今回は、プログラムで他のプログラムを起動する方法を学ぶ。プログラムを起動する方式には大きく分けて、**spawn 方式**（スポーン：卵を産む）と、**fork-exec 方式**（分岐-実行？）がある。

1. spawn 方式

   Windows 等で使用される方式である。新しい「(1) プロセスを作り」、「(2) プログラムを実行する」の二つの仕事を一つの spawn システムコールで行うので分かりやすい。

   fork-exec 方式を実現するためには、メモリ再配置のためのハードウェア機構を CPU が持っている必要がある。そのため組込用の小さなコンピュータ等では spawn 方式しか選択肢がない場合がある。

   最近は、UNIX 系 OS でも spawn 方式も使用できるようになっている。次に Mac の `posix_spawn` システムコールを紹介する。

```
書式：#include <spawn.h>
       int posix_spawn(int * pid, char *path,
         posix_spawn_file_actions_t *file_actions,  posix_spawnattr_t *attrp,
         char * argv[], char * envp[]);

解説：新しいプロセスを作り path で指定したプログラムを実行する。
       pid は新しいプロセスのプロセス番号を格納する変数を指すポインタ。
       file_actions, attrp はプロセスの初期化を指示するデータ構造へのポインタ。
       argv, envp は実行されるプログラムの main 関数の argv, envp に渡すデータ。
```

2. fork-exec 方式

   UNIX 系の OS 用いられてきた方式のこと。まず「(a) 新しいプロセスを作り (fork システムコール)」、次にユーザが記述したプログラムで初期処理を行い、最後に「(b) 新しいプログラムをロード・実行 (exec システムコール)」する。初期化処理をユーザが自由にプログラムで記述できるので柔軟性が高い。

   (a) プログラムをロード・実行 (exec システムコール)

   プロセスが新しいプログラムの実行を開始するシステムコールである。プロセスが新しいプログラムを実行するプロセスに変身する（変身の術）。以下に、UNIX の exec システムコール（execve システムコール）の解説を示す。

```
書式：#include <unistd.h>
       int execve(char *path, char *argv[], char *envp[]);

解説：自プロセスで path で指定したプログラムを実行する。
       argv, envp は実行されるプログラムの main 関数の argv, envp に渡すデータ。
       正常時には execve を実行したプログラムは新しいプログラムで上書きされる。
       execve システムコールが戻る (次の行が実行される) のはエラー発生時だけである。
```

   図 1 に `exec` システムコールを実行するプロセスの様子を示す。プロセスが `exec` システムコールを実行すると、そのプロセスのメモリ空間に新しいプログラムが実行形式ファイルからロードされる。`exec` システムコールを実行したプログラムは、新しいプログラムで上書きされる。プロセスの仮想 CPU はリセットされ、新しいプログラムの実行が開始される。

   リスト 1 に `execve` システムコールを使用して、`/bin/date` プログラムを実行するプログラムの例を示す。

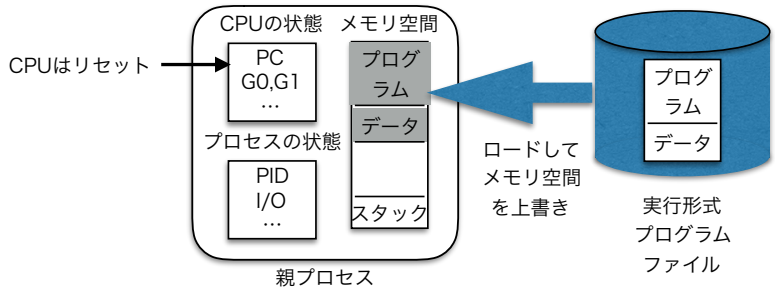

図 1: exec の仕組み

リスト 1: execve の使用例 (その１)

```
#include <stdio.h>          // perror のために必要
#include <unistd.h>         // execve のために必要

extern char **environ;
char *args[] = { "date", NULL };
char *execpath="/bin/date";

int main(int argc, char *argv[], char *envp[]) {
  execve(execpath, args, environ);   // /bin/date を自分と同じ環境変数で実行
  perror(execpath);                  // exec が戻ってきたらエラー！
  return 1;
}

/* 実行例(英語表示、日本時間で表示される)
$ exectest1
Sat Jul 16 22:25:33 JST 2016
*/
```

