16 UNIXプロセス間通信(3)
——シグナル——

16–1 シグナル機構の必要性

これまでに紹介したプロセス間通信の機構だけでは、意識的に通信操作を行わないプロセスに対しては通信手段が全く無い。　しかし、

- 例えば実行時にゼロ除算を起こしたプロセスに対しては、割り込み信号を受けたカーネルがその事実を知らせ、プロセス自身にそれに対処してもらいたい時もある。

  (安易に強制終了で済まない場合もある。)

⇨ 割り込みの様にプロセスに対して何らかの特殊な事象の生起を知らせる、言わばソフトウェア割込みの機構が用意されている。

- この機構の下でプロセスに伝えられるものをシグナルと呼ぶ。

16–2 シグナルによるソフトウェア割り込みの機構

p.35からの引用：

計算機を稼働している際には、ハードウェアの誤動作, 電源異常, ユーザプログラムのエラー, 並行して独立に動作させている入出力機器からのメッセージ到着、といった事象が起こり得る。 これらの事象は緊急の処置を要するものが多いので、通常はこれらの事象に対して

(1) 計算機がその時点で実行している処理を一旦中断し、

(2) 起こった事象に対する処置を完了した上で、

(3) 元々実行していた処理を再開する、

といったことを行う。 これらの機能を一般に割込みと言う。

シグナルの機構：

割り込み信号の受け手がカーネルであるのに対してシグナルの受け手はプロセスであるという違いはあるが、その他の点では、シグナルの機構は割り込みの機構とほぼ同じである。

シグナルの機構：

割り込みの機構とほぼ同じ。

- 各々のプロセスは、カーネルや他プロセスから種々の事象 (i.e. メモリ例外, ゼロ除算, 強制終了の要求がキーボード等からあった, ...) の生起を随時知らせてもらうための窓口を持っている。
  (プロセスに伝えられるものをシグナルと呼ぶ。)

- シグナルの種類を識別するために、各々のシグナルには番号が付けられている。 (⇨ 16.3節。)

- プロセスは届いたシグナルの種類を見てそのシグナルを無視したり処理に応じたりすることが出来る。 (⇨ 16.4節。)

- 処理に応ずる各々のシグナルに対して、各プロセスはシグナル処理のルーチン (シグナルハンドラと呼ぶ) を予め用意しておくことが出来る。 (⇨ 16.4節。)

- 処理に応ずるシグナルを受け取ると、プロセスは生起した事象を処理するルーチンを持っているかどうかを調べ、持っていれば、

  (1) プロセスが実行中の処理を一旦中断し、
  (2) 起こった事象に対する処置を完了した上で、
  (3) 元々実行していた処理を再開する、

  といったことを行う。

  生起事象を処理するルーチンを持っていなければ、システムがデフォルトに用意した処置(大抵はプロセスの終了)が施される。

- `alarm()` システムコールを用いて
  指定した時間後に自プロセスに`SIGALRM`シグナルを送ったり、

  `kill()` システムコールを用いて
  他プロセスに任意の番号のシグナルを送ったり出来る。

(⇨ 16.5節。)

16–3 シグナルの種類

どんなシグナルが用意されているか：

例えば、**Vine Linux 2.1.5** では次の通り。

但し
- SIG で始まる文字列 ··· シグナルの名前
- その前の番号 ··· シグナルの番号

```
[motoki@x205b]$  kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS
[motoki@x205b]$
```

補足： 用意されているシグナルはUNIXの種類によって少し違う。例えば、情報工学科実習室のUNIX(SunOS 5.8)の場合は次の通り。

```
sv01_41:  kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM
USR1 USR2 CLD PWR WINCH URG POLL STOP TSTP CONT TTIN TTOU VTALRM PROF
XCPU XFSZ WAITING LWP FREEZE THAW CANCEL LOST RTMIN RTMIN+1 RTMIN+2
RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
sv01_42:
```

VineLinux2.1.5では、使えるシグナルの名前はヘッダファイル/usr/include/asm/signal.hの中でマクロとして次の様に定義されている。

```
[motoki@x205b]$  more /usr/include/asm/signal.h
    ......(途中省略)......
#define SIGHUP          1
#define SIGINT          2
#define SIGQUIT         3
#define SIGILL          4
#define SIGTRAP         5
#define SIGABRT         6
#define SIGIOT          6
#define SIGBUS          7
    ......(以下省略)......
[motoki@x205b]$
```

