計算機プログラミング'95 ＃3

久野　靖*

1995.9.21

1　プロセスの操作

1.1　プロセスの生成

Unix の 1 つの特徴として、プロセスを多数使ったプログラムが比較的簡単に作れるということがある。(そのことは、どういう利点をもたらすか?)　今回はその各種側面を見て見よう。

まず、Unix ではプロセスを作るのは常に**既存のプロセスが 2 つに分裂する**ことによって行う。つまり「卵」や「胎児」よりは「細胞分裂」だと思えばいい。そのためには具体的には fork() というシステムコールを使う。

```
/* t31.c -- simplest fork() example */

main() {
  int pid;
  pid = fork();
  printf("Hello.\n");
}
```

しかし、このままでは分裂した 2 つのプロセスが同一物なので、まったく同じ動作しかできない。プロセスを作るからには、複数プロセスでうまく「分業」したいはずである。

そのヒントは、fork() の返り値にある。プロセスを生成するときには、親プロセスと子プロセスがあるという話をしましたね?　実は、fork() で分裂した後親プロセスも子プロセスもその次の命令から実行を続けるが、ただし fork() の返り値だけは親と子で違っていて、親では子の PID、子では 0 が返される。上のプログラムでは変数 pid を参照していないが、その値を見れば「自分が親か子か」を区別できる。

練習 1　上のプログラムをそのまま動かせ。また、fork() の回数を増やしてみよ。

練習 2　printf の前に「sleep(10);」を入れてすぐには終わらないようにして、実行中に別の窓から ps でプロセスの様子を観察してみよ。また、printf で pid を打つようにしてみよ。

練習 3　親と子で別のメッセージを打つようにしてみよ。

練習 4　さらに、子供をちょうど 10 個作るようにしてみよ。

なおヒントだが、プロセスを終了させるにはその中で「exit(0);」を実行させればよい。前に言いましたね?

*筑波大学大学院経営システム科学専攻

1.2 待ち合わせ

さて、fork() で分裂するだけでは、分かれっぱなしであとは勝手に実行するしかない。それでは不便なので、同期を取るための手段として「親プロセスが子プロセスの実行終了を待つ」というシステムコール wait が用意されている。使い方は「wait(0)」とすると、子プロセスが 1 つ終わるまで待ち、その子プロセスの PID を返す。

```
/* t32.c -- fork and wait */

main() {
  int pid, wpid;
  if(pid = fork()) { /* parent */
    int wpid = wait(0);
    printf("child pid=%d\n", wpid);
  } else { /* child */
    printf("Hello.\n");
    sleep(5);
  }
}
```

練習 5 上のプログラムをそのまま動かせ。また、子供を 10 個作って全部終わるまで待つように直して見よ。

1.3 exec システムコール

これまでのところでは、プロセスは増えてもすべて同じプログラムを実行しているだけだった。次に、プロセスは増えないがその代わりに別のプログラムに「とりかえる」システムコール exec を学ぶ。exec にはいく通りかの呼び出し方が用意されているが、次の 2 つが基本である。

```
execl("パス名", "引数 0", "引数 1", …, 0);
execv("パス名", argv);  --- ただし argv は文字列の配列で最後が 0。
```

「パス名」は実行したいプログラムの入ったファイルの絶対パス名である。execl ではその後に引数を順次書くので、プログラム中にそのまま引数文字列を書きたい時に便利である。

```
/* t33.c -- execl */

main() {
  execl("/bin/ls", "ls", "-l", 0);
}
```

一方、execv の方は引数リストを自分で組み立てるときに便利である。

```
/* t34.c -- execv */

main(int argc, char *argv[]) {
  int i;
  int myargc = 0;
  char *myargv[100];
```

```
  myargv[myargc++] = "ls";
  myargv[myargc++] = "-l";
  for(i = 1; i < argc; ++i) myargv[myargc++] = argv[i];
  myargv[myargc] = 0;
  execv("/bin/ls", myargv);
}
```

練習 6 上のプログラムの好きな方を動かせ。また、tr を exec することで「大文字を小文字に変換する」プログラムを書け。(つまり、「tr A-Z a-z」と同じになる。)