リスト 2 は、環境変数の一部を書き換えた上で、/bin/date プログラムを実行するプログラムの例である。

リスト 2: execve の使用例 (その２)

```
#include <stdio.h>          // perror のために必要
#include <unistd.h>         // execve のために必要
#include <stdlib.h>         // putenv のために必要

extern char **environ;
char *args[] = { "date", NULL };
char *execpath="/bin/date";

int main(int argc, char *argv[], char *envp[]) {
  putenv("LC_TIME=ja_JP.UTF-8");     // 自分の環境変数を変更する
  execve(execpath, args, environ);   // /bin/date を自分と同じ環境変数で実行
  perror(execpath);                  // exec が戻ってきたらエラー！
  return 1;
}

/* 実行例(日本語表示、日本時間で表示される)
$ exectest3
2016 年 7 月 16 日 土曜日 22 時 34 分 10 秒 JST
*/
```

リスト 3 は、全く新しい環境変数の一覧を渡して/bin/date プログラムを実行するプログラムの例である。

リスト 3: execve の使用例 (その３)

```
#include <stdio.h>          // perror のために必要
#include <unistd.h>         // execve のために必要

char *args[] = { "date", NULL };
char *envs[] = { "LC_TIME=ja_JP.UTF-8", "TZ=Cuba", NULL};
char *execpath="/bin/date";

int main(int argc, char *argv[], char *envp[]) {
  execve(execpath, args, envs);       // /bin/date を上記の環境変数で実行
  perror(execpath);                   // exec が戻ってきたらエラー！
  return 1;
}

/* 実行例(日本語表示、キューバ時間で表示される)
$ exectest2
2016 年 7 月 16 日 土曜日 09 時 24 分 40 秒 CDT
*/
```

リスト 4 は、複数のコマンド行引数がある場合の例である。/bin/echo プログラムを“aaa”、“bbb” を引数にして実行する。args 配列にプログラムの名前 (argv[0]:“echo”) を忘れないように注意すること。

リスト 4: execve の使用例 (その４)

```
#include <stdio.h>          // perror のために必要
#include <unistd.h>         // execve のために必要

extern char **environ;
char *args[] = { "echo", "aaa", "bbb", NULL };  // "$ echo aaa bbb" に相当
char *execpath="/bin/echo";

int main(int argc, char *argv[], char *envp[]) {
  execve(execpath, args, environ);    // /bin/echo を自分と同じ環境変数で実行
  perror(execpath);                   // exec が戻ってきたらエラー！
  return 1;
}

/* 実行例
$ exectest4
aaa bbb                 <--- /bin/echo の出力
*/
```

exec する際、プロセス状態の一部は引き継がれる。例えば、オープン中のファイルや、「無視」に設定されたシグナルハンドリング等は、新しいプログラムがロードされてもそのまま引き継がれる。この仕組みを使用して、プログラム実行開始前に標準入力 (0)、標準出力 (1)、標準エラー出力 (3) がオープンされる。

シェルは fork-exec を使用してプログラム（外部コマンド）を起動している。シェルのリダイレクト（プログラムの入出力を切替える仕組み）は、リダイレクト先のファイルを標準入力・出力としてオープンした状態で外部プログラムを exec することで実現で

きる。リスト 5 は、標準出力を “aaa.txt” にリダイレクトした状態で/bin/echo を実行するプログラムである。

リスト 5: execve の使用例 (その５)

```
#include <stdio.h>          // perror のために必要
#include <unistd.h>         // execve のために必要
#include <fcntl.h>          // open のために必要

extern char **environ;
char *args[] = { "echo", "aaa", "bbb", NULL };    // "$ echo aaa bbb" に相当
char *execpath="/bin/echo";
char *outfile="aaa.txt";

int main(int argc, char *argv[], char *envp[]) {
  close(1);                                       // 標準出力をクローズする
  int fd = open(outfile,                          // 標準出力を "aaa.txt" と
                O_WRONLY|O_CREAT|O_TRUNC, 0644);  //   してオープンしなおす
  if (fd<0) {                                     // オープンできなかった
    perror(outfile);                              //   エラーメッセージを出力
    return 1;                                     //   エラー終了
  }
  if (fd!=1) {                                    // 標準出力以外になってる
    fprintf(stderr, "何か変！\n");                //   原因が分からないが...
    return 1;                                     //   何か変なのでエラー終了
  }
  execve(execpath, args, environ);                // /bin/echo を実行
  perror(execpath);                               // exec が戻ってきたらエラー！
  return 1;
}

/* 実行例
$ exectest5                 <-- echo が実行されたはずなのに何も出力されない
$ cat aaa.txt               <-- "aaa.txt" に
aaa bbb                     <--   echo の出力が保存されていた
*/
```

