# minitalk

## 概要

SIGUSR1とSIGUSR2、この2つだけでデータの送受信ができるような CLIENT と SERVER を作る。具体的には、送りたい文字列 をビットに変換して、1ビットずつ CLIENT から SERVER にデータを送信する。

## 送りたい文字列のビット変換

`int32_t型` のビット配列は、こうなっている。(int型は環境によってビット数が異なるので、32ビットに固定した。)

```
00000000000000000000000000000000
```

具体的に考えてみよう。`char型` の 'a'(ascii==97) を `int32_t型` に変換した場合のビット配列は、こうなっている。

```
00000000000000000000000001100001
```

で、例えば、一番右のビットを取りたい場合、どうすればいいか？ビット演算で取ることができる。

- ビットのONを取りたい場合上は'a'(ascii==97) のビット配列、下は1のビット配列。

  ```
  00000000000000000000000001100001
  00000000000000000000000000000001
  ```

  この2つでビット演算 `&` を行うと、1が返ってくる。`&` はどちらも1の時に1を返すから、どちらも1である一番右のビットだけが1となる。それを`int32_t型`(`int型`でも可)で解釈すると、1となる。

  `00000000000000000000000000000001`

- ビットのOFFを取りたい場合上は'a'(ascii==97) のビット配列をビット演算 `>>` で1桁右にシフトしたもの、下は1のビット配列。

  ```
  00000000000000000000000001100000
  00000000000000000000000000000001
  ```

  この2つでビット演算の `&` を行うと、0が返ってくる。`&` はどちらも1の時に1を返すが、すべて該当しないので、すべてのビットが0となる。それを`int32_t型`(`int型`でも可)で解釈すると、0となる。

