

# システムプログラミング

## シグナル、入出力

芝浦工業大学 情報工学科  
菅谷みどり



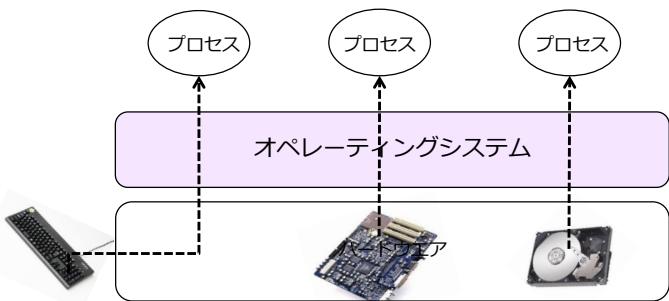
## 授業計画

- 9/27 イントロダクション, 歴史, シェルスクリプト
- 10/4 システム管理, 開発環境プログラムのコンパイル, リンク
- 10/11 ファイル（高水準なファイル操作）入出力ハードウェアと制御, ファイルシステム, ファイル構造
- 10/18 プロセス
- 10/25 プロセス2
- 11/8 プロセス3
- 11/15 スレッド, デッドロック1, 研究紹介1
- 11/29 研究紹介2, スレッド, デッドロック
- 12/6 シグナル, 入出力**
- 12/13 ネットワークプログラミングI クライアントプログラム
- 12/20 ネットワークプログラミングII サーバプログラム
- 1/10 グループワーク（中間報告）
- 1/17 グループワーク
- 1/24 発表, 提出

システムプログラミング

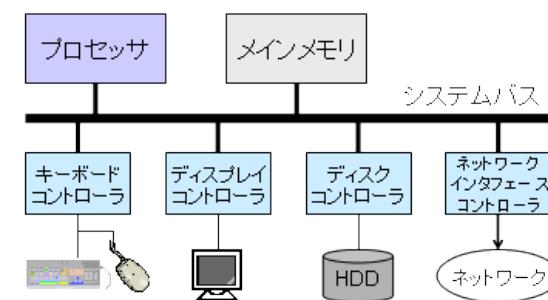
## シグナル

- プロセス
  - OSがコンピュータを抽象化し、使いやすくしたもの
- 入出力装置からのイベント通知
  - 割り込みにより受け取る
- シグナル
  - イベント通知のメカニズム（ソフトウェア割り込み）



## 入出力機器

- 割り込み
  - 入出力機器からのイベント通知の仕組み
- 入出力機器
  - PCの場合、コンピュータにI/Oコントローラを通じて接続されている



2

4

## 入出力(I/O)へのアクセスのタイミング

- タイミングの決め方
  - I/O から情報を受け取るためには、何らかの方法で受け取るタイミングを決める必要がある
- 日常生活にある話として、
  - あなたがダイニングテーブルで勉強をしている。やかんに水を入れて、火にかけて、お湯を沸かしている。お湯が沸いたら、コーヒーブレークを取るつもりでいる。



## 割り込み

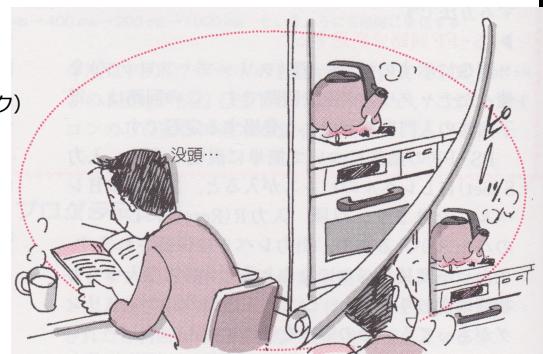
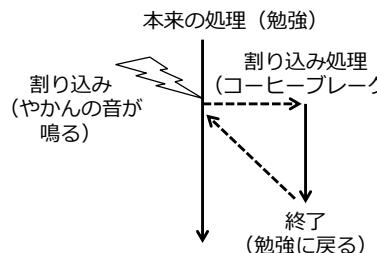
普通のやかん → 気になって、何度もやかんを見る



