

# UNIXシステムプログラミング

## 第7回 シェルの動作とプロセス制御 (2)

2019年11月8日

情報工学科

寺岡文男

# pipe( ): プロセス間通信のための ファイル記述子作成

```
#include <unistd.h>

int
pipe(int filedes[2]);
```

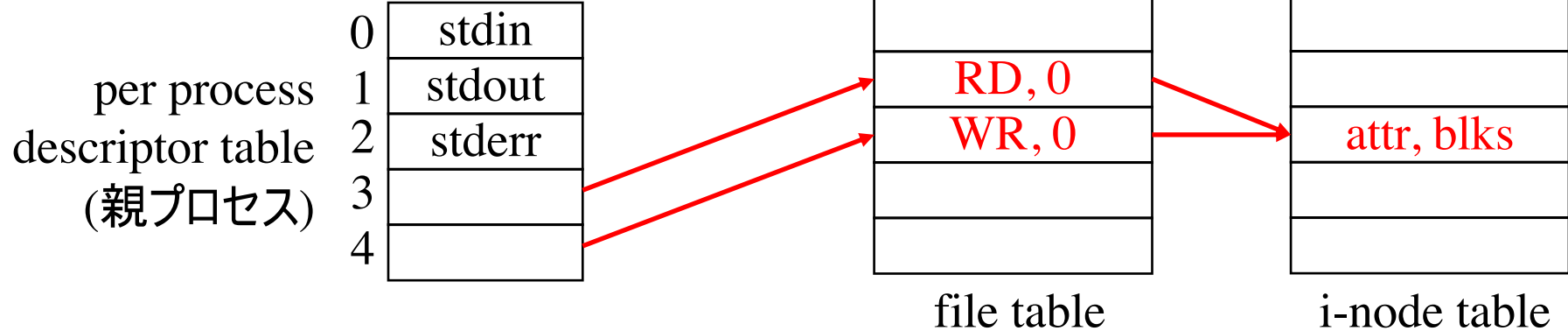
- 機能: プロセス間通信のためのパイプを作成し, ファイル記述子のペアを作成する.
- 引数:
  - filedes: ファイル記述のペアが返される領域を示す.
    - 通常, filedes[1]が出力用, filedes[0]が入力用.
- 返回值
  - 正常の場合は0, 異常の場合は -1.
- シェルのパイプ処理に利用される (後述).

int pfd[2]; と  
宣言しておく

# パイプの仕組み(1)

pipe(pfd);

