(10);
}
```

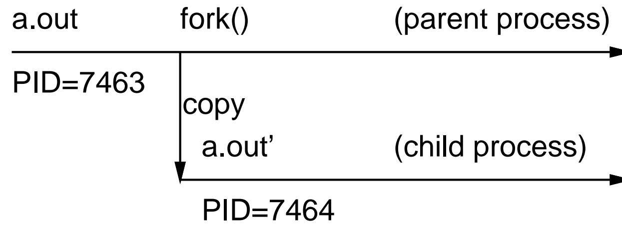

図 1: fork() の意味づけ

```
% cc t13.c
% a.out &
[1] 7463
% ps lx
   F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
8201  20  7456  7455  4  15  0  48  424 pause    S    p4  0:00 -csh
8201  20  7463  7456  6  15  0  24  176 pause    S    p4  0:00 a.out
8201  20  7464  7463  0  15  0  24  136 pause    S    p4  0:00 a.out
   1  20  7465  7456 25  31  0 152  416          R    p4  0:00 ps lx
%
```

確かに a.out が 2 つあり、1 つは csh の子供で、もう 1 つはその 1 つ目の子供になっていることがわかる。

では、プログラムの中では親と子はどうやって区別したらいいだろう? (分かりますか?) 実は、fork() が返す値というのは親と子では違っている!子にとっては fork() の値は 0 なのだが、親にとっては fork() の値はできた子プロセスの PID になっている。だから、その値に応じて分岐することにより、子供と親とで別の仕事をすることが可能なわけである。たとえば次のような具合である。

```
/* t14.c -- create process, distinguish parent and child. */
main() {
  int pid = fork();
  if(pid != 0) {
    printf("I'm a parent.  child pid = %d.\n", pid);
    sleep(10); }
  else {
    sleep(5);
    printf("I'm a child (pid = %d).\n", pid); }
}
```

これは何をすると思うか? さっそく見てみよう。

```
% cc t14.c
% a.out
I'm a parent.  child pid = 7432.
I'm a child (pid = 0).
%
```

これだけではつまらないので、ps で観察してみる。

```
% a.out &
[1] 7433
% I'm a parent.  child pid = 7434.
ps lx
   F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
8201  20  7424  7423  3  15  0  48  424 pause    S    p4  0:00 -csh
```

*筑波大学経営システム科学専攻

```
8201  20  7433  7424  8  15  0  40  208 pause    S    p4  0:00 a.out
8201  20  7434  7433  0  15  0  24  136 pause    S    p4  0:00 a.out
   1  20  7435  7424 24  31  0 152  416          R    p4  0:00 ps lx
% I'm a child (pid = 0).
```

確かに、子のほうは PID=7434 で、親が打ち出した値の通りである。

18.3 プロセスの待ち方

さて、先のプログラムは親も子も適当な頃合いを見計らって出力したり終ったりする、といういい加減なものだったが、現実の仕事ではそれではちょっと困る。そこで、親プロセスが子プロセスの終了を待つ、というシステムコール `wait()` が用意されている。これを使った例を示す:

```
/* t15.c -- the parent wait for the child. */
main() {
  int pid, wpid;
  printf("parent start.\n");
  if((pid = fork()) == 0) {
    printf("child start.\n"); sleep(5); printf("child end.\n");
    exit(0); }
  wpid = wait(0);
  printf("parent end.  pid = %d, wpid = %d.\n", pid, wpid);
}
```

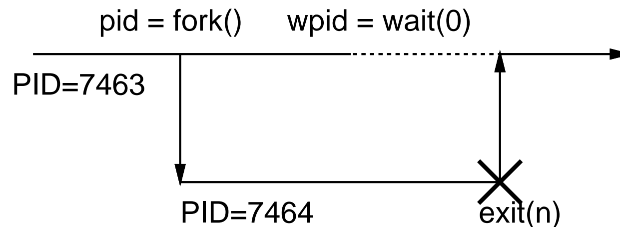

図 2: fork() と wait()

`wait` システムコールは、1 個引数を取り、1 個値を返す。引数の方はもし待ち合わせた子プロセスに関して詳しい情報が知りたければその情報を入れる場所へのポインタを渡すのだが、ここではとりあえずそういう情報はもらわなくていいことにしたので 0 を渡す。一方、値の方は終了した子プロセスの PID である。さっそく動かしてみよう。

