UNIXシステムプログラミング

第6回 シェルの動作とプロセス制御(1)

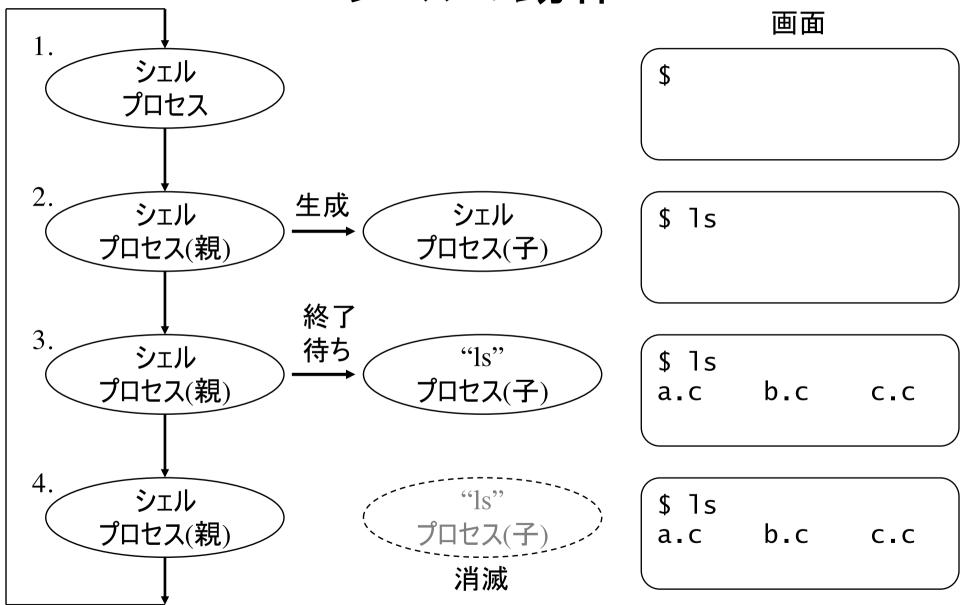
2019年11月1日 情報工学科 寺岡文男

シェルとは

シェル: コマンドインタプリタ

- シェルの基本動作
 - 1. "\$"を表示し、ユーザからの入力を待つ.
 - 2. 入力文字列の構文解析を行い、子プロセスを生成する.
 - 3. 子プロセスは指定されたコマンド(実行形ファイル)を実行し、実行終了後には消滅する.
 - 4. 親プロセスは子プロセスの終了を待ち、1. へ戻る.

シェルの動作



プロセス制御(1)

- シェルの動作をプロセス制御の観点から見る.
- プロセスの生成
 - UNIXでは、プロセスはあるプロセスの複製として生成される.
 - 最初のプロセスは init (/sbin/init, pid = 1)
 - 複製の元のプロセスを「親プロセス」と呼び、複製された方を「子プロセス」と呼ぶ。
 - fork() システムコール
- プロセスの属性
 - プロセス識別子(pid), 親プロセスの識別子(ppid)
 - プロセスグループの識別子(pgid), セッション識別子(sid)
 - 状態,制御端末,など

プロセス制御(2)

- プログラムの実行
 - fork() はプロセスを複製するのみで、他のプログラムを実行する わけではない。
 - execve() システムコール, execvp() ライブラリ関数
- プロセスの終了と終了待ち
 - シェルの親プロセスは子プロセスの終了を待つ.
 - wait() システムコール
 - 子プロセスは exit() システムコールによって実行を終了.
 - 子プロセスの終了状態を親プロセスに返すことができる。
 - 制プロセスが子プロセスの終了をwait()システムコールで待たないと、子プロセスは"ゾンビ"状態になる。

プロセス制御(3)

- シグナル:プロセスにイベントを知らせるための信号
- プロセス外部から送信されるもの
 - キーボードから ctrl-C 入力 → SIGINT送信 → 実行終了
 - キーボードから ctrl-Z 入力 → SIGTSTP送信 → 実行中断
- プロセス自身の動作に起因するもの
 - 例: 不正なポインタの先を参照 → SIGSEGV → 実行終了→ core dumped
- 32種類が定義されている
- 関連するシステムコール、ライブラリ関数
 - sigaction(), signal(), kill(), killpg(), etc.

