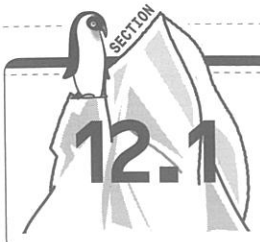
第12章

プロセスにかかわるAPI

この章では3大基本概念最後の1つ、プロセスに関係するAPIを紹介します。



12.1 基本的なプロセスAPI

我々が最も直接的にプロセスと接するのは、シェルからプログラムを起動するときでしょう。本章ではまず、シェルがプログラムを実行するときに使っている基本APIを紹介していきます。

さて、UNIXでは、一見遠回りに見えるほど単機能のシステムコールをいくつか用意して、それを組み合わせることでプログラムの起動という作業を実現しています。そのシステムコールとは、次の3つです。

- fork(2)
- exec(2)
- wait(2)

順番に見ていきましょう。

fork(2)

fork()は、自プロセスを複製して新しいプロセスを作るシステムコールです。

```
#include <unistd.h>

pid_t fork(void);
```

fork()を呼び出すと、カーネルはそのプロセスを複製し、2つのプロセスに分裂させます（図12.1）。この時点で、「複製前のプロセス」と「複製後のプロセス」はどちらもfork()を呼び出した状態になっています。そして、2つのプロセスの両方にfork()の呼び出しが戻るので、両方のプロセスでfork()以後のコードが実

行されます。このとき、元から存在しているプロセスのほうを**親プロセス**（parent process）、複製して作られたプロセスのほうを**子プロセス**（child process）と言います。

子プロセスでのfork()の戻り値は0です。親プロセスでのfork()の戻り値は子プロセスのプロセスID（正の整数値）です。fork()が失敗した場合には子プロセスは作成されず、親でのみ-1が戻ります。

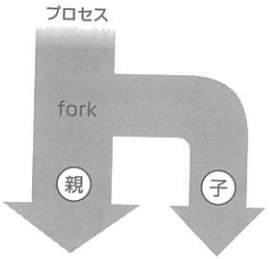


図12.1 fork()のイメージ

exec

execは、自プロセスを新しいプログラムで上書きするシステムコールです。execを実行すると、その時点で現在実行しているプログラムが消滅し、自プロセス上に新しいプログラムをロードして実行します（図12.2）。

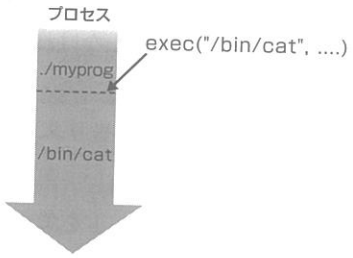


図12.2 execのイメージ

execの典型的な使用例は、fork()して即座にexecすることです。これで新しいプログラムを実行したことになります。

```
#include <unistd.h>

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

コマンドライン引数と環境変数の渡し方によって少しずつ語尾の違うexecXXがありますが、普通はこれらのAPIを全部まとめて「exec」あるいは「exec族」と呼びます。manページも「man exec」で見られます。Linuxも含めてほとんどのUNIX系OSでは、execve()だけがシステムコールで、残りはライブラリ関数です。

各APIの引数の違いは次のとおりです。

語尾に「l（list）」が付くバージョンでは、コマンドライン引数を引数リストとして渡します。引数リストの最後には目印としてNULLを置かなければいけません。

「v（vector）」の付くバージョンでは、コマンドライン引数を文字列の配列で渡します。配列argvの最後の要素はNULLにしなければいけません。

いずれの場合も、起動するプログラムに渡すコマンドライン引数リストは、そのままmain()のargvになります。つまり、引数リストの第1要素はプログラムの名前を表しているので、ほとんどの場合において第1要素と第2要素で同じコマンド名を繰り返す必要があります。次のコードを参考にしてください。