各々のシグナルの意味：

Linuxにおける各々のシグナルの発生要因、デフォルトの処置等を次に示す。

シグナルの種類	発生要因, 等	デフォルトの処置
SIGHUP	HangUPの意。 端末の回線が切れた時に発生する。 （実際のシグナルの受け手等がどうなっているかについては、UNIXの種類/セッション管理の実装方法によって少しずつ違っている。SVR4の場合、端末ドライバが通信の途切れを発見したらセッションリーダだけにSIGHUPシグナルを送る。）	終了
SIGINT	INTerruptの意。 ユーザがキーボードを使って割り込みの指示をした。(通常は Ctrl-c キー。)	終了
SIGQUIT	ユーザがキーボードを使って割り込みの指示をした。(通常は Ctrl-\ キー。)	coreを生成して終了。

シグナルの種類	発生要因, 等	デフォルトの処置
SIGILL	ILLegal instructionの意。 不正な型式の機械語命令を実行しようとした。	coreを生成して終了。
SIGTRAP	プロセスが(デバッグモード中に)ブレイクポイントに出会った。	coreを生成して終了。
SIGIOT	I/O Trapの意。	coreを生成して終了。
SIGBUS	ワード境界を無視してメモリアクセスを行った。	coreを生成して終了。
SIGFPE	Floating Point Exceptionの意。 0による除算, オーバーフロー または アンダーフローが起こった。	coreを生成して終了。
9番 SIGKILL	強制終了の指令が出た。	終了 (無視することも、他の処置で置き換えることも出来ない。)

シグナルの種類	発生要因, 等	デフォルトの処置
SIGUSR1	ユーザが定義したシグナル1。	終了
SIGSEGV	**SEGmentation Violation**の意。 プロセスに割り当てられていない領域へのアクセス、または書き込み禁止領域への書き込みを行おうとした。	core を生成して終了。
SIGUSR2	ユーザが定義したシグナル2。	終了
SIGPIPE	読み手のないパイプへ書き込もうとした。	終了
SIGALRM	**ALaRM**の意。 alarm( ) システムコールが設定したタイマーによって発生する。	終了
SIGTERM	**TERMinate**の意。 終了の要求が出た。 (kill プロセスID とコマンド入力した時に出されるのがこのシグナル。)	終了

シグナルの種類	発生要因, 等	デフォルトの処置
SIGCHLD	CHiLDの意。 子プロセスの状態が変更された。	無視する。
SIGCONT	CONTinueの意。 プロセスの続行要求が出た。 (割り込みキー Ctrl-q を押すとこのシグナルが出る。Ctrl-s で(仮想)端末の画面表示が停止されていた場合は、これによって表示が再開される。)	停止中なら再開し、それ以外の場合は無視する。
SIGSTOP	停止の指令が出た。 (割り込みキー Ctrl-s を押すとこのシグナルが出る。これによって、(仮想)端末に送ると画面表示が停止されキーを受けつけなくなった様に見える。)	停止する。 (無視することも、他の処置で置き換えることも出来ない。)
SIGTSTP	Terminal SToPの意。 ユーザがキーボードを使って停止の指示をした。 (通常は Ctrl-z キー。)	停止する。 (バックグラウンドへ。)

シグナルの種類	発生要因, 等	デフォルトの処置
SIGTTIN	バックグラウンドで走るプロセスでstdinからの入力が試みられた。	停止する。
SIGTTOU	バックグラウンドで走るプロセスでstdoutへの出力が試みられた。	停止する。
SIGURG	ソケットに緊急事態発生。	無視する。
SIGXCPU	プロセスに割り当てられたCPU時間の上限を超過した。	coreを生成して終了。
SIGXFSZ	プロセスがアクセスしようとしたファイルの大きさが決められた上限を越えていた。	coreを生成して終了。
SIGVTALRM	Virtual Timer ALaRMの意。setitimer()システムコールが設定したプロセス仮想時間のタイマーによって発生する。	終了

シグナルの種類	発生要因, 等	デフォルトの処置
SIGPROF	**PROFiling timer alarmの意。** **profiling timerよって発生する。** (profiling timer はプロセスのプロファイリング(e.g.各々の関数の呼ばれた回数, 実行時間等を測る)を行うために用意されているタイマで、プロセス仮想時間と システムがプロセスの代わりに実行している時間の両方を別々に測ることが出来る。このタイマもsetitimer()システムコールを使って利用できる。)	終了
SIGWINCH	ターミナル・ウィンドウの大きさが変更された。	無視する。
SIGIO	非同期I/Oイベントが発生。	無視する。
SIGPWR	システム電源異常を検知した。	無視する。
SIGSYS	システムコールに対して誤った引数が与えられた。	coreを生成して終了。