プロセスの属性

• プロセスはさまざまな属性を持つ(以下は抜粋)

- UID: プロセスを起動したユーザ識別子

- PID: プロセスの識別子

PPID: 親プロセスの識別子

- PGID: プロセスグループの識別子

- SID: セッションの識別子

- PRI: プロセスの優先度

- STAT: プロセスの状態

R: 実行中 (running)

I: 20秒以上スリープ (idle) U: 割込禁止

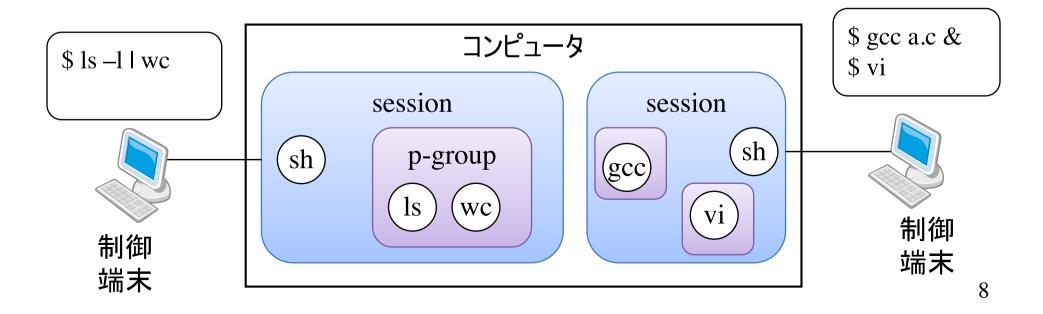
S: 20秒未満のスリープ (sleep) Z: ゾンビ (zombie)

- TTY: プロセスの制御端末

T: 停止 (stopped)

プロセスグループ, セッション

- セッション:制御端末を共有するプロセス群
 - 通常, シェルがセッションリーダ
 - sid (session id)で識別 (= セッションリーダのpid)
- シェルが起動したプロセス(群):プロセスグループ
 - e.g., "ls -l | wc"
 - pgid (process group id)で識別 (= プロセスのpid)



ps コマンド

プロセスの状態を表示,多くのスイッチがある

```
$ ps l
F    UID    PPID    PRI NI    VSZ    RSS    WCHAN    STAT    TTY    TIME    COMMAND
0    24542    27890    20    0    117416    2332    wait         Ss    pts/3    0:00 -csh
0    24542    27891    20    0    108124    1004 -         R+    pts/3         0:00    ps l
%
```

- "a.out | a.out | a.out &" を実行 (sleep(1)の無限ループ)

```
$ ps j
PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND
27890 27891 27891 pts/3 28226 Ss 24542 0:00 -csh
27891 28223 28223 27891 pts/3 28226 S 24542 0:00 ./a.out
27891 28224 28223 27891 pts/3 28226 S 24542 0:00 ./a.out
27891 28225 28223 27891 pts/3 28226 S 24542 0:00 ./a.out
27891 28225 28223 27891 pts/3 28226 S 24542 0:00 ./a.out
27891 28226 28226 27891 pts/3 28226 R+ 24542 0:00 ps j
$
```

プロセスの属性に関するシステムコール

- getpid(): プロセスID (pid) の取得
- getppid(): 親プロセスのID (ppid) の取得
- getpgrp(): プロセスグループID (pgid) の取得
- setpgrp(): pgidの設定
- getsid(): セッションID (sid) の取得
- setsid(): 新しいセッションの作成
- tcgetpgrp(): フォアグラウンドの pgid の取得
- tcsetpgrp(): フォアグラウンドの pgid の設定

プロセスの属性に関するシステムコールの構文

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
pid_t getpgrp(void);
int setpgrp(pid_t pid, pid_t pgrp);
pid_t getsid(pid_t pid); // pidが示すプロセスのsidを返す
pid_t setsid(void); // 新しくセッションを作成し、sidを返す
pid_t tcgetpgrp(int fd); // fdが示す制御端末のフォアグラウンド
                     // プロセスグループのpgidを返す
pid_t tcsetpgrp(int fd, pid_t pgrp);
                     // fdが示す制御端末のフォアグラウンド
                     // プロセスグループとしてpgrpを設定
```

fork(): プロセスの生成

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

- 機能:プロセスの複製を生成する
 - 一元のプロセスを親プロセスと呼び、生成された方を子プロセスと呼ぶ。
- 返り値
 - 親プロセスには子プロセスのpidが返る
 - 子プロセスには 0 が返る
 - エラーの場合は -1が返る
- 子プロセスはオープンしているファイル記述子などを引き継ぐ

fork()の使用例

子プロセスはこの代 入文から実行を開 始する。