  `00000000000000000000000000000000`

この処理を、送りたいビット数だけループで繰り返せばいい。例えば8ビット送りたい場合、こんな感じになる。

- ビット配列を 7ビット 右シフトして、右から 8ビット目を取り、送信。
- ビット配列を 6ビット 右シフトして、右から 7ビット目を取り、送信。

  ：
- ビット配列を 1ビット 右シフトして、右から 2ビット目を取り、送信。
- ビット配列を 0ビット 右シフトして、右から 1ビット目を取り、送信。

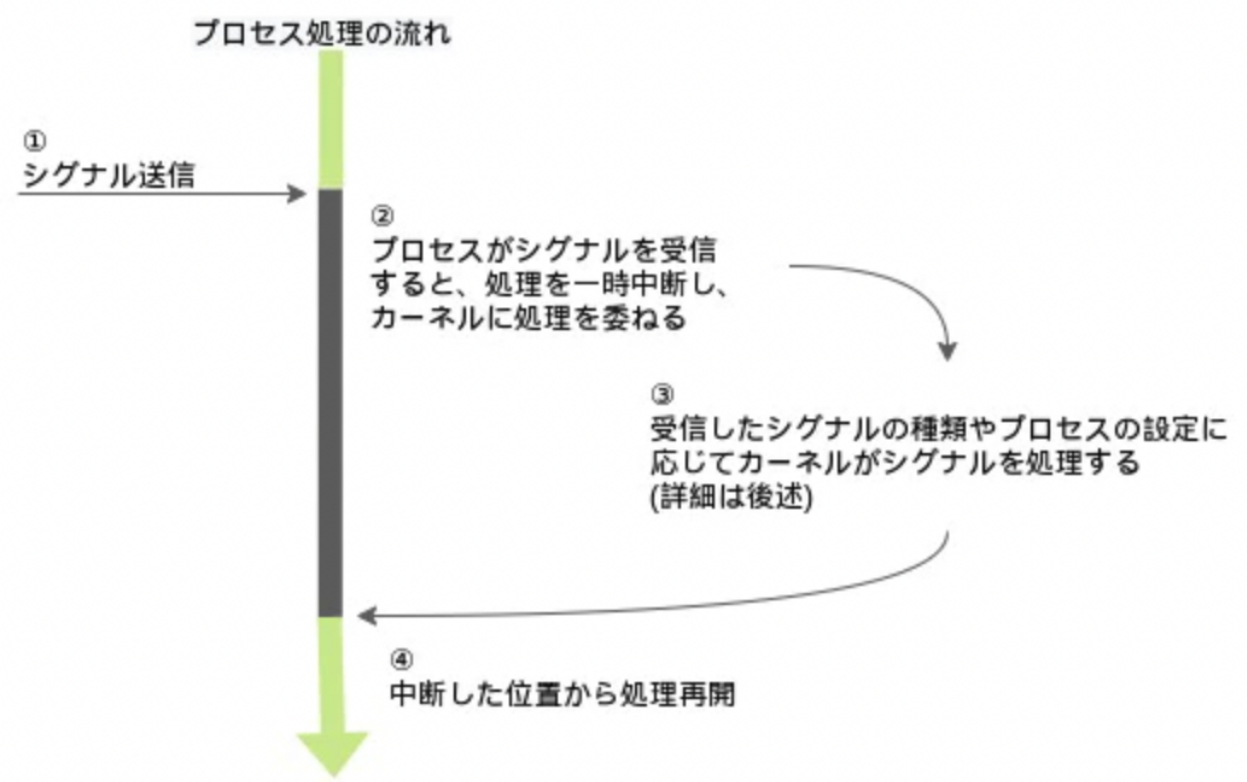
# 変換したビットの送信

CLIENT側 から SERVER側 へ送信するには signal を使う。具体的には、`SIGUSR1` と `SIGUSR2` をビットの ON / OFF に見立てて、CLIENT側から送信してやればいい。signal を送るには `kill` 関数を使う。第1引数には 送りたい相手のProcessID、第2引数には 送りたいシグナルをセットする。

# シグナル

プロセスとプロセスの間で通信を行う際に使用される“信号”のことで，シグナルを受け取ったプロセスは“何らかの動作”を行う．

あらかじめ定義された条件分岐命令とは異なり，あるプロセスが他のプロセスに何かを通知し，外部から実行の流れを強制的に変えるための仕組み




例）

［CTRL］＋［C］→「SIGINT」というキーボードからの割り込みシグナル

［CTRL］＋［Z］→「SIGTSTOP」という一時停止シグナル

# SIGUSR1，SIGUSR2

UNIX は多くのシグナルを定義しており，そのほとんどは特定の目的のために確保されているが、 SIGUSR1 と SIGUSR2 の２つは、自由に用途を決めることができる．

# killコマンド

killコマンドは指定したプロセスIDのプロセスを終了させる．

```
kill [オプション] プロセスID
```

**主なオプション**

| | |
|---|---|
| -s シグナル | 指定したシグナル名またはシグナル番号を送信する |
| -シグナル | 指定したシグナル名またはシグナル番号を送信する |
| -l [] | シグナル名とシグナル番号の対応を表示する |

### 主なシグナル

| | | |
|---|---|---|
| 1 | SIGHUP | 再起動 |
| 6 | SIGABRT | 中断 |
| 9 | SIGKILL | 強制終了 |
| 15 | SIGTERM | 終了 |
| 17 | SIGSTOP | 停止 |
| 18 | SIGCONT | 再開 |

シグナル名を指定する場合は、最初の３文字「SIG」は省略して入力します。たとえば、SIGABRTシグナルを送りたい場合は、以下のように入力します。

**$ kill -ABRT プロセスID**

# kill関数

### 書式

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

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

### 説明

任意のプロセスグループまたはプロセスに，シグナルを送ることができる．第一引数に指定されたプロセスpidに，第二引数のsigが送られる．

| *pid* | 意味 |
|---|---|
| *pid* ＞ 0 | *pid* で指定された ID を持つプロセスに，*sig* で指定したシグナルが送られる. |
| *pid* = 0 | 呼び出し元のプロセスのプロセスグループに属するすべてのプロセスに，*sig* で指定したシグナルが送られる． |
| | |

| | |
|---|---|
| *pid* = -1 | 呼び出し元のプロセスがシグナルを送る許可を持つ全てのプロセスに，*sig* で指定したシグナルが送られる。但し，プロセス番号 1 (*init*) へはシグナルは送られない。 |
| *pid* < -1 | ID が *-pid* のプロセスグループに属するすべてのプロセスに，*sig* で指定したシグナルが送られる。 |