(b) 新しいプロセスを作る (fork システムコール)

exec システムコールはプロセスを新しいプログラムに変身させる。変身して新しいプログラムを実行したプロセスは終了してしまう。新しいプロセスを作って、新しいプログラムを実行させる仕組みが必要である。fork システムコールは新しいプロセスを作成し自身をコピーする。つまり、**分身**を作るシステムコールである（分身の術)。

```
書式： #include <unistd.h>
       int fork(void);

解説： プロセスのコピーを作る。親プロセスには子プロセスの PID が返される。
       エラー時は、親プロセスに-1 が返され子プロセスは作られない。
```

fork システムコールを実行した「プロセスがコピーされ」る。元のプロセスを**親プロセス**、コピーして作ったプロセスを**子プロセス**と呼ぶ。図 2 に fork の様子を、リスト 6 に fork システムコールを実行するプログラムの例を、以下に fork の処理手順を示す。

i. 親プロセスがプログラム実行中に fork システムコールを実行する。

ii. 新しいプロセス（子プロセス）が作られ、親プロセスの内容がコピーされる。

iii. 子プロセスには、メモリ空間（プログラム、変数（データ）、スタック）、プロセスの状態（どのファイルをオープン中か、シグナルハンドラの登録等）、CPU の状態（CPU レジスタの値、SP の値、PC の値、フラグの値）等、全ての情報がコピーされる。ただし、プロセス番号 (PID:Process ID) 値は親子プロセスで異なる。

iv. 親プロセスは `fork` システムコールを終了しプログラムの実行を再開する。この時、`fork` システムコールは子プロセスの PID を返す。

v. 子プロセスは `fork` システムコールを呼出した瞬間のコピーなので、`fork` システムコールが終了するところからプログラム実行を開始する。この時、`fork` システムコールは `0` を返す。

最終的に親プロセスと子プロセスが同時に並行して実行される状態になる。

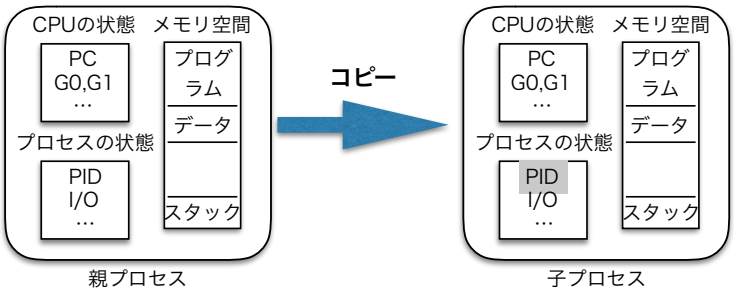

図 2: fork の仕組み

リスト 6: fork の使用例

```
#include <stdio.h>          // printf, fprintf のために必要
#include <unistd.h>         // fork のために必要

int main() {
  int x = 10;
  int pid;

  pid = fork();                                  // この瞬間にプロセスがコピーされる
  if (pid<0) {
    fprintf(stderr, "fork でエラー発生\n");   // エラーの場合
    return 1;
  } else if (pid!=0)   {                         // 親プロセスだけが以下を実行する
    x = 20;                                      //   親プロセスの x を書き換える
    printf("親␣pid=%d␣x=%d\n", pid, x);
  } else {                                       // 子プロセスだけが以下を実行する
    printf("子␣pid=%d␣x=%d\n", pid, x);       //   子プロセスの x は初期値のまま
  }
  return 0;
}

/*  実行例
$ forktest
親 pid=8079 x=20              // 親プロセスの出力
子 pid=0 x=10                 // 子プロセスの出力(x の値に注目)
*/
```

