SECTION

2-1 fork

新しい子プロセスを作成する

書　式

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

引　数　なし

戻り値　正の値：子プロセスのプロセスID（forkに成功した親プロセスに返される）
0：forkに成功（子プロセスに返される）
-1：forkに失敗

解説

新しいプロセスを作成するにはforkを使います。forkを呼び出すと、forkを呼び出したプロセスを親プロセスとして、親プロセスの属性のほとんどを引き継いだ子プロセスが新たに作成されます。forkを呼び出すプログラム側から見ると、自分自身が親と子の2つに分身するかのように見えます。forkからリターンする際の戻り値として、親プロセスには子プロセスのプロセスIDが、子プロセスには0が返されます。そこで、プログラム側ではforkの戻り値を見て場合分けして、親プロセスの場合の処理と子プロセスの場合の処理をそれぞれ記述するようにします。

新たなプログラムを実行するためにforkを呼び出す場合、forkはexecveと組み合わせて使用するのが普通です。具体的には、forkで子プロセスを作成後、子プロセス側がexecveで目的のプログラムを実行します。一方、サーバプログラムなどが複数のクライアントに対して同時にサービスを行うような場合は、execveを行わずに、forkだけを単独で利用することがあります。

forkのサンプルプログラム

List2-1-1はforkで子プロセスを作成した後、親プロセスと子プロセスの両方からwriteでテストメッセージを出力する例です。forkの戻り値はif文を使って場合分けして、エラーの場合、子プロセスの場合、親プロセスの場合のそれぞれの処理を記述しています。

子プロセスを終了する際には、main()関数をreturnしたり、Cライブラリ関数のexit()を呼び出すのではなく、システムコールのexitを直接呼び出す関数である、_exit()を呼び出すべきです。Cライブラリ関数のexit()では、親プロセスから引き継いだ標準入出力ライブラリのバッファをフラッシュするなどの事後処理が誤って実行されてしまいます。

List2-1-1をコンパイルして実行すると、両方のプロセスのメッセージが表示されることが確認できます。

List 2-1-1　**forkのサンプルプログラム**

```
#include <sys/types.h>      /* forkのために必要なヘッダファイル */
#include <unistd.h>         /* forkのために必要なヘッダファイル */

#include <stdio.h>          /* perror()のために必要なヘッダファイル */

int
main()
{
  pid_t pid;                    /* プロセスID格納用変数 */

  if ((pid = fork()) < 0) {     /* forkで新しいプロセスを作成 */
    perror("fork");             /* エラーが発生した場合はエラーメッセージを表示 */
    return 1;                   /* 戻り値1で異常終了 */
  } else if (pid == 0) {                /* 子プロセスの場合 */
    write(1, "chlid process\n", 14);    /* 子プロセス側からメッセージを出力 */
    _exit(0);                           /* 戻り値0で子プロセスを正常終了 */
  }
  write(1, "parent process\n", 15);     /* 親プロセス側からメッセージを出力 */
  return 0;                             /* 戻り値0で親プロセスを正常終了 */
}
```

実行結果　**List 2-1-1の実行結果**

```
$ ./List2-1-1 ↩
chlid process       ←子プロセスからのメッセージ
parent process      ←親プロセスからのメッセージ
```