第5章　シェルの動作とプロセス制御

UNIX システムにログインすると，シェルと呼ばれるプロセスが起動する．シェルは “$” または “%” というプロンプトを表示してユーザからの入力を待ち，入力されたコマンドを実行する．本章ではシェルの動作を通じて UNIX システムのプロセス制御について概観する．本章は最後にシェルを作成するという課題 3 を課す．

5.1　シェルの動作

5.1.1　基本動作

第 1.4 節で述べたように，シェルというプログラム (たとえば “/bin/csh”) を実行しているプロセスを，通常は省略してシェルプロセスと呼ぶ．シェルプロセスは基本的に以下の動作を繰り返す．

1. プロンプト (“$”) を表示し，ユーザからの入力を待つ．
2. ユーザからの入力の構文を解釈し，まず自分自身の複製プロセス (子プロセス) を生成する．
3. 子プロセスはユーザが入力したコマンド (実行形ファイル，たとえば/bin/ls) を実行する．実行し終わると子プロセスは消滅する．
4. 親プロセスは子プロセスの実行終了を待ち，再び 1. へ返る．

図 5.1 は ls コマンドを実行する例である．図において一番右側の四角は画面である．第 1 段階では，シェルは “$” というプロンプトを表示してユーザからの入力を待っている．第 2 段階ではシェルプロセスは自分の複製 (子プロセス) を生成する．この段階では子プロセスはシェルというプログラムを実行しているが，第 3 段階では子プロセスはシェルの代わりに/bin/ls というプログラムを実行し，その結果を画面に表示している．親プロセス (シェル) は子プロセス (ls) の実行終了を待っている．第 4 段階では子プロセス (ls) の実行が終了し，子プロセスは消滅する．実行は第 1 段階に戻り，親プロセス (シェル) はまたプロンプトを表示してユーザの入力を待つ．

5.1.2　標準入出・標準出力のリダイレクト

シェルの重要な機能の 1 つに標準入出・標準出力のリダイレクトがある．たとえば図 5.2 のように実行すると，ls コマンドの出力は output というファイルにセーブされる．ls コ

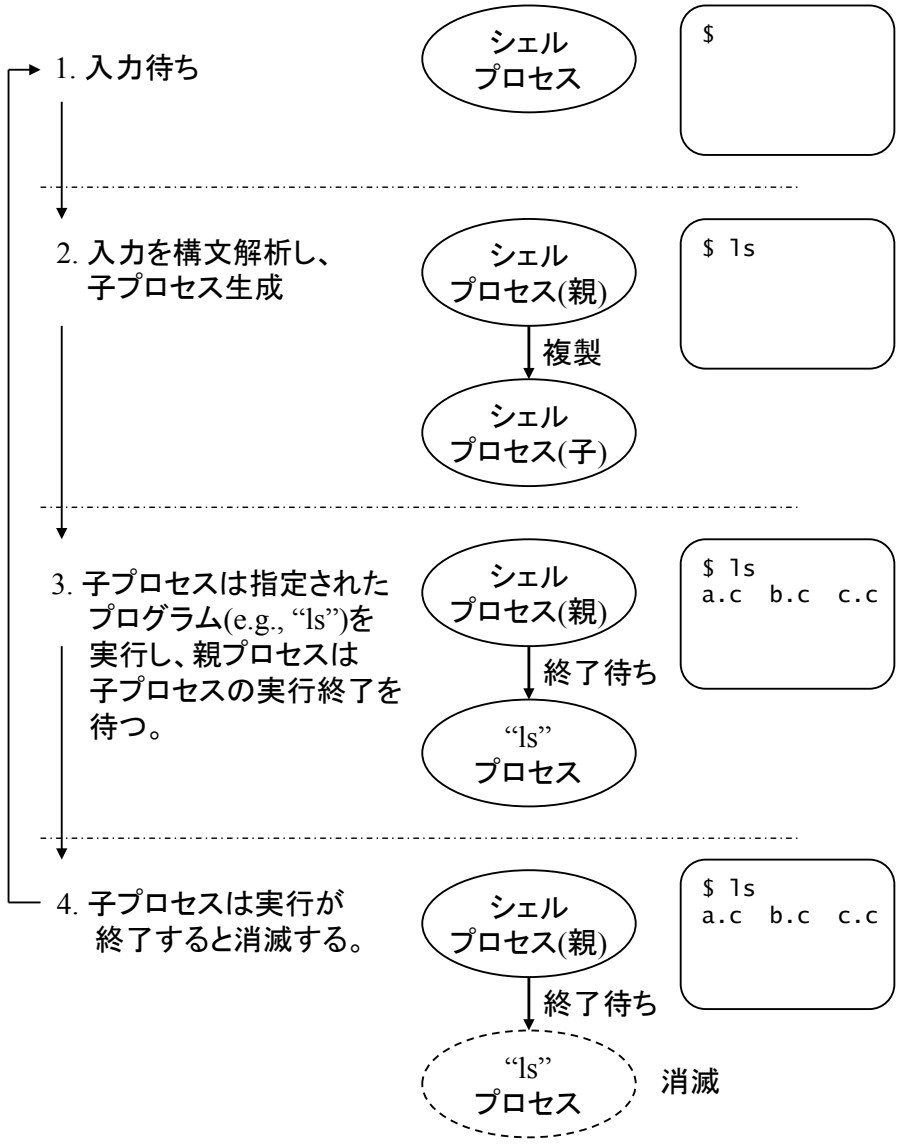

図 5.1: シェルの基本動作

マンドは標準出力に実行結果を出力するようにプログラムされている．標準出力は通常は画面になっている．それが，実行時に “>” を指定することで，実行結果が画面ではなく指定されたファイルにセーブされるようになる．不思議とは思わないだろうか．

これは，シェルが標準出力に対応するファイルの付け替え (リダイレクト) を行っているからである．具体的には以下のような処理が行われている．まず，UNIX においては通常のファイル (ディスクに格納されている情報の塊り) も入出力デバイスも “ファイル” として扱われることを思い出して欲しい．図 5.1 の第 2 段階において，子プロセスは標準出力を画面という “ファイル” からリダイレクトで指定されたファイルへ付け替えるという処理をする．その後，第 3 段階で子プロセスは `ls` プログラムの実行を開始するが，その際，標準出力はリダイレクトで指定されたファイルのままとなっている．したがって，`ls` コマンドの実行結果はリダイレクトで指定されたファイルに格納される．以上のように，実行するコマンドのプログラムにはまったく変更を加えずに，シェルの子プロセスの操作によって出力先の “ファイル” を切り替えることができるのである．標準入力のリダイレ

```
$ ls > output
$
```

図 5.2: 標準出力のリダイレクト

```
$ ls | wc -l
3
$
```

図 5.3: パイプ

クトに関しても同様に，図 5.1 の第 2 段階において標準入力をリダイレクトで指定されたファイルに付け替えることにより実現している．

5.1.3 パイプ

リダイレクトと並ぶシェルのもう 1 つの重要な機能がパイプである．たとえば図 5.3 のように実行すると，現在のディレクトリにあるファイルの数を知ることができる．この例では，“|” を指定することによって `ls` コマンドの標準出力が `wc` コマンドの標準入力に接続されている．これも不思議とは思わないだろうか．

図 5.4 にパイプの処理手順を示す．リダイレクトの仕組みから想像できると思うが，具体的には以下のような処理が行われている．図 5.4 の第 2 段階において “`ls | wc -l`” が入力されている．するとシェルの親プロセスはまず **pipe** と呼ばれる特殊なファイルを作成し，その後 2 つの子プロセスを生成する．子プロセス 1 は標準出力を pipe に付け替える．一方，子プロセス 2 は標準入力を pipe に付け替える．第 3 段階において子プロセス 1 は `ls` プログラムの実行を開始し，子プロセス 2 は `wc` プログラムの実行を開始する．すると，`ls` の実行結果は pipe に書かれ，`wc` はこの実行結果を pipe から読み込むことになる．第 4 段階において，親プロセスは両方の子プロセスの終了を待ち，両方の子プロセスの終了後，第 1 段階に返ってまたプロンプトを表示して次の入力を待つ．