補足：

どのキーに割り込み機能が割り当てられているかを調べるには、次の様にstty -aとコマンド入力すればよい。

```
[motoki@x205b]$  stty -a
speed 38400 baud; rows 43; columns 61;
line = 0; intr = ^C; quit = ^\; erase =
^H; kill = ^U; eof = ^D; eol = M-^?; eol2
= M-^?; 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
[motoki@x205b]$
```

ここで、例えばDELキーにSIGINTシグナル発生の機能を割り当てるには stty intr 0x7F またはstty intr ^? とコマンド入力すればよい。 (0x7Fは DEL という機能文字のASCIIコード。)

16–4 シグナルに対する処理の指定

シグナルに対する処置を指定するシステムコール signal( ) :

- 関数プロトタイプは

```
void (*signal(int signum, void (*handler)(int)))(int);
```

- ヘッダファイル<signal.h> を必要とする。

補足 : この関数プロトタイプは次の様に見る。

(第1引数のデータ型) int

(第2引数のデータ型) void (*handler)(int),
すなわち int型引数を取る void 型関数へのポインタ。

(関数値のデータ型) void (*signal(...))(int),
すなわち int型引数を取る void 型関数へのポインタ。

従って、signalは整数と関数(へのポインタ)を引数に取って、関数(へのポインタ)を返す関数ということになる。

- 関数引数の signum には、処置を設定したいシグナルの番号または名前を指定する。
- 関数引数の handler には、シグナルに対する処置内容を表す、
  - マクロ名 SIG_DFL,
  - マクロ名 SIG_IGN, または
  - シグナルハンドラの名前

  のいずれかを指定する。
- signal(signum, SIG_DFL) が呼ばれると、番号がsignumのシグナルに対する処置がデフォルトなものに設定される。
- signal(signum, SIG_IGN) が呼ばれると、番号がsignumのシグナルが無視される様になる。
- signal(signum, handler_name) が呼ばれると、番号がsignumのシグナルに対する処置が関数handler_nameによって行われる様になる。
- いずれの場合も、成功すると 元の処置を行っていた関数へのポインタを関数値として返し、失敗するとSIG_ERR を返す。

例16. 1 (シグナルに対する処置の指定)

シグナル番号を引数として受け取り

I (PID=...) received a signal of #... just now.

というメッセージを書き出す関数を、Ctrl-cキー, Ctrl-\キーによって発生する2つのシグナル SIGINT, SIGQUIT のハンドラとする例を示す。

[motoki@x205a]$ nl set-sig-handler-for-SIGINT.c

```
 1 /*************************************************************
 2 /* Operating-Systems/C-Programs/set-a-handler-for-SIGINT.c
 3 /*-----------------------------------------------------------
 4 /* シグナル番号を引数として受け取り                          *
 5 /*     I (PID=...) received a signal of #... just now.
 6 /* というメッセージを書き出す関数を定義して、これをシグナル */
 7 /* SIGINT(Ctrl-C), SIGQUIT(Ctrl-\)のシグナルハンドラと
 8 /* するプログラム例を示す。                                  *
 9 /* 塚越一雄「Linuxシステムコール」(技術評論社,2000)4.2.10節 */
10 /*************************************************************
11 #include <stdio.h>
```

```
12 #include <signal.h>   /* for signal() system call */
13 #include <unistd.h>   /* for sleep() and getpid() system ca
14 #include <sys/types.h> /* for getpid() system call */

15 void verbose(int signum);

16 int main(void)
17 {
18   int i;
                          ハンドラ
19   signal(SIGINT, verbose);
20   signal(SIGQUIT, verbose);

21   for (i=0; i<10; i++) {
22     printf("%03d: A program is running.\n", i);
23     sleep(2);
24   }
```

```
25    return 0;
26 }

27 void verbose(int signum)
28 {
29    printf("I(PID=%d) received a signal of #%d just now.\n",
30           getpid(), signum);
31    /* signal(signum, verbose); */ 次のシグナルのための再設定
32 }
[motoki@x205a]$  gcc set-sig-handler-for-SIGINT.c
[motoki@x205a]$  ./a.out
000: A program is running.
001: A program is running.
002: A program is running.
Ctrl-c
I (PID=17378) received a signal of #2 just now.
```

```
003: A program is running.
Ctrl-c
I (PID=17378) received a signal of #2 just now.
004: A program is running.
005: A program is running.
Ctrl-\
I (PID=17378) received a signal of #3 just now.
006: A program is running.
Ctrl-\
I (PID=17378) received a signal of #3 just now.
007: A program is running.
008: A program is running.
Ctrl-c
I (PID=17378) received a signal of #2 just now.
009: A program is running.
[motoki@x205a]$
```