```
int pid;
if ((pid = fork()) < 0){
    /* エラー処理 */
} else if (pid == 0) {
    /* 子プロセスの処理 */
    ....
} else {
    /* 親プロセスの処理 */
    ....
}
```

親プロセスの処理と 子プロセスの処理を 別々にプログラムする ことができる。

execve(): プログラムの実行

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

- 機能:新しいプログラムの実行を開始する.
 - path: 実行形ファイルのパス名
 - argv[]: プログラムの引数へのポインタの配列. 最後はNULLポインタ.
 - envp[]: 環境変数へのポインタの配列. 最後はNULLポインタ.
 - "main(int argc, char *argv[], char *envp[])" という形式でプログラムの実行が開始される.
- 実行が成功すればexecve()はリターンしない.
- オープンしているファイル記述子などを引き継ぐ.

execvp(): プログラム実行(簡易版)

```
#include <unistd.h>
extern char **environ;
int
execvp(char *file, char *argv[]);
```

- 機能:新しいプログラムの実行を開始する.
 - file: 実行形ファイルのパス名の最後のコンポーネント.
 - argv[]: プログラムへの引数へのポインタの配列. 最後はNULLポインタ.
- execvp()は環境変数PATHが示すサーチパスをたどって実 行形ファイルを探す.

wait(): 子プロセスの終了待ち

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

- ・ 機能: 子プロセスの終了を待つ.
 - status: このint型ポインタが指す領域に、子プロセスの終了状態が返る.
- 返り値
 - 終了した子プロセスのpid
 - エラーの場合は-1

exit(): プロセスの実行終了

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

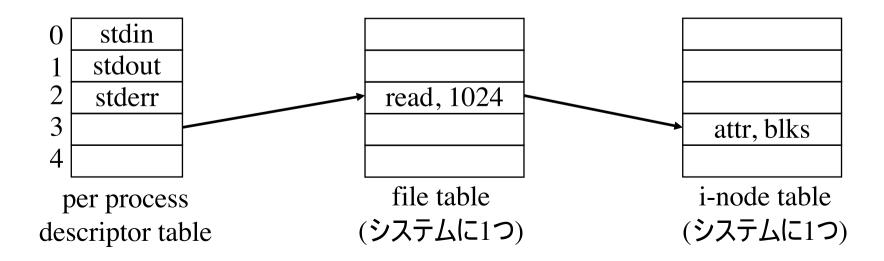
- 機能: プロセスを終了する.
 - status: プロセスの終了状態. 親プロセスに返る. (cf. wait())
 - EXIT_SUCCESS or EXIT_FAILURE
 - #include <sysexits.h> はさまざまな値を定義している.
 - 正常の場合: 0 (EX_OK)
 - EX_USAGE (64): コマンドの使用方法の誤り
 - etc.
- 返り値:なし(exit()は戻らない)

"ファイル"による入出力デバイスの抽象化

- UNIXシステムでは、通常のファイルに加えて入出力デバイスを "ファイル" として抽象化している。
 - 通常のファイル (ディクスに格納されている情報の塊り)
 - ディレクトリ (ファイル名とOS内のファイル識別子の対応付けを管理)
 - 画面(出力デバイス: /dev/console)
 - ディスク (ブロック単位の入出力: /dev/sda1 etc.)
 - etc.

ファイル関連のカーネル内のデータ構造

- Per process descriptor table
 - プロセスごとにもつ. オープンしているファイルを管理.
- File table
 - システム内でオープンされているファイルを管理.
 - read/writeのモードやオフセットなど.
- i-node table
 - ファイルの属性やデータブロックを管理する.

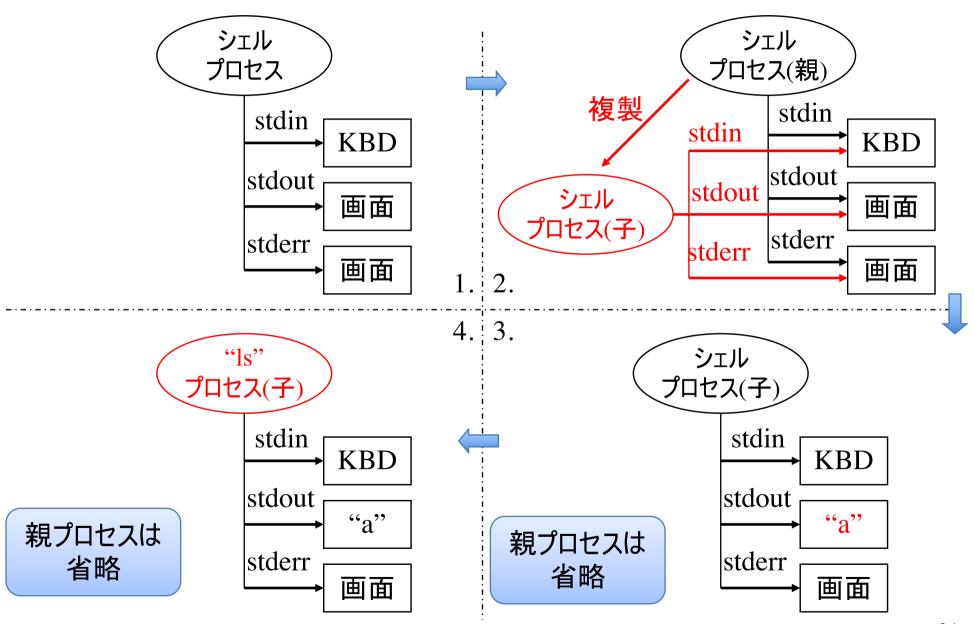


リダイレクト

- 多くのコマンドは標準入力と呼ばれる"ファイル"からデータを 入力し、標準出力と呼ばれる"ファイル"にデータを出力する ようにプログラミングされている。
 - 通常は:
 - 標準入力: キーボード
 - 標準出力: 画面
- "<"や">">"を指定することにより、シェルが標準入力や標準出力のファイルを切り替える。