5.2 プロセス制御

第 5.1 節で述べたシェルの処理を UNIX のプロセス制御の観点から述べる．プロセス制御に関するシステムコールやライブラリ関数の詳細は次節で解説する．

5.2.1 プロセスの生成

UNIX システムでは，プロセスの生成はすでに存在しているプロセスの複製という方法で行われる．このように書くと，それでは一番最初のプロセスはどのようにして生成され

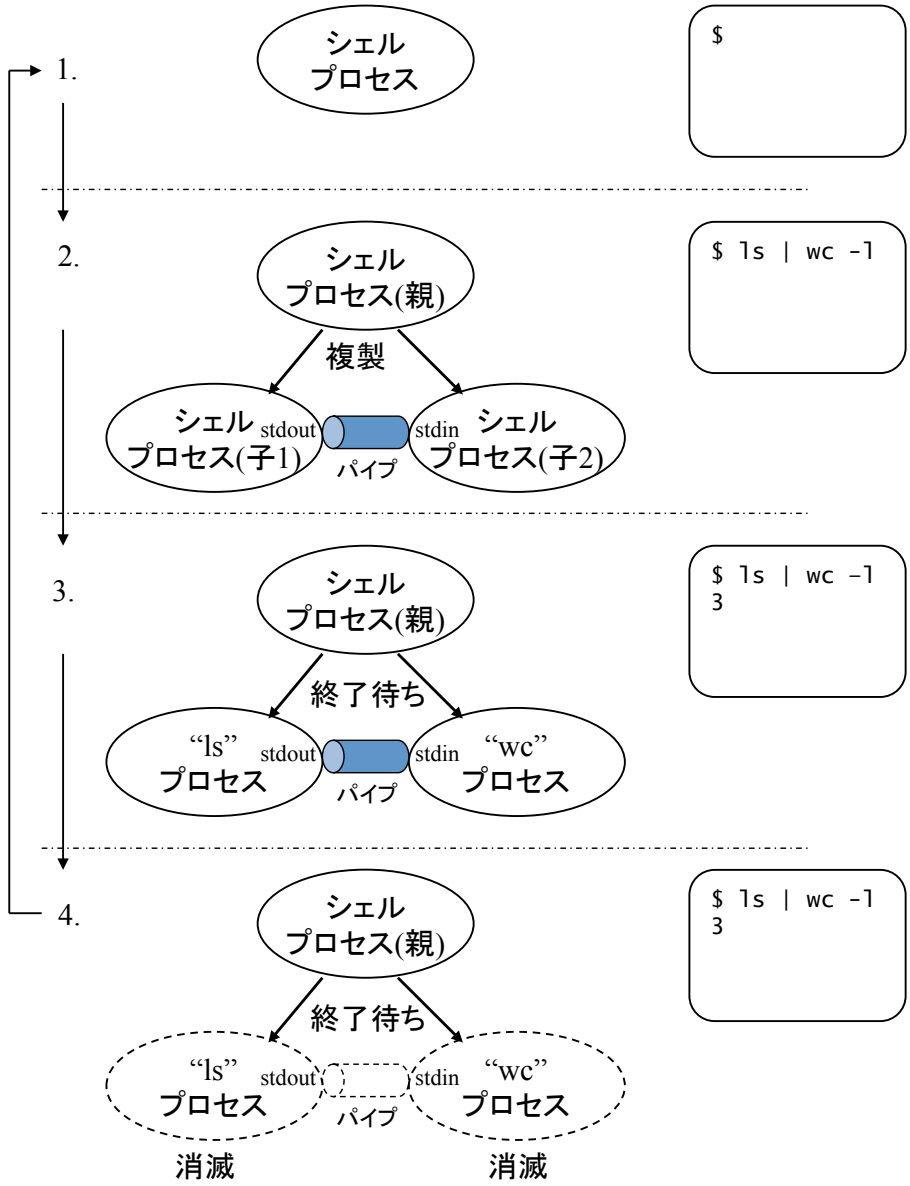

図 5.4: シェルのパイプの処理

るのか，という疑問を持つことと思う．本講義の本質から外れるので詳細は省略するが，UNIX システムの立ち上げ時には特殊な方法を用いて `init` と呼ばれるプロセスが生成される．すなわち，UNIX システムでは `init` がすべてのプロセスの祖先ということになる[1]．UNIX システムにおいては各プロセスはプロセス識別子 (pid) という整数で識別されている．`init` プロセスの pid は 1 である．`ps` コマンドを使って確かめてみるとよい (たとえば “`ps ax`” を実行)．

プロセスの複製のためには，`fork()` というシステムコールが用意されている．`fork()` を実行すると，実行中のプロセスとまったく同じ内容をもつプロセスが生成される．元のプロセスを**親プロセス** (parent process) と呼び，新しく生成されたプロセスを**子プロセス** (child process) と呼ぶ．シェルの場合，たとえば `/bin/csh` というシェルのプログラムを実行している親プロセスが `fork()` を実行する．新しく生成された子プロセスも `/bin/csh`

[1] UNIX システムにはさまざまな派生システムがあり，`init` プロセスが存在しないものもある．

```
$ ps l
F   UID   PID  PPID PRI NI    VSZ  RSS WCHAN  STAT TTY      TIME COMMAND
0 24542 27891 27890  20  0 117416 2332 rt_sig Ss   pts/130 0:00 -csh
0 24542 28167 27891  20  0 108124 1004 -      R+   pts/130 0:00 ps l
$
```

図 5.5: “ps l” の実行例

というプログラムを実行するわけであるが，親プロセスが実行した fork() の次の命令から実行を開始する．

5.2.2 プロセスの属性

“ps l” や “ps u” を実行するとプロセスのさまざまな情報が表示される．図 5.5 に “ps l” を実行したときの出力例を示す．この図からは，現在この制御端末 (ログインしている端末) ではシェル (csh) と ps コマンドが動作中であることが分かる．この実行結果の例が示すようにプロセスは多くの属性情報をもつが，ここでは主なものを紹介する．

- UID: このプロセスを起動したユーザの識別子 (user id)
- PID: このプロセスの識別子 (process id)
- PPID: このプロセスの親プロセスの pid
- PRI: このプロセスの優先度レベル
- VSZ: このプロセスが占める仮想メモリサイズ (1024 バイト単位)
- RSS: このプロセスが占める物理メモリサイズ (1024 バイト単位)
- STAT: このプロセスの状態
- TTY: このプロセスの制御端末

たとえば，ps プロセスは csh プロセスから起動されているので，ps プロセスの PPID は csh プロセスの PID と等しくなっている．また ps プロセスは現在実行中であるので STAT は現在実行中を示す “R” となっており，csh プロセスは ps プロセスの実行終了を待っているので STAT はスリープ状態を示す “S” となっている．

また，“ps j” を実行するとプロセスグループやセッション，制御端末の情報を調べることができる．たとえば，無限に sleep(1) を実行するプログラムである a.out を a.out | a.out | a.out のようにパイプで接続して起動し，“ps j” を実行すると図 5.6 のように表示される．ここでは，PID，PPID に加えて PGID や TPGID などが表示されている．これらは以下のような意味を持つ．

- PGID: このプロセスが属すプロセスグループの識別子 (process group id)

```
$ ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
27890 27891 27891 27891 pts/130  28226 Ss   24542   0:00 -csh
27891 28223 28223 27891 pts/130  28226 S    24542   0:00 ./a.out
27891 28224 28223 27891 pts/130  28226 S    24542   0:00 ./a.out
27891 28225 28223 27891 pts/130  28226 S    24542   0:00 ./a.out
27891 28226 28226 27891 pts/130  28226 R+   24542   0:00 ps j
$
```

図 5.6: “ps j” の実行例

- TPGID: このプロセスが接続している端末 (tty) 上のフォアグラウンドプロセスグループの ID (セッション ID).

