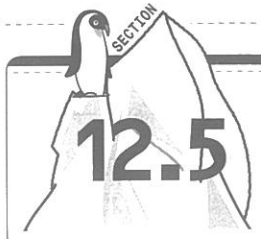


12.5 練習問題

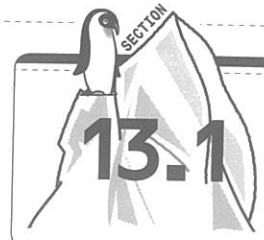
1. fork() したらプロセスが使うメモリは倍になるのでしょうか。調べなさい。
2. fork() とexec() を使ってプログラムを起動する、簡単なシェルを書きなさい。
3. 問題 2 で作ったシェルにパイプとリダイレクトを実装しなさい。（難しい）

※解答は本書のサポートページで公開しています。

第13章

シグナルにかかわるAPI

シグナルのことはこれまでも何回か触れてきました。本章では、シグナルの送信と捕捉について話します。



13.1 シグナル

シグナルは、ユーザ（端末）やカーネルがプロセスに何かを通知する目的で使われます。シグナルにはいくつもの種類があり、マクロで名前が付けられていますが、その実体はint型の整数です。

シグナルがプロセスに配送されると、あらかじめ設定されている3つの処理のいずれかが実行されます。次の3つです。

1. 無視する
2. プロセスを終了する
3. プロセスのコアダンプを作成して異常終了する

1つ目の処理方法は、シグナルを単に無視することです。例えば、子プロセスが終了したときなどに生成されるSIGCHLDはデフォルトで無視されます。

どうせ無視されるならそんなシグナルは必要ないじゃないかと思うかもしれませんが、そうではありません。実は、シグナルを受けたときの挙動はプロセスが変更することもできるのです。プロセス実行中に挙動を変えてやれば、デフォルトでは無視されるシグナルも役に立つようになります。

2つ目の処理方法は、プロセスを終了させることです。例えば、コマンドラインで Ctrl + C を打ったときに生成されるSIGINTがそうです。

3つ目の処理方法は、コアダンプを生成してプロセスを異常終了させることです。例えば、第11章で触れたSIGSEGVがそうです。対応する物理アドレスのないメモリにアクセスするとこのシグナルが生成され、プロセスは異常終了します。これはsegmentation faultによる終了とも言います。

コアダンプ（core dump）というのは、プロセスのメモリのスナップショットのことです。通常は「core」というファイルに保存されます。

ちなみに、core dumpのcoreはメモリのことを指しています。昔のコンピュータは磁気コアというもので作った「コアメモリ」を使っていたので、メモリのことをコアと呼んでいたのです。しかしそのあとコアメモリは廃れてしまい、現在はその名前だけが残っています。

特に頻繁に使われるシグナルを表13.1にまとめておきます。

表13.1 よく使われるシグナル

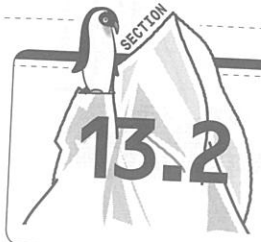
シグナル名	捕捉可能	デフォルトの挙動	生成原因と用途
SIGINT	○	終了	割り込み。主に Ctrl + C で生成され、プログラムを中止したいときに使う
SIGHUP	○	終了	ユーザがログアウトしたときなどに生成する。デーモンプロセスでは設定ファイルの読み直しに使うことが多い
SIGPIPE	○	終了	切れたパイプに書き込むと生成される
SIGTERM	○	終了	プロセスを終了させるときに使う。killでシグナルを指定しなかったときのデフォルトの値
SIGKILL	×	終了	確実にプロセスを終了させるために使う
SIGCHLD	○	無視	子プロセスが停止または終了したときに生成される
SIGSEGV	○	コアダンプ	アクセスが禁止されているメモリ領域にアクセスした。初期化していないポインタを参照したり、バッファオーバーフローが起きた場合などに生成される。プログラムのバグによって起こる
SIGBUS	○	コアダンプ	アラインメント違反。ポインタ操作を間違えたときなどに生成される。プログラムのバグによって起こる
SIGFPE	○	コアダンプ	算術演算でのエラー。ゼロ除算や浮動小数点数のオーバーフローなどで生成される