```
$ 1s > output $ 標準出力を"output"という ファイルにリダイレクト
```

例: "Is > a" の実行

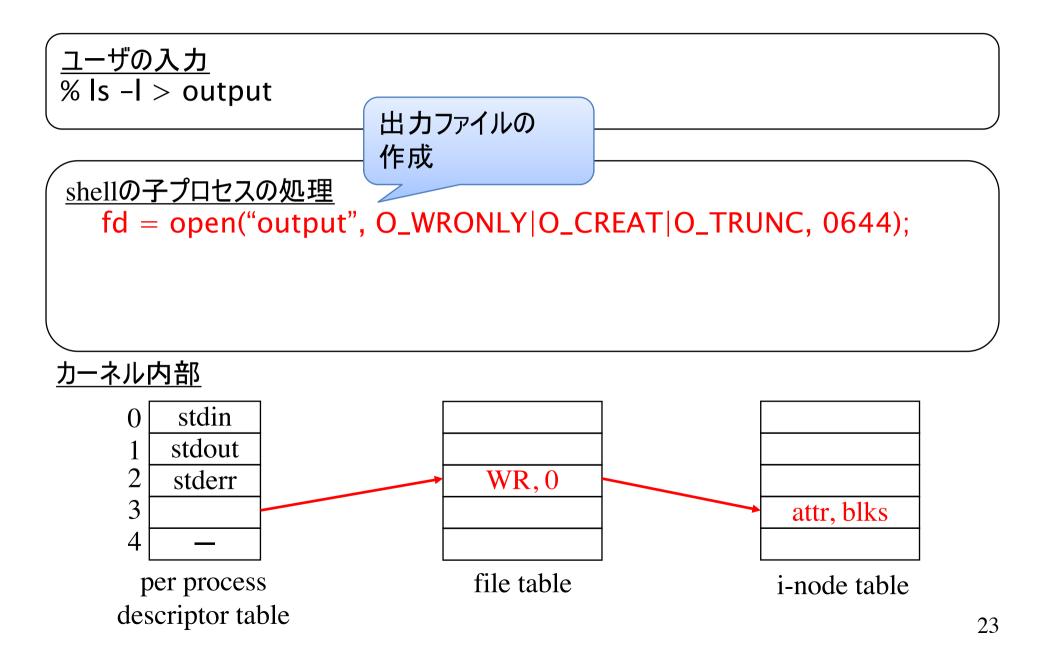


dup(): ファイル記述子の複製