UNIX システムはプロセスをプロセスグループとセッションという単位で管理している．セッションには通常制御端末が割り当てられる．図 5.6 を見ると，すべてのプロセスは 1 つのセッション (TPGID = 28226) に属していることが分かる．csh の STAT の項目に “s” が表示されており，csh がこのセッションのリーダであることが分かる．このセッションの中で，csh は 1 プロセスでプロセスグループ (PGID = 27891) を構成し，パイプで接続して起動した 3 つの a.out プロセスで 1 つのプロセスグループ (PGID = 28223) を構成し，その後起動した ps プロセスでまた 1 つのプロセスグループ (PGID = 28226) を構成していることが分かる．コマンドを単独で実行した場合はそのプロセスの PID が PGID に設定され，パイプで接続して実行した場合は，最初に実行したプロセスの PID が全体の PGID に設定されている．さらにすべてのプロセスは pts/130 を制御端末としていることが分かる．

5.2.3 プログラムの実行

fork() は現在と同じ内容を持つプロセスの複製を生成するだけであり，別のプログラムの実行を開始するわけではない．あるプログラムを実行しているプロセスが別のプログラムを実行するために，execve() というシステムコールが用意されている．また，execve() をより使いやすくしたライブラリ関数として execvp() などが用意されている．execve() を呼び出すことにより，プロセスは現在実行しているプログラムの実行をやめ，指定されたプログラムの先頭から実行を開始する．

5.2.4 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実行終了を待つ必要がある．子プロセスの実行終了を待つために，wait() というシステムコールが用意されている．wait() により，子プロセスの終了時の状態 (整数) を得ることができ，これによって親プロセスは子プロセスが正常終了したのか，異常終了したのかを知ることができる．

一方，子プロセスは exit() というライブラリ関数を呼び出すことにより実行を終了する．exit() を明示的に呼び出さなくても，main() を最後まで実行すると自動的に exit() が呼び出されて，プロセスの実行を終了するようになっている．しかし，正常に終了したことを親プロセスに通知するため，main() 関数の最後では明示的に return 0; を実行すべきである．exit() の引数 (整数) が，wait() で子プロセスの終了待ちをしている親プロセスに返される．

5.2.5 シグナル

プロセス制御のため，UNIX システムにはプロセスにシグナル (signal) と呼ばれる信号を送る機構が用意されている．シグナルには，カーネルや他のプロセスから非同期的に送られるものと，そのプロセスの処理に起因して送られるものの 2 種類がある．前者の例としてはキーボードからの crtl-C や ctrl-Z の入力によるシグナルが挙げられる．キーボードから ctrl-C を入力することによって実行中のプログラムを終了できることは知っていることと思う．キーボードから ctrl-C を入力すると，SIGINT (interrupt signal) と呼ばれるシグナルがその端末を制御端末とするフォアグラウンドのプロセスグループに送られる．デフォルトでは，プロセスが SIGINT を受け取るとカーネルはそのプロセスの実行を終了するようになっている．また，キーボードから ctrl-Z を入力すると SIGTSTP (stop signal from keyboard) がその端末を制御端末とするフォアグラウンドのプロセスグループに送られる．デフォルトでは，プロセスが SIGTSTP を受け取るとカーネルはそのプロセスの実行を一時中断するようになっている．

プロセス自身の処理に起因したシグナルは，多くはプログラムのバグによるものである．たとえば不正なポインタの先をアクセスすると SIGBUS (bus error) または SIGSEGV (segmentation violation) が発生する．デフォルトでは，プロセスが SIGBUS や SIGSEGV を受け取るとカーネルはそのプロセスを強制終了し，メモリイメージを core ファイルとして作成する．

UNIX システムには 32 種類のシグナルが定義されており，多くのシグナルについてはデフォルトの動作はプロセスの終了である．また，sigaction() システムコールやライブラリ関数 signal() によってシグナルを無視したり，シグナル受信時に実行する関数を指定したりできる (5.3.12 節参照)．

5.2.6 プロセス制御とシグナルの処理

プロセスの強制終了

前述したように，キーボードから ctrl-C を入力するとその端末を制御端末とするフォアグラウンドのプロセスグループに SIGINT が送られる．シェルからたとえば a.out を実行しているとき，ユーザが a.out プロセスの実行を終了させようとしてキーボードから ctrl-C を入力すると，SIGINT は a.out プロセスのみならずシェルプロセスにも送られる．SIGINT を受信したときのデフォルトの動作はプロセスの終了であるが，シェル自体が終了しては困る．そこでシェルプロセス自体は SIGINT を受信してもそれを無視するように設定しておく必要がある．

バックグラウンドでの実行

シェルでコマンドを起動するとき最後に “&” を指定すると，コマンドをバックグラウンドで実行する．このときシェルの親プロセスは子プロセスの実行終了を wait() システムコールで待つことなく即座にプロンプトを表示して次のコマンド入力を待つ．このときキーボードから ctrl-C を入力しても，バックグラウンドプロセスグループには SIGINT は送信されない．

一方，子プロセスの実行が終了すると親プロセスには SIGCHLD (child status has changed) が送られる．親プロセスは SIGCHLD を受け取ると wait() システムコールを呼び出して子プロセスの終了状態を確認する必要がある．親プロセスが子プロセスの実行終了を wait() システムコールで確認しないと，実行を終了した子プロセスの状態がカーネル内に残ったままになってしまう．このようなプロセスの状態を通常 “ゾンビ” と呼ぶ．

実行の中断と再開

コマンドをフォアグラウンドで実行中にキーボードから ctrl-Z を入力すると実行中のプロセスは実行を一時中断する．ctrl-Z の入力により SIGTSTP がその端末を制御端末としているフォアグラウンドのプロセスグループに送られる．フォアグランドの各プロセスは SIGTSTP を受け取ることにより実行を一時中断する．シェルの親プロセスは wait() システムコールによって子プロセスの実行終了を待つが，この間に SIGTSTP を受け取ることになる．wait() システムコールはシグナルを受信すると異常終了する．そしてシェルの親プロセスはプロンプトを表示して次の入力を待つ．ここでユーザが “fg” または “bg” を入力すると，シェルの親プロセスは実行中断中のプロセスグループに SIGCONT (continue after stop) を送信する．これを受け取ったコマンドプロセスは実行を再開する．このようにシェルがフォアグラウンドのプロセスグループを変更する際には，ライブラリ関数 tcsetpgrp() を用いてどのプロセスグループがフォアグラウンドかを指定する．

5.3 プロセス制御のシステムコール

5.3.1 fork(): プロセスの生成

図 5.7 に fork() システムコールの構文を示す．fork() は fork() を実行したプロセスの複製を作成する．元のプロセスを親プロセス (parent process) と呼び，新しく作成されたプロセスを子プロセス (child process) と呼ぶ．子プロセスは親プロセスとは別のプロセス識別子 (pid) および親プロセス識別子 (ppid: parent pid) を持つ．すなわち親プロセスの pid が子プロセスの ppid となる．また，子プロセスは親プロセスがオープンしていたファイル記述子 (file descriptor) を引き継ぐ．リダイレクトやパイプの実現はこの性質を利用している．

親プロセスにおいては fork() は子プロセスの pid を返す．一方，子プロセスにおいては fork() は 0 を返す．図 5.8 に fork() の使用例を示す．この例では 03 行目で fork() を実行し，返り値を変数 pid に代入している．子プロセスはここから実行を開始する．04，05 行目はエラー処理である．06 行目は子プロセスかどうかの判断であり，子プロセスは

```
01: #include <unistd.h>
02:
03: pid_t
04: fork(void);
```