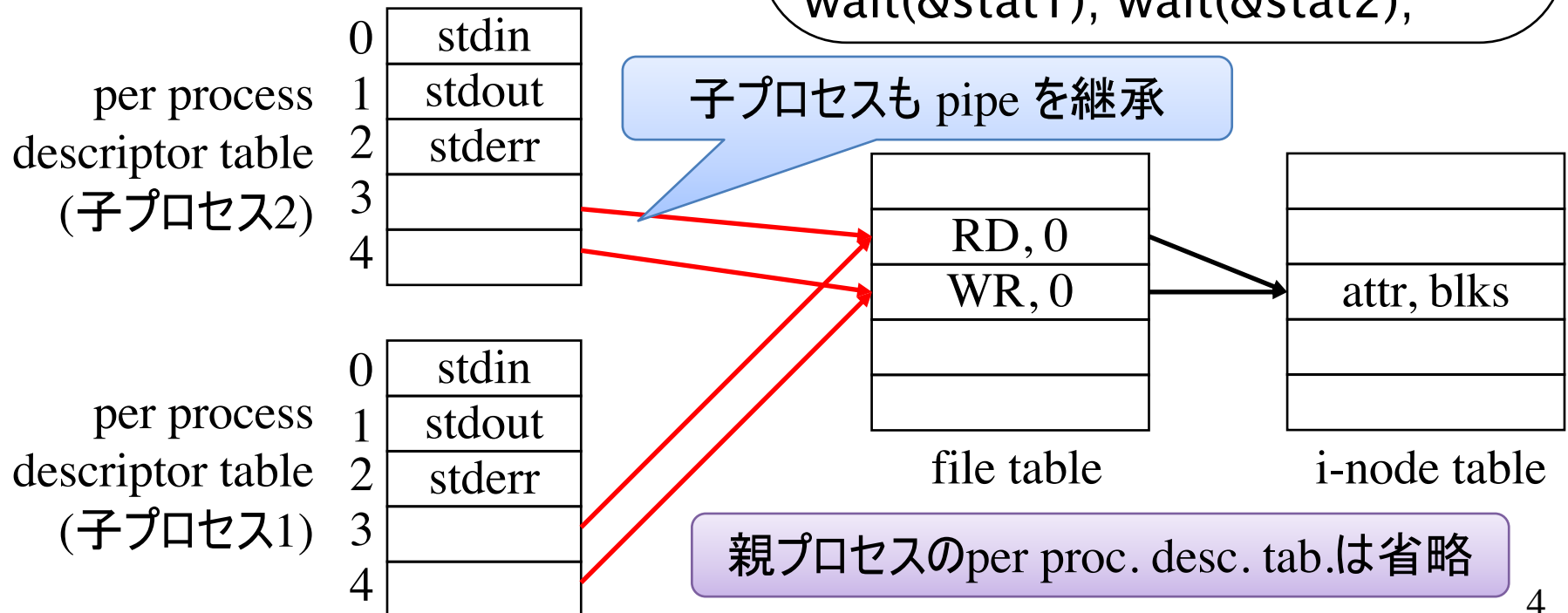
親プロセスが  
pipe を作成



# パイプの仕組み(2)

```
pipe(pfd);  
if (fork() == 0) { // 子プロセス1  
  
}
```

```
if (fork() == 0) { // 子プロセス2  
  
}  
close(pfd[0]); close(pfd[1]);  
wait(&stat1); wait(&stat2);
```



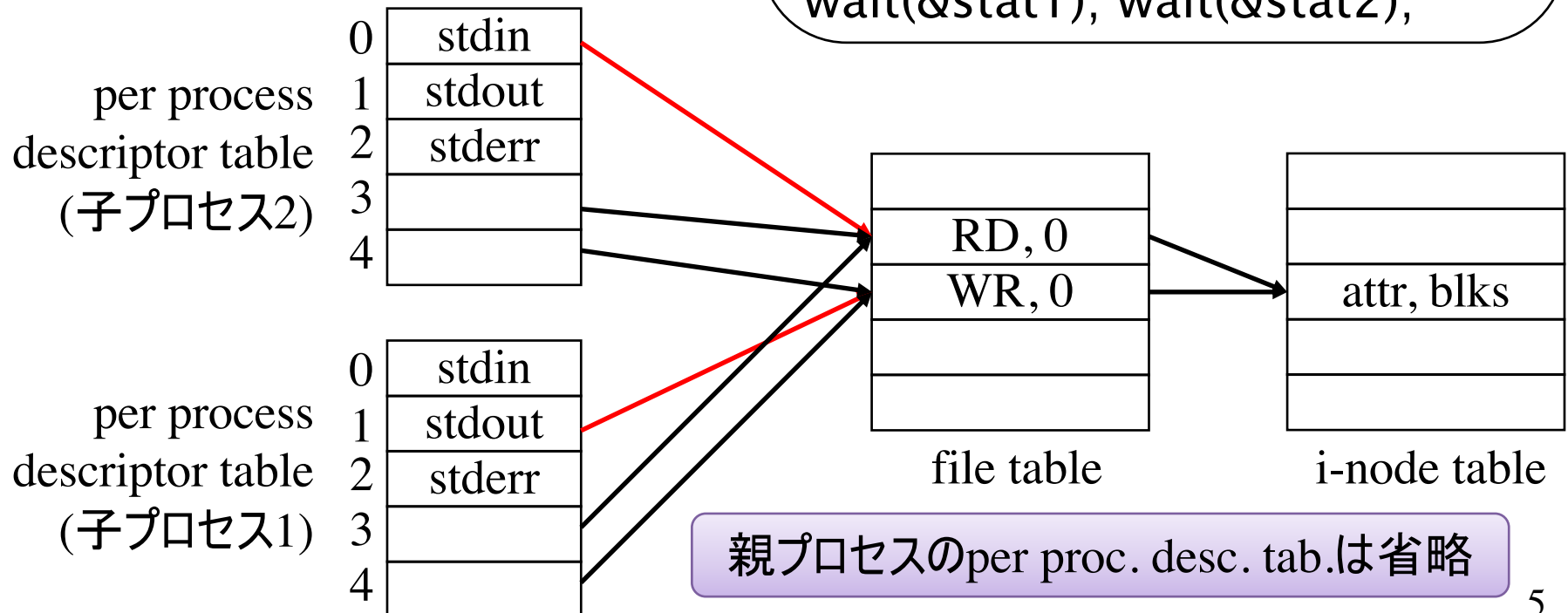
# パイプの仕組み(3)

```
pipe(pfd);  
if (fork() == 0) { // 子プロセス1  
    close(1);  
    dup(pfd[1]);  
}
```