3. プロセスの終了と待ち合わせ

親プロセスは子プロセスを幾つか作成しそれらに同時に並行して処理を行わせる。子プロセスは処理を終えると終了する。子プロセスが処理を終えると親プロセスは各子プロセスが正常に終了したかチェックする。全ての子プロセスが正常に終了していれば処理全体が完了である。このような処理ができるように、子プロセスが処理結果と共に自身を終了する `exit` システムコールと、親プロセスが子プロセスの終了を待つ `wait` システムコールが準備されている。(正確には `exit` は `_exit` システムコールを呼び出すライブラリ関数)

子プロセスは `exit` システムコールを実行してもすぐに消滅するわけではない。子プロセスは、親プロセスが `wait` システムコールを実行して終了ステータスを取り出してくれるまで待ち状態になる。この状態を zombi 状態と呼ぶ。

なお C プログラムの `main` 関数は、スタートアップルーチンから `exit(main(argc, argv, envp));` のように呼び出されている。`main` 関数を `return n;` で終了すると、`exit(n);` が実行されることになる。つまり、`main` 関数中では `return n;` と `exit(n);` が同じ意味になる。

書式：
```
#include <stdlib.h>
void exit(int status);
```

解説：自プロセスを終了する。親プロセスは `wait` システムコールで `status` の値を受け取る。
終了ステータス (`status`) は下位 `8bit` が有効である。(`0 <= status <= 255`)
`exit` を呼び出すとプロセスが終了するので `exit` は戻らない。

書式：
```
#include <sys/wait.h>
void wait(int *status);
```

解説：子プロセスの終了を待つ。`status` に子プロセスが終了した理由等が格納される。
`status` の下位 `8bit` には、子プロセスが `exit` に渡した終了ステータスが格納される。
その他のビットで終了の理由 (`exit`、シグナル等) が分かるようになっている。

4. fork-exec 方式のプログラム例

リスト 7 に新しいプロセスで新しいプログラムを実行するプログラムの基本的な例を示す。このプログラムは、`fork`、`exec`、`exit`、`wait` を組み合わせて使用する一般的な例である。図 3 は、リスト 7 のプログラムの動きを解説したものである。

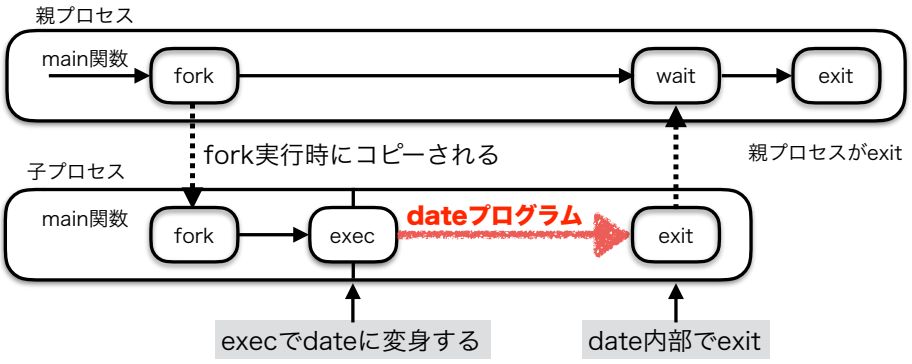

図 3: fork-exec の仕組み

リスト 7: fork-exec の例

```
#include <stdio.h>                    // perror のために必要
#include <stdlib.h>                   // exit のために必要
#include <unistd.h>                   // fork, execve のために必要
#include <sys/wait.h>                 // wait のために必要

extern char **environ;
char *args[] = {"date", NULL};
char *execpath="/bin/date";

int main(int argc, char *argv[], char *envp[]) {
  int pid;
  if ((pid=fork())<0) {               // 分身を作る
    perror(argv[0]);                  // fork がエラーなら
    exit(1);                          // 親プロセスをエラー終了
  }
  if (pid!=0) {                       // pid が 0 以外なら自分は親プロセス
    int status;
    wait(&status);                    //   子プロセスが終了するのを待つ
  } else {                            // pid が 0 なら自分は子プロセス
    execve(execpath, args, environ);  //   date プログラムを実行
    perror(execpath);                 //   exec が戻ってくるならエラー
    exit(1);                          //   エラー時はここで子プロセスを終了
  }
  exit(0);                            // 親プロセスを正常終了
}
```

