

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

オペレーティングシステムの機能を使ってみよう

1 / 24

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** (スプーン：卵を産む方式)
Windows 等で使用されてきた。
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)
UNIX 系の OS で使用されてきた。
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)。

オペレーティングシステムの機能を使ってみよう

2 / 24

spawn 方式

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[]);
```

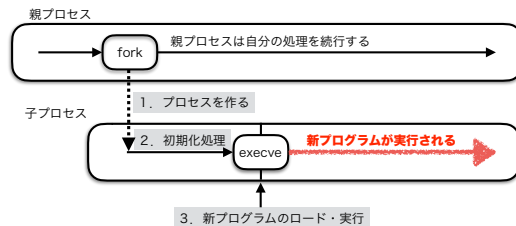
解説 新しいプロセスを作り path で指定したプログラムを実行する。

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。

オペレーティングシステムの機能を使ってみよう

3 / 24

fork-exec 方式 (1)



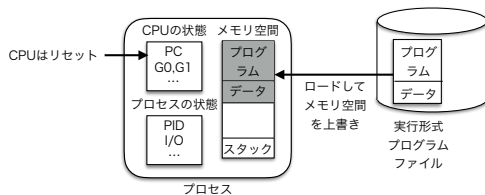
1. 新しいプロセス (子プロセス) を作る (fork システムコール)。
 2. ユーザプログラムに従い子プロセスが自ら初期化処理を行う。
 3. 新しいプログラムをロード・実行 (execve システムコール) する。
- プログラムで初期化処理を行うので柔軟性が高い。

オペレーティングシステムの機能を使ってみよう

4 / 24

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。

オペレーティングシステムの機能を使ってみよう

5 / 24

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。

オペレーティングシステムの機能を使ってみよう

6 / 24

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)
% ./executest1
Sun Apr  2 09:20:24 JST 2023
*/
```

- /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。

オペレーティングシステムの機能を使ってみよう

7/24

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)
% ./executest3
2023年 4月 2日 日曜日 09時24分29秒 JST
*/
```

- (環境変数を変更＝初期化処理) をした上で execve をする。
- 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。

オペレーティングシステムの機能を使ってみよう

8/24

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)
% ./executest2
2023年 4月 1日 土曜日 20時25分52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。

オペレーティングシステムの機能を使ってみよう

9/24

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
% ./executest4
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。

オペレーティングシステムの機能を使ってみよう

10/24

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 execv(char *path, char *argv[]);
int execvp(char *file, char *argv[]);
int execl(char *path, char *argv0, *argv1, ... ,*argvn, NULL);
int execlp(char *file, char *argv0, *argv1, ... ,*argvn, NULL);
```

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

オペレーティングシステムの機能を使ってみよう

11/24

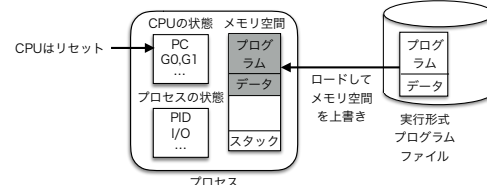
fork-exec 方式 (9)

入出力のリダイレクト 1

- リダイレクトの復習

```
% echo aaa bbb          <--- echoの出力が表示される
aaa bbb
% echo aaa bbb > a.txt   <--- echoの出力が表示されない
% cat a.txt
aaa bbb                  <--- 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。



オペレーティングシステムの機能を使ってみよう

12/24

fork-exec 方式 (10)

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いる
        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;
}
```

オペレーティングシステムの機能を使ってみよ

13 / 24

課題 No.9

- 前のページ (リスト 9.5) のプログラムを入力し実行してみる。
- 入力のリダイレクトをするプログラム例を作る。
ヒント：標準入力のファイルディスクリプタは 0 番である。
- `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。

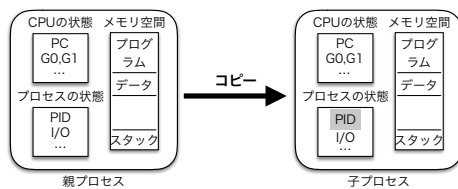
```
% ./myenv LC_TIME=ja_JP.UTF-8 TZ=Cuba ls -l
```

オペレーティングシステムの機能を使ってみよ

14 / 24

fork-exec 方式 (11)