stdout を pipe に接続

```
if (fork() == 0) { // 子プロセス2  
    close(0);  
    dup(pfd[0]);  
}  
close(pfd[0]); close(pfd[1]);  
wait(&stat1); wait(&stat2);
```

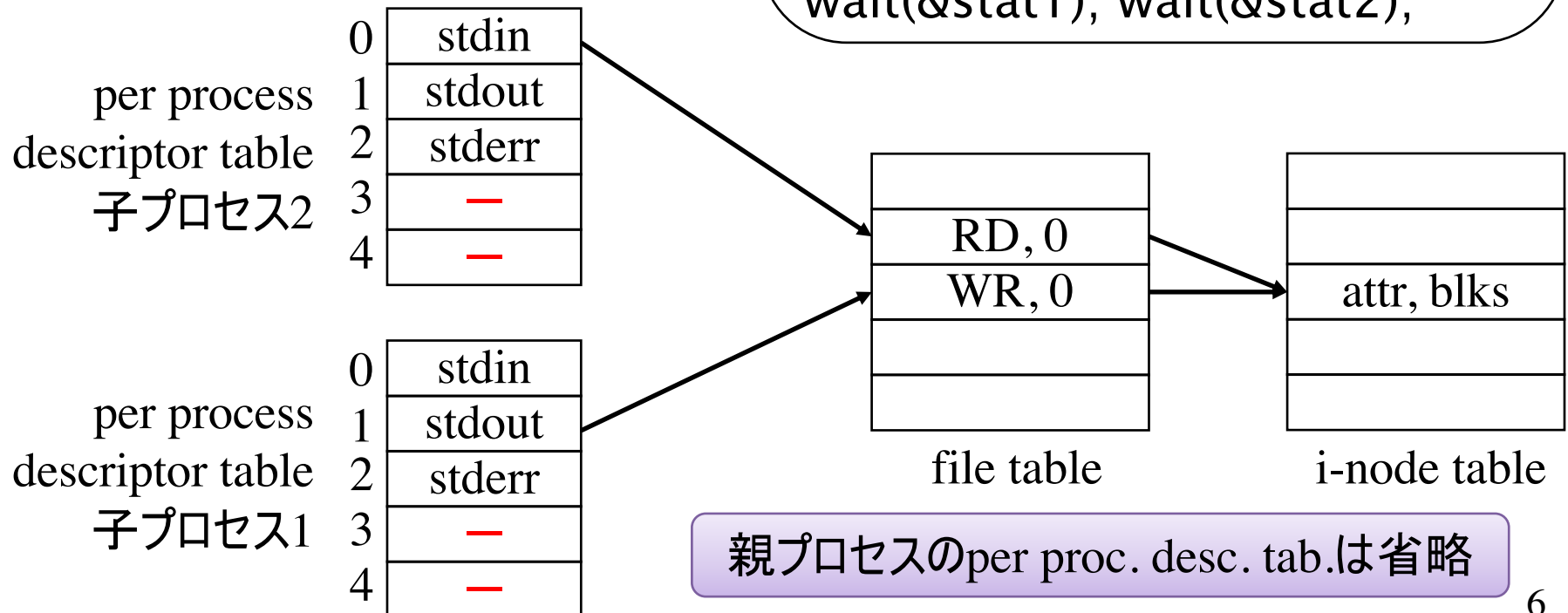
stdin を pipe に接続



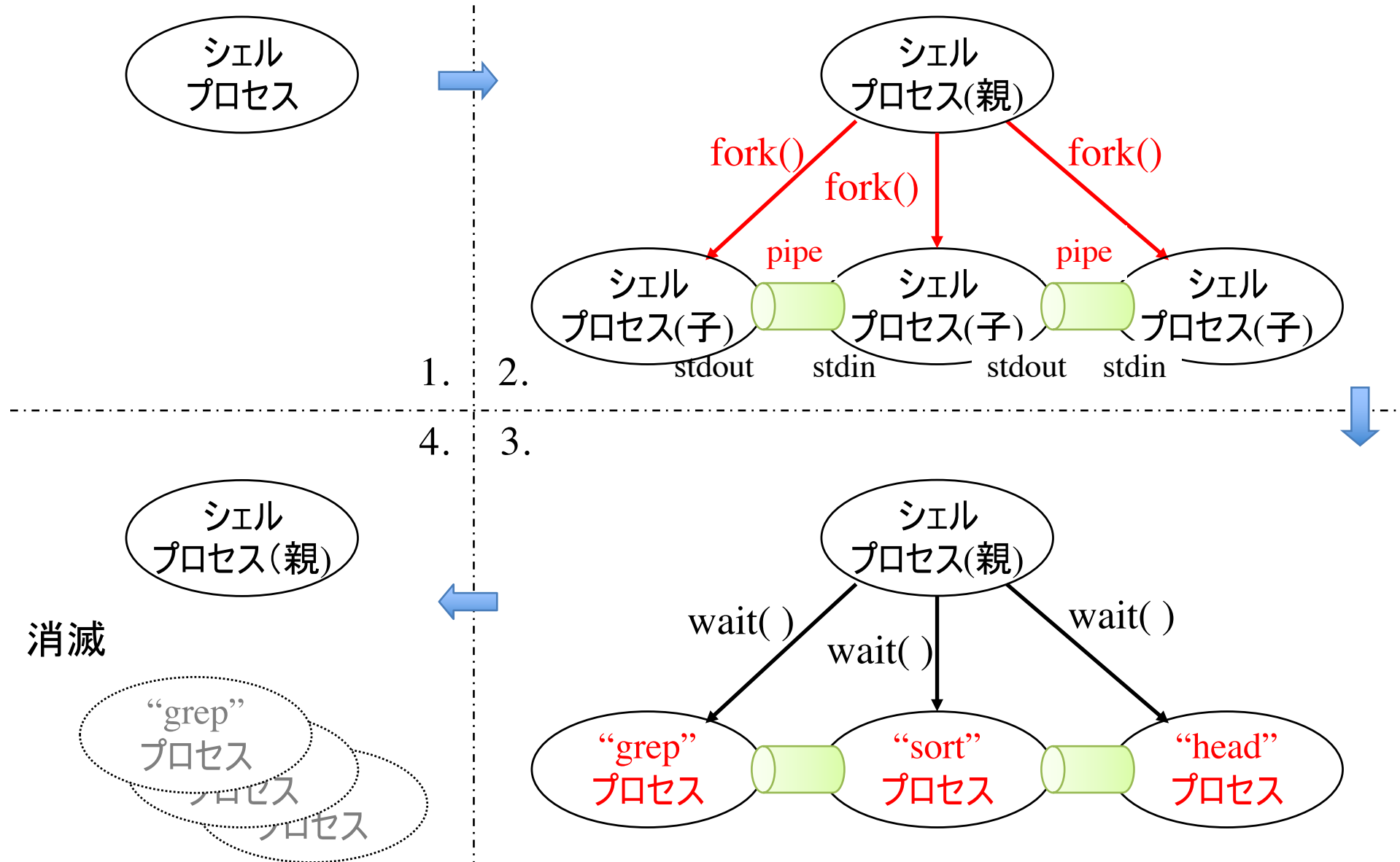
# パイプの仕組み(4)

```
pipe(pfd);  
if (fork() == 0) { // 子プロセス1  
    close(1);  
    dup(pfd[1]);  
    close(pfd[0]); close(pfd[1]);  
    execvp("ls", ...);  
}
```

```
if (fork() == 0) { // 子プロセス2  
    close(0);  
    dup(pfd[0]);  
    close(pfd[0]); close(pfd[1]);  
    execvp("wc", ...);  
}  
close(pfd[0]); close(pfd[1]);  
wait(&stat1); wait(&stat2);
```