```
/* execl()の使用例 */
execl("/bin/cat", "cat", "hello.c", NULL);
```



```
/* execv()の使用例 */
char *argv[3] = { "cat", "hello.c", NULL };
execv("/bin/cat", argv);
```

また、「e（environment）」が付いていると、最後の引数として環境変数envpが追加されます。「e」が付いていないAPIでは、現プロセスの環境変数がそのまま使われます（環境変数については第14章で詳しく説明します）。

最後に、「p（path?）」が付いていると、第1引数programを環境変数PATHから自動的に探します。つまりシェルと同じように動作します。「p」が付いていないexecの場合は、常に第1引数pathを絶対パスまたは相対パスで指定しなければいけません。

execは成功すると呼び出しが戻らないので、呼び出しが戻った場合は常に失敗です。そのときは-1を返してerrnoをセットします。

wait(2)

fork()したプロセスの終了を待つには、wait()かwaitpid()を使います。いずれもシステムコールです。

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

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

wait()は、子プロセスのうちどれか1つが終了するのを待ちます。waitpid()は、第1引数pidと同じプロセスIDを持つプロセスが終了するのを待ちます。

statusにNULL以外を指定した場合は、そのアドレスに子プロセスの終了ステータスが格納されます。終了ステータスとは、終了の仕方を表すフラグとexit()の引数に渡した値（終了コード）を合成した値で、次のマクロを使うと個々

の値を取り出せます。

表 12.1　終了の仕方を調べるマクロ

マクロ	意味
WIFEXITED(status)	exit で終了していたら非 0、それ以外なら 0
WEXITSTATUS(status)	exit で終了していたとき、終了コードを返す
WIFSIGNALED(status)	シグナルで終了していたら非 0、それ以外なら 0
WTERMSIG(status)	シグナルで終了していたとき、シグナル番号を返す

プログラムの実行

ここでまず、以上の 3 つのシステムコールを使って「プログラムを実行して結果を待つ」操作を実行してみましょう。手順を日本語で書くと次のとおりです。

1. fork() する
2. 子プロセスで新しいプログラムを exec する
3. 親プロセスは子プロセスを wait する

リスト 12.1 に具体的な例を示します。このプログラム spawn は、コマンドライン引数を 2 つ受け取り、第 1 引数をプログラムのパス、第 2 引数をその引数と解釈して実行します。また、プログラムが終了したらその終了方法を出力します。

リスト 12.1　spawn.c

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

int
main(int argc, char *argv[])
{
    pid_t pid;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <command> <arg>\n", argv[0]);
        exit(1);
    }
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "fork(2) failed\n");
        exit(1);
    }
    if (pid == 0) { /* 子プロセス */
        execl(argv[1], argv[1], argv[2], NULL);
        /* execl() が呼び出しから戻ったら失敗 */
        perror(argv[1]);
        exit(99);
    }
    else {          /* 親プロセス */
        int status;

        waitpid(pid, &status, 0);
        printf("child (PID=%d) finished; ", pid);
        if (WIFEXITED(status))
            printf("exit, status=%d\n", WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("signal, sig=%d\n", WTERMSIG(status));
        else
            printf("abnormal exit\n");
        exit(0);
    }
}
```

今回の難所はなんと言っても fork() でしょう。fork() は子プロセスと親プロセスの両方で呼び出しが戻るということを意識して読んでください。

また、exec には種類がいろいろありましたが、今回は引数を直接書くのが楽な execl() を選びました。exec 類の説明のときにも書きましたが、プログラム名（子プロセスにとっての argv[0]）を渡すのを忘れないように気を付けてください。

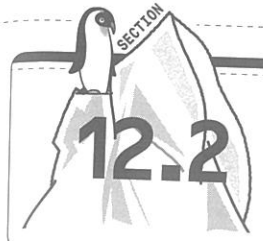
では、ビルドして実行してみましょう。

```
$ gcc -g -Wall -o spawn spawn.c
$ ./spawn /bin/echo OK
OK
child (PID=20092) finished; exit, status=0
```

うまくいきました。
ちなみに、存在しないプログラムを指定すると、次のようにexecが失敗します。

```
$ ./spawn /usr/bin/notexist xxx
/usr/bin/notexist: No such file or directory
child (PID=20339) finished; exit, status=99
```

なお、ライブラリ関数のsystem(3)を使うと、もっと簡単にプログラムを実行できます。ただしこちらはシェルを一度起動してプログラムを実行させるので、負荷が大きくなるほか、セキュリティ上の問題が格段に発生しやすくなります。十分に注意してください。



12.2 プロセスの一生

先ほどはfork()でプロセスを作る方法を覚えたので、プロセスを終了する方法についても話しておきましょう。

_exit(2)

プロセスを自発的に終了するシステムコールが、_exit()です。

```
#include <unistd.h>

void _exit(int status);
```

_exit()は、statusを終了ステータスとしてプロセスを終了します。_exit()は絶対に失敗しないので、呼び出したら決して戻りません。

exit(3)