```
% cc t15.c
% a.out
parent start.
child start.
child end.
parent end.  pid = 8772, wpid = 8772.
%
```

この通り、ちゃんと子プロセスが終わると親プロセスの `wait()` が完了することがわかる。ところで、この `exit(0);` がないとどういうことが起きるか分かりますか?

18.4 プロセスの成り換わり方

さて、ここまでのやり方だと確かに親と子では違う動作ができるものの、親も子も `a.out` という同じプログラムを実行していることに変わりはない。そうではなくて、別の新しいプログラム

を起動する方法がないと困りますよね?実は Unix ではそういうことがしたい場合には、あるプロセスが別のプログラムのプロセスに**成り換わる**というシステムコール exec を利用する。PID は成り換わったあとでも元と同じである。つまり、exec はプロセスはそのままにその上で走っているプログラムを取り替えるシステムコールだ、と考えれば良い。

実は exec にはいくつかの異なる呼び出し方のものが用意されているが、代表を 2 つだけ挙げておく:

```
execl(パス名, 文字列 0, 文字列 1, ... , 文字列 n, 0);
execv(パス名, argv); -- ただし、argv は文字列の配列で最後に 0。
```

どの形式もまず実行すべきプログラムを指定し、その後にそのプログラムに渡すコマンド引数を指定することに変わりはない。では、一番簡単な例を見ていただこう。

```
/* t16.c -- the simplest example of exec. */
main() {
  execl("/bin/ls", "ls", "-l", 0);
}
```

このプログラムは始まるとすぐ ls に成り換わってしまうだけである。ちなみに"ls"が 2 箇所に現れるのが奇異に思えるかも知れないが、最初の文字列は実行すべきプログラムのパス名であり、その後はそのプログラムが走り出した時の argv[0],argv[1],... になる。(argv[0] がプログラム名、という習慣になっていたのを覚えていますよね?) では実行例。

```
% cc t16.c
% ls
a.out   t16.c
% a.out
total 25
-rwxr-xr-x  1 kuno        24576 Jan 12 11:53 a.out
-rw-r--r--  1 kuno           91 Jan 12 11:52 t16.c
%
```

このように、execl() はプログラムの中でその場に引数文字列を書くのに便利な形式である。一方、プログラムの中で引数を組み立てたりしたい場合には execv() 形式が便利である。次のプログラムは上のプログラムを改良してファイル名が指定できるようにしたものである。

```
/* t17.c -- handling argv[] in my program. */
main(argc, argv)
  int argc; char *argv[]; {
  int i, myargc = 0;
  char *myargv[100];
  myargv[myargc++] = "ls";
  myargv[myargc++] = "-l";
  for(i = 1; i <= argc; ++i) myargv[myargc++] = argv[i];
  execv("/bin/ls", myargv);
}
```

このプログラムは myargv[0] は"ls"、myargv[1] は"-l"、その先は argv[1] 以下のコピーとしてからこれを渡して ls を起動するだけである。

```
% cc t17.c
% a.out a.out
-rwxr-xr-x  1 kuno        24576 Jan 12 12:05 a.out
% a.out t17.c t16.c
-rw-r--r--  1 kuno           91 Jan 12 11:52 t16.c
-rw-r--r--  1 kuno          274 Jan 12 12:04 t17.c
%
```

確かに、指定したファイルだけの ls -l が取られている。

19 プロセスとファイルディスクリプタ、パイプ

19.1 入出力の切替え

さて、なんで「1 つのプロセスでありながら成り換わる」というのがいいのだと思いますか? 実はこのしくみを使うと入出力の切替え (シェルで<、>で指定するやつ) はとっても簡単にできる。次のプログラムを見ていただこう。

```
/* t18.c -- redirect stdin and stdout. */
#include <stdio.h>
#include <fcntl.h>

main(argc, argv)
  int argc; char *argv[]; {
  int d0, d1;
  if(argc < 4) { fprintf(stderr, "too few arguments.\n"); exit(1); }
  close(0); d0 = open(argv[1], O_RDONLY);
  if(d0 < 0) { perror("input file: "); exit(1); }
  close(1); d1 = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, 0644);
  if(d1 < 0) { perror("output file: "); exit(1); }
  execvp(argv[3], argv + 3);
}
```

このプログラムは、まず `argv[1]` と `argv[2]` で指定されたファイルを開く。ときに、`open` はディスクリプタ番号として今使われていない一番若い番号を選ぶようになっているので、`close(0);` た直後の `open` は必ず 0 番にファイルをつなげることになる。1 でも同様。従って、最後の行に来

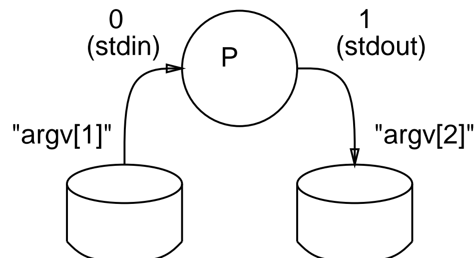

図 3: execvp() の直前でのディスクリプタ

る時はディスクリプタの 0 番と 1 番は図 3 のようにファイルにつながっていることになる。この状態で成り換わると... ときに、前の例では成り換わるプログラムとして絶対パス名を指定したが、`execvp` だと `sh/csh` のように`$path` を参照して探してくれる。従ってこの `execvp` で `argv[3]` に指定した指令を実行することになる。その引数は `argv[3], argv[4],...` ということになる。では結果。