**戻り値**

成功した場合は0、失敗した場合には-1が返されerrnoにエラー内容が格納される。エラー内容は以下のとおり

| 名前 | 意味 |
|---|---|
| SINVAL | 無効なシグナル |
| EPERM | シグナルを送る許可を持っていない |
| ESRCH | 指定したプロセスまたはプロセスグループが存在しなかった |

**プロセスIDの取得**

kill関数の第二引数にはプロセスIDを指定するが、Linuxの場合であれば「ps -ef」を実行すれば、取得することができる。

たとえば、以下ではサーバープロセスが立ち上がっているが、左から2番目目がプロセスIDで，左から3番目が親プロセスのプロセスIDである。

```
kimoto   20220 25715  0 Dec25 ?        00:05:35 /home/kimoto/labo/perltweet/script/perltweet
```

上記では、左から2番目の「20220」という番号がプロセスIDです。これがkillに指定するプロセスIDになる。

# signal関数

**書式**

```
#include <signal.h>

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

**説明**

signal 関数は sig で指定されるシグナルを受け取った時に，どう扱うかを３つの手段から選ぶ．

- func の値が `SIG_DFL` であれば，既定の対応をする．
- `SIG_IGN` であれば，そのシグナルは無視される．
- どちらでもなければ，func は関数へのポインタと見なされ，その関数はシグナル発生時に呼ばれるものとして設定される．

宣言が読みにくいが、こう翻訳できる。

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

ここで、__p_sig_fn_tは、「void (int)型へのポインタ」型である。つまり関数のポインタとなる。

signal()の第2引数に関数へのポインタを渡すかわりに、以下のマクロを使うこともできる。

- SIG_DFL 規定の対応をする。例えば致命的な実行時エラーなら、処理を中断する。
- SIG_IGN そのシグナルを無視する。この場合致命的なエラーでも実行が続いてしまうことに注意。

**戻り値**

返り値は、関数のセットが成功すればその関数のポインタ値、成功しなければSIG_ERRである。 成功しなかった場合は正の数がerrnoにセットされる。

# 送られたシグナルの受信

CLIENT側 から SERVER側 へ送信された signal を受信するには `signal` 関数を使う。第1引数には 受け取りたいシグナル、第2引数には シグナルを受け取った時の処理への関数ポインタをセットする。