それからもちろん、これまで何度も使ってきたexit()でもプロセスを終了できます。こちらはライブラリ関数です。

```
#include <stdlib.h>

void exit(int status);
```

exit()は、statusを終了ステータスとしてプロセスを終了します。exit()は絶対に失敗せず、呼び出しからも戻りません。

exit()と_exit()の違いは次の2点です。

- exit()はstdioのバッファを全部フラッシュする
- exit()はatexit()で登録した処理を実行する

つまり、libcに関連した各種の後始末をするかしないかの違いです。このような違いは結局のところexit()と_exit()の素性の差から来ています。つまり、exit()はlibcの関数だが、_exit()はシステムコールだという違いです。libcの関数であるexit()がlibcの後始末をするのは当然ですが、システムコールである_exit()はlibcのことなど知りませんから、その後始末をするわけがありません。

なお、main()からreturnしてプログラムを終える場合もあると思いますが、そのときも実はmain()を呼んだコードがこっそりexit()を呼んでいます。

終了ステータス

これまでexit()するときはexit(0)やexit(1)のように即値を使ってきましたが、「0は成功」「1はエラー」というのはLinux（UNIX）に特有の決まりごとです。成功・失敗のどちらかを表現するだけでよければ、EXIT_SUCCESSとEXIT_FAILUREというマクロを使ったほうがより広い環境で通用します。

一般にはEXIT_xxxxと即値のどちらを使うべきでしょうか。ポリシーにもよりますが、プログラムが成功・失敗のいずれかしか返さないのであれば、EXIT_xxxxを使うのがよいでしょう。しかし、成功・失敗以上に細かくステータスを分けたい場合は直接数値を書くべきでしょう。

プロセスの一生

以上で話したことをまとめると、プロセスの一生を描くことができます。まずfork()で開始し、場合により途中でexecして_exit()で終了、その終了ステータス

は親プロセスのwait()で出てくる…という流れです（図12.3）。

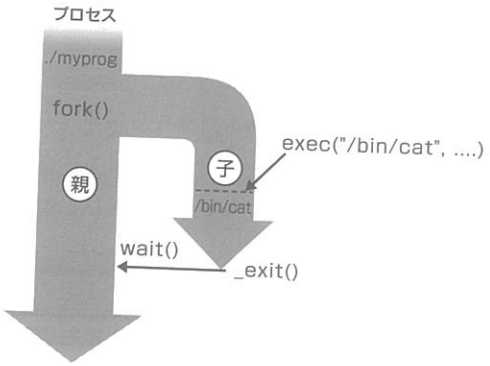


図12.3　fork()+exec+wait()

ゾンビ

ところで、先ほどの図ではうまくfork()とwait()が対になっているので_exit()の値が回収されましたが、もし親プロセスがいつまでたってもwait()を呼ばないまま動作し続けたらどうなるのでしょうか。

カーネルにしてみれば、親プロセスがwait()を呼ぶかどうかは予測できません。wait()を呼ばないと見せかけて実はしばらくたってからwait()するかもしれませんし、そのときはちゃんと子プロセスのステータスコードを返してやる必要があります。ですから、カーネルは親プロセスが終了するか、またはwait()を呼ぶまでは子プロセスのステータスコードを保存しておかなければなりません。この状態になった子プロセスのことを**ゾンビプロセス**（zombie process）と呼びます。ゾンビになると、psコマンドの出力に「zombie」とか「defunct」と表示されます。

すぐに終了するプログラムならば、その子プロセスが1つ2つゾンビになったところで問題はありません。しかし、後述するデーモンプロセスのように長時間動き続けるプロセスの子プロセスがゾンビになってしまうと、いつまでたってもゾンビが成仏できず、システムがゾンビだらけになってしまいます。プロセス数が増えればそれだけカーネルの負担が増えますから、システムにとっていいこと