図 5.7: fork() の構文

```
01:     int pid;
02:     ...
03:     if ((pid = fork()) < 0) {
04:         /* エラー処理 */
05:         ...
06:     } else if (pid == 0) {
07:         /* 子プロセスの処理 */
08:         ...
09:     } else {
10:         /* 親プロセスの処理 */
11:         ...
12:     }
```

図 5.8: fork() の利用例

07，08 行目を実行する．親プロセスは 10，11 行目を実行する．このようにして，親プロセスと子プロセスの実行を別々にプログラムすることができる．

5.3.2 execve(): プログラムの実行

図 5.9 に execve() システムコールの構文を示す．execve() はプロセスにおいて新しいプログラムの実行を開始する．path は新しく実行するファイルのパス名である．argv は新しく実行するプログラムへの引数リストへのポインタである．argv[0] は慣習としてプログラム名 (パス名の最後のコンポーネント) の文字列へのポインタである．argv[] の最後のエレメントは NULL ポインタでなければならない．

envp は環境変数リストへのポインタであり，新しく実行するプログラムの引数となる．main() 関数の引数は main(int argc, char *argv[]) であると教わっていると思うが，実はもう 1 つ引数がある．正確には main(int argc, char *argv[], char *envp[]) である．printenv コマンドを実行すると環境変数とその値が表示されることをコンピュータ実習で触れたが，覚えているだろうか[2]．図 5.10 に printenv コマンドの実行例を示す．環境変数は “変数名=値” という形式で表現され，環境変数名には慣習的に大文字が使わ

[2]寺岡は 2005 年度までコンピュータ実習を担当していた．

```
01: #include <unistd.h>
02:
03: int
04: execve(const char *path, char *const argv[], char *const envp[]);
```

図 5.9: execve() の構文

```
01: $ printenv
02: LANG=ja_JP.utf-8
03: USER=tera
04: LOGNAME=tera
05: HOME=/home/st-admin/tera
06: PATH=/usr/local/bin:/usr/X11R6/bin:/usr/bin:/bin
07: MAIL=/var/spool/mail/tera
08: SHELL=/bin/csh
09: SSH_CLIENT=115.162.176.3 50249 22
10: SSH_CONNECTION=115.162.176.3 50249 131.113.108.53 22
11: SSH_TTY=/dev/pts/130
12: TERM=xterm-256color
13: HOSTTYPE=x86_64-linux
14: VENDOR=unknown
15: OSTYPE=linux
16: MACHTYPE=x86_64
17: ...
10: $
```

図 5.10: printenv の実行例

れる．この例の場合，env[0] は “LANG=ja_JP.utf-8” という文字列へのポインタであり，env[1] は “USER=tera” という文字列のポインタである．envp[] の最後のエレメントは NULL ポインタである．環境変数の値を参照することにより，同じプログラムでも環境によって動作を変えるようにプログラムすることができる．

execve() の実行前後において保存されるプロセスの属性のうち主なものは以下のとおりである．

- 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)
- 制御端末

```
01: #include <unistd.h>
02: extern char **environ;
03:
04: int
05: execvp(const char *file, char *const argv[]);
```

図 5.11: execvp() の構文

```
01: #include <sys/types.h>
02: #include <sys/wait.h>
03:
04: pid_t
05: wait(int *status);
```

図 5.12: wait() の構文

- シグナルマスク
- オープンしているファイル記述子

execve() の実行が成功するとプロセスが実行しているプログラムを変更してしまうため，返り値はない．しかし実行中にエラーが発生すると execve() は-1 を返し，errno にエラーの原因がセットされる．

5.3.3 execvp(): プログラムの実行 (簡易版)

上記の execve() をもっと便利に呼び出せるようにしたライブラリ関数がいくつか用意されているが，ここでは execvp() を紹介する．図 5.11 に execvp() の構文を示す．

execve() の第 1 引数である path は実行するプログラムファイルの絶対パス名あるいは相対パス名を指定する必要がある．しかし我々はシェルでコマンドファイルを指定する際に通常は絶対パス名や相対パス名 (たとえば “/bin/ls” や “./a.out”) を指定せず，単にコマンド名 (たとえば ls) のみを指定することが多い．このような場合，シェルは環境変数の PATH に設定されているサーチパスをたどって行き，“/bin/ls” を見つける (図 5.10 の 04 行目参照)．execvp() は実行するプログラムファイルに関して内部でサーチパスの探索を行う．たとえば第 1 引数である file に単に “ls” という文字列へのポインタを設定しておけば，execvp() 内部でサーチパスの探索を行い，“/bin/ls” を実行する．また，環境変数に関しては外部変数 environ が指す領域で受け渡されるようになっている．

```
01: #include <stdlib.h>
02:
03: void
04: exit(int status);
```

図 5.13: exit() の構文

```
01: #include <sys/types.h>
02: #include <unistd.h>
03:
04: pid_t
05: getpid(void);
```

図 5.14: getpid() の構文

5.3.4 wait(): 子プロセスの終了待ち

図 5.12 に wait() システムコールの構文を示す．wait() は子プロセスの終了を待ち，返り値として終了した子プロセスのプロセス識別子 (pid) を返す．また，子プロセスが exit()(次節参照) の引数として渡した値が，int *status が指す領域に返される．何度も繰り返すが，status は int 型のメモリオブジェクトを指していなければならない．エラーの際は wait() は-1 を返す．

5.3.5 exit(): プロセスの終了

図 5.13 にライブラリ関数 exit() の構文を示す．exit() はオープンしているファイルをすべてクローズするなどの後始末をしてプロセスを終了する．引数 status によって終了状態を示す．この値は親プロセスが実行している wait() の引数が指す int 型の領域に設定される．

終了状態の数値は，<sysexits.h>ファイルに定義されている値を使用することが推奨されている．正常終了の場合には 0 または EX_OK を使用する．エラーの場合は EX_USAGE (64) などの正の数値が使用される．詳しくは “man sysexits” を参照して欲しい[3]．

5.3.6 getpid(): プロセス識別子の取得

図 5.14 に getpid() システムコールの構文を示す．getpid() は自分自身のプロセス識別子を返す．getpid() の実行にはエラーは発生しない．

[3]矢上の Linux マシンでは sysexits の man は存在しないようである．

```
01: #include <sys/types.h>
02: #include <unistd.h>
03:
04: pid_t
05: getppid(void);
```

図 5.15: getppid() の構文

```
01: #include <unistd.h>
02:
03: int
04: setpgid(pid_t pid, pid_t pgrp);
```

図 5.16: setpgid() の構文

5.3.7 getppid(): 親プロセスの識別子の取得

図 5.15 に getppid() システムコールの構文を示す．getppid() は親プロセスのプロセス識別子を返す．getppid() の実行にはエラーは発生しない．

5.3.8 setpgid(): プロセスグループの設定

図 5.16 に setpgid() システムコールの構文を示す．setpgid() は第 1 引数である pid で示すプロセスのプロセスグループ識別子を第 2 引数である pgrp に設定する．pgrp の値としては，親プロセスの pid や自分自身の pid を使用する場合が多い．正常の場合は setpgid() は 0 を返す．エラーの場合は-1 を返し，エラーの原因は errno に設定される．

シェル (csh) においては，子プロセスは自分自身のプロセス識別子をプロセスグループ識別子として指定して setpgid() を実行しているようである．パイプが指定されて複数のプロセスが同時に実行される場合は，これらのプロセス群を 1 つのプロセスグループとし，プロセスグループ識別子としては最初に生成された子プロセスのプロセス識別子を利用しているようである．このプロセスグループ識別子はキーボードから ctrl-C が入力された場合のシグナル (後述) の送信先として使われる．

5.3.9 getpgrp(): プロセスグループ識別子の取得

図 5.17 に getpgrp() システムコールの構文を示す．getpgrp() は現在属しているプロセスグループの識別子を返す．getpgrp() の実行にはエラーは発生しない．

```
01: #include <unistd.h>
02:
03: pid_t
04: getpgrp(void);
```