シグナルの捕捉

表13.1の「捕捉可能」という欄に「○」が付いているシグナルは、そのシグナルが配送されたときの挙動を変更することができます。

例えば、SIGCHLDを受けたときの挙動はデフォルトでは「無視」ですが、「捕捉可能」の欄に○が付いています。ですから、後述するsigaction()というAPIを使ってシグナルを受けたときの挙動を変えることができます。

一方、SIGKILLを見てください。SIGKILLは「捕捉可能」の欄が×で、しかもデフォルトの挙動が終了ですね。ですから、SIGKILLを受信したプロセスは常にカーネルによって終了させられるのだとわかります。このシグナルは、どうしようもなくなったときにシステムの管理者がプロセスを確実に終了させるための最後の手段として用意されています。



13.2 シグナルを捕捉する

先ほど、シグナルを受けたときの挙動はプロセスが変更できるという話をしました。シグナルの処理をカーネル任せにせず、自分でシグナルを処理することを「シグナルを**捕捉する**（catch）」あるいは「シグナルを**トラップする**（trap）」と言います。UNIXの最初期から存在するシグナル捕捉インターフェイスが、signal(2)です。「signal」という字面からすると一見シグナルを送るAPIのように見えるのですが、実際には捕捉するAPIなので注意してください。

signal(2)

```
#include <signal.h>

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

このままでは何が何やらわかりませんね。typedefを使ってもう少しわかりやすくしてみましょう。

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

これならなんとかわかるでしょうか。1行目のtypedefは、「sighandler_t型とは、int型を引数にとり、戻り値がvoidの関数、へのポインタである」という意味です。つまり、signal()は第2引数に関数ポインタをとり、関数ポインタを返す関数なのです。なお、関数ポインタについてはすぐあとで復習をするので、慣れていない方は先にそちらを読んでおいてください。

signal()は、シグナル番号sigのシグナルを受けたときの挙動を変更します。具体的には、第2引数funcの関数を呼ぶように変更します。

なお、このとき第2引数funcに渡す関数を、シグナルを処理する関数という意味で**シグナルハンドラ**（signal handler）と言います。先ほど使った「sighandler_t」の「sighandler」もsignal handlerのことです。

また、第2引数funcには表13.2に示す特別な値も使えます。

表13.2　第2引数で用いられる特別な値

定数	意味
SIG_DFL	カーネルのデフォルトの挙動に戻す
SIG_IGN	カーネルレベルでシグナルを無視するように指示する

最後に、signal()は直前までのシグナルハンドラを返します。

関数ポインタ

関数ポインタが出てきたので、少しだけおさらいをしましょう。

まずは文法から確認します。「int *n」とあれば、nはint型の値を指すポインタです。「char *str」とあれば、strはchar型の値を指すポインタです。void (*f)(int)とあれば、fは「intを引数に取り戻り値のない関数」を指すポインタです。

関数ポインタ第1の難関は「関数を指す」という表現です。この「関数」とは要するに何なのでしょうか。実はこれは、「関数をコンパイル・アセンブルしてできた機械語列」のことです。その機械語列はビルドが済むとオブジェクトファイルに格納されており、execの過程で各プロセスのメモリ空間内でテキスト領域に配置されます。その機械語列の先頭アドレスこそが関数ポインタの値です。ポインタというのは結局メモリアドレスのことであり、関数ポインタもその例外ではないということを認識してください。

関数ポインタ第2の難関は文法です。これにはあまりよい方法はないので、「こう書くとこう動く」と覚えてしまうしかありません。

まず、関数ポインタ変数のほうは、「type (*f)(type)」という形があったらfが関数ポインタです。「(*f)」に着目して考えるとよいでしょう。

それから、普通に定義した関数でも、その名前だけを書くとその関数の関数ポインタが取れるという仕組みがあります。例えば次のような場合です。

