

オペレーティングシステムの機能を使ってみよう

第9章 プロセスの生成とプログラムの実行

spawn 方式と fork-exec 方式

新しいプログラムを実行する方式は次の二種類がある。

- **spawn 方式（スポン：卵を産む方式）**

次の 3 ステップを一つのシステムコールで行う。

1. プロセスを作る。
2. プロセスを初期化する。
3. プログラムを実行する。

- **fork-exec 方式（分岐-実行方式）**

次の 3 ステップを二つのシステムコールとプログラムで行う。

1. プロセスを作る（fork システムコール）。
2. プロセスを初期化する（ユーザプログラム）。
3. プログラムを実行する（exec システムコール）。

spawn 方式

Windows 等で使用されてきた.
posix_spawn の例

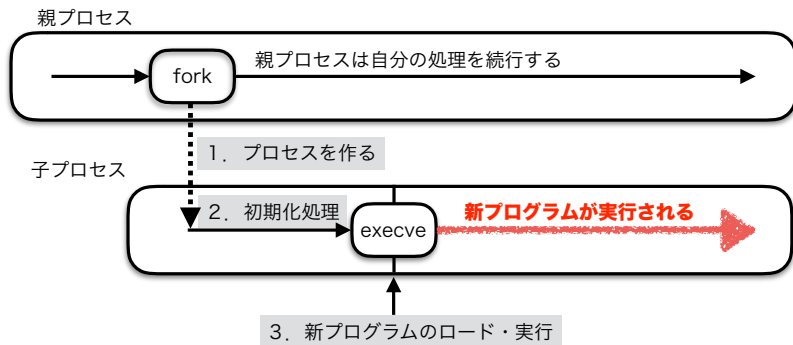
書式 次の通りである.

```
#include <spawn.h>
int posix_spawn(pid_t *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.

引数 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. path は実行するプログラムを格納したファイルのパスである. 絶対パスでも相対パスでも良い.
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に渡すコマンド行引数と環境変数である.

fork-exec 方式 (1)

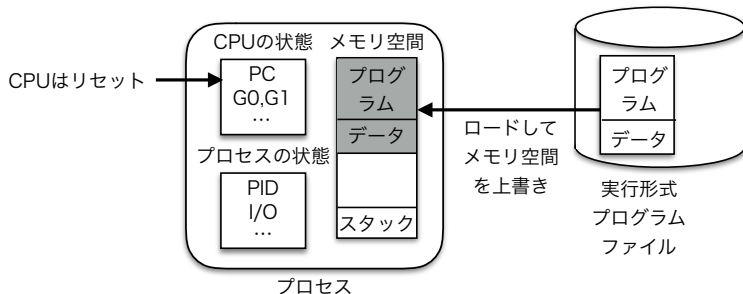


1. 新しいプロセス (子プロセス) を作る (**fork システムコール**).
2. ユーザプログラムに従い子プロセスが自ら初期化処理を行う.
3. 新しいプログラムをロード・実行 (**execve システムコール**) する.

プログラムで初期化処理を行うので柔軟性が高い。

fork-exec 方式 (2)

プログラムのロードと実行 (execve システムコール) の概要



- プロセスのメモリ空間に新しいプログラムをロードする.
- execve システムコールを発行したプログラムは上書きされて消える.
- プロセスの仮想 CPU はリセットされプログラムの先頭から実行.
- プロセスが新しいプログラムに**変身**した.

fork-exec 方式 (3)

execve システムコール

書式 execve システムコールの書式は次の通りである.

```
#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. 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消える. 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.

引数 path は実行するプログラムを格納したファイルのパスである. 絶対パスでも相対パスでも良い. argv, envp は新しいプログラムに渡すコマンド行引数と環境変数である.

fork-exec 方式 (4)

execve システムコールの使用例 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);                  // ※ execve 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
*/
```

- /bin/date プログラムをロード・実行する.
- date プログラムの argv 配列を準備して渡す.
- 環境変数は自身のものを渡す.
- execve が戻ってきたら無条件にエラー処理をする.

fork-exec 方式 (5)

execve システムコールの使用例 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);                 // execve 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
*/
```

- (環境変数を変更＝初期化処理) をした上で execve する.
- putenv() 関数を用いて自身の環境変数を書き換え.
- execve に自身の環境変数を渡す.