はありません。デーモンプロセスなどでは子プロセスがゾンビになってしまわないよう注意する必要があります。

子プロセスがゾンビになるのを回避するには、次の3つの方法があります。

1. fork()したらwait()する
2. ダブルfork
3. sigaction()を使う

1.が一番正統な方法です。fork()したらwait()するのが親の務めです。

2.のダブルforkというのは、途中に余分なfork()を挟む方法です。これは口で言ってもわかりにくいので図12.4を見てください。

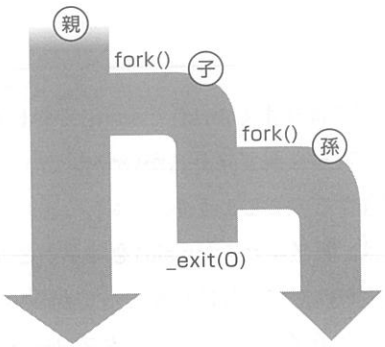
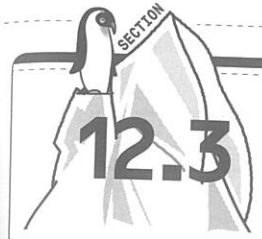


図12.4　ダブルfork

この状態になると、孫プロセスにとっての親(「子」プロセス)はいなくなります。wait()する権利があるのは直接の親だけなので、子プロセスがいなくなった時点で孫プロセスをwait()する権利を持つプロセスは存在しなくなります。したがって、カーネルも孫プロセスをゾンビにせず、終了したらすぐに始末してくれます。

最後の3.は、次章で述べるsigaction()というAPIを使って、「自分はwait()をしない」とカーネルに知らせることです。

ゾンビ問題は第17章で再度検討します。



12.3 パイプ

Linuxのシェルには、単にプログラムを起動するだけでなく、複数のプログラム間（プロセス間）をパイプでつなぐことができるという重要な特徴がありました。今度はプロセス間にパイプをつなぐ方法を紹介していきましょう。

パイプ

プログラマの目から見ると、パイプとは、プロセスからプロセスにつながったストリームのことです。ファイルにつながったストリームと同じように、パイプもファイルディスクリプタを使って表現されます（図12.5）。

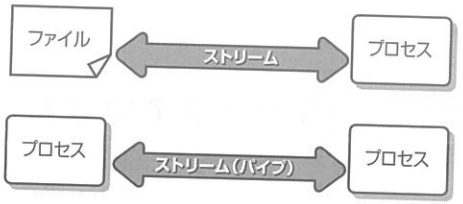


図12.5　パイプのイメージ

なお、普通のファイルにつながったストリームは読み書き両用にできましたが、パイプは基本的に一方向です。つまり、1つのファイルディスクリプタに対しては読むか書くかどちらかしかできません。