```
 1  int
 2  plus1(int n)
 3  {
 4      return n + 1;
 5  }
 6
 7  int
 8  main(int argc, char *argv[])
 9  {
10      int (*f)(int);   /* 関数を指すポインタ変数fを定義 */
11      int result;
12
13      f = plus1;              /* ここが問題！ */
14      result = f(5);          /* fに代入した関数（plus1）を実行 */
15      printf("%d\n", result); /*「6」と出力される */
16      exit(0);
17  }
```

13行目ではplus1()のポインタを変数fに代入しています。plus1()を呼んでいるのではありません。

これは、char*とchar[]の関係に少し似ています。

「char *buf」はbufというポインタ変数だけを定義しますが、char buf[64]はcharが64個分のメモリを確保するのと同時にbufというポインタ変数を定義しています。ですから、どちらでもbufと書くだけで配列先頭へのポインタが得られます。

「void (*print)(char *msg)」はprintというポインタ変数しか定義しませんが、「void print(char *msg){....}」は機械語列を置くメモリを確保するのと同時にprintというポインタ変数を定義しています。ですから、どちらでもprintと書くだけで関数の機械語列先頭へのポインタが得られます。

signal(2)の実装上の問題

シグナルの話に戻りましょう。

シグナルというのはこちらの状態を考えずにどんなときでも飛んできますし、通常の実行に割り込んで実行されるため、本質的に厄介です。それに加えて、signal(2)の実装にはさまざまな問題があります。いくつか重要な論点を解説します。

- **ハンドラの再設定**

  OSによっては、signal()は一度シグナルハンドラを起動するとハンドラの設定を元に戻してしまう場合があります。そのような実装では、次にsignal()を呼ぶまでの間に飛んできたシグナルを捕捉できなくなってしまいます。

- **システムコールの再起動**

  シグナルはこちらが何をしていようと飛んでくるので、read(2)やwrite(2)を実行している途中で割り込んでくることもあります。signal(2)では、そのような場合の挙動もOSによってまったく違います。あるOSはread(2)などがエラーを返して終了します。あるOSは自動的にシステムコールをやり直して、プログラマからは見えないようにします。前者のような仕様になっている場合、read()やwrite()を使うたびにシグナルを意識してコーディングしなければならず、非常に不便です。

- **重複して呼び出せない関数**

  シグナルはこちらが何をしていようと飛んできます。ということは、ある関数の実行中にシグナルハンドラが実行されて、その関数を再度呼んでしまうこともあるでしょう。このとき、その関数内でグローバル変数を使っていたりすると非常に困ったことになります。標準Cライブラリの中にすらそのような関数は大量に存在しているのです。

- **シグナルのブロック**

  何度も言いますが、シグナルはいつでも飛んできます。つまり、シグナル

ハンドラを実行している間にも飛んできます。そういう場合にシグナルハンドラが複数同時に実行されてしまうのは一般にはあまりありがたくない挙動なので、現在のシステムではシグナルハンドラ実行中には同種のシグナルの配送を保留できるようになっています。このことを「シグナルをブロックする」と言います。このときの「ブロック」はバレーボールのブロックと同じです。signal(2)ではブロックに関する挙動を設定することもできません。

sigaction(2)

以上のように、signal(2)は単純ではあるのですがいろいろな点で問題があり、移植性にも欠けるので、新しいシステムコールが用意されました。それがsigaction()です。新しいプログラムでは黙ってsigaction()一本でいくのがよいでしょう。

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oldact);

struct sigaction {
    /* sa_handler, sa_sigaction は片方のみ使う */
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t*, void*);
    sigset_t sa_mask;
    int sa_flags;
};
```

sigaction()は、シグナルsigのシグナルハンドラを登録します。第2引数のactには、シグナルハンドラを指定します。具体的には、定数SIG_IGN、SIG_DFL、もしくは任意の関数ポインタのいずれかです。第3引数のoldactには、sigaction()呼び出し時に設定されていた元のシグナルハンドラが返ります。不要ならNULLを指定してください。

