オペレーティングシステムの機能を使ってみよう
第７章 シグナル

シグナル

前の章で使用して見た kill コマンドや JOB 制御等で使用される.
プロセスに非同期的にイベントの発生を知らせる仕組み..

1) プロセスや OS がプロセスにイベントを通知
   kill コマンド（kill プロセス）が他のプロセスにシグナルを送る.
   Ctrl-C や Ctrl-Z が押された時，OS がプロセスにシグナルを送る.
2) プロセス自身の異常を通知
   0 での割り算 → Floating point exception シグナル（SIGFPE）
   ポインタの初期化忘れ → Segmentation fault シグナル（SIGSEG）
3) プロセスが予約した時刻になった
   アラームシグナル（SIGALRM）

シグナルの一覧

番号	記号名	デフォルト	説明
1	SIGHUP	終了	プロセスが終了していないときログアウトした.
2	SIGINT	終了	ターミナルで Ctrl-C が押された.
3	SIGQUIT	コアダンプ	ターミナルで Ctrl-\が押された.
4	SIGILL	コアダンプ	不正な機械語命令を実行した.
8	SIGFPE	コアダンプ	演算でエラーが発生した.
9	SIGKILL	終了	強制終了（**ハンドリングの変更ができない**）.
10	SIGBUS	コアダンプ	不正なアドレスをアクセスした場合など.
11	SIGSEG	コアダンプ	不正なアドレスをアクセスした場合など.
14	SIGALRM	終了	alarm() で指定した時間が経過した.
15	SIGTERM	終了	終了.
17	SIGSTOP	停止	一時停止（**ハンドリングの変更ができない**）.
18	SIGTSTP	停止	ターミナルで Ctrl-Z が押された.
19	SIGCONT	無視	一時停止中なら再開する.
20	SIGCHLD	無視	子プロセスの状態が変化した.

- 番号，記号名，デフォルト（デフォルトのシグナルハンドリング）
- SIGKILL と SIGSTOP はデフォルトから変更できない

シグナルハンドリング

プロセスが，受信したシグナルをどう扱うか．

1) 無視（*ignore*）そのシグナルを無視する．
2) 捕捉・キャッチ（*catch*）そのシグナルを受信し，登録しておいたシグナル処理ルーチン（シグナルハンドラ関数）を呼び出す．
3) デフォルト（*default*）シグナルの種類ごとに決められている初期のハンドリングであり，以下の四種類のどれかである．各シグナルのデフォルトが四種類のうちのどれかは一覧表から分かる．

停止 プロセスは一時停止状態になる．
無視 プロセスはそのシグナルを無視する．
終了 プロセスは終了する．
コアダンプ プロセスは core ファイルを作成してから終了する．

シグナルの扱い方 = シグナルハンドリング

signal システムコール

自プロセスのシグナルハンドリングを設定する.

書式 sig シグナルの種類, func は新しいハンドリング.

```
#include <signal.h>
sig_t signal(int sig, sig_t func);
```

解説 sig はハンドリングを変更するシグナル
SIG_IGN はシグナルを無視するようにする.
SIG_DFL はシグナルハンドリングをデフォルトに戻す.
シグナルハンドラ関数を指定すると捕捉になる.
シグナルハンドラ関数のプロトタイプ宣言は次の通り.

```
void func(int sig);
```

戻り値 正常なら変更前のハンドリングが,
エラーなら SIG_ERR が返される.

プログラム例：シグナルを無視する例

SIGINT を一時的に無視するプログラムの例を示す.

```
1  #include <signal.h>
2
3  int main() {
4    ...
5    signal(SIGINT, SIG_IGN);  // ここから
6    ...
7    signal(SIGINT, SIG_DFL);  // ここまで
8    ...
9  }
```

5 行 Ctrl-C を無視するようにハンドリングを変更する.

6 行 Ctrl-C を押してもプログラムが終了しない状態.

7 行 ハンドリングを元に戻し Ctrl-C で終了するようにする.

プログラム例：シグナルを捕捉する例

SIGINT を捕捉するプログラムの例を示す.

```
1  #include <signal.h>
2
3  void handler(int n) {          // シグナルハンドラ（プロトタイプ宣言どおり）
4    ...                          // シグナル処理
5  }
6
7  int main() {
8    ...
9    signal(SIGINT, handler);     // ここから（void f(int)型の関数を引数にする）
10   ...
11   signal(SIGINT, SIG_DFL);     // ここまで
12   ...
13 }
```

9 行 SIGINT のハンドリングを捕捉にする.

10 行 Ctrl-C が押される度に handler() 関数を実行.

11 行 SIGINT のハンドリングをデフォルト（終了）に戻す.

シグナルハンドラの制約

1) 制約がある理由
   ハンドラ関数はいつ呼ばれるか分からない.
   例えば `printf()` がバッファ操作中にシグナル捕捉！！
   ハンドラ関数中で `printf()` 実行 → 何か悪いことが起こる

2) やってもよいこと
   1. シグナルハンドラ関数のローカル変数の操作
   2. `volatile sig_atomic_t` 型変数の読み書き
      `sig_atomic_t` 型は macOS では `int` 型
      (「読み書き」は単純な参照と代入のことだけ指す)
      `volatile` を付けると C コンパイラの最適化の対象外
      (最適化は非同期にアクセスを前提にしていない.)
   3. 非同期シグナル安全な関数の呼び出し
      `_exit()`, `alarm()`, `chdir()`, `chmod()`, `close()`, `creat()`, `dup()`, `dup2()`, `execle()`, `execve()`, `fork()`, `kill()`, ...