図 5.17: getpgrp() の構文

```
01: #include <sys/types.h>
02: #include <unistd.h>
03:
04: int
05: tcsetpgrp(int fd, pid_t pgrp_id);
```

図 5.18: tcsetpgrp() の構文

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

tcsetpgrp() は第 1 引数である fd が指す制御端末のフォアグランドプロセスグループ ID を第 2 引数である pgrp_id に設定する．第 1 引数 fd は tcsetpgrp() を呼び出すプロセスの制御端末でなければならない．さらに第 2 引数 pgrp_id は呼び出しを行うプロセスと同じセッションに属するプロセスグループでなければならない．プログラムの実行中にキーボードから ctrl-Z を入力してそのプロセスを中断し，bg コマンドや fg コマンドなどでフォアグラウンドプロセスグループを変更した際に，シェルはこの関数によってフォアグラウンドのプロセスグループを切り換える．正常の場合，tcsetpgrp() は 0 を返す．エラーの場合は-1 を返し，エラーの原因は errno に設定される．

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

tcgetpgrp() は引数である fd が指す制御端末のフォアグラウンドプロセスグループ ID を返す．エラーの場合は-1 を返し，エラーの原因は errno に設定される．

```
01: #include <sys/types.h>
02: #include <unistd.h>
03:
04: pid_t
05: tcgetpgrp(int fd);
```

図 5.19: tcgetpgrp() の構文

表 5.1: 主なシグナルの種類

番号	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 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

5.3.12 sigaction(): シグナル処理の指定

現在の UNIX システムには 32 種類のシグナルが定義されている．大部分のシグナルに関しては，シグナル受信時のデフォルトの動作は実行の終了である．シグナルの種類によってはシグナルの受信を無視したり，またはシグナル受信によって指定した関数を実行することもできる．またシグナルの種類によってはシグナル受信を無視したり指定した関数を実行することはできず，必ず実行が終了してしまうものもある．

32 種類のシグナルのうち，主なものを表 5.1 に示す．SIGINT や SIGTSTP についてはキーボードからの ctrl-C や ctrl-Z の入力に関連しており，プロセスの実行とは非同期に送られる．また，ポインタを使ったプログラムにバグがあると，実行中に “segmentation violation: core dumped” のような表示があり，プロセスのコアイメージのファイルが作られるのを経験したことがあると思う．このような場合，バグのあるプログラムの実行が原因となって SIGBUS や SIGSEGV がプロセスに送られているのである．余談であるが，SIGHUP は，コンピュータの遠隔端末を電話回線で接続していたときの名残であり，電話回線が切断したことを知らせるシグナルであった．しかし現在は電話回線で遠隔端末を接続することはないので，別の用途に使われている．

プロセスへのシグナルの送信はハードウェア割り込みの発生に似ている．シグナルがプロセスに送信されると，さらなる同じ種類のシグナル送信は通常はブロックされる．シグナルが配送されると現在のプロセスのコンテキストは退避され，新しいコンテキストが生成される．プロセスはシグナルを処理するハンドラを指定したり，シグナルの受信を無視することもできる．またはシグナルによるデフォルトのアクションを選択することもできる．ブロックされているシグナルはブロックが解除されると配送される．通常，シグナルハンドラはプロセスの現在のスタック上で動作する．

カーネルはプロセスがどのシグナルをブロックするかを示す変数 (signal mask) を保持している．signal mask の値は親プロセスから継承される．通常 signal mask の値はエンプティであるので，すべてのシグナルが配送されることになる．signal mask の値は sigprocmask() によって変更できる．

シグナル配送の条件が発生すると，シグナルは一種の待ち行列に加えられる．そのシ

```
01: #include <signal.h>
02:
03: struct sigaction {
04:     union {
05:         void (*__sa_handler)(int);
06:         void (*__sa_sigaction)(int, struct __siginfo *, void *);
07:     } __sigaction_u;   /* singal handler */
08:     int      sa_flags; /* signal options */
09:     sigset_t sa_mask;  /* signal mask to apply */
10: };
11:
12: #define sa_handler   __sigaction_u.__sa_handler
13: #define sa_sigaction __sigaction_u.__sa_sigaction
14:
15: int
16: sigaction(int sig, const struct sigaction * restrict act,
              struct sigaction * restrict oact);
```

図 5.20: setsigaction() の構文

グナルがブロックされていない場合，シグナルがプロセスに配送される．プロセスが，たとえばシステムコールの実行，ページフォルトやトラップまたはクロック割込みの発生などでカーネル内を実行しているときに，シグナルが配送される．複数のシグナルが配送可能なときには，トラップによって生じたシグナルが最初に配送される．どのシグナルが配送待ちになっているかは sigpending() によって得ることができる．シグナルが配送されると，ハンドラが実行されている間は新しい signal mask が設定される．新しい signal mask の値は，現在の singal mask の値，配送されたシグナル，ハンドラに関連付けられた signal mask の値の論理和となる．

図 5.20 に sigaction() システムコールの構文を示す．sigaction() は第 1 引数 sig を受信した際の処理方法を設定する．第 2 引数 act がゼロで無い場合は，処理方法 (SIG_DFL，SIG_IGN またはハンドラ関数) と signal mask が設定される．第 3 引数 oact がゼロで無い場合は以前に設定した処理方法の情報が返される．処理が成功すると sigaction() はゼロを返す．エラーの場合は-1 を返し，エラーの原因は errno に設定される．

sigaction() で設定した処理方法は次の sigaction() または execve() が実行されるまでは変化しない．sa_handler に SIG_IGN を設定するとペンディング中のものも含めて指定されたシグナルが破棄される．sa_handler に SIG_DFL を設定すると，指定されたシグナルのデフォルトの動作が設定される．デフォルトの動作については表 5.1 を参照のこと．

sa_flags によってシグナル処理の細かい動作を指定することができる．本テキストでは SA_RESTART のみを説明する．その他のフラグについては man ページを参照のこと．システムコールの実行中にシグナルを受信すると，システムコールは強制終了され，errno に

```
01: #include <signal.h>
02:
03: void (*signal(int sig, void (*func)(int)))(int);
```

図 5.21: signal() の構文

は EINTR が設定される．データ転送を伴うシステムコールの場合，要求したサイズに満たないサイズのデータ転送でシステムコールが終了する場合もある．このとき，sa_flags に SA_RESTART が設定されているとシステムコールは再実行される．再実行に関係するシステムコールは以下のとおりである．open()，read()，write()，sendto()，recvfrom()，sendmsg()，recvmsg()．

fork() の実行後，signal mask や再実行のフラグは子プロセスに継承される．execve() の実行後，無視するように設定されたシグナルや signal mask もそのままになる．

シグナルハンドラで使用が許されるシステムコールとそうでないものがある．それらをいちいち挙げるのは煩雑なので，詳細は man ページを参照のこと．

5.3.13 signal(): シグナル処理の指定 (簡易版)

singal() は sigaction() システムコールの使い方を簡略化したライブラリ関数である．図 5.21 に signal() の構文を示す．signal() はシグナルを受信したときの動作を指定するための関数であり，シグナルを送るための関数ではない．この構文は理解しづらいかもしれないが，第 1 引数 sig は int 型でシグナル番号を指定し，第 2 引数 func は返り値のない関数へのポインタであり，signal() 自身は返り値のない関数へのポインタを返す，という定義である．

第 2 引数 func によってシグナルを受信したときの動作を指定することができる．デフォルトの動作を指定する場合は SIG_DFL を指定する．シグナルを無視する場合には SIG_IGN を指定する．シグナルの受信によって特定の関数を実行したい場合は，その関数へのポインタを指定する．一度設定した動作は，次に別の動作を指定するまで有効である．singal() の返り値は，指定した sig に関する現在の func の値である．