リスト 8 に少し実用的な例を示す。このプログラムは、実行例に示すようにコマンド行引数に指示された環境変数の変更を行った上で date プログラムを次々に実行する。環境変数の変更は子プロセス側で行うようになっているので、親プロセスの環境変数は変化しない。

リスト 8: 子プロセスで環境変数を次々変更しながら date を次々実行

```
#include <stdio.h>                    // perror のために必要
#include <stdlib.h>                   // exit のために必要
#include <unistd.h>                   // fork, execve のために必要
#include <sys/wait.h>                 // wait のために必要
extern char **environ;
char *args[] = {"date", NULL};
char *execpath="/bin/date";

int main(int argc, char *argv[], char *envp[]) {
  int pid;
  for (int i=1; argv[i]!=NULL; i++) {
    if ((pid=fork())<0) {               // 分身を作る
      perror(argv[0]);                  // fork がエラーなら
      exit(1);                          // 親プロセスをエラー終了
    }
    if (pid!=0) {                       // pid が 0 以外なら自分は親プロセス
      int status;
      wait(&status);                    //   子プロセスが終了するのを待つ
    } else {                            // pid が 0 なら自分は子プロセス
      putenv(argv[i]);                  //   環境変数を変更する
      execve(execpath, args, environ);  //   date プログラムを実行
      perror(execpath);                 //   exec が戻ってくるならエラー
      exit(1);                          //   エラー時はここで子プロセスを終了
    }
```

```
  }
  exit(0);                                  // 親プロセスを正常終了
}
/* 実行例
$ forkexec2 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
2016年 7月 18日 月曜日 21時 27分 55秒 JST                          # 日本語、日本時間
понедельник, 18 июля 2016 г. 21:27:55 (JST)                       # ロシア語、日本時間
Mon Jul 18 08:30:00 CDT 2016                                      # 英語、キューバ時間
*/
```

リスト 9 は、環境変数の変更を親プロセスが fork 前に行うように変更したものである。リスト 8 の実行結果との違いに注目して欲しい。

リスト 9: 親プロセスで環境変数を次々変更しながら date を次々実行

```
#include <stdio.h>                      // perror のために必要
#include <stdlib.h>                     // exit のために必要
#include <unistd.h>                     // fork, execve のために必要
#include <sys/wait.h>                   // wait のために必要
extern char **environ;
char *args[] = {"date", NULL};
char *execpath="/bin/date";

int main(int argc, char *argv[], char *envp[]) {
  int pid;
  for (int i=1; argv[i]!=NULL; i++) {
    putenv(argv[i]);                      // 環境変数を変更する
    if ((pid=fork())<0) {                 // 分身を作る
      perror(argv[0]);                    // fork がエラーなら
      exit(1);                            // 親プロセスをエラー終了
    }
    if (pid!=0) {                         // pid が 0 以外なら自分は親プロセス
      int status;
      wait(&status);                      //   子プロセスが終了するのを待つ
    } else {                              // pid が 0 なら自分は子プロセス
      execve(execpath, args, environ);    //   date プログラムを実行
      perror(execpath);                   //   exec が戻ってくるならエラー
      exit(1);                            //   エラー時はここで子プロセスを終了
    }
  }
  exit(0);                                // 親プロセスを正常終了
}
/* 実行例
$ forkexec3 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
2016年 7月 18日 月曜日 22時 25分 51秒 JST
понедельник, 18 июля 2016 г. 22:25:51 (JST)
понедельник, 18 июля 2016 г. 09:25:51 (CDT)
*/
```

5. 問題

コマンド行引数で「環境変数とファイル名」の組を複数指定し、環境変数を変更した上で出力をファイルにリダイレクトし date を実行するプログラムを作りなさい。例えば次のように実行すると、現在時刻をキューバ時間で表したものが c.txt にローマ時間で表したものが r.txt に格納される。

```
$ a.out TZ=Cuba c.txt TZ=Europe/Rome r.txt
```