これに対して、音のなるやかんを使うと

音のなるやかん → 音がなるまで勉強に集中できる

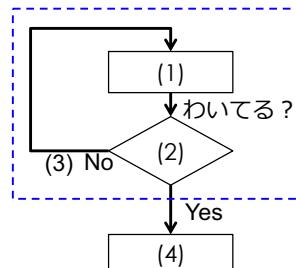
→ 割り込み



## ポーリング

普通のやかん → 気になって、何度もやかんを見る

→ ポーリング

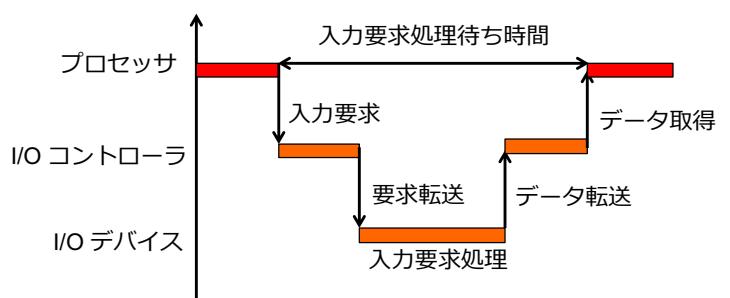


- (1) 読書をやめてガスレンジの前にゆく
- (2) お湯の状態から、沸いているかを確認する
- (3) 沸いていない場合は、(1) へもどる
- (4) 沸いていたら、コーヒーを入れる

気になって何度も見る期間



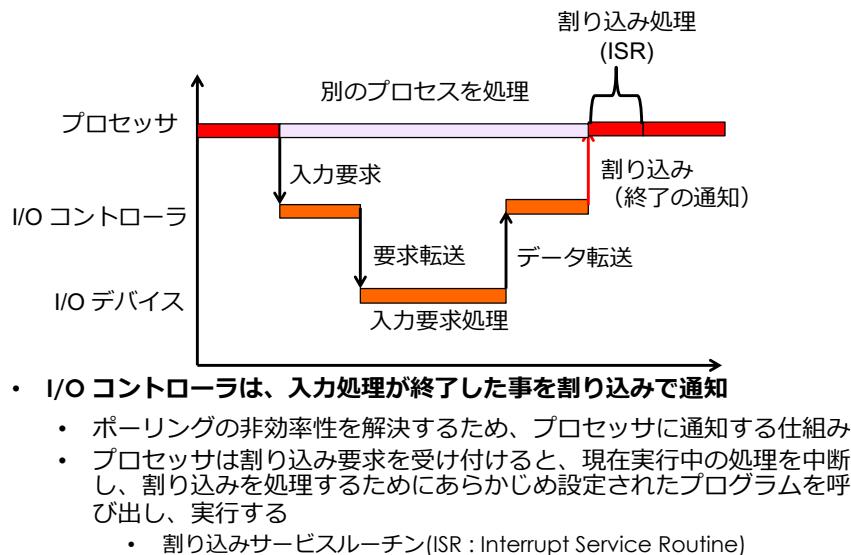
## ポーリング処理による I/O 制御



- プロセッサは、I/O コントローラの処理の終了をチェックし続ける

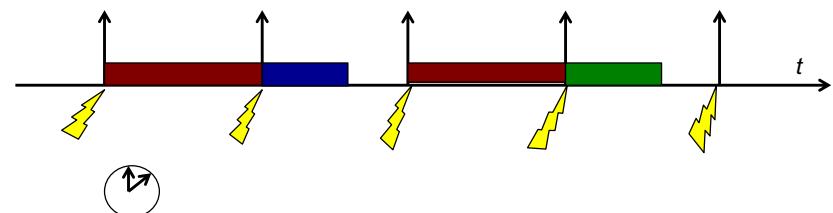
- I/O デバイスはプロセッサの処理速度と比較すると非常に遅い
- プロセッサの使用効率を著しく低下させる

## 割り込み処理による I/O 制御



## 時間依存の処理

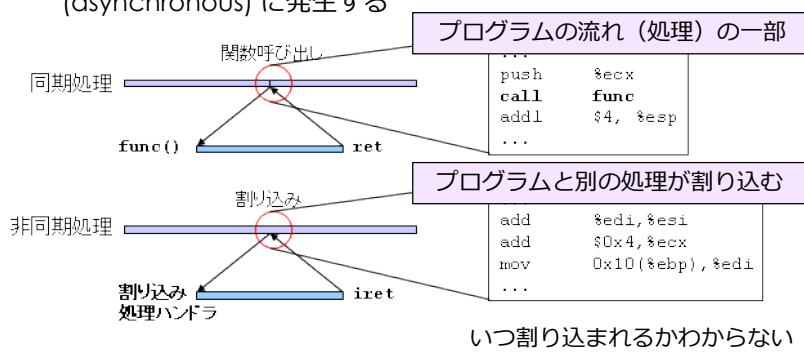
- 割り込みにより、時間依存の処理が可能になる
  - タイマデバイスから一定周期（例えば10ms）で、割り込み
  - このタイミングにあわせて
    - 実行プロセスの切り替え
    - 各種統計処理（定期的に行う処理）
    - バッファのディスクへのフラッシュ