練習 7 コマンド引数ではなく、標準入力から 1 行読み込み、それを空白のところで分割して argv を作って exec で実行するようにしてみよ。1 行読み込むのには「fgets(buf, BUFSIZE, stdin)」を使えばよい。

練習 8 このままだと、1 個コマンドを実行したらそれでおしまいになってしまう。exec する前に fork して、子プロセスだけ exec を実行し、親プロセスは子プロセスを wait するようにしてみよ。

1.4 ファイルディスクリプタと exec

しかし、なぜ「新しいプロセスとプログラム」ではなく「新しいプログラム」だけなのだろう? それは、exec する前にファイルディスクリプタ等の操作をやっておくと、その状態が exec した後のプログラムに引き継がれるからである。

例えば、「入出力リダイレクションをやるだけのプログラム」を考える。このプログラムは

```
a.out 入力ファイル 出力ファイル コマンド 引数 …
```

のようにすると、指定されたコマンドを実行するが、ただし標準入力と標準出力が指定したファイルに切り替わった状態で実行される。シェルのリダイレクションはこうして実現されているのだった。

```
/* t35.c -- redirection to files. */
#include <fcntl.h>
#include <stdio.h>

main(int argc, char *argv[]) {
  int fd0, fd1;
  if(argc < 4) { fprintf(stderr, "too few arguments.\n"); exit(1); }
  close(0); fd0 = open(argv[1], O_RDONLY);
  if(fd0 < 0) { perror("input file: "); exit(1); }
  close(1); fd1 = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0644);
  if(fd1 < 0) { perror("input file: "); exit(1); }
  execvp(argv[3], &argv[3]);
  perror("cannot exec: "); exit(1);
}
```

なお、open が返すディスクリプタ番号は、現在使用中でない最小の番号になる。だから close(0) の直後に open すれば必ず 0 番になるし、close(1) の直後では 1 番になる (成功すればの話)。あと、execvp というのは execv の親戚で、ただし$PATH に従ってプログラムを探してくれるので絶対パスで指定しなくてもすむ。どうしてもプログラムが見つからなかった場合には exec の次の命令に進むので、そこではエラーメッセージを出す。

1.5 パイプ

さて、Unix シェルには「|」(パイプライン) という機能もあった。これは pipe システムコールでパイプを作ることによって実現できる。パイプというのは OS 内部のバッファで、片方から書き込んだものがもう片方から読めるような仕掛けである。

```
/* t36.c -- pipe */
#define BUFSIZE 1024
char buf[BUFSIZE];

main() {
  int n, pid, fd[2];
  pipe(fd);
  if(pid = fork()) { /* parent */
    close(fd[1]);
    while((n = read(fd[0], buf, BUFSIZE)) > 0) {
      write(1, buf, n); write(1, buf, n); }
  } else { /* child */
    close(fd[0]);
    while((n = read(0, buf, BUFSIZE)) > 0) {
      write(fd[1], buf, n); write(fd[1], buf, n); }
  }
}
```

しかし、このままだと fd[0] や fd[1] は 0 や 1 ではないので、直ちに exec しても標準入力や標準出力としては使えない。その場合には

```
dup2(旧 fd, 新 fd)
```

というシステムコールでファイルディスクリプタ番号のコピーができるので、0 番や 1 番にコピーしてからいらない番号を close すればよい。

練習 9 「ls -l | tr a-z A-Z」を実行するようなプログラムを pipe、fork、exec を使って書け (/bin/tr ではなく/usr/ucb/tr を使うように)。

ここまでくれば、シェルがどうやって仕事をしているか分かるようになりましたね?

2 おまけ: スレッドプログラム

さて、ここまでのプロセスプログラムでは基本的に各プロセスは独立していて、互いに連絡できるとしてもパイプやファイルなどを通じてだけだった。しかし現在では、1 つの計算機に複数の CPU がついているものも多くなってきたので、「プロセスは 1 つだがその中で複数の命令列が並列実行される」というのも欲しくなってきた。この「実行される命令列」のことを「スレッド」という。複数のスレッドは並行して走るが、ただしメモリは共有している。従って広域変数などはどのスレッドでも共通してアクセスできる。