```
% cc t18.c
% cat t
How are you?
% a.out t t1 tr a-z A-Z
% cat t1
HOW ARE YOU?
%
```

つまり、ご存じ `tr a-z A-Z` が実行されるのだが、ただし入力はファイル `t` から取られ、出力はファイル `t1` にできたわけである。

19.2 パイプ

入出力の切り替えができたら、次はパイプである。`Unix` ではパイプはその名もズバリ `pipe` システムコールで作る。`pipe` システムコールには大きさ 2 の整数配列を渡す。すると、図 4 上のよ

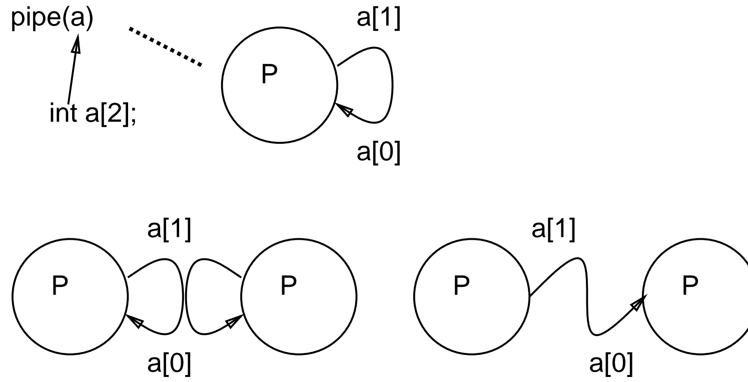

図 4: パイプの作り方と配線

うに、例えば配列名が a だとすると、ディスクリプタ番号 a[1] から書くと a[0] からそれが読めるようなパイプが作られる。これだけだと自分で書いて自分で読むしかないのであまり面白くないが、そこで fork() すれば図 4 下左のように 2 つのプロセスの間をまたがったパイプになる。余計な線が邪魔なら close してしまえば図 4 下右のように普通の形の (?) パイプになる。ではさっそく例題。

```
/* t19.c -- creating a pipe. */
#define BUFSIZE 1024
char buf[BUFSIZE];

main() {
  int n, pid, a[2];
  pipe(a);
  if((pid = fork()) == 0) {
    while((n = read(0, buf, BUFSIZE)) > 0) {
      write(a[1], buf, n); write(a[1], buf, n); }
  }
  else {
    while((n = read(a[0], buf, BUFSIZE)) > 0) {
      write(1, buf, n); write(1, buf, n); }
  }
}
```

さて、このプログラムは何をすると思いますか?

```
% cc t19.c
% cat t
How are you?
% a.out <t
How are you?
How are you?
How are you?
How are you?
%
```

つまり、子は stdin から読んだものを 2 回パイプに書き、親はパイプから読んだものを 2 回 stdout に書くのでこうなるわけである。

19.3　組み合わせると...

さて、それでは別々のプログラムのパイプラインを作るにはどうしたらいいか?　それは、上のようにして配線した後でそれぞれ exec を使って成り換わればいいのである。例えば下のプログラ

ムは

```
tr a-z A-Z | sed 's/[AEIOU]/**/g'
```

という指令行と同じことをやる。

```
/* t20.c -- pipe and exec. */
#define BUFSIZE 1024
char buf[BUFSIZE];

main() {
  int pid, a[2];
  pipe(a);
  if((pid = fork()) == 0) {
    dup2(a[1], 1); close(a[0]); close(a[1]);
    execl("/bin/tr", "tr", "a-z", "A-Z", 0); }
  else {
    dup2(a[0], 0); close(a[0]); close(a[1]);
    execl("/bin/sed", "sed", "s/[AEIOU]/**/g", 0); }
}
```

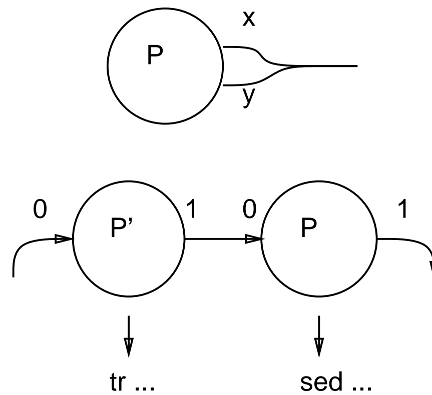

図 5: dup を使った番号かえ

ちなみに、dup2(x, y) というシステムコールは図 5 上のように、ディスクリプタ番号 x と同じものを番号 y でも使えるようにする。これを使って子プロセスについてはパイプの入り口を 0 に配線し、親についてはパイプの出口を 1 に配線した後でいらない口は全部 close してしまう。そしてその後それぞれが tr、sed に成り換わればいいわけである。