5.3.14 sigprocmask(): signal mask の操作

図 5.22 に sigprocmask() システムコールの構文を示す．第 2 引数 set が non-zero の場合は対象となるシグナルを示し，第 1 引数 how で以下の 3 つ動作のうちの 1 つを指定する．

- SIG_BLOCK: 指定されたシグナルがマスクされるように signal mask が設定される．
- SIG_UNBLOCK：指定されたシグナルのマスクが解除されるように signal mask が設定される．
- SIG_SETMASK：指定された値が signal mask の値になる．

```
01: #include <signal.h>
02:
03: int
04: sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

図 5.22: sigprocmask() の構文

```
01: #include <sys/types.h>
02: #include <signal.h>
03:
04: int
05: kill(pid_t pid, int sig);
```

図 5.23: kill() の構文

第 3 引数 oact が non-zero の場合は以前の signal mask が返される．第 2 引数 act が null の場合は signal mask の値は変更されず，現在の値が oact に返される．

処理が正常に終了すると sigprocmask() は 0 を返す．エラーの場合は-1 を返し，errno にエラーの原因が設定される．

5.3.15 kill(): シグナルの送信

図 5.23 に kill() システムコールの構文を示す．pid が 0 より大きい場合は pid が指定するプロセスに sig で指定したシグナルが送られる．pid が 0 のときは kill() を呼び出したプロセスのプロセス識別子をグループ識別子とするプロセスグループ全体に sig で指定したシグナルが送信される．正常の場合は kill() は 0 を返し，エラーの場合は-1 を返す．エラーの原因は errno に設定される．

```
01: #include <signal.h>
02:
03: int
04: killpg(pid_t pgrp, int sig);
```

図 5.24: killpg() の構文

```
01: #include <unistd.h>
02:
03: int
04: dup(int oldfd);
```

図 5.25: dup() の構文

5.3.16 killpg(): プロセスグループへのシグナルの送信

図 5.24 に killpg() システムコールの構文を示す．pgrp が指定するプロセスグループに sig が指定するシグナルが送信される．pgrp が 0 の場合は killpg() を呼び出したプロセスが属するプロセスグループにシグナルが送信される．正常の場合は killpg() は 0 を返し，エラーの場合は-1 を返す．エラーの原因は errno に設定される．

5.3.17 システムコールの実行とシグナル受信

前述のように，シグナルにはプロセスの動作とは非同期に送られるものがある．システムコールには，実行中にシグナルを受信すると実行を中断して-1 を返すものと自動的に再実行するものがある．シグナルによって実行が中断されると，errno には EINTR が設定される．一方，自動的に再実行されるシステムコールには，read()，write()，sendto()，recvfrom()，sendmsg()，recvmsg() などがある．詳しくはそれぞれのシステムコールの man ページを参照のこと．

5.4 リダイレクト・パイプのためのシステムコール

5.4.1 dup(): ファイル記述子の複製

図 5.25 に dup() システムコールの構文を示す．dup() は oldd が指定するファイル記述子の複製を作成し，そのファイル記述子を返す．返されるファイル記述子の値は，現在使われていない最も小さいファイル記述子の値である．エラーの場合，dup() は-1 を返し，errnor にエラーの原因が設定される．

シェルのリダイレクトやパイプは dup() を使って実現されているが，上記の説明だけではどのようにすればよいのか分からないと思うので，以下で詳しく説明する．図 5.26 から図 5.29 にリダイレクトの手順を示す．図において上部はユーザ空間であり，1 つのプロセスが動作しているものとする．図の下部はカーネル空間である．

カーネル空間にはプロセスごとに **per process descriptor table** と呼ばれるテーブルがあり，各エントリにオープンしているファイルが登録される．per process descriptor table のエントリのインデックスがファイル記述子となる．標準入力，標準出力，標準エラー出力のファイル記述子はそれぞれ 0，1，2 であることは述べたが，これはすなわち per process descriptor table の最初の 3 つのエントリが標準入力，標準出力，標準エラー出力

のためのエントリであることを示す．またカーネル空間には**file table**と**i-node table**と呼ばれる2つのテーブルがある．これらのテーブルはシステムに1つだけある．file tableの各エントリはオープンされているファイルのread/writeのモードや，次にどの位置からread/writeを行うかを示すオフセットなどの情報が保持される．i-node tableの各エントリは対応するファイルがディスク上のどのブロックから構成されているかという情報やファイルの属性情報を保持している．

図5.26 (手順1)はプロセスが標準出力のリダイレクト先として指定された`output`というファイルを書き込みのためにオープンしたところである．カーネル内部ではper process descriptor tableの空きエントリを先頭から探し，3のエントリが空いていたのでこのエントリを使用したところである．すなわち，この`open()`はファイル記述子として3を返す．図5.27 (手順2)では標準出力である1をクローズしている．そのため，カーネル空間ではper process descriptor tableの1のエントリが解放されて未使用状態になっている．図5.28 (手順3)では`dup()`システムコールにより，手順1でオープンしたファイル記述子を複製している．カーネル空間ではper process descriptor tableの空きエントリを先頭から探し，最初に見つかったエントリが複製のために使われる．この例では1のエントリが見つかるので，このエントリは手順1でオープンした`output`というファイルのためのfile tableエントリを指すようになる．この状態では，ファイル記述子の1(標準出力)と3が共に`output`というファイルを指している．手順1でオープンして得たファイル記述子はすでに不要になったため，図5.29 (手順4)ではこのファイル記述子をクローズしている．結果として，標準出力は`output`というファイルを指すようになっている．このあとに，たとえば`execve()`システムコールで`ls`コマンドを実行すると，`ls`コマンドの実行結果は`output`というファイルに書き込まれる．

5.4.2 `pipe()`: パイプの作成

図5.30に`pipe()`システムコールの構文を示す．パイプとは双方向のデータ通信を提供する機構である．`pipe()`は`fildes[0]`にreadのためのファイル記述子を返し，`fildes[1]`にwriteのためのファイル記述子を返す．パイプは双方のファイル記述子がクローズされるまで存在する．また，片方のファイル記述子がクローズされたパイプへ書き込むと`SIGPIPE`が発生する．`pipe()`は正常の場合は`0`を返し，エラーの場合は`-1`を返す．

シェルのパイプの機能は`pipe()`システムコールと`dup()`システムコールを利用して実現されている．図5.31から図5.34に手順を示す．図の上部はユーザ空間であり，シェルの動作を示している．図の下部はカーネル空間である．

図5.31 (手順1)ではシェルの親プロセスが`pipe()`を実行している．するとカーネル空間では親プロセス用per process descriptor tableの2つのエントリ(3と4)がパイプ用に使われ，`pfd[0]`には3が返り，`pfd[1]`には4が返る．図5.32 (手順2)ではシェルの親プロセスは`fork()`を2回実行し，2つの子プロセスを生成している．たとえば“`ls | wc`”を実行する場合，1番目の子プロセスが`ls`を実行し，2番目の子プロセスが`wc`を実行する．図ではカーネル空間の親プロセスのper process descriptor tableの記述は省略している．2つの子プロセスのper process descriptor tableは，親プロセスのものを複製したものとなる．図5.33 (手順3)では，1番目の子プロセスはリダイレクトと同様の手法により，

標準出力を pfd[1] につなぎ換えている．同様に 2 番目の子プロセスは標準入力を pfd[0] につなぎ換えている．図 5.34 (手順 4) では，双方の子プロセスは余分なファイル記述子をクローズし，それぞれのプログラムの実行を開始している．最終的には，1 番目の子プロセスの標準出力はパイプに対して行われ，2 番目の子プロセスはパイプに書き込まれたデータを標準入力から読み込むことになる．

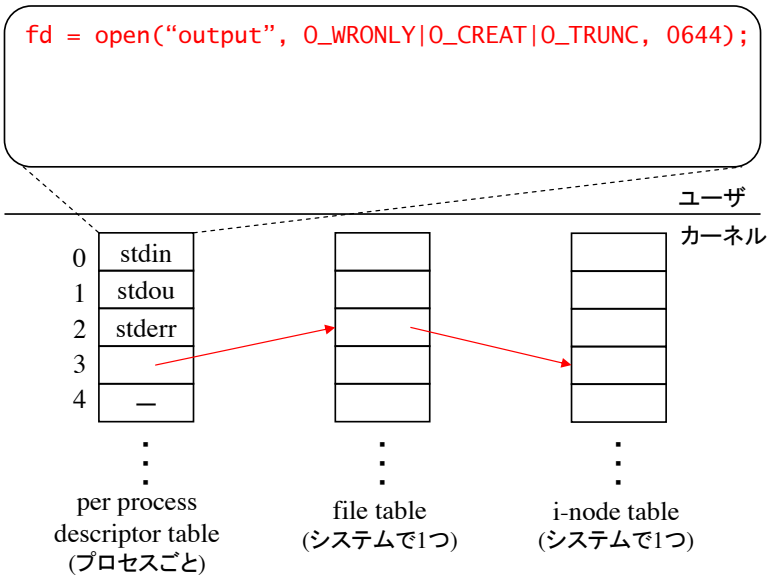

図 5.26: リダイレクト：手順 1

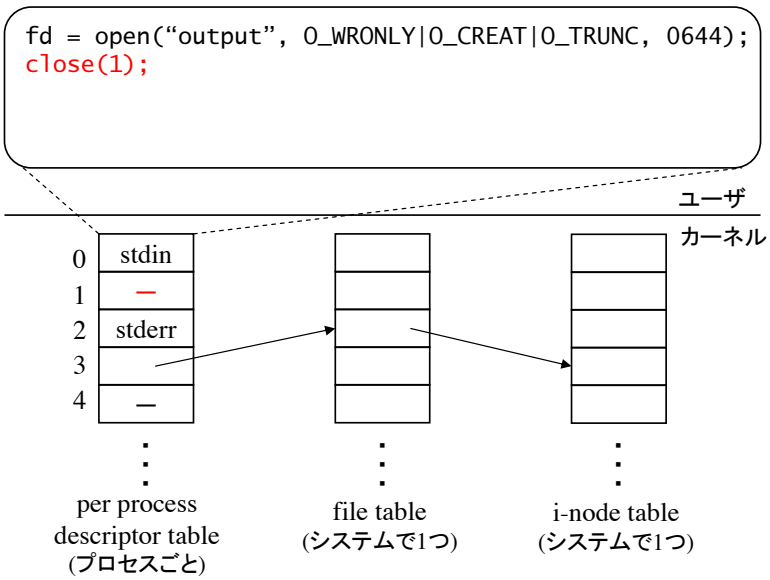

図 5.27: リダイレクト：手順 2

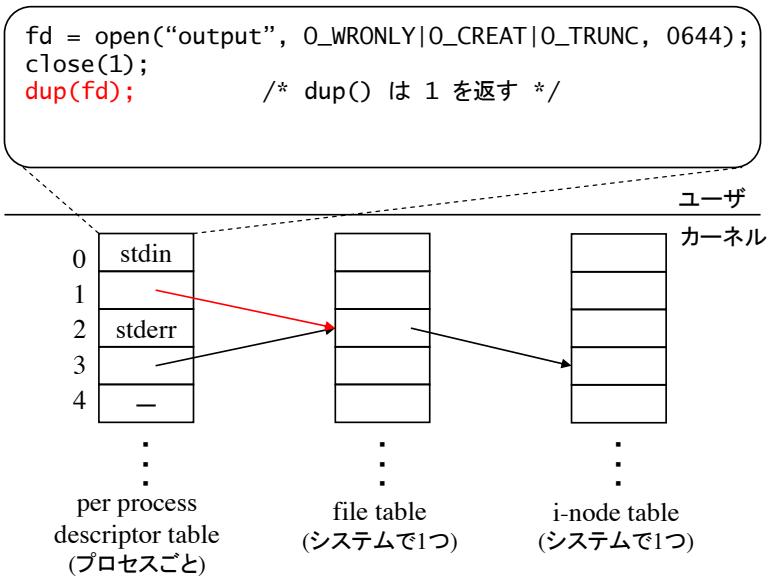

図 5.28: リダイレクト：手順 3

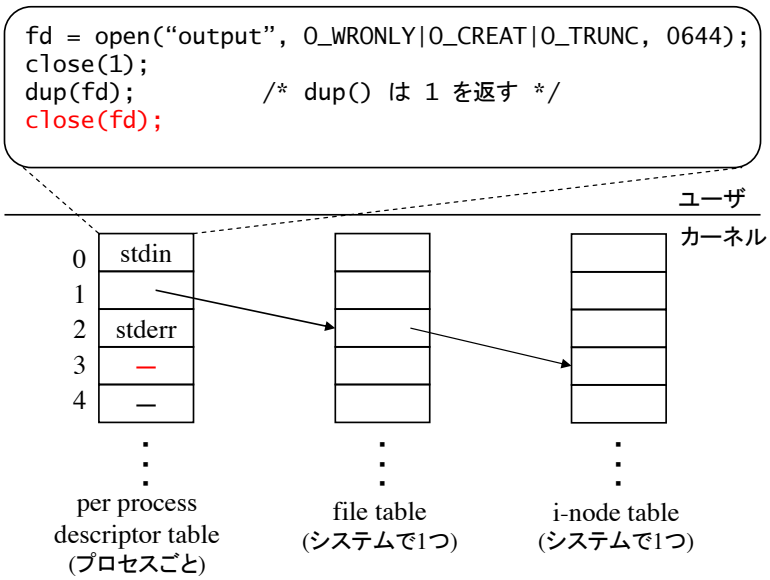

図 5.29: リダイレクト：手順 4