10

## 同期処理と非同期処理

- 関数呼び出しは同期処理
  - 明示的に関数を呼び出し、呼び出しされた関数で処理が行われ、関数呼び出しから帰ってくる
- 割り込みは非同期処理
  - 本来のプログラムの流れとは無関係に、非同期的 (asynchronous) に発生する



## 例外

- 例外 (Exception)
  - プログラムの実行中に発生する、そのプログラムに起因するエラー
    - 例
    - アクセスが許可されていない番地にアクセスしようとした
    - 0による除算を行う
    - 不正な命令（例えば特権が必要な命令）を実行しようとした
- 例外処理
  - 例外発生時に、OS カーネルの例外ハンドラが起動され、処理を行う
- 割り込み処理との比較
  - 類似点：
    - 発生時に OS カーネルのハンドラが起動される、ハンドラを呼び出す命令はプログラム中には含まれていない点
  - 相違点：
    - 割り込みの場合は、デバイスなどのプログラム外の要因、例外はプログラム自体の要因

12

## シグナル

- 割り込みと例外
  - OS のみが取り扱う
- ユーザプロセスが割り込みなどを使いたい場合は?
  - 例えば、現在の処理を継続しながら別の処理を行いたいケース
    - 一定周期の割り込みを使った、プロンプトの点滅アニメーションの実行、ポーリング
- シグナル - 割り込みを抽象化したもの
  - 割り込みハンドラ → シグナルハンドラ
    - プロセスはそれぞれのシグナルの種類に対し、シグナルハンドラをOSに登録
  - プロセス
    - 実行中のプログラムに起因する例外や、プロセス外部からのイベントをシグナルとして受け取る
    - シグナル発生時には、OS カーネルに登録したシグナルハンドラが起動される

13



## シグナルの種類と動作

- POSIX.1-1990 に定義されているシグナル (man 7 signal)

シグナル	値	動作	コメント
SIGHUP	1	Term	制御端末(controlling terminal)のハングアップ検出、または制御しているプロセスの死
SIGINT	2	Term	キーボードからの割り込み (Interrupt)
SIGQUIT	3	Core	キーボードによる中止 (Quit)
SIGILL	4	Core	不正な命令
SIGABRT	6	Core	abort(3) からの中断 (Abort) シグナル
SIGFPE	8	Core	浮動小数点例外
SIGKILL	9	Term	Kill シグナル
SIGSEGV	11	Core	不正なメモリ参照
SIGPIPE	13	Term	パイプ破壊: 読み手の無いパイプへの書き出し
SIGALRM	14	Term	alarm(2) からのタイマーシグナル
SIGTERM	15	Term	終了 (termination) シグナル

シグナル SIGKILL と SIGSTOP はキャッチ、ブロック、無視できない。

14



## シグナル処理方法

- シグナルはそれぞれ現在の「処理方法 (disposition)」を保持
  - この処理方法によりシグナルが配送された際にプロセスがどのような振舞いをするかが決まる。
- “動作” の欄のエントリ
  - 各シグナルのデフォルトの処理方法を示しており、以下のような意味を持つ。

Term デフォルトの動作はプロセス終了。

Ign デフォルトの動作はこのシグナルの無視。

Core デフォルトの動作はプロセス終了とコアダンプ出力 (core(5) 参照)。

Stop デフォルトの動作はプロセスの一時停止。

Cont デフォルトの動作は、プロセスが停止中の場合にその実行の再開。

15



## Signal システムコール

- Signal システムコール
  - シグナルをサポートするために最初に作られたシステムコール

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t sighandler);
```

- Signal()
  - シグナルハンドラを変更するシグナル番号
  - 新しいシグナルハンドラを引数に指定して呼び出す
  - 古いシグナルハンドラが返される
  - 引数と、戻り値の型は sighandler\_t
    - int 型を引数に取り、値を返さない (void型の) 関数へのポインタ (アドレス)
- シグナルハンドラは、プロトタイプ宣言される関数
  - void sighandler (int)
    - この関数へのポインタが、signal システムコールの引数や戻り値になる。ここで引数は、送られてきたシグナルの番号。

16



## Signal システムコール(cond't)

- 古いシグナルハンドラの値
  - SIG\_IGN シグナルの無視
  - SIG\_DFL でフォルトの動作
- pause()**
  - シグナルと一緒に利用されるシステムコール