# 例: “grep ... | sort | head” の実行



# シグナルとは

- **シグナル**: プロセスにイベントを通知する仕組み
- 2種類のシグナルがある
  - 他のプロセスやカーネルから非同期に送られるもの
  - プロセスの動作に起因するもの
- 前者の例
  - キーボードから “ctrl-C” → **SIGINT** (interrupt signal) → プロセスの終了
  - キーボードから “ctrl-Z” → **SIGTSTP** (stop signal from keyboard) → プロセスの一時実行中断
- 後者の例
  - 不正なポインタの先を参照 → **SIGSEGV** (segmentation violation) → core dumped



# 代表的なシグナルとデフォルトの動作

No.	シグナル名	デフォルトの動作	説明
1	SIGHUP	実行終了	端末ラインのハングアップ
2	SIGINT	実行終了	キーボードからの ctrl-C の入力
9	SIGKILL	実行終了	プロセスの終了
10	SIGBUS	コアイメージの作成	バスエラー
11	SIGSEGV	コアイメージの作成	セグメンテーションバイオレーション
13	SIGPIPE	実行終了	読み込みプロセスがないパイプへの書き込み
14	SIGALRM	実行終了	実時間タイマーの時間切れ
18	SIGTSTP	実行中断	キーボードからの ctrl-Z の入力
19	SIGCONT	無視	中断後の再実行
20	SIGCHLD	無視	子プロセスの終了
26	SIGVTALRM	実行終了	仮想時間タイマーの時間切れ

# プロセス制御とシグナル (1)

- キーボードからのシグナル(SIGINT, SIGTSTPなど)はその制御端末のフォアグラウンドプロセスグループに送信される.
  - バックグラウンドプロセスには送信されない.
- “fg” や “bg” コマンドでフォアグラウンドプロセスグループを変更する場合は tcsetpgrp( )でフォアグラウンドプロセスグループIDを設定する.
- バックグラウンドプロセスが終了すると親プロセスには SIGCHLDが送信される.
  - 親プロセスはwait( )によって子プロセスの終了状態を得る必要がある.

## プロセス制御とシグナル (2)

- カーネルはブロックするシグナルを示す変数 `signal mask` を保持する.
  - `signal mask` は親プロセスから継承される.
- シグナルごとに受信時のデフォルトの動作が決まっている.
  - 終了, 無視など.
- `sigaction( )` や `signal( )` によってシグナル受信時の動作を指定できる.
  - 無視する.
  - デフォルトの動作を行う.
  - 指定した関数(シグナルハンドラ)を実行する.

# sigaction( ): シグナル受信時の 動作の指定 (1)

```
#include <signal.h>

struct sigaction {
    union {
        void (*__sa_handler)(int);
        void (*__sa_sigaction)(int, struct __siginfo *,
                                void *);
    } __sigaction_u;    /* signal handler */
    int      sa_flags;   /* signal options */
    sigset_t sa_mask;    /* signal mask to apply */
};
#define sa_handler    __sigaction_u.__sa_handler
#define sa_sigaction  __sigaction_u.__sa_sigaction

int sigaction(int sig, struct sigaction *act,
              struct sigaction *oact);
```