```
01: #include <unistd.h>
02:
03: int
04: pipe(int *fildes);
```

図 5.30: pipe() の構文

```
pipe(pfd);                  // パイプを作成
```

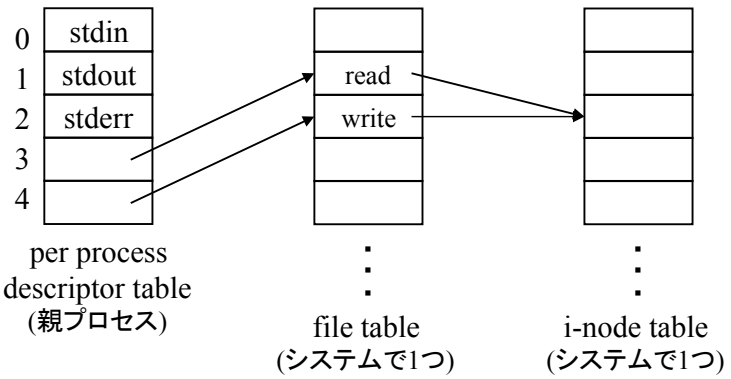

図 5.31: パイプ：手順 1

```
pipe(pfd);                  // パイプを作成
if (fork() == 0) {          // 子プロセス1, e.g., “ls”



}
if (fork() == 0) {          // 子プロセス2, e.g., “wc”