```
static void handle_signal(int signal)
{
```

```
  g_receive_signal = signal;
}

void  set_signal(void)
{
  signal(SIGUSR1, &handle_signal);
  signal(SIGUSR2, &handle_signal);
}
```

# 受信確認システム

SERVER側 でビット(シグナル)を受け取って処理が完了したら、CLIENT側 に対して「受け取れた！」というシグナル(一般的にACKと呼ばれるもの)を送って、それをCLIENT側 で受け取れたら、次のビットを送る。これを最後まで繰り返せば、確実に送信することができる。

ただ、CLIENT側 にシグナルを送るためには CLIENT の ProcessID が必要になるので、文字列を送る前にそれを送る処理を追加する必要がある。

# getpid() , getppid()- プロセス ID の取得

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

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

**引数**

なし

**戻り値**

正の値：自分のプロセスID (getpid) / 親のプロセスID (getppid)

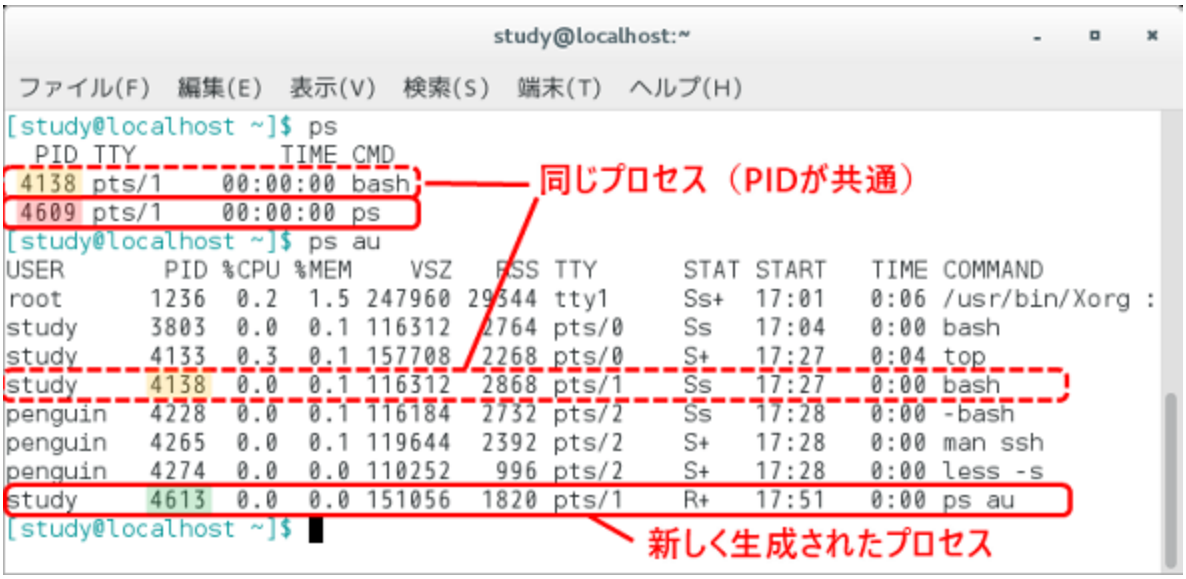
# プロセスID

## プロセスIDとは？

プロセスには、それぞれ“固有の番号”が割り振られます。これを「**プロセスID（PID）**」と呼びます。

例えば、「pts/1」で1回目に実行したpsコマンドのプロセスIDは「4609」になっています（**画面3**）。

**画面3**　「ps」と「ps au」を実行した結果（画面1、画面2でpsコマンドを実行している画面）




psコマンドは、プロセスを表示してすぐに終了します。2回目に実行したときは新たに「4613」というプロセスIDが割り振られています。

一方、ログインシェルであるbashは、ログアウトするまで（端末アプリケーションの場合はアプリケーション終了をするまで）メモリに滞在するので、「pts/1」のbashのプロセスIDは「4138」ままです。

## 親プロセスと子プロセスの関係

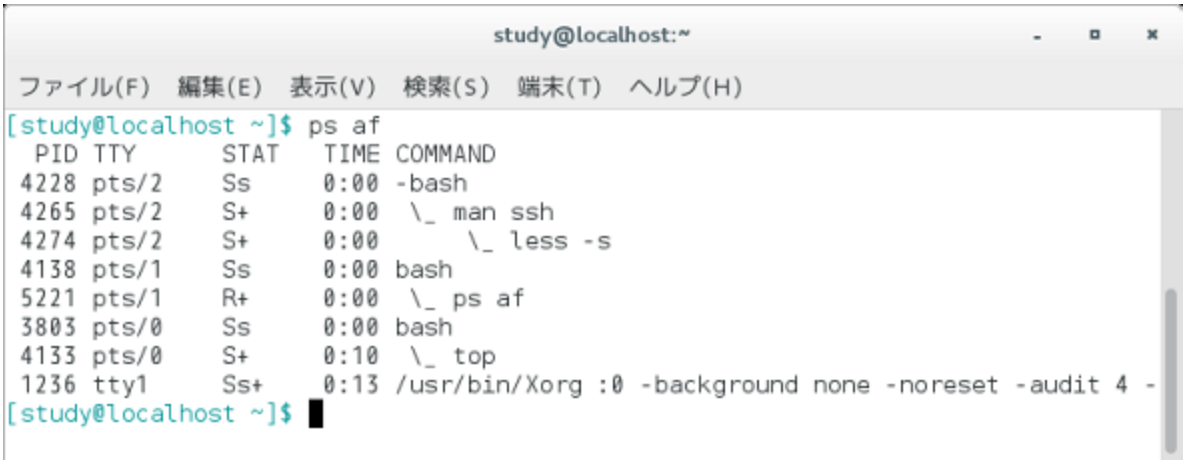
プロセスには“親と子”の関係があります。例えば、bashのプロンプトでpsを動かしている場合、“bashのプロセスが親”で“psのプロセスが子”になります。それぞれ「**親プロセス**」と「**子プロセス**」と表現します。

プロセスの親子の階層は、psコマンドの「f」または「--forest」オプションで表示することができます（**画面4**）。

**ps af**

（端末のプロセスを階層表示する）

**画面4**　「ps af」で端末のプロセスを階層表示できる



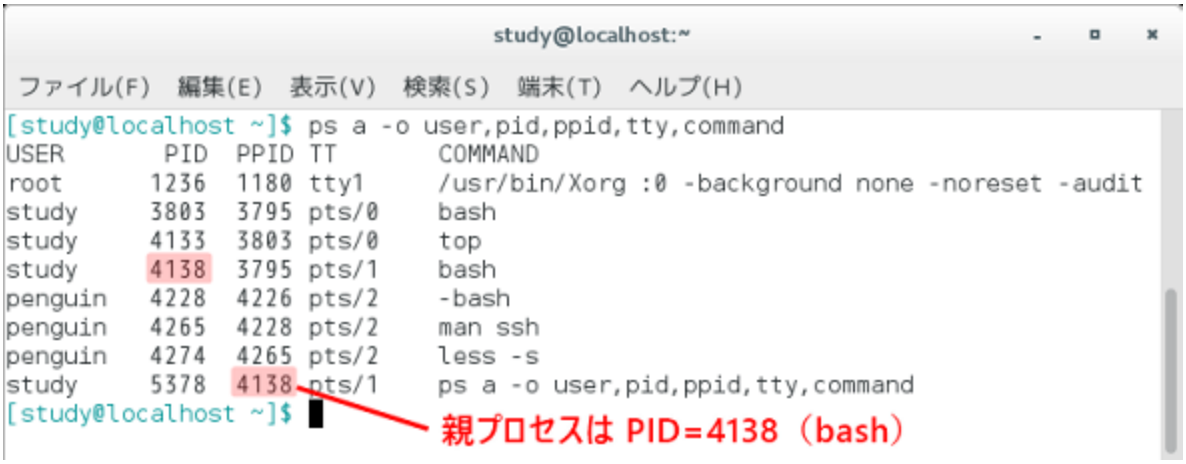

## 親プロセスのID

それぞれの子プロセスは、親プロセスのIDも保持しています。「-o」オプションで表示してみましょう。

以下のコマンド例では、USERとPID、PPID（親プロセスのID）、そしてTTYとCOMMAND欄を表示しています（**画面5**）。

**ps a -o user,pid,ppid,tty,command**

（USER、PID、PPID、TTY、COMMANDを表示）



# pause() - シグナルを保留中のプロセスの中断

### 形式

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

int pause(void);
```