以降では、先ほど述べたポイントについてsigaction()の実装を説明しつつ、struct sigactionの役割に触れます。

- **ハンドラの再設定**

  sigaction()は、OSにかかわらずシグナルハンドラの設定を保持し続けることが保証されています。

- **システムコールの再起動**

  sigaction()は、デフォルトではシステムコールを再起動しません。sa_flagsメンバにフラグSA_RESTARTを追加すると、再起動する設定になります。すでに書いたとおり、一般には再起動されるほうが便利なので、SA_RESTARTは常に追加しておくのが無難です。

- **シグナルのブロック**

  struct sigactionのメンバsa_maskで、ブロックするシグナルを指定できます。さらに、シグナルハンドラの起動中は処理中のシグナルを自動的にブロックしてくれるので、ほとんどの場合はsa_maskは空にしておけば十分です。sa_maskを空にするには、後述するsigemptyset()を使います。

なお、struct sigactionのsa_sigactionもsa_handlerと同じくシグナルハンドラを指定するメンバですが、こちらを使うと、シグナルを受信したときにシグナル番号以外の詳細な情報を得ることができます。詳しくは「man sigaction」を参照してください。

sigactionの使用例

以上のポイントを踏まえて、一般的なsigaction()の使用例を次に示します。

```
#include <signal.h>

typedef void (*sighandler_t)(int);
```

```
sighandler_t
trap_signal(int sig, sighandler_t handler)
{
    struct sigaction act, old;

    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_RESTART;
    if (sigaction(sig, &act, &old) < 0)
        return NULL;

    return old.sa_handler;
}
```

まず、シグナルハンドラをsa_handlerにセットします。sa_handlerとsa_sigactionは片方しか使えないので、sa_sigactionは無視します。

次に、sa_maskは空にしておけばよいので、sigemptyset()で空に初期化します。

そして最後に、システムコールは自動的に再起動されるほうが便利なので、sa_flagsにはSA_RESTARTをセットします。

sigset_t操作API

sigaction()でsigset_tを操作しなければならないので、一応sigset_t操作用のAPIを簡単に紹介しておきます。必要になったら見てください。

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
int sigismember(const sigset_t *set, int sig);
```

sigemptyset()は、setを空に初期化します。

sigfillset()は、setをすべてのシグナルを含む状態にします。

sigaddset()は、シグナルsigをsetに追加します。

sigdelset()は、シグナルsigをsetから削除します。

sigismember()は、シグナルsigがsetに含まれるとき真を返します。

シグナルのブロック

シグナルのブロックの設定は、sigaction()のsa_maskメンバでできました。他に必要なのは、ブロックしていたシグナルを配送してもらうためのAPIです。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

sigprocmask()は、自プロセスのシグナルマスクをセットします。セット方法はフラグhowで決まります。howに指定できる値は表13.3のとおりです。

表13.3　sigprocmask()の第1引数howの値

値	効果
SIG_BLOCK	setに含まれるシグナルをシグナルマスクに追加する
SIG_UNBLOCK	setに含まれるシグナルをシグナルマスクから削除する
SIG_SETMASK	シグナルマスクをsetに置き換える

sigpending()は、保留されているシグナルをsetに書き込みます。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。

sigsuspend()は、シグナルマスクmaskをセットすると同時にプロセスをシグナル待ちにします。ブロックしていたシグナルを解除して、保留されていたシグナルを処理するときに使います。

sigsuspend()は、常に-1を返します。いつでもシグナルに割り込まれて終了する（EINTR）からです。

13.3 シグナルの送信

シグナルはカーネルやユーザが送信することが圧倒的に多く、通常のプログラムでシグナルを送る必要はほとんどありません。しかし、APIは一応簡単に紹介しておきます。

kill(2)

シグナルを送信するシステムコールは、kill()です。

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

int kill(pid_t pid, int sig);
```

kill()は、プロセスIDがpidのプロセスにシグナルsigを送信します。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。