```
int pause(void);
```

- pause を呼び出すと、シグナルを受け取るまで、プロセスの実行をブロックする
- シグナルを受け取ると、シグナルハンドラの実行後に pause から戻る

17

## 解説と実行結果

- 解説
  - 4-11行目 :シグナルハンドラの定義、3回起動すると、exit
  - 18行目: SIGINT 対応するシグナルハンドラとして sigint\_handler を設定
  - 22行目: pause でシグナルを受け取るまでブロック
    - Ctrl-C をたたくたびに sigint\_handler が起動し、3回で終了

```
[doly@yli008 08_Samples]$ ./sigint
main: sigint_count(3), calling pause...
^C
sigint_handler(2): sigint_count(3)
main: return from pause. sigint_count(2)
main: sigint_count(2), calling pause...
^C
sigint_handler(2): sigint_count(2)
main: return from pause. sigint_count(1)
main: sigint_count(1), calling pause...
^C
sigint_handler(2): sigint_count(1)
sigint_handler: exiting..
```

19

## Signal プログラムを書いてみよう

```
1 #include <stdio.h>
2 #include <signal.h>
3 #include <stdlib.h>
4 int sigint_count = 3;
5 void
6 sigint_handler(int signum)
{
7     printf("sigint_handler(%d): sigint_count(%d)%n", signum,
8         sigint_count);
9     if (--sigint_count <= 0) {
10         printf("sigint_handler:
11             exiting ... %n");
12         exit(1);
13     }
14 #if 0 /* For original System V signal */
15 signal(SIGINT, &sigint_handler);
16 #endif
17 }
18 signal(SIGINT, &sigint_handler);
19
20 for (;;) {
21     printf("main: sigint_count(%d),
22         calling pause ...%n", sigint_count);
23     pause();
24     printf("main: return from pause.
25         sigint_count(%d)%n",
26         sigint_count);
27 }
```

シグナルハンドラの定義

18

## System V 当初の仕様の問題点と対策

- システムコールを呼び出し、ブロックしている時の動作
  - シグナルが呼ばれると、システムコールがキャンセルされる
    - Read() でブロック => システムコールが無かったことに(TT)
  - 対応 (BSD UNIX) :
    - システムコールをキャンセルしないよう、シグナルハンドラ実行後にブロック状態に戻るようにした
- シグナルハンドラのマスクとリセット
  - シグナルハンドラの実行中に同じシグナルが通知されると、そのシグナルはリセットされてしまう
    - その都度設定しなくてはならなかった
  - 対応(BSD UNIX) :
    - 同じシグナルが通知されたら、そのシグナルは保留される。現在実行中のシグナルハンドラ処理の終了を待ち、もう一度、シグナルハンドラを起動する。リセットされない
- Linux は、当初は SystemV の API を採用、現在は BSD UNIX
  - そのため、12-14行目は不要

20

## Sigaction システムコール

- Signal システムコールの混乱の解決

- POSIX では、sigaction という新しいシステムコールを導入
- POSIX 準拠であれば、sigaction を使用する必要がある

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- Sigaction システムコール

- 第1引数: シグナル番号
- 第2引数: 設定するシグナルの動作
- 第3引数: 古い設定が返される

- 問題点への対応

- ブロック時の動作: キャンセルされてしまう
- ハンドラのマスクとリセット: シグナルは保留される

21

## sigactionプログラムを書いてみよう

```
1 #include <stdio.h>
2 #include <signal.h>
3 int sigint_count = 3;
4 void
5 sigint_handler(int signum)
6 {
7     printf("sigint_handler(%d): sigint_count(%d)\n", signum, sigint_count);
8     if (--sigint_count <= 0) {
9         printf("sigint_handler: exiting ... \n");
10    exit(1);
11 }
12 #if 0 /* For original System V signal */
13 signal(SIGINT, &sigint_handler);
14 #endif
15 }
```

シグナルハンドラの定義