これを利用した行列かけ算プログラムを見てみていただこう。まずスレッド版でないものから。

```
/* t37.c -- matrix multiply */
#define MATSIZE 400
```

```
float a[MATSIZE][MATSIZE];
float b[MATSIZE][MATSIZE];
float c[MATSIZE][MATSIZE];

main() {
  uni(a); uni(b); clr(c);
  mul(0, MATSIZE, 0, MATSIZE, 0, 0, 0, 0);
}

clr(float m[][MATSIZE]) {
  int i, j;
  for(i = 0; i < MATSIZE; ++i)
    for(j = 0; i < MATSIZE; ++i) m[i][j] = 0.0;
}

uni(float m[][MATSIZE]) {
  int i;
  clr(m);
  for(i = 0; i < MATSIZE; ++i) m[i][i] = 1.0;
}

mul(int xlow, int xhigh, int ylow, int yhigh,
    int xaoff, int yaoff, int xboff, int yboff) {
  int x, y, z;
  for(x = xlow; x < xhigh; ++x) {
    for(y = ylow; y < yhigh; ++y) {
      float v = 0.0;
      for(z = xlow; z < xhigh; ++z)
        v += a[x+xaoff][z+yaoff]*b[z+xboff][y+yboff];
      c[x][y] += v;
    }
  }
}
```

単に行列 a、b を単位行列に初期設定し、その後その積を c に計算している。さて、各行列を 4 つの部分行列に分割して並列に積を求めるように改造してみる。次のプログラムは計算内容は変わらないが、4CPU の smb では 4 倍速く実行されるはずである。

```
/* t37b.c -- matrix multiply, thread version */
#include <thread.h>
#define MATSIZE 400
#define MATHALF 200
float a[MATSIZE][MATSIZE];
float b[MATSIZE][MATSIZE];
float c[MATSIZE][MATSIZE];

sub1() {
```

```
  mul(0, MATHALF, 0, MATHALF, 0, 0, 0, 0);
  mul(0, MATHALF, 0, MATHALF, MATHALF, 0, 0, MATHALF);
}
sub2() {
  mul(MATHALF, MATSIZE, 0, MATHALF, -MATHALF, 0, 0, 0);
  mul(MATHALF, MATSIZE, 0, MATHALF, 0, 0, 0, MATHALF);
}
sub3() {
  mul(0, MATHALF, MATHALF, MATSIZE, 0, 0, 0, -MATHALF);
  mul(0, MATHALF, MATHALF, MATSIZE, MATHALF, 0, 0, 0);
}
sub4() {
  mul(MATHALF, MATSIZE, MATHALF, MATSIZE, -MATHALF, 0, 0, -MATHALF);
  mul(MATHALF, MATSIZE, MATHALF, MATSIZE, 0, 0, 0, 0);
}

main() {
  thread_t t1, t2, t3, t4;
  uni(a); uni(b); clr(c);
  thr_create(0, 0, (void*(*)(void*))sub1, 0, THR_NEW_LWP, &t1);
  thr_create(0, 0, (void*(*)(void*))sub2, 0, THR_NEW_LWP, &t2);
  thr_create(0, 0, (void*(*)(void*))sub3, 0, THR_NEW_LWP, &t3);
  thr_create(0, 0, (void*(*)(void*))sub4, 0, THR_NEW_LWP, &t4);
  thr_join(t1, 0, 0);
  thr_join(t2, 0, 0);
  thr_join(t3, 0, 0);
  thr_join(t4, 0, 0);
}

clr(float m[][MATSIZE]) {
  int i, j;
  for(i = 0; i < MATSIZE; ++i)
    for(j = 0; i < MATSIZE; ++i) m[i][j] = 0.0;
}

uni(float m[][MATSIZE]) {
  int i;
  clr(m);
  for(i = 0; i < MATSIZE; ++i) m[i][i] = 1.0;
}

mul(int xlow, int xhigh, int ylow, int yhigh,
    int xaoff, int yaoff, int xboff, int yboff) {
  int x, y, z;
  for(x = xlow; x < xhigh; ++x) {
    for(y = ylow; y < yhigh; ++y) {
```

```
      float v = 0.0;
      for(z = xlow; z < xhigh; ++z)
        v += a[x+xaoff][z+yaoff]*b[z+xboff][y+yboff];
      c[x][y] += v;
    }
  }
}
```