pipe(2)

```
#include <unistd.h>

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

pipe()は、両端とも自プロセスにつながったストリームを作成し、その両端のファイルディスクリプタ2つを第1引数fdsに書き込んで返します。

図12.6を見てください。pipe()の呼び出しが成功すると、図12.6のようにストリームがつながります。fds[0]は読み込み専用、fds[1]は書き込み専用です。

図12.6　pipe(2)

親子プロセス間をパイプでつなぐ

pipe()は単独ではほとんど意味がありません。fork()と組み合わせることで初めて意味が出てきます。

鍵になるのは、fork()はプロセスを複製するときにストリームもすべて複製するということです。pipe()でパイプを作ったあとfork()すると、図12.7のようになります。

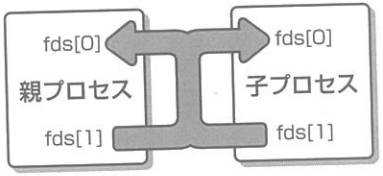


図12.7　pipe()のあとfork()

ここで、親が読み込み側をclose()し、子が書き込み側をclose()すると、図12.8のようになります。これで見事に親から子へのパイプができました。

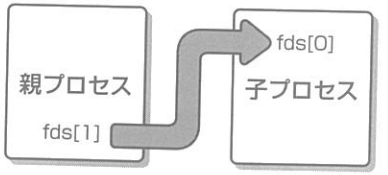


図12.8　両端でclose()

この仕組みは何かだまされたような気がするのも確かです。あまり直感的とは言えません。

dup(2), dup2(2)

ここまでで、pipe()とfork()でプロセスの間にパイプを作ることはできるようになりましたが、狙ったファイルディスクリプタにパイプをつなぐことがまだできません。例えばシェルを作る場合は、正確に標準入力と標準出力にパイプをつなぐ必要があるでしょう。そのためには、第5章でごく簡単に説明したシステムコール、dup()が必要です。

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup()とdup2()は、ファイルディスクリプタoldfdを複製します。「dup」はduplicate（複製する）のdupです。

dup()は、使われていない最小のファイルディスクリプタへoldfdを複製してそれを返します。dup2()は、ファイルディスクリプタoldfdをファイルディスクリ

プタnewfdに複製してそれを返します。また、エラーが起きた場合は-1を返します。

「複製する」というのは、図12.9のように、1つのストリームをカーネル内で2つに分岐するという意味です。

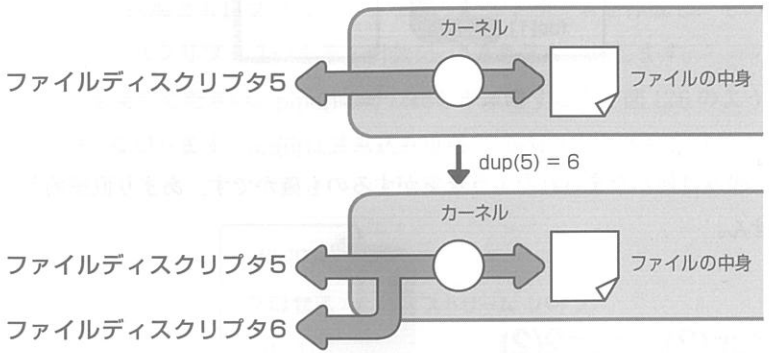


図12.9　dup()

それはどういうことかと言うと、図12.9のファイルディスクリプタ6に対して操作をすると、ファイルディスクリプタ5に対して操作をするのと同じ効果になるということです。例えばファイルディスクリプタ6に対してread()すると、その分はファイルディスクリプタ5でもすでに読み込まれたことになります。lseek()でファイルオフセットを操作した場合も、両方に効果が波及します。また、close()する場合は、両方をclose()して初めてストリーム全体が後始末されます。

さて、狙った番号のファイルディスクリプタにパイプをつなぐ方法を説明しましょう。dup()とdup2()のどちらでもできますが、dup2()のほうが確実で簡単なので、dup2()を使います。例えば3番につながっているパイプを0番に移したければ、次のような順番でAPIを使用します。

1. close(0);
2. dup2(3, 0);
3. close(3);

簡単ですね。

1つだけ注意するなら、最後に3番をclose()することを忘れないでください。3番にパイプをつないだままにしておくと、パイプがいつまでたっても始末されなくなってしまいます。

popen(3)

pipe()はfork()やdup2()とうまく組み合わせて使わなければいけませんでした。stdioにはもう少し扱いやすいパイプ接続用APIがあります。それが、popen(3)です。

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

popen()は、プログラムcommandを起動してそれにパイプをつなぎ、そのパイプを表すstdioストリームを返します。失敗したらNULLを返します。

modeは文字列"r"か"w"です。"r"ならパイプは読み込み用に開かれ、"w"なら書き込み用に開かれます。残念ながら、読み書き両用にはできません。読み書きを両方したい場合は、やはり自分でpipe()とfork()を使い、パイプをつなぐ必要があります。

なお、popen()ではプログラムがシェル経由で実行されるので、第1引数commandのコマンドはPATHから探されますし、リダイレクトやパイプも使えます。

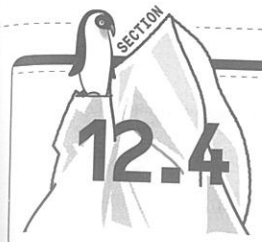
pclose(3)

popen()と対になるAPIがpclose()です。

```
#include <stdio.h>

int pclose(FILE *stream);
```

pclose()は、popen()でfork()した子プロセスをwait()し、そのあとにストリームを閉じます。popen()で開いたFILE*は必ずpclose()で閉じなければいけません。



12.4 プロセスの関係

Linuxではプロセス同士がいろいろな形でつながりを持ちます。本節では、そのようなプロセス同士の関係に着目してみましょう。

親子関係

最初は親子関係です。Linux上では、どんなプロセスもfork()かそれに類するAPIで生成されます。ということは、Linux上のプロセスを親子関係でつないでいくと、1つのツリー構造にまとめることができるはずです。pstreeコマンドを使うと、この親子関係のツリーを表示することができます。