# sigaction( ): シグナル受信時の 動作の指定 (2)

- 引数:
  - int sig: シグナルの種類
  - struct sigaction \*act: 動作を指定 (SIG\_DFL, SIG\_IGN, または handler と mask)
  - struct sigaction \*oact: ノンゼロの場合, 以前の handler の情報が返される.
  - sa\_flags (例):
    - SA\_NODEFER: 後続の同種のシグナル配送の遅延なし
    - SA\_RESETHND: handler の実行後, 設定を解除
    - SA\_RESTART: システムコール実行中にシグナルを受信してシステムコールが中断した場合, 再実行する.
  - sa\_mask: 適用されるシグナルマスク

## sigaction( ): シグナル受信時の 動作の指定 (3)

- 返り値
  - 正常: 0
  - 異常: -1 (int errnoが原因を示す)
- sigset\_t を操作するマクロ
  - `int sigaddset(sigset_t *set, int signo);`
  - `int sigdelset(sigset_t *set, int signo);`
  - `int sigemptyset(sigset_t *set);`
  - `int sigfillset(sigset_t *set);`
  - `int sigismember(sigset_t *set, int signo);`

# sigprocmask( ): シグナルマスクの設定

```
#include <signal.h>
```

```
int sigprocmask(int how, sigset_t *set,  
                sigset_t *oset);
```

- 引数

- int how: 以下の3種類の機能の1つ
  - SIG\_BLOCK: 以前のマスクと引数のマスクのOR
  - SIG\_UNBLOCK: 以前のマスクと引数のマスクのAND
  - SIG\_SETMASK: マスクの値を直接指定
- sigset\_t \*set: 設定するマスク
- sigset\_t \*oset: ノンゼロの場合, 以前のマスクの値が返る

- 返り値

- 正常: 0, 異常: -1 (int errnoが原因を示す)

# シグナルハンドラの指定 (簡易版)

```
#include <signal.h>
```

```
void (*signal(int sig, void (*func)(int)))(int);
```

- sig: シグナルの種類
- func: シグナルを処理する関数へのポインタ
  - func(int)の引数はシグナルの種類を示すint型
  - SIG\_IGN: シグナルを無視する
  - SIG\_DFL: シグナル受信時のデフォルト
- 返り値は以前に設定したハンドラへのポインタ
- sigaction( )を使う方が良い



# tcsetpgrp( ): フォアグラウンド プロセスグループIDの設定

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

```
int tcsetpgrp(int fd, pid_t pgrp_id);
```

- 引数

- int fd: 制御端末を示すファイル記述子
  - fd = open(“/dev/tty”, O\_RDWR); とするとよい
  - /dev/tty は制御端末を示すデバイスファイル
- pid\_t pgrp\_id: プロセスグループID

- 返回值

- 正常: 0
- 異常: -1 (int errnoが原因を示す)