また、pidが負数のときは、IDが-pidのプロセスグループ全体にシグナルを送ります。なお、プロセスグループにシグナルを送るには、killpg()という専用のシステムコールも使えます。

ちなみに、kill()と言えば「なんでkillなんて名前なんだ、sendsigにしろ」という話に持っていくのがお約束ですが、ことUNIXに関してはそういうことを言っているときりがありません。あきらめて先に進みましょう。

13.4 Ctrl + C

ここまででシグナル自体の話は終わりです。本章の最後に、第3章で予告した Ctrl + C の話をしましょう。 Ctrl + C がシグナルに変換されて対象プロセスに届くまでの仕組みを、図13.1にまとめました。この図を見ながら説明していきます。

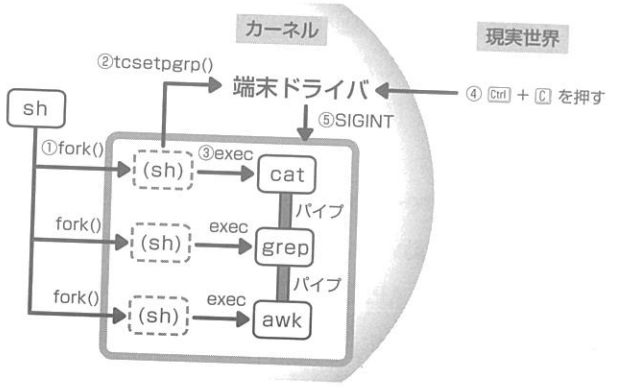


図13.1 Ctrl + C がプロセスを停止させるまで

まず、ユーザがキーボードで Ctrl + C を押すと、これをカーネルの端末ドライバが検出します。端末ドライバはモードによって動きが違いますが、普通にシェルを使っているときは**cookedモード**（cooked mode）になっているので、特殊な働きをするキーが存在します。例えば文字を消すバックスペースや、割り込みをかける Ctrl + C などです。どのキーがどの働きをするのかは、sttyコマンドを-aオプション付きで実行してみるとわかります。

```
$ stty -a
speed 38400 baud; rows 47; columns 74; line = 80;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

「intr」がinterrupt、つまり割り込みのことです。「^C」はもちろん Ctrl + C を意味しています。

さて、intrと設定されている Ctrl + C が押されたので、端末ドライバは Ctrl + C を割り込みだと認識しました。あとはSIGINTを端末で動作中のプロセスに送信すればよいわけですが、「端末で動作中のプロセス」はどうやったらわかるでしょうか。

実は、シェルがパイプでつながれたプロセス群を起動するたびに「この端末ではいま、このプロセスが動いています」と端末に教えているのです。それを教えるには、tcsetpgrp()というAPIを使います。ちなみに、tcsetpgrp()はioctl()を使って実装されています。

さらに正確に言えば、端末に教えるのはプロセスではなく、プロセスグループです。パイプでつながれたプロセス群は1つのプロセスグループになることを思い出しましょう。割り込みではパイプでつながれたプロセス全体を止めるのが望ましい挙動でしょうから、プロセスグループ全体にシグナルを送信すべきです。

そして、SIGINTが発送されると、そのデフォルトの動作はプロセス終了なので、パイプ全体が終了します。もちろん、プロセスがsigaction()などでSIGINTを捕捉しているなら話が別です。無視するかもしれませんし、何か他の動作をするかもしれません。

最後にもう一度、順を追ってまとめましょう。

1. シェルがパイプを構成するプロセスをfork()する
2. シェルがパイプのプロセスグループIDをtcsetpgrp()で端末に通知する
3. forkされた各プロセスがそれぞれのコマンドをexecする
4. ユーザが Ctrl + C を押す
5. カーネル内の端末ドライバがそれをSIGINTに変換、動作中のプロセスグループに発送する
6. プロセスグループがデフォルトの動作に従って終了する

普段から気軽に使っている Ctrl + C ですが、その裏ではこんな仕組みが動いているのです。単純な機能のように見えますが、意外にも多くのモジュールが関係する、複雑な機能なのですね。