```
$ pstree
systemd─┬─ModemManager─┬─{gdbus}
        │              └─{gmain}
        ├─NetworkManager─┬─dhclient
        │                ├─dnsmasq
        │                ├─{gdbus}
        │                └─{gmain}
        ├─accounts-daemon─┬─{gdbus}
        │                 └─{gmain}
        ├─acpid
        ├─agetty
        ├─avahi-daemon───avahi-daemon
        ├─colord─┬─{gdbus}
        │        └─{gmain}
        ├─cron
        ├─cups-browsed─┬─{gdbus}
        │              └─{gmain}
        ├─cupsd
        ├─dbus-daemon
                :
```

注目したいのは、左端に表示されているsystemdです。systemdはブート時

にカーネルが直接起動するプログラムであり、すべてのプロセスの始まりです。UNIXで最初に起動するプログラムと言えば数十年の永きに渡ってinitだったのですが、ここ数年でinitの代わりにsystemdが使われるようになりました。

getpid(2), getppid(2)

プログラムから自分のプロセスIDや、親プロセスのプロセスIDを知ることもできます。

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

pid_t getpid(void);
pid_t getppid(void);
```

getpid()は自分のプロセスIDを返します。getppid()は親プロセスのプロセスIDを返します。

他のプロセスの情報

自分と、自分と直接の親子関係にあるプロセスについてはここまで説明してきたAPIでわかるとして、他のプロセスの情報はどうやったらわかるのでしょうか。

Linuxの場合は、プロセスファイルシステム/procが最も「正しい」APIです。正しいというのは、カーネルからそれ以上直接的に情報を得る手段はないという意味です。psコマンドやpstreeコマンドもプロセスファイルシステムを使って情報を集めています。

プロセスグループとセッション

Linuxには、親子関係以外にもプロセスを体系付ける別の概念もあります。それが**プロセスグループ**（process group）と**セッション**（session）です。すべてのプロセスはそれぞれただ1つのプロセスグループとセッションに所属しています。

この2つの概念を理解する上では、それが「何か」を説明してもあまり意味がありません。「何のために」作られたかを知ることが大切です。

一言で言えば、プロセスグループはシェルのためにあります。例えばシェルを使ってコマンドをたくさん使ったパイプを起動したとしましょう。しかし、思ったより時間がかかっているので見直してみると、コマンドの一部を打ち間違えていたことがわかりました。これでは意味がないので、Ctrl + C を打って処理を中断します。

さて、この場合にどのプロセスが止まるべきでしょうか。もちろん、パイプを構成する全プロセスが止まるべきです。

そこで考案されたのがプロセスグループです。例えば「パイプでつながれたプロセス群」を1つのプロセスグループとしてまとめ、そのグループのプロセスにシグナルをまとめて送れるようにすればよいのです。

一方、セッションというのは、ユーザのログインからログアウトまでの流れを管理するための概念です。ログインシェルを起点に、ユーザが同じ端末から起動したプロセスを1つにまとめる働きをします。結果として、1つのセッションは複数のプロセスグループをまとめるような形になります（図 12.10）。

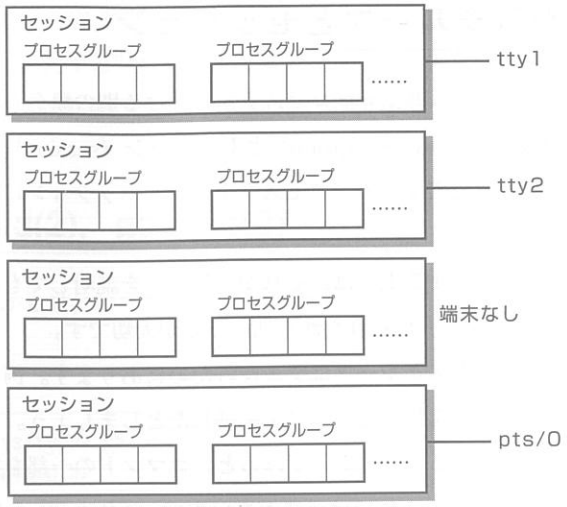


図12.10　プロセスグループとセッション

このとき、セッションと関連付けられた端末のことを、プロセスの**制御端末**（controlling terminal）と言います。

プロセスグループとセッションのリーダー

psコマンドをjオプション付きで実行するとプロセスグループやセッションを見ることができます。

```
$ ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 4548  4549  4549  4549 pts/17    7394 Ss    1001   0:00 -bash
 6992  6993  6993  6993 pts/0     7380 Ss    1001   0:00 -zsh
 6993  7380  7380  6993 pts/0     7380 S+    1001   0:00 sleep 1000
 6993  7381  7380  6993 pts/0     7380 S+    1001   0:00 tail
 7502  7394  7394  4549 pts/17    7394 R+    1001   0:00 ps j
 4549  7502  7502  4549 pts/17    7394 S     1001   0:03 zsh
```