# tcgetpgrp( ): フォアグラウンド プロセスグループIDの取得

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);
```

- 引数
  - int fd: 制御端末を示すファイル記述子
    - fd = open(“/dev/tty”, O\_RDWR); とするとよい
    - /dev/tty は制御端末を示すデバイスファイル
- 返り値
  - 正常: フォアグラウンドプロセスID
  - 異常: -1 (int errnoが原因を示す)

# shell作成における シグナルに関する注意事項

- プロセスの強制終了
  - キーボードからctrl-C → SIGINTがforeground proc groupへ.
  - 親プロセスはSIGINTを無視しなければならない.
- Backgroundでの実行
  - 親プロセスは子プロセスの終了をwait( )しない.
  - 子プロセスが終了 → SIGCHLDが親プロセスに送信.
  - 親プロセスはSIGCHLDをキャッチしなければならない.
- 子プロセスは自分のpidをpgidとして設定する.
- パイプに関連する複数の子プロセスは同一のpgidを持つようにする.

# 課題3：シェル(mysh)の作成 (1)

- ステップ1 (必須)
  - コマンドの実行ができる.
  - cdコマンドが実行できる (fork( )しないで chdir( )を実行).
  - “exit” と入力することで終了する.
  - エラーの検出をきちんと行う.
- ステップ2 (必須)
  - 標準入力と標準出力のリダイレクトができる.
  - 1段のパイプができる (e.g., “ls -l | wc -l”).
- ステップ3 (必須)
  - 多段のパイプができる.
  - ctrl-Cの入力により, 実行中のコマンドを終了する.

## 課題3：シェル(mysh)の作成 (2)

- ステップ4 (必須)
  - コマンドのバックグラウンドでの実行ができる.
  - `execvp( )` の代わりに `execve( )` を使用し, コマンドのサーチパスをたどる.
- ステップ5 (オプション)
  - 現在使用しているシェルの便利な機能を組み込む.

history, -\t'でoption一覧  
jobsでbgを示す

# 課題3：シェル(mysh)の作成 (3)

- 提出方法

- 締切: 12月5日(木) 20:00JST
- 1つのディレクトリ(e.g., mysh\_d)に必要なすべてのファイルを保存.
  - 余分なファイルは消すように (\*.oファイルなど).
- main.c の先頭にコメントとして学籍番号, 氏名を記入.
- ステップ5については, memo.txt に何を実装したかを明記.
- “mysh”という実行形ファイルを生成するMakefileを作成する.
- このディレクトリ(e.g, mysh\_d)の1つ上のディレクトリで  
“tar czf mysh\_d.tgz mysh\_d” を実行.
- keio.jp に mysh\_d.tgz ファイルをアップロード.

# ヒント (1): 入力解析

- `int gettoken(char *token, int len)`
  - `char *token;`      トークンを格納する領域へのポインタ
  - `int len;`            上記領域のサイズ
- 返り値: トークンの種類
  - `TKN_NORMAL`            英数字からなる文字列
  - `TKN_REDIR_IN`          標準入力のリダイレクト文字 ( '`<`' )
  - `TKN_REDIR_OUT`        標準出力のリダイレクト文字 ( '`>`' )
  - `TKN_PIPE`              パイプ文字 ( '`|`' )
  - `TKN_BG`                バックグラウンド文字 ( '`&`' )
  - `TKN_EOL`                end of line
  - `TKN_EOF`                end of file
  - etc.
- `getc( )` と `ungetc( )` を使う.

## ヒント(2): 多段パイプ処理方法の一例

- “\$ ... | ... | ... ” のような入力において,
  - パイプ文字で区切られた部分を1まとまりとしてループで処理
- ループ本体では,
  - `int getargs(int &ac, char *av[], ...)` でコマンド名と引数を得る
  - 返り値は終了のトークン種別 (改行, パイプ, その他)
- パイプ文字で終了 → 次段へのパイプの準備
- `fork( )` する
- 子プロセスでの処理
  - 前段からのパイプあり → `stdin` をパイプへ
  - 次段へのパイプあり → `stdout` をパイプへ
  - リダイレクトあり → `stdin` or `stdout` をファイルへ
  - `execve( )` でコマンドを実行



# 注意点 (1)

- プロセスグループ, フォアグラウンド
  - コマンドを実行する子プロセスは, shellの親プロセスとは別のプロセスグループに属するようにする.
  - フォアグラウンドプロセスグループも, 子プロセスのものとする.
  - パイプにより同時に動作するプロセス群は同一のプロセスグループに属するようにする.
- バックグラウンド
  - コマンドをバックグラウンドで実行した場合, フォアグラウンドのプロセスグループは, shellの親プロセスのプロセスグループになるようにする.

## 注意点 (2)

1. 親プロセスは子プロセス(群)を `fork( )` する
2. 親プロセスは, `tcsetpgrp( )` で子プロセスグループをフォアグラウンドに指定
3. 親プロセスは子プロセスの終了を待つ
4. 親プロセスは, `tcsetpgrp( )` で自分のプロセスグループをフォアグラウンドに指定
5. 親プロセスに **SIGTTOU** が送信される
  - デフォルトの動作は実行停止



- 親プロセスは **SIGTTOU** を無視するようにしておく