```
16 main()
17 {
18     struct sigaction sa_sigint;
19     memset(&sa_sigint, 0, sizeof(sa_sigint));
20     sa_sigint.sa_handler = sigint_handler;
21     sa_sigint.sa_flags = SA_RESTART;
22     if (sigaction(SIGINT, &sa_sigint, NULL) {
23         perror("sigaction");
24         exit(1);
25     }
26     for (;;) {
27         printf("main: sigint_count(%d), calling pause ... \n", sigint_count);
28         pause();
29         printf("main: return from pause. sigint_count(%d)\n", sigint_count);
30     }
31 }
32 
```

23

## シグナルの動作

- シグナルの動作の定義

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

- 各メンバ値の意味

- sa\_handler: シグナルハンドラ
- sa\_sigaction: より詳しいシグナル情報を引数に受け取ることができるシグナルハンドラ、sa\_flags にSA\_SIGINFO を加えることで、sa\_handler の代わりに使用できるようになる
- sa\_mask : シグナルハンドラ起動時に、対応するシグナルの他にマスクしたいシグナル
- sa\_flags ; 動作の詳細を設定するフラグ
- sa\_restorer: 使用不可

22

## sigactionプログラムを書いてみよう (2)

```
1 #include <stdio.h>
2 #include <signal.h>
3 int sigint_count = 3;
4 void
5 sigint_action(int signum, siginfo_t *info, void *ctx)
6 {
7     printf("sigint_handler(%d): sigint_count(%d)\n", signum, sigint_count);
8     if (--sigint_count <= 0) {
9         printf("sigint_handler: exiting ... \n");
10    exit(1);
11 }
12 #if 0 /* For original System V signal */
13 signal(SIGINT, &sigint_handler);
14 #endif
15 }
```

シグナルハンドラの定義

```
16 main()
17 {
18     struct sigaction sa_sigint;
19     memset(&sa_sigint, 0, sizeof(sa_sigint));
20     sa_sigint.sa_sigaction = sigint_action;
21     sa_sigint.sa_flags = SA_RESTART | SA_SIGINFO;
22
23     if (sigaction(SIGINT, &sa_sigint, NULL) {
24         perror("sigaction");
25         exit(1);
26     }
27     for (;;) {
28         printf("main: sigint_count(%d), calling pause ... \n", sigint_count);
29         pause();
30         printf("main: return from pause. sigint_count(%d)\n", sigint_count);
31     }
32 }
33 
```

24

## シグナルの無視

```
1 #include <stdio.h>
2 #include <signal.h>
3
4 int sigint_count = 3;
5
6 main()
7 {
8     struct sigaction sa_ignore;
9
10    memset(&sa_ignore, 0,
11           sizeof(sa_ignore));
12    sa_ignore.sa_handler = SIG_IGN;
13
14    if (sigaction(SIGINT, &sa_ignore,
15                  NULL) < 0) {
16        perror("sigaction");
17        exit(1);
18    }
19}
```

```
18 while (1) {
19     printf("main: sigint_count(%d), calling
20           pause() .....%n", sigint_count );
21     pause();
22
23     printf("main: return from pause().
24           sigint_count(%d)%n", sigint_count );
25 }
```

/usr/include/bits/signum.h

/\* Fake signal functions. \*/
#define SIG\_ERR ((\_\_sighandler\_t)-1) /\* Error return. \*/
#define SIG\_DFL ((\_\_sighandler\_t)0) /\* Default action. \*/
#define SIG\_IGN ((\_\_sighandler\_t)1) /\* Ignore signal. \*/

SIG\_IGN の値は 1、デフォルトは 0
Sigaction は古い設定を返すが、signal を無視、またはデ
フォルト設定の場合は、これらの値が sa\_handler に入る

25



## kill

- Kill システムコール

- プロセスにシグナルを送る (man kill で確認)

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

26



## Kill プログラムを書いてみよう