```
% cc t20.c
% cat t
How are you?
% a.out <t
H**W **R** Y****?
%
```

さて、ここでようやく「プロセスが分裂する」と「新しいプログラムに成り換わる」という一見おかしな機能が Unix ではプロセスを扱う基本になっているのかがわかったと思う。つまり、fork() した時点ではまだ親子は互いに相手が何であるかよく分かっているので、ここで様々な細工をすることができる。そして準備ができたところで新しいプログラムに取り換わることで、そのプログラム自身は「自分がどういう細工をされているか」など気にしないでも素直に走ることができるのであった。

A　演習および課題

今回もものによってはかなり資源を消費するので、他人の迷惑にならないように注意してください。

A.1　18節の演習

18-1. t12.c を変更して、16回 Hello と言うようにしてみよ。2のべき乗じゃ簡単だから、10回なんていうのはどうかな?そのプログラムのふるまいに予想外な点があれば (たぶん何か1つくらいあると思うが)、その理由を考えてみよ。

18-2. ただ適当な秒数寝るだけの子プロセスを n 個作り出すようなプログラムを書け。[1] そしてできれば、あなたの使っているシステムではいくつくらいプロセスが作り出せるか実験してみよ。n がオプションとして指定できるとかっこいいと思うよ。[2]

18-3. t15.c で、exit(0); を忘れるとどういうことが起きるか (どのような出力が得られるか、またその理由は何故か) を考え、実際その通りになるか確認してみよ。[3]

18-4. t15.c を変更して、n 個子供を作り、それらの子供が全部終るまで待つようにしてみよ。また子供が終る順番は作られた順番と一致するかどうか調べよ。さらに、sleep() で寝る子供ではなく、計数ループで時間を潰す子供だとどうか?

18-5. 通常は wait() は親プロセスが子プロセスの完了まで待つために使われるが、逆に親プロセスが wait() を実行するよりも子プロセスが終るのが早いとどういうことが起きるか? [4] また、その極端な場合として、親が結局 wait() を実行してくれることなく終ってしまった場合はどういうことが起きるか?それらはどういう考えのもとにそういう風になっているのか?実験を行ない疑問に答えよ。

A.2　19節の演習

19-1. 分裂したあとで、親と子とが同じディスクリプタ番号に書いたり、同じディスクリプタ番号から読もうとしたりするとどういう風になるか?まず結果を予想し、それを確かめる実験を行なえ。

19-2. 複数 (1個でもよい!) のプロセスをパイプでぐるっと輪につないで、データがぐるぐる巡回する、というのを作ってみよ。データがだんだん増殖するようにするとどうなるか。やってみる前に予測を立てること。

19-3. パイプでつながったプロセスで、読む方が書く方より速かったらどうなるか。またその逆だとどうなるか。プログラムを書いて実験せよ。答えはできるだけ定量的に求めること。

19-4. 指定した指令を指定した回数だけ繰り返し実行する、というプログラムを作ってみよ。例えば「a.out 10 ps x」というと ps x が10回実行されるわけ。できればn個の実行が一斉に起きる版と順番に1個ずつ起きる版と両方作ってみよ。

---

[1] 子プロセスの仕事が終ったところで exit(0); しないと恐ろしいことになるから注意!
[2] オプションの文字列を整数にするには関数 atoi(s) を呼ぶ。「man atoi」で調べてみよ。
[3] wait() のマニュアルページをよく読めば完全に推理できます。
[4] 親が wait() してくれる前に子があとかたもなく無くなってしまったら困るような気がしませんか?

19-5. 上と同様だが、指定した指令がずらっと指定した数だけパイプラインで直列に並ぶ、というのを作ってみよ。例えば、`a.out 3 sed 's/a/aa/g'` というと、入力の中の「`a`」が全部 8 個の `a` になるわけ。あまり役に立たないかな?

A.3 本日の課題

本日の課題は、`18-1.`〜`18-5.`、および `19-1.`〜`19-5.` あわせたうちから 2 問選択とします。