fork-exec 方式 (6)

execve システムコールの使用例 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);                 // execve 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)
$ ./execetest2
2016 年 7 月 16 日 土曜日 09 時 24 分 40 秒 CDT
*/
```

- 全く新しい環境変数の一覧を渡す例.
- date プログラムが必要とする環境変数だけの配列を渡す.

fork-exec 方式 (7)

execve システムコールの使用例 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);               // execve が戻ってきたらエラー！
    return 1;
}
/* 実行例
$ ./exectest4
aaa bbb                                <---- /bin/echo の出力
*/
```

- 複数のコマンド行引数をもつプログラム (echo) の実行例.
- argv[0] にプログラムの名前を入れ忘れないように.

fork-exec 方式 (8)

execve システムコールのラッパー関数

- 関数の内部で execve システムコールを発行 (wrapper)

書式 四つのラッパー関数の書式をまとめて掲載

```
#include <unistd.h>
int execl(char *path, char *argv[]);
int execlp(char *file, char *argv[]);
int execl(char *path, char *argv0, *argv1, ... ,*argvn, NULL);
int execlp(char *file, char *argv0, *argv1, ... ,*argvn, NULL);
```

意味 execl("/bin/date", argv);
→ execve("/bin/date", argv, environ);
execlp("date", argv);
→ execve("/bin/date", argv, environ);
execl("/bin/echo", "echo", "aaa", "bbb", NULL);
execlp("echo", "echo", "aaa", "bbb", NULL);

fork-exec 方式 (9)

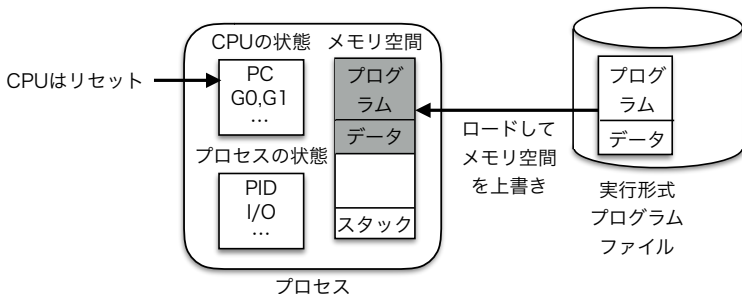
入出力のリダイレクト 1

● リダイレクトの復習

```
$ echo aaa bbb
aaa bbb
$ echo aaa bbb > a.txt
$ cat a.txt
aaa bbb
```

<--- echo の出力が表示される
<--- echo の出力が表示されない
<--- echo の出力が a.txt に格納されてた

- プログラムは標準入出力を自らオープンする必要が無かった.
- プロセスの状態が execve 前のプログラムから引き継がれるから.



入出力のリダイレクト2

- リダイレクトはプログラムのロード・実行前にシェルが行う。
- シェルが標準入出力をファイルに接続してから `execve` している。
- 原理を表すプログラム例

```
#include <stdio.h>           // perror のために必要
#include <unistd.h>          // execl のために必要
#include <fcntl.h>           // open のために必要
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,
                    O_WRONLY|O_CREAT|O_TRUNC, 0644); // 標準出力をオープンしなおす
    if (fd!=1) {             // 標準出力以外になってる
        fprintf(stderr, "something wrong\n"); // 原因が分からないが...
        return 1;           // 何か変なのでエラー終了
    }
    execl(execpath, "echo", "aaa", "bbb", NULL); // /bin/echo を実行
    perror(execpath);        // execl が戻ったらエラー！
    return 1;
}
```

課題 No.9

1. env コマンドのクローン myenv

putenv() がエラーになるまでコマンド行引数を環境変数の設定と思って使う。残りが実行するコマンドを表している。下の実行例では

```
putenv(argv[1]);
```

```
putenv(argv[2]);
```

```
putenv(argv[3]);
```

```
execvp(argv[3], &argv[3]);
```

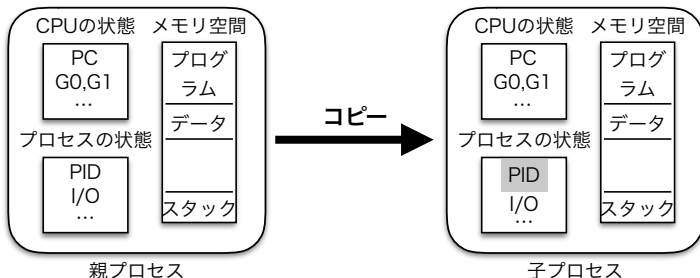
(三回目の putenv() はエラーになる)