シグナルハンドラの例

```
1  #include <unistd.h>
2  #include <signal.h>
3  volatile sig_atomic_t flg = 0;          // シグナルハンドラが操作しても良い
4  void handler(int n) {
5    flg = 1;                              // 単純な代入
6    write(1, "Ctrl-C\n", 7);              // 非同期シグナル安全な関数の実行
7  }
8  int main(int argc, char **argv) {
9    int cnt = 0;
10   signal(SIGINT, handler);
11   while (cnt<3) {
12     if (flg) {                          // 単純な参照
13       cnt++;
14       flg = 0;                          // 単純な代入
15     }
16   }
17   return 0;
18 }
```

3 行 ハンドラ関数が操作できる変数 `flg` を宣言

4 行 ハンドラ関数（限られた操作しかできない）

kill システムコール

プロセスがプロセスにシグナルを送信するシステムコール

書式 pid は送り先プロセス，sig はシグナルの種類

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

解説 送信先のプロセスとシグナルの種類を指定してシグナルを送信する．（シグナルの種類は前出の表の通り）

戻り値 正常時は 0，異常時-1 が返される．

プログラム例 kill システムコールの使用例として，kill コマンドを簡単化したプログラム（mykill）を示す．

使用方法：`mykill <シグナル番号> <プロセス番号>`

mykill プログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>                    // atoi のために必要
3  #include <signal.h>                    // kill のために必要
4
5  int main(int argc, char *argv[]) {
6    if (argc!=3) {
7      fprintf(stderr, "Usage : %s SIG PID\n", argv[0]);
8      return 1;
9    }
10
11   int sig = atoi(argv[1]);             // 第 1 引数
12   int pid = atoi(argv[2]);             // 第 2 引数
13
14   if (kill(pid,sig)<0) {
15     perror(argv[0]);
16     return 1;
17   }
18   return 0;
19 }
```

- atio() は，文字列"123"を整数 123 に変換

mykill の実行例

```
$ ./mykill                                  <-- 使い方が分からない
Usage : ./mykill SIG PID                    <-- 使い方を表示してくれる
$ sleep 1000 &
[1] 13589                                   <-- sleep が JOB=1, PID=13589
だと分かる
$ ./mykill 2 13589                          <-- PID=13589のプロセスにSIGINT
を送る
$                                           <-- Enter をもう一度入力
[1]+  Interrupt: 2    sleep 1000
$ ./mykill 100 13589
./mykill: Invalid argument                  <-- シグナル番号が不正
$ ./mykill 2 13589
./mykill: No such process                   <-- プロセス番号が不正
$
```

- Grapher は macOS のグラフ作成アプリ
- Grapher を使用する実行例

sleep システムコール

自プロセスを指定された時間，待ち状態にする．

書式 seconds は待ち時間（秒単位）

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

解説 決められた時間待ち状態になる．途中でシグナルが届いた場合は終了する．（ハンドリングが無視以外の場合）

戻り値 sleep 予定だった残りの秒数が返される．（通常は 0 のはず）

プログラム例 1 秒に一度 hello と表示するプログラム（Ctrl-C で終了）

```
#include <stdio.h>
#include <unistd.h>
int main() {
  for (;;) {                // 無限ループ
    printf("hello\n");      // hello表示
    sleep(1);               // 1 秒待つ
  }
  return 0;
}
```

pause システムコール

自プロセスを待ち状態にする（時間制限なし）.

書式　#include <unistd.h>
int pause(void);

解説　時間を定めず待ち状態になる．途中でシグナルが届いた場合は終了する．(ハンドリングが無視以外の場合)

戻り値　常にエラーで終了するので-1

プログラム例　SIGINT のハンドリングを捕捉にした例

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void handler(int n) {}          // 何もしないハンドラ関数
int main() {
  signal(SIGINT, handler);      // SIGINTを捕捉に変更する
  pause();                      // 1回目の  Ctrl-C を待つ
  pause();                      // 2回目の  Ctrl-C を待つ
  pause();                      // 3回目の  Ctrl-C を待つ
  return 0;                     // Ctrl-C 3回で終了する
}
```

alarm システムコール

アラームシグナルの発生を予約する.

書式 `#include <unistd.h>`

`unsigned int alarm(unsigned int seconds);`

解説 seconds 秒後に SIGALRM が発生する.
予約するだけでプロセスが待ち状態になるわけではない.

戻り値 前回の alarm システムコールの残り時間（通常 0）

プログラム例 SIGALRM のハンドリングを捕捉にした例

```
#include <stdio.h>
#include <unistd.h>                    // alarm, pause のために必要
#include <signal.h>                    // signal のために必要
void handler(int n) {}                 // 何もしないハンドラ関数
int main(int argc, char *argv[]) {
  signal(SIGALRM, handler);            // SIGALRMを捕捉に変更する
  alarm(3);
  printf("pause()します\n");
  pause();                             // プロセスが停止する
  printf("pause()が終わりました\n");
  return 0;
}
```

課題 No.6

1. リスト 7.3 のプログラムは，Ctrl-C が押された回数を `main()` 関数側でカウントしている．そのため，`main()` 関数の処理が忙しくて `flg` 変数を頻繁にチェックできない場合に，Ctrl-C の回数を正確にカウントできないかもしれない．
   `main()` 関数の力を借りずに Ctrl-C の回数をカウントし，三回目の Ctrl-C のとき `flg` 変数を 1 にするようにプログラムを改良しなさい．
   シグナルハンドラ中ではグローバル変数のインクリメントはできないものとする．
   ヒント：シグナルハンドラ中で `signal()` を実行してもよい．
2. sleep システムコールを用いないで，指定秒数プログラムを待ち状態にする関数 `mysleep(int seconds)` を作りなさい．