表示されている欄の意味は、PIDがプロセスID、PGIDがプロセスグループID、SIDがセッションIDです。

PID（プロセスID）とPGID（プロセスグループID）が等しいプロセスは**プロセスグループリーダー**（process group leader）です。最初にそのプロセスグループを作ったプロセスがリーダーになります。

また、PID（プロセスID）とSID（セッションID）が等しいプロセスは**セッションリーダー**（session leader）です。こちらもプロセスグループリーダーと同じく、最初にそのセッションを作ったプロセスがセッションのリーダーになります。

プロセスグループリーダーにしてもセッションリーダーにしても、リーダーだからといって特に権限が強いわけではありません。むしろその逆に「新しいプロセスグループやセッションを作れない」という制限があります。

デーモンプロセス

さらに、今度は「ps -ef」と打ってみましょう。「-ef」オプション付きだとpsコマンドはマシンで動作している全プロセスを表示するのでしたね。私のマシンでは次のように表示されました。

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Aug07 ?        00:00:04 /lib/systemd/systemd --system --deserialize 20
                    ⋮
root       221     1  0 Aug07 ?        00:00:01 /lib/systemd/systemd-journald
root       698     1  0 Aug07 ?        00:02:13 /usr/sbin/cupsd -l
syslog     703     1  0 Aug07 ?        00:00:00 /usr/sbin/rsyslogd -n
root       704     1  0 Aug07 ?        00:00:00 /usr/sbin/cron -f
                    ⋮
```

すると、TTYの欄が「?」になっているプロセスがたくさん並んでいることに気付きます。これは制御端末のないプロセスであることを示しています。このような、制御端末を持たないプロセスのことを**デーモンプロセス**（daemon process）、あるいは単に**デーモン**（daemon）と言います。

なぜデーモンプロセスのような存在が必要になるのでしょうか。端的に言うと、

サーバを作るためです。

サーバ（server）というのは、聞いたことくらいはあると思いますが、他のプロセスに何かのサービスを提供するプロセスのことです。普通はネットワーク経由で使います。

サーバも誰かが起動しなければなりません。ですが、サーバはコンピュータが動いている間はずっと起動しておくものです。プロセスを起動した人がログアウトするとその端末が制御端末になっているプロセスは止められてしまいますから、サーバプロセスを起動した人はログアウトできないことになってしまいます。これでは不便ですね。

そこでデーモンプロセスが必要になってくるのです。どの端末ともまったく関係なく動作することができれば、サーバを起動した人は気兼ねなくログアウトできますし、うっかり Ctrl + C でサーバを止めてしまうこともありません。

第17章ではデーモンプロセスになるための方法を紹介します。

setpgid(2)

最後に、新しいプロセスグループとセッションを作るシステムコールを紹介してこの章を終わりにしましょう。まずは新しいプロセスグループを作るために利用されるAPI、setpgid()から説明します。

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

setpgid()は第1引数pidのプロセスIDを持つプロセスのプロセスグループIDを、第2引数pgidへ変更します。成功したら0を返します。失敗したら-1を返してerrnoをセットします。

第1引数pidが0のときは自プロセスが対象になります。また第2引数pgidが0のときは自プロセスのプロセスIDがプロセスグループIDとして使われます。

つまり、自分がリーダーになって新しいプロセスグループを作りたいときは、両方の引数を0にすればいいわけです。そしてそれが代表的な使いかたです。

setsid(2)

続いて、新しいセッションを作るsetsid()を紹介します。setsid()はデーモンを作るときにお世話になります。

```
#include <unistd.h>

pid_t setsid(void);
```

setsid()は新しいセッションを作成し、自分がセッションリーダーになります。同時に、そのセッションで最初のプロセスグループを作成し、そのグループリーダーになります。また、setsid()で作成した新しいセッションは制御端末を持ちません。つまり、セッションリーダーになると同時にデーモンにもなるわけです。戻り値は作成したセッションのID（通常は自プロセスのプロセスIDと同じ）です。失敗したときは-1を返してerrnoをセットします。

自プロセスがプロセスグループリーダーだとsetsid()は失敗してしまいます。setsid()を確実に成功させるためには、あらかじめ1回余分にfork()して、プロセスグループリーダーではなくなっておくのが常套手段です。

setsid()の利用例は第16章で紹介します。