16–5 シグナルの送信

他プロセスに思い通りのシグナルを送ったり、
指定した時間後に自プロセスにSIGALRM シグナルを送ったり
することも可能である。

⇨ このシグナル送信の機能とシグナルハンドラ指定の機能を組み合わせれば、プロセス間の同期制御もある程度は出来る。

プロセスに思い通りのシグナルを送るシステムコール

int kill(pid_t pid, int signum) :

- <signal.h> と <sys/types.h> が必要。
- 番号signumのシグナルが第1引数で指定されたプロセスに送られる。
- 関数引数の pid はシグナルの送付先を表す。

  (場合1 : pid≥1)

  ID番号がpidのプロセスに。

  (場合2 : pid=0)

  呼び出しプロセスと同じプロセスグループ内のプロセス全てに。

  (場合3 : pid=−1) ..... UNIXの種類に依存。

  呼び出しプロセスの実効ユーザIDと同じ実ユーザIDを持つプロセス全てに。

  (場合4 : pid≤−2)

  プロセスグループIDが |pid| のプロセス全てに。

補足：

UNIXでは、端末とプロセスの間のアクセス関係,ログインセッションの管理をうまく行うために、互いに関連し合うプロセスの集まりを考えそれらをプロセスグループと呼んでいる。プロセスグループ内のプロセスは全て同じプロセスグループID(PGID) を持つ。 通常、プロセスグループIDは親プロセスから子プロセスへと受け継がれる。

ただ、実際にプロセスグループをどの様に定義するかは、UNIXのバージョンによってかなり違う。例えば、

(古いSVR3では) 同じ制御端末を共有する (i.e.1つのログインセッションに属する) プロセスの集合がプロセスグループになっていた。

$\Longrightarrow$ `kill(0,SIGTERM)` とすることによって、同一ログインセッションに属する全プロセスをまとめて終了させることが出来る。

(古い4.3BSDでは) ログインセッションの中の各々のジョブがプロセスグループに相当する。シェル環境下では、通常コマンドライン毎に1つのプロセスグループが構成されるので、パイプラインで連続したコマンド列を実行する場合にはパイプで繋がれたプロセス群がプロセスグループが構成される。

$\Longrightarrow$ `kill(0,SIGTERM)` とすることによって、パイプで繋がれたプロセスをまとめて終了させることが出来る。

(SVR4では) .....

指定した時間後に自プロセスにSIGALRMシグナルを送るシステムコール unsigned int alarm(unsigned int seconds) :

- <unistd.h> を必要とする。
- seconds秒後に自プロセスにSIGALRMシグナルが送られる。
- alarm( )実行時に既にアラームクロックがセットされていた場合はシグナル送信までの残りの秒数が返され、アラームがセットされていなかった場合は0 が返される。

シグナル受け取りを待つシステムコールint pause(void) :

- この関数を使うためには<unistd.h> を必要とする。
- シグナルを受け取るまでプロセスを待ち状態にする。
- 関数値は常に -1 である。

例16. 2 (シグナルを使って親子のプロセス間で会話をする; 失敗例)