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。
- もとも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。

オペレーティングシステムの機能を使ってみよ

15 / 24

fork-exec 方式 (12)

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(); // 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
    }
```

オペレーティングシステムの機能を使ってみよ

16 / 24

fork-exec 方式 (13)

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 }
```

オペレーティングシステムの機能を使ってみよ

17 / 24

fork-exec 方式 (14)

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。

- 親プロセスは子プロセスをいくつか作成し、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、親プロセスは子プロセスが正常に終了したかチェックする。

% 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と共に自身を終了する。
→ `exit` システムコール
- 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

オペレーティングシステムの機能を使ってみよ

18 / 24

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 ≤ status ≤ 255)
- 親プロセスは wait で終了ステータスを受け取る。
- 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;
    ...
}
```

オペレーティングシステムの機能を使ってみよう

19 / 24

fork-exec 方式 (16)

wait システムコール：親プロセスが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。

オペレーティングシステムの機能を使ってみよう

20 / 24

fork-exec 方式 (17)

プログラム例

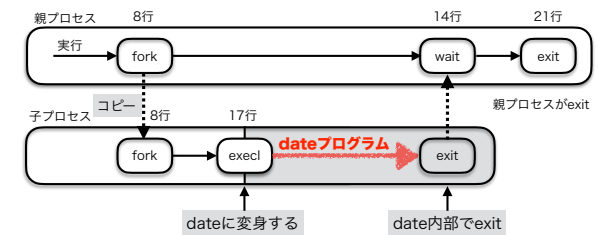
```
1 #include <stdio.h>           // perror のために必要
2 #include <stdlib.h>          // exit のために必要
3 #include <unistd.h>          // fork, execve のために必要
4 #include <sys/wait.h>        // wait のために必要
5 char *execpath="/bin/date";
6 int main(int argc, char *argv[], char *envp[]) {
7     int pid;
8     if ((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        while (wait(&status)!=pid) // 子プロセスが終了するのを待つ
15            ;
16    } else {                   // pid が 0 なら自分は子プロセス
17        execl(execpath, "date", NULL); // date プログラムを実行 (execl を使用して)
18        perror(execpath);     // exec が戻ってくるならエラー
19        exit(1);              // エラー時はここで子プロセスを終了
20    }
21    // 親プロセスを正常終了
22    exit(0);
23 }
```

オペレーティングシステムの機能を使ってみよう

21 / 24

fork-exec 方式 (18)

プログラムの解説



オペレーティングシステムの機能を使ってみよう

22 / 24

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            while (wait(&status)!=pid) // 子プロセスが終了するのを待つ
16                ;
17        } else {              // pid が 0 なら自分は子プロセス
18            putenv(argv[i]);   // 環境変数を変更する
19            execl(execpath, "date", NULL); // date プログラムをロード・実行
20            perror(execpath); // execl が戻ってくるならエラー
21            exit(1);          // エラー時はここで子プロセスを終了
22        }
23    }
24    // 親プロセスを正常終了
25    exit(0);
26 }
```

```
26 /* 実行例
27 % ./forkexec2 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
28 2023年 4月 2日 日曜日 10時47分31秒 JST
29 воскресенье, 2 апреля 2023 г. 10:47:31 (JST)
30 Sat Apr 1 21:47:31 CDT 2023
31 */
```

オペレーティングシステムの機能を使ってみよう

23 / 24

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            while (wait(&status)!=pid) // 子プロセスが終了するのを待つ
17                ;
18        } else {              // pid が 0 なら自分は子プロセス
19            execl(execpath, "date", NULL); // date プログラムを実行 (execl を使用して)
20            perror(execpath); // execl が戻ってくるならエラー
21            exit(1);          // エラー時はここで子プロセスを終了
22        }
23    }
24    // 親プロセスを正常終了
25    exit(0);
26 }
```

```
26 /* 実行例
27 % ./forkexec3 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
28 2023年 4月 2日 日曜日 10時49分17秒 JST
29 воскресенье, 2 апреля 2023 г. 10:49:17 (JST)
30 суббота, 1 апреля 2023 г. 21:49:17 (CDT)
31 */
```

オペレーティングシステムの機能を使ってみよう

24 / 24