が実行されるようにプログラムを作る。

```
$ ./myenv LC_TIME=ja_JP.UTF-8 TZ=Cuba ls -l
```

fork-exec 方式 (1 1)

新しいプロセスを作る (fork システムコール) 1



- fork システムコールはプロセスのコピー**分身**を作る。
- もととのプロセスが**親プロセス**，分身が**子プロセス**。
- **分身**は PID 以外は同じ (**CPU の PC も同じ**)。
- 子プロセスは fork システムコールの途中から実行を開始する。

fork-exec 方式 (1 2)

新しいプロセスを作る (fork システムコール) 2

書式 fork の書式を示す.

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

解説 fork システムコールが終了する際、親プロセスには子プロセスの PID が返され、子プロセスには 0 が返される。プログラムはこの値を目印に自分が親か子か判断できる。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。

```
int main() {
    int x = 10;
    int pid;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "fork でエラー発生\n"); // この瞬間にプロセスがコピーされる
        return 1;
    } else if (pid != 0) {
        x = 20; // 親プロセスだけが以下を実行する
               // 親プロセスの x を書き換える
    }
```


fork-exec 方式 (1 3)

新しいプロセスを作る (fork システムコール) 3

使用例 親プロセスと子プロセスが並行実行される状態になる。

```
1  #include <stdio.h>           // printf, fprintf のために必要
2  #include <unistd.h>          // fork のために必要
3  int main() {
4      int x = 10;
5      int pid;
6      pid = fork();             // この瞬間にプロセスがコピーされる
7      if (pid < 0) {
8          fprintf(stderr, "fork でエラー発生\n"); // エラーの場合
9          return 1;
10     } else if (pid != 0) {     // 親プロセスだけが以下を実行する
11         x = 20;                // 親プロセスの x を書き換える
12         printf("親 pid=%d x=%d\n", pid, x);
13     } else {                  // 子プロセスだけが以下を実行する
14         printf("子 pid=%d x=%d\n", pid, x);    // 子プロセスの x は初期値のまま
15     }
16     return 0;
17 }
```

プロセスの終了と待ち合わせ

例えば次のような手順で処理がされる.

- 親プロセスは複数の子プロセスに並行して処理をさせる.
- 子プロセスは自分の処理をして終了する.
- 親プロセスは全ての子プロセスが正常に終了したか確認する.

`$ ls -l | grep rwx` ← 二つのプロセスが並列実行される

- 子プロセスが処理結果 (正常／エラー) を親に知らせる.
→ `exit` システムコール
- 親プロセスが子プロセスの終了を知る.
→ `wait` システムコール

fork-exec 方式 (15)

exit システムコール：自プロセスを終了する。

書式 *exit* システムコールの書式を示す。

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

解説

- プロセスが終了するので *exit* は戻らない。
- *status* はプロセスの**終了ステータス**である。
($0 \leq \text{status} \leq 255$)
- 終了ステータスは親プロセスが *wait* で受け取る。
- C プログラムの *main* 関数は *exit* の引数で実行。
exit(main(argc, argv, envp));
- 次の C プログラムは同じ結果になる。

```
int main() {
    ...
    exit(1);
    ...
}
```

=

```
int main() {
    ...
    return 1;
    ...
}
```

fork-exec 方式 (16)

wait システムコール：子プロセスの終了を待つ.

書式 *wait* システムコールの書式を示す.

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

解説

- 終了したプロセスの番号 (PID: 正の値) を返す.
- エラーが発生した場合に -1 を返す.
- *status* にプロセスの終了理由等が格納される.
status の下位 8bit が子プロセスの終了ステータス
他のビットで終了の理由 (exit, シグナル等) を表す.

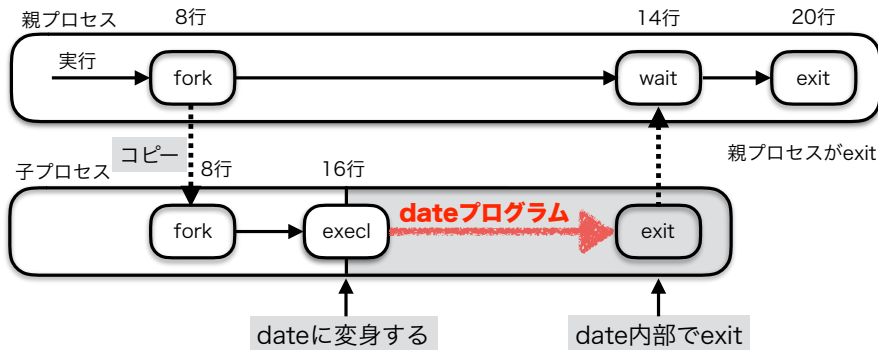
fork-exec 方式 (17)