例14.3のプログラムとほぼ同等のことをシグナルの機構を用いて行おうとしたが、...。

ダメプログラムの例を次に示す。

補足：
プロセススケジューリングのタイミングによっては親子両方のプロセスがハングアップしてしまう可能性がある。

```
[motoki@x205a]$  nl ipc-by-signal-unstable.c
 1 /**************************************************/
 2 /* Operating-Systems/C-Programs/ipc-by-signal-unstable.c*/
 3 /*------------------------------------------------*/
 4 /* シグナルを使って親プロセスと子プロセス                 */
 5 /* の間でコミュニケーションする例                      */
 6 /*  --------                                      */
 7 /*  ダメな例：一見問題ないように見えるが、プロセススケ   */
```

```
 8 /*  --------  ジューリングのタイミングによっては pause()*/
 9 /*            中でないプロセスにシグナルを送信してしまい*/
10 /*            結果的に親子両方のプロセスが同時に pause()*/
11 /*            中になってしまう可能性もある。              */
12 /*********************************************************/
13 #include <stdio.h>
14 #include <stdlib.h>     /* for exit() library function */
15 #include <unistd.h>     /* for fork(), getppid() and pause(
16 #include <signal.h>     /* for signal() and kill() system c
17 #include <sys/types.h>  /* for kill(), getppid() and wait()
18 #include <sys/wait.h>   /* for wait() system call */

19 void brief_on_signal(int signum);

20 int main(void)
21 {
22   int k, status;
```

```
23   pid_t childpid, parentpid;
24
25   if (signal(SIGUSR1, brief_on_signal) == SIG_ERR) {
26     perror("signal");
27     exit(EXIT_FAILURE);
28   }

29   if ((childpid=fork())==-1) {
30     perror("can't fork");
31     exit(EXIT_FAILURE);
32   }else if (childpid==0) {  /*子プロセス*/
```

```
32    }else if (childpid==0) {  /*子プロセス*/
33      printf("(child) PID=%d\n", getpid());
34      parentpid=getppid();この時点で親からのシグナルが入ると...
35      for (k=0; k<6; k++) {
36        pause();
37        printf("(child) It's my %d-th turn to process.\n",
                  k);
38        sleep(3-k%3);   /* 子プロセスの処理の代わり */
39        if (kill(parentpid, SIGUSR1)==-1) {
40          perror("kill");
41          exit(EXIT_FAILURE);
42        }
43      }
44      exit(EXIT_SUCCESS);
45    }else {                          /*親プロセス*/
```

```
45    }else {                                  /*親プロセス*/
46      printf("<PARENT> PID=%d\n", getpid());
47      for (k=0; k<6; k++) {
48        printf("<PARENT> It's my %d-th turn to process.\n",
                  k);
49        sleep(k%2+1);    /* 親プロセスの処理の代わり */
50        if (kill(childpid, SIGUSR1)==-1) {
51          perror("kill");
52          exit(EXIT_FAILURE);
53        }                          この時点で子からのシグナルが入ると...
54        pause();
55      }
56      wait(&status);
57      exit(EXIT_SUCCESS);
58    }
59  }
```

```
60 void brief_on_signal(int signum)  シグナルハンドラ
61 {
62   printf("                Signal(#%d) --> Process(ID%d)\n",
63          signum, getpid());
64 }
[motoki@x205a]$  gcc ipc-by-signal-unstable.c
[motoki@x205a]$  ./a.out
(child) PID=17404
<PARENT> PID=17403
<PARENT> It's my 0-th turn to process.
              Signal(#10) --> Process(ID17404)
(child) It's my 0-th turn to process.
              Signal(#10) --> Process(ID17403)
<PARENT> It's my 1-th turn to process.
              Signal(#10) --> Process(ID17404)
(child) It's my 1-th turn to process.
              Signal(#10) --> Process(ID17403)
```

```
<PARENT> It's my 2-th turn to process.
                Signal(#10) --> Process(ID17404)
(child) It's my 2-th turn to process.
                Signal(#10) --> Process(ID17403)
<PARENT> It's my 3-th turn to process.
                Signal(#10) --> Process(ID17404)
(child) It's my 3-th turn to process.
                Signal(#10) --> Process(ID17403)
<PARENT> It's my 4-th turn to process.
                Signal(#10) --> Process(ID17404)
(child) It's my 4-th turn to process.
                Signal(#10) --> Process(ID17403)
<PARENT> It's my 5-th turn to process.
                Signal(#10) --> Process(ID17404)
(child) It's my 5-th turn to process.
                Signal(#10) --> Process(ID17403)
[motoki@x205a]$
```

例16. 3 (シグナルを使って親子のプロセス間で会話をする; 改良版)

例14.3のプログラムとほぼ同等のことをシグナルの機構を用いて行いたい。

しかし、
`pause()` と `kill()` を組み合わせて使うというのでは、例16.2と同じ失敗に陥りそう。

⇨
- 親プロセスと子プロセスの各々に
  実行を進めて良いかどうかのフラッグ　を設け、
- 各々のプロセスで自分の処理が終れば
  フラッグをFALSEにした上でシグナルを使って相手方にそれを知らせる、
- シグナルを受け取ったプロセスのシグナルハンドラは
  フラッグをTRUEに設定する、
- そして、pause() する代わりに
  フラッグが FALSE である間 sleep(1) を繰り返す、