```
#include <unistd.h>
int
dup(int oldfd);
```

- 機能:存在するファイル記述子の複製を作成する.
- 引数:
 - oldfd: 存在するファイル記述子.
- 返り値:複製されたファイル記述子.
 - エラーの場合は -1 が返され、errnoが原因を示す.
- リダイレクトやパイプ処理に利用される(後述).

リダイレクトの仕組み (1)

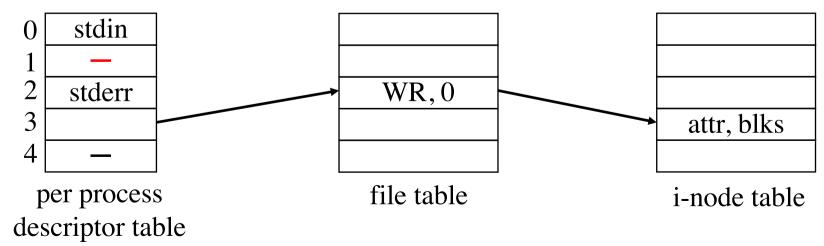


リダイレクトの仕組み(2)

```
<u>ユーザの入力</u>
% ls -l > output
```

```
shellの子プロセスの処理
fd = open("output", O_WRONLY|O_CREAT|O_TRUNC, 0644);
close(1); 標準出力を
クローズ
```

カーネル内部

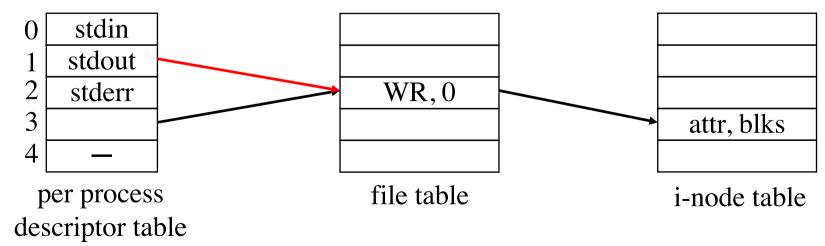


リダイレクトの仕組み (3)

```
<u>ユーザの入力</u>
% ls -l > output
```

```
shellの子プロセスの処理
fd = open("output", O_WRONLY|O_CREAT|O_TRUNC, 0644);
close(1);
dup(fd);
fdの複製を作成
→ stdout が "output" に接続される
```

カーネル内部

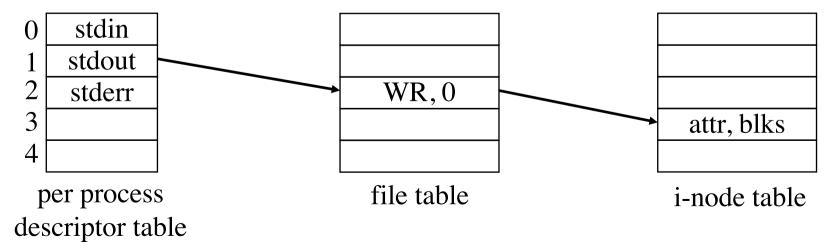


リダイレクトの仕組み (4)

```
<u>ユーザの入力</u>
% ls -l > output
```

```
shellの子プロセスの処理
fd = open("output", O_WRONLY|O_CREAT|O_TRUNC, 0644);
close(1);
dup(fd);
close(fd);
余分な fd をクローズ
この後 ls をexec する
```

カーネル内部



dup2(): ファイル記述子の複製

#include <unistd.h>
int
dup(int olfd, int newfd);

- 機能:newfdをoldfdの複製として作成する.
- 引数:
 - oldfd: 存在するファイル記述子.
 - newfd: 新しいファイル記述子. 必要であれば, まずnewfdをclose する.
- 返り値:複製されたファイル記述子.
 - エラーの場合は —1 が返され, errnoが原因を示す.

dup() vs. dup2()

• dup()を使う場合

```
shellの子プロセスの処理
fd = open("output", O_WRONLY|O_CREAT|O_TRUNC, 0644);
close(1);
dup(fd);
close(fd);
余分な fd をクローズ
この後 ls をexec する
```

• dup2()を使う場合

```
shellの子プロセスの処理
fd = open("output", O_WRONLY|O_CREAT|O_TRUNC, 0644);
dup2(fd, 1):
close(fd); 余分な fd をクローズ
この後 ls をexec する
```

演習問題

- 基本的なシェル "mysh" の作成
 - 1. プロンプト ("mysh\$ ") を表示して入力を待つ
 - "exit"が入力された終了する
 - 2. コマンドが入力されたら fork() し, 子プロセスは execvp() でコマンドを実行する.
 - 3. 親プロセスは wait() で子プロセスの終了を待つ.
 - 4.1に戻る.
- 入出力のリダイレクトができるようにする.