```
1 #include <stdio.h>
2 #include <signal.h>
3 #include <sys/types.h>
4
5 main(int argc, char *argv[])
6 {
7     pid_t pid;
8
9     if (argc != 2) {
10         printf("Usage: %s pid%n", argv[0]);
11         exit(1);
12     }
13     pid = atoi(argv[1]);
14     if (pid <= 0) {
15         printf("Invalid pid: %d%n", pid);
16         exit(1);
17     }
18     if (kill(pid, SIGINT) < 0) {
19         perror("kill");
20         exit(1);
21     }
22 }
```

27



## インターバルタイマ

- Setitimer : 一定周期で割り込みを発生する

- 割り込みのタイミングで
  - 現在の処理を継続しながら別の処理を行う

```
int setitimer(int which, const struct itimerval *value, struct itimerval
*ovalue);
```

- 使い方

- 第2引数で指定した時間がたつと、シグナルで通知。時間の内容は、第1引数で指定
  - 第1引数:
    - ITIMER\_REAL : 指定した時間が経過し 0 になったら SIGALRM を通知
    - ITIMER\_VIRTUAL : プロセスがユーザプログラムを実行している時間だ  
け減少し、残り時間が 0 になったら SIGVALARM が通知される
    - ITIMER\_PROF: カーネル内も含めて、プロセスが実行している時間だ  
け減少し、残り時間が 0 になったら SIGPROF が通知される
  - 第2引数 : 時間を指定するために使用する struct timeval の定義

28

# インターバルタイマ

- **struct\_itimerspec**

```
struct itimerspec {
    struct timespec it_interval; /* next value */
    struct timespec it_value; /* current value */
};

struct timespec {
    long tv_sec; /* seconds */
    long tv_nsec; /* microseconds */
};
```

- **動作**

- `it_value` が 0 になると、`which` で指定されるタイマに対応するシグナルが通知される
- `it_interval` に値が次の時間として設定される。
- `it_value, it_interval` どちらの値も 0 になったタイマは停止する

- **プログラムを書いてみよう**

- ITIMER\_REAL で1秒ごとにシグナルを受け、10回シグナルを受け取ると終了する

29

## 演習

- **演習1**

- 3秒おきに、アラームを表示させてみる。表示させた結果（テキスト）とプログラムを保存

- **演習2**

- プログレスバーを作ってみよう
  - 何秒（何ms）おきに適当な記号を表示
    - E.g. #####, \*\*\*\* etc..
  - 表示がうまくいかない → バッファの操作のヒント
    - `setbuf(stdout, NULL)` を `main` に追加する

- **演習3**

- 3秒おきに、異なる顔文字を表示させてみよう
  - 異なる顔文字の表示のさせ方のヒント
    - ランダム : `rand()`, `randl()`

- **演習4**

- キーボードから1文字入力を読み込む関数 `getchar` にタイムアウト機能を追加した関数 `mygetchar` を作成する
  - `mygetchar` は、ある一定時間内にキー入力があればそれを返し、なければ -2 を返す関数とする (-2 なのは EOF が -1 のため)
  - タイムアウト時間は、ユーザ入力、引数、固定のいずれでも良いので、ユーザが利用しやすい方法を考えてみること

- `man` や `web` を参照しながら書いてみよう

31



## インターバルタイマのプログラムを書いてみよう

```
1 #include <stdio.h>
2 #include <signal.h>
3 #include <time.h>
4 #include <sys/time.h>
5 #include <string.h>
6 #include <stdlib.h>
7 void alarm()
8 {
9     printf("alarm: %ld\n", time(NULL));
10}
11
12 main()
13{
14    struct sigaction sa_alarm;
15    struct itimerspec itimer;
16    int counter = 10;
17    memset(&sa_alarm, 0, sizeof(sa_alarm));
18    sa_alarm.sa_handler = alarm;
19    sa_alarm.sa_flags = SA_RESTART;
20
21    if (sigaction(SIGALRM, &sa_alarm, NULL) < 0) {
22        perror("sigaction");
23        exit(1);
24    }
25
26    itimer.it_value.tv_sec = 1;
27    itimer.it_value.tv_nsec = 0;
28
29    if (setitimer(ITIMER_REAL, &itimer, NULL) < 0) {
30        perror("setitimer");
31        exit(1);
32    }
33
34    while (counter--) {
35        pause();
36        printf("main: %d\n", time(NULL));
37    }
38 }
```

30



## 演習提出

- **README**

- 1から4までのプログラム、説明、実行結果および考察を含む README を入れること

- **演習の提出〆切**

- Exec06 フォルダに、学籍番号\_名前（ローマ字）で提出

32