### 機能説明

呼び出しスレッドの実行を中断します。シグナル・ハンドラーを実行するか、スレッドを終了するシグナルが送達されるまで、スレッドは実行を再開しません。プロセスの *thread* によって一部のシグナルをブロックできます。詳細は、sigprocmask() - スレッドの検査または変更を参照してください。

着信ブロック解除シグナルでスレッドが終了すると、pause() は 二度と呼び出し元へは戻りません。着信シグナルがシグナル・ハンドラーによって処理される場合、pause() は シグナル・ハンドラーが戻ってから戻ります。

### 戻り値

pause() が戻るとき、通常 -1 の値を戻し、errno が EINTR に設定されます。これは、シグナルが受け取られ、正常に処理されたことを示します。

# usleep() - マイクロ秒単位で実行を延期する

### 形式

```
#include <unistd.h>

int usleep(useconds_t usec);
```

### 機能説明

**usleep**() 関数は (少なくとも) *usec*
マイクロ秒の間、 呼び出し元スレッドの実行を延期する。 システムの動作状況や呼び

出しによる時間の消費やシステムタイマーの粒度によって、 停止時間は設定した値よりも少し延ばされるかもしれない。

**戻り値**

**usleep**() 関数は成功すると 0 を返す。 エラーの場合、 -1 が返され、 *errno* にエラーの原因を示す値が設定される。

**エラー**

EINTR　：　シグナルによって中断された。

EINVAL　：　*usec* が 1000000 以上だった。 (これをエラーとみなすシステムのみ)