補足資料

ファイルディスクリプタ

ファイルディスクリプタ (file descriptor) とは，プロセスが計算機資源に I/O を行うための識別子である．ファイルなどにアクセスする際には，プロセスはまずそのファイルに対するファイルディスクリプタを OS に要求し，その値を使って I/O を行う．たとえば，読み書きしたいファイルを open() の引数として渡して実行すると，成功した場合には返り値としてファイルディスクリプタ (正の整数) が返ってくる．プロセスはこの番号を使って読み書き (つまり，read() や write()) を行う．ちなみに，プロセスが起動すると，0: 標準入力 (stdin)，1: 標準出力 (stdout)，そして 2: 標準エラー出力 (stderr) の 3 つのファイルディスクリプタがデフォルトで割り当てられる．この番号を指定すれば，標準入力や標準出力に I/O を行うことができる (read(0, buf, 1024)，write(1, "test n", 5)，など).

ファイルディスクリプタの生成，削除はシステムコールを経由して行う．新たなファイルディスクリプタの生成には open() や socket() を，削除には close() を用いる．ファイルディスクリプタを生成する際には，もっとも若い整数値が与えられる．たとえば，0，1，2 が使われている状況で新たなファイルディスクリプタを生成すると，その番号は 3 となる．また，1 を close() して 0，2 が使われている状況であれば，新たなファイルディスクリプタには 1 が割り当てられる．

システムコール: fork()

プロセスを生成するためには，fork() というシステムコールを用いる．これが Linux カーネルで新規にプロセスを生成する唯一の方法である（init などの特別なプロセスは，ブートストラップの過程でカーネルが特別に生成する）．fork() を呼ぶことで生成された新規プロセスを子プロセスと呼ぶ．これに対し，新規プロセスを生成した側のプロセスを親プロセスと呼ぶ.

子プロセスは親プロセスの複製として稼働しはじめ，親と子のどちらも fork() を呼んだ次の命令から実行を続ける．子プロセスは親プロセスのデータ空間やスタックなどのコピーを得る．これは子にデータがコピーされたのであり，メモリ領域を共有するのではない.

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

pid_t fork(void);
```

成功するとプロセスのコピーが生成され，子プロセスには 0，親プロセスには子プロセスのプロセス ID が返る．失敗すると -1 を返す．親プロセスと子プロセスとの返り値が異なるため，これを利用してプログラミングを行う.

システムコール: wait()

子プロセスの終了は wait() というシステムコールで待つことができる．wait() には waitpid() などのバリエーションがあるが，子プロセスの終了や中断 (suspend) など子プロセスの状態を親プロセスが把握できるようになっている．プロセスは終了すると，一旦ゾンビプロセスとなる．その後，そのプロセスの親プロセスが wait() システムコールを呼び出すまで存在し続ける．つまり，子プ

ロセスを生成しまくった後，wait() を呼ばなければシステム内はゾンビだらけとなる．親プロセスが wait() システムコールを呼び出すと，ゾンビプロセスの終了時の終了状態が親プロセスに返され，同時にゾンビプロセスも消滅する．このように，子プロセスの終了状態を親プロセスに伝える仕組みとして，ゾンビプロセスが存在する．

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

成功したら子プロセスの ID と終了状態を，失敗したら -1 を返す．以下に両システムコールを用いたコードを掲載する．それぞれのシステムコールの挙動を確認してもらいたい．

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

int global_val = 3;

int main()
{
  int local_val;
  pid_t p_pid, c_pid;

  local_val = 10;

  printf("[%d]: global_val = %d, local_val = %d\n",
         (int)getpid(), global_val, local_val);

  if((c_pid = fork()) < 0) {
    perror("fork");
    return 1;
  }
  else if(c_pid == 0) { /* child */
    global_val++;
    local_val++;
    printf("[%d]: global_val = %d, local_val = %d\n",
           (int)getpid(), global_val, local_val);
    return 0;
  }
  wait(NULL);
  printf("[%d]: global_val = %d, local_val = %d\n",
         (int)getpid(), global_val, local_val);
  return 0;
}
```

システムコール: exec()

あるコマンドを実行するために用いられるシステムコールは，一般に exec() と総称される．exec() は，成功するとそれを呼んだプロセスの上に，新しいプログラムが読み込まれる．プロセスが exec() のひとつを呼ぶと，そのプロセスはまったく新しいプログラムに置き換えられ，新しいプログラムが main() から実行される．新しいプロセスが作られるわけではないので，プロセス ID は変わらない．

exec() は現プロセス（現プロセスのテキスト，データ，ヒープ，スタック領域）を，ディスクからまったく新しいプログラムで置き換える．すなわち，そのプロセスが使っていたメモリは解放され，その場所に新しいプログラムがロードされる．そして一度 exec() が成功すると，呼び出したプログラムには制御が戻らず，終了状態となる．