プログラム例

```
1  #include <stdio.h>                                // perror のために必要
2  #include <stdlib.h>                                // exit のために必要
3  #include <unistd.h>                                // fork, execve のために必要
4  #include <sys/wait.h>                              // wait のために必要
5  char *execpath="/bin/date";
6  int main(int argc, char *argv[], char *envp[]) {
7      int pid;
8      if ((pid=fork())<0) {                            // 分身を作る
9          perror(argv[0]);                            // fork がエラーなら
10         exit(1);                                     // 親プロセスをエラー終了
11     }
12     if (pid!=0) {                                    // pid が 0 以外なら自分は親プロセス
13         int status;
14         wait(&status);                               // 子プロセスが終了するのを待つ
15     } else {                                         // pid が 0 なら自分は子プロセス
16         execl(execpath, "date", NULL);              // date プログラムを実行(execl を使用してみた
17         perror(execpath);                            // exec が戻ってくるならエラー
18         exit(1);                                     // エラー時はここで子プロセスを終了
19     }
20     exit(0);                                       // 親プロセスを正常終了
21 }
```

fork-exec 方式 (1 8)

プログラムの解説



fork-exec 方式 (19)

```
5 char *execpath="/bin/date";
6 int main(int argc, char *argv[], char *envp[]) {
7     int pid;
8     for (int i=1; argv[i]!=NULL; i++) {
9         if ((pid=fork())<0) { // 分身を作る
10             perror(argv[0]); // fork がエラーなら
11             exit(1); // 親プロセスをエラー終了
12         }
13         if (pid!=0) { // pid が 0 以外なら自分は親プロセス
14             int status;
15             wait(&status); // 子プロセスが終了するのを待つ
16         } else { // pid が 0 なら自分は子プロセス
17             putenv(argv[i]); // 環境変数を変更する
18             execl(execpath, "date", NULL); // date プログラムをロード・実行
19             perror(execpath); // execl が戻ってくるならエラー
20             exit(1); // エラー時はここで子プロセスを終了
21         }
22     }
23     exit(0); // 親プロセスを正常終了
24 }
25 /* 実行例
26 $ forkexec2 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
27 2016 年 7 月 18 日 月曜日 21 時 27 分 55 秒 JST
28 понедельник, 18 июля 2016 г. 21:27:55 (JST)
29 Mon Jul 18 08:30:00 CDT 2016
30 */
```

fork-exec 方式 (20)

```
5 char *execpath="/bin/date";
6 int main(int argc, char *argv[], char *envp[]) {
7     int pid;
8     for (int i=1; argv[i]!=NULL; i++) {
9         putenv(argv[i]); // 環境変数を変更する
10        if ((pid=fork())<0) { // 分身を作る
11            perror(argv[0]); // fork がエラーなら
12            exit(1); // 親プロセスをエラー終了
13        }
14        if (pid!=0) { // pid が 0 以外なら自分は親プロセス
15            int status;
16            wait(&status); // 子プロセスが終了するのを待つ
17        } else { // pid が 0 なら自分は子プロセス
18            execl(execpath, "date", NULL); // date プログラムを実行(execl を使用してみた)
19            perror(execpath); // exec が戻ってくるならエラー
20            exit(1); // エラー時はここで子プロセスを終了
21        }
22    }
23    exit(0); // 親プロセスを正常終了
24 }
25 /* 実行例
26 $ forkexec3 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
27 2016 年 7 月 18 日 月曜日 22 時 25 分 51 秒 JST
28 понедельник, 18 июля 2016 г. 22:25:51 (JST)
29 понедельник, 18 июля 2016 г. 09:25:51 (CDT)
30 */
```


課題 No.10

1. 次々リダイレクトして `date` を実行するプログラム

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
```

2. `system` 関数のクローン `mysystem`

`system()` 関数の仕様を調べて、なるべく同じものを作りなさい。C言語中で `system("...");` の関数呼出しをすると、シェアルに以下のように入力したのと同じことが起こる。

```
$ /bin/sh -c "..."
```