}
/* 親プロセス */
close(pfd[0]); close(pdf[1]); // pipeをクローズ
wait(&stat1); wait(&stat2);   // 子プロセスの終了待ち
```

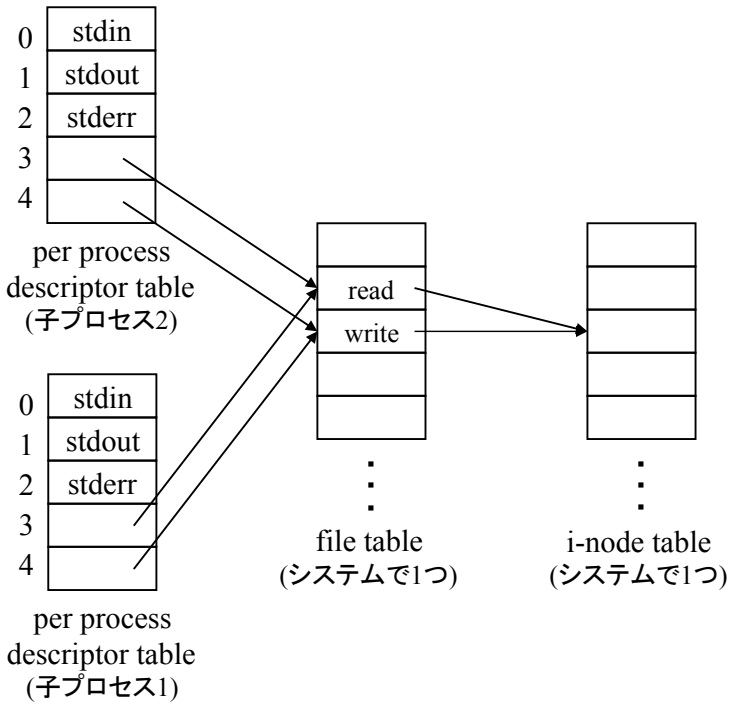

図 5.32: パイプ：手順 2

```
pipe(pfd);                  // パイプを作成
if (fork() == 0) {          // 子プロセス1, e.g., “ls”
    close(1);               // 標準出力をクローズ
    dup(pfd[1]);            // 標準出力をパイプへ


}
if (fork() == 0) {          // 子プロセス2, e.g., “wc”
    close(0);               // 標準入力をクローズ
    dup(pfd[0]);            // 標準入力をパイプへ


}
/* 親プロセス */
close(pfd[0]); close(pdf[1]); // pipeをクローズ
wait(&stat1); wait(&stat2);   // 子プロセスの終了待ち
```

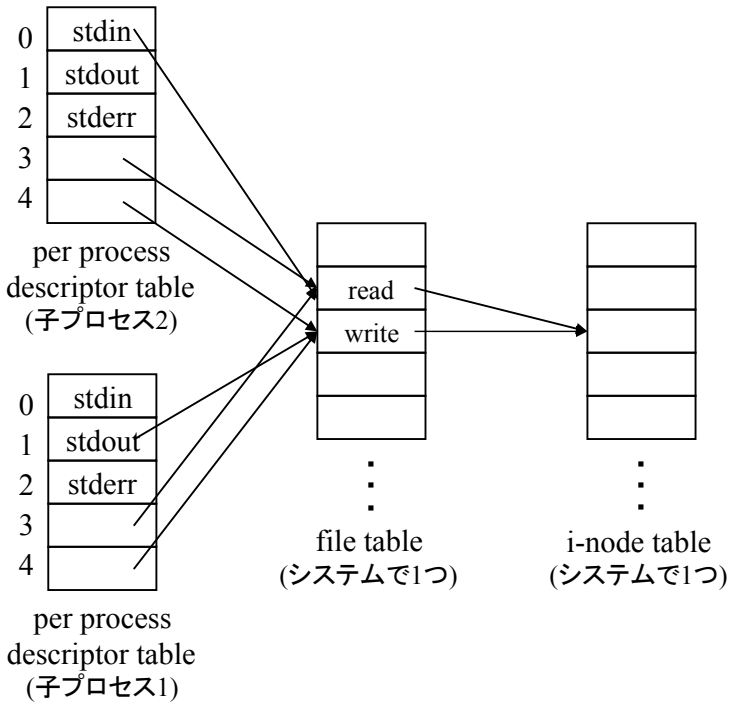

図 5.33: パイプ：手順 3

```
pipe(pfd);                  // パイプを作成
if (fork() == 0) {          // 子プロセス1, e.g., “ls”
    close(1);               // 標準出力をクローズ
    dup(pfd[1]);            // 標準出力をパイプへ
    close(pdf[0]); close(pfd[1]);
    execvp(“ls”, ...);
}
if (fork() == 0) {          // 子プロセス2, e.g., “wc”
    close(0);               // 標準入力をクローズ
    dup(pfd[0]);            // 標準入力をパイプへ
    close(pdf[0]); close(pfd[1]);
    execvp(“wc”, ...);
}
/* 親プロセス */
close(pfd[0]); close(pdf[1]); // pipeをクローズ
wait(&stat1); wait(&stat2);   // 子プロセスの終了待ち
```

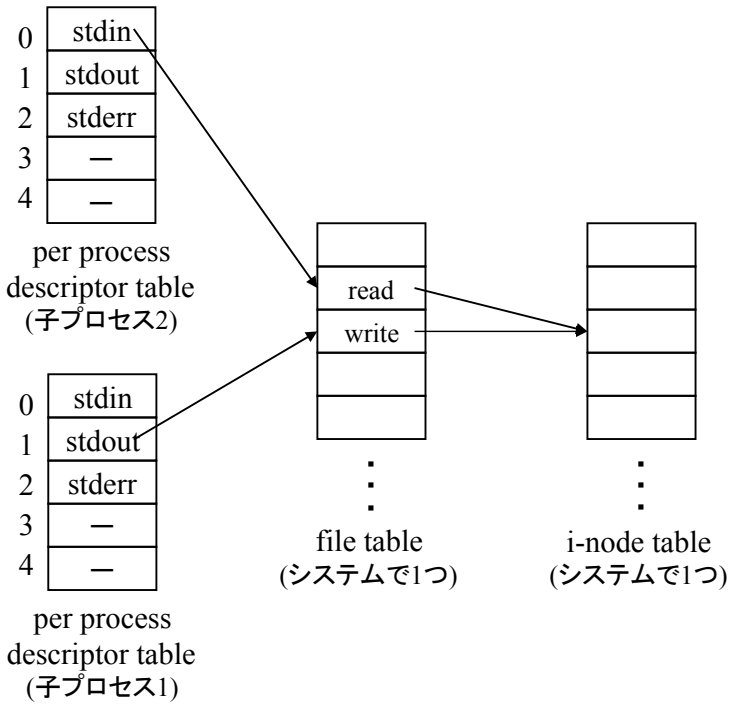

図 5.34: パイプ：手順 4

課題3：シェルの作成

ステップ1 (必須)

以下の機能をもつシェル mysh を C 言語で作成しなさい．

- コマンドの実行ができること．
- cd コマンドが実行できること．
- “exit” と入力することによって mysh が終了すること．
- エラーの検出もきちんと行うこと．

コマンドファイルの実行に関しては，execve() システムコールの代わりにライブラリ関数 execvp() を使用してよい．cd コマンドは chdir() システムコールを利用すればよい．構文は以下のとおりである．cd コマンドを実現する上での注意点は，子プロセスを生成せずに直接 chdir() システムコールを実行しなければならないことである．理由は各自で考えて欲しい．

```
01: #include <unistd.h>
02:
03: int
04: chdir(const char *path);
```

ステップ2 (必須)

ステップ 1 に加えて以下の機能を実現しなさい．

- 標準入力と標準出力のリダイレクトができること．
- 1 段のパイプができること (例：“ls | wc -l”)．

ステップ3 (必須)

ステップ 2 に加えて以下の機能を実現しなさい．このステップまでを必須とする．

- 多段のパイプができること．
- ctrl-C の入力により，mysh 自体は終了せず実行中のコマンドを終了できること．

ステップ4 (オプション)

このステップ以降はオプションである.

- コマンドのバックグラウンドでの実行ができること.
- ライブラリ関数 execvp() の代わりに execve() システムコールを使用すること. すなわち, mysh の中で環境変数 PATH に設定されているサーチパスをたどり, コマンドファイルを見つけて実行する.

ステップ5 (オプション)

普段使用しているシェル (たとえば csh) の機能をどんどん取り込むこと.