なお，ファイルディスクリプタは exec() を用いても初期化されない．つまりファイルディスクリプタは他のプログラムに化けた後も，化ける前のプロセスで設定した状態が維持される．これがシェルを実装する上で重要な性質となり，リダイレクトやパイプを実現する上でのキーとなる (標準出力 (1) をファイルへの出力やパイプへの出力にすり替えるなど).

```
#include <unistd.h>

extern char ** environ

int execl(const char *path, const char *arg, ..., NULL);
int execlp(const char *file, const char *arg, ..., NULL);
int execle(const char *path, const char *arg , ..., NULL, char * const envp[]);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv [], char *const envp[]);
int execvp(const char *file, char *const argv[]);
```

成功すると該当プログラムに化けるため特に返り値を返さない．失敗したら -1 を返す．これらの 6 つの関数のうち execve だけがカーネルに対するシステムコールで，残りの 5 つはこのシステムコールを起動する単なるライブラリ関数である．以下に exec() 使った例を示す．引数にコマンドを受け取り，それを exec() する．

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  if(argc < 2) {
    printf("Error: Command not found\n");
    return 1;
  }

  if(execvp(argv[1], &argv[1])) {
    perror("execvp");
    return 1;
  }

  return 0;
}
```

システムコール: dup()

リダイレクトを実現する上では，プログラムの標準出力をファイルにつなぐなどをすれば，目的の挙動を得られそうである．しかし，それぞれのファイルディスクリプタ番号が異なるため，どのように繋げばよいかという疑問が生じる．ここで活躍するのが，dup() システムコールである．これは，ファイルディスクリプタの複製を作成するシステムコールである．

```
#include <unistd.h>

int dup(int filedes);
```

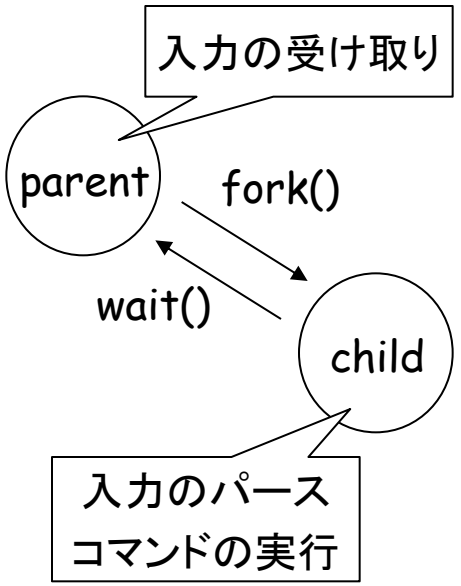
引数で渡したファイルディスクリプタを複製し，新しいファイルディスクリプタの番号を返す．失敗したら -1 を返す．

ここでファイルディスクリプタの生成規則について思い出して欲しい．Linux カーネルは，ファイルディスクリプタを生成すると，そのときに使われていないもっとも若い整数値を割り当てる．この性質をうまく使えば，標準出力 1 にファイルをつなげたり，パイプをつなげることができる．以下がサンプルコードである．標準出力をファイルにすり替えている．そのため，printf() の標準出力がファイルへの出力となっている．

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
  int fd;

  if((fd = open("./test.txt", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) < 0) {
    perror("open");
    return -1;
  }

  close(1);
  dup(fd);
  close(fd);

  printf("test\n");
  return 0;
}
```

例: 簡易シェル on Linux (simple_shell.c) を題材に・・・

- 使った主たるシステムコール
  - fork(): プロセス(子プロセス)を生成する
    - 自分の複製を作るようなイメージ
    - 返り値は親と子で異なる
      - 親: 成功: 子プロセスのプロセス ID、失敗: マイナス値
      - 子: 0
  - execvp(): 引数で渡されたプログラムに化ける
    - 呼び出したプロセスのメモリ内容が上書きされる
    - 返り値: 成功: 0、失敗: マイナス値
  - wait(): 子プロセスの終了を待つ
    - 子プロセスの状態が変わるまで呼び出し地点で止まる
    - 返り値: 子プロセスの PID



- 挙動としては・・・
  1. 入力を受け取る(67: getcmd())
  2. 子プロセスを生成する(68: fork())
  3. 親: 子プロセスが終わるまで待つ(70: wait())
  3. 子: 入力をパースして execvp() (45: parsecmd(), 46: execvp())
  4. 子プロセスの終了を確認して 1 へ