ことにした。

⇨ この改良版を次に示す。

```
[motoki@x205a]$  nl ipc-by-signal.c
 1 /**************************************************************/
 2 /* Operating-Systems/C-Programs/ipc-by-signal.c               */
 3 /*------------------------------------------------------------*/
 4 /* シグナルを使って                                           */
 5 /* 親プロセスと子プロセスの間でコミュニケーションする例 */
 6 /*  ------                                                    */
 7 /*  改良版：親子両方のプロセスが同時にpause() 中になら  */
 8 /*  ------  ない様に、pause() の代わりに sleep() のルー */
 9 /*          プを用いた。                                      */
10 /**************************************************************/
11 #include <stdio.h>
12 #include <stdlib.h>     /* for exit() library function */
13 #include <unistd.h>     /* for fork(), getppid() and sleep(
14 #include <signal.h>     /* for signal() and kill() system c
```

```
15 #include <sys/types.h>  /* for kill(), getppid() and wait()
16 #include <sys/wait.h>   /* for wait() system call */

17 #define  TRUE  1
18 #define  FALSE 0
19 typedef int Boolean;

20 Boolean My_turn;

21 void flip_semaphore_and_brief_on_signal(int signum);

22 int main(void)
23 {
24   int k, status;
25   pid_t childpid, parentpid;
26
27   if (signal(SIGUSR1, flip_semaphore_and_brief_on_signal)
```

```
                                        == SIG_ERR) {
28      perror("signal");
29      exit(EXIT_FAILURE);
30    }

31    if ((childpid=fork())==-1) {
32      perror("can't fork");
33      exit(EXIT_FAILURE);
34    }

35    My_turn = FALSE;
36    if (childpid==0) {          /*子プロセス*/
```

```
36    if (childpid==0) {          /*子プロセス*/
37      printf("(child) PID=%d\n", getpid());
38      parentpid=getppid();
39      for (k=0; k<6; k++) {
40        while (My_turn == FALSE)  ビジーウェイト
41          sleep(1);
42        printf("(child) It's my %d-th turn to process.\n",
                k);
43        sleep(3-k%3);   /* 子プロセスの処理の代わり */
44        My_turn = FALSE;
45        if (kill(parentpid, SIGUSR1)==-1) {
46          perror("kill");
47          exit(EXIT_FAILURE);
48        }
49      }
50      exit(EXIT_SUCCESS);
51    }else {                       /*親プロセス*/
```

```
51    }else {                              /*親プロセス*/
52      My_turn = TRUE;
53      printf("<PARENT> PID=%d\n", getpid());
54      for (k=0; k<6; k++) {
55        printf("<PARENT> It's my %d-th turn to process.\n",
                  k);
56        sleep(k%2+1);    /* 親プロセスの処理の代わり */
57        My_turn = FALSE;
58        if (kill(childpid, SIGUSR1)==-1) {
59          perror("kill");
60          exit(EXIT_FAILURE);
61        }
62        while (My_turn == FALSE)  ビジーウェイト
63          sleep(1);
64      }
65      wait(&status);
66      exit(EXIT_SUCCESS);
```

```
67    }
68 }

69 void flip_semaphore_and_brief_on_signal(int signum)
70 {
71    My_turn = TRUE;
72    printf("                Signal(#%d) --> Process(ID%d)\n",
73           signum, getpid());
74 }
[motoki@x205a]$  gcc ipc-by-signal.c
[motoki@x205a]$  ./a.out
(child) PID=17419
<PARENT> PID=17418
<PARENT> It's my 0-th turn to process.
              Signal(#10) --> Process(ID17419)
(child) It's my 0-th turn to process.
              Signal(#10) --> Process(ID17418)
```

```
<PARENT> It's my 1-th turn to process.
                Signal(#10) --> Process(ID17419)
(child) It's my 1-th turn to process.
                Signal(#10) --> Process(ID17418)

   ... (途中省略) ...

<PARENT> It's my 5-th turn to process.
                Signal(#10) --> Process(ID17419)
(child) It's my 5-th turn to process.
                Signal(#10) --> Process(ID17418)
[motoki@x205a]$
```