Linux システムコール演習 3

岡野浩三

令和 2 年 6 月 2 日

ここではシグナルと Long jump による例外処理を扱う．通常のプログラムの授業では扱わない OS 特有の話題である．

1 シグナル

UNIX ではプロセス間通信としてシグナルを使うことができる．CPU における割込み信号と割り込みハンドラによく似ているが，CPU のレベルの割り込みとは基本的には異なり，OS レベルで OS の機能として実装されているものである．

簡単なプロセス通信の処理や Terminal(キーボード) からの

Ctl+Cに (よって発生する割り込みシグナル SIGINT) よる割り込み処理のプログラムとして使われる．

歴史的には シグナルの処理には Signal システムコールが長らく用いられてきたが，種々の理由により，現在ではこのシステムコールを使うことは推奨されない．現在では代わりに sigaction システムコールが使われる．

sigaction システムコールはシグナルの発生時の処理をより明確に定義しなおした反面，今までのシステムコールとの互換性を取るため，種々の細かな振る舞いの違いを吸収できるように，引数のデータ型などにおいて若干複雑になってしまった．通常の単純な使い方をする分には sigaction システムコールは複雑すぎるため，以下のようなラッパー関数を用意しておいて，このラッパー関数を呼び出したいところで使うことが多い．

```
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;
}
```

ここで *sighandler_t という型を「int 型を引数にとり void 型を返す関数ポインタ」として定義していることに注意．ラッパー関数として定義された trap_signal は sigaction システムコール以前に標準として使われていた signal システムコールと似たような引数をとる．簡単には第一引数にシグナル番号，第二引数に対応するシグナルハンドラーを取る．

このラッパー関数を用いて，以前，シェルスクリプトとして実現した「基本的にループ番号を表示しながら，割り込み処理ごと異なる文字列を表示する」プログラムを C 版で記述してみると以下のようになる．

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#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;
}

void
signalHandlerForInt(int sig) {
  puts("I'm interrupted.\n");
}

#define TRUE 1
```

```
int main(int argc, char *argv[]) {
  trap_signal(SIGINT, signalHandlerForInt);

  int i=0;
  while(TRUE) {
    sleep(1);
    fprintf(stdout, "%d-th loop\n", i++);
  }
  exit(0);
}
```

最初にラッパー関数 trap_signal() を用いて SIGINT(すなわち Ctl+C) に対して関数 signalHandlerForInt をそのシグナルハンドラーとして定義している．ちょうどシェルスクリプトにおける trap コマンドと同じ処理をしている．

演習 1
ここまでを実行せよ．

同様に以前，シェルスクリプトとして実現した「基本的にループ番号を表示しながら，割り込み処理で異なる文字列を表示するが，2 回目以降の割り込み処理では終了する」プログラムを C 版で記述してみると以下のようになる．以前の割り込みハンドラの値を覚えておくため大域変数として old を用いている．

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#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;
```

```
}

sighandler_t old;

void
signalHandlerForInt(int sig) {
  puts("I'm interrupted.\n");
  if (old == signalHandlerForInt)
    exit(0);
  old = trap_signal(SIGINT, signalHandlerForInt);
}

#define TRUE 1


int
main(int argc, char *argv[]) {

  old = trap_signal(SIGINT, signalHandlerForInt);
  int i=0;
  while(TRUE) {
    sleep(1);
    fprintf(stdout, "%d-th loop\n", i++);
  }
  exit(0);
}
```

演習 2
ここまでを実行せよ.

問 1
なぜこのプログラムで上述のように振舞うのかを説明せよ.

2 Alarm

定期的な処理をするには alarm システムコールが有用である. alarm で時間を秒で指定するとその時間後に SIGALRM のシグナルが発生する. 例えば 5 秒おきに所定の処理をしたければ以下のようなプログラムを記述すればよい.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#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;
}

void
signalHandlerForAlarm(int sig) {
  alarm(5);
  puts("I catch a signal.");
  if (sig == SIGALRM)
    puts("5 sec. elapsed");
}

#define TRUE 1


int
main(int argc, char *argv[]) {

  trap_signal(SIGALRM, signalHandlerForAlarm);
  alarm(5);
  int i=0;
  while(TRUE) {
    sleep(1);
    fprintf(stdout, "%d-th loop\n", i++);
```

```
  }
  exit(0);
}
```

演習 3
ここまでを実行せよ.

演習 4
定期処理を用いた簡単な自作プログラムを作ってみよ．例えば時計とか連続画像再生 *(*表示などは外部プログラムをプロセス呼び出しするなどで対応してみよ*)* とか.

問 2
signalHandlerForAlarm() の本体の *1* 行目 *alarm(5);* の役割を説明せよ.

3 Long Jump

通常 goto 文は「スパゲティープログラム」の元凶として，とりわけ構造的プログラムの立場では忌避される．ただ，実際のプログラムをする際に,「適切に用いる」のであれば有効な状況もないわけではない.

上記の観点より C 言語では break, continue に並んで goto 文がサポートされている.「適切に用いる」の具体例としては，正常処理，異常処理にかかわらず，共通にしなければならない，最終処理群があるとして，それを共通にまとめて記述しておき，そこに goto 文で飛ぶ，という使い方である．以下に例を記す.

```
int main(int argc, char *argv[]) {
  FILE *outf;
  outf = fopen(...)
  normal_proc0();

  if (something bad) {
    Something_do();
    goto final;
  }
  normal_proc1();
    :
  goto final;

final:
  close (outf)
  exit(0);
```

```
}
```

この例では異常処理，正常処理にかかわらず，最後に outf をクローズするという意図を明確にしている．また，この最終処理に変更があった際に，変更箇所が 1 つですむのでバグを作りにくくなるという利点が生じる．

ただし，このような goto 文は同じ関数内でしか有効でない．関数を超えた goto をこのような異常処理 (例外処理) の記述に応用したいという要求も当然現れる．それをある程度術実現するのが long jump である．Long jump をするには前処理として set jump が必要である．

Long jump の記述例が以下である．

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf jmpbuf;

void sub(void) {
  printf("sub start\n");
  printf("do something normal\n");
  longjmp(jmpbuf, 1); // some error
  printf("sub end\n");
}

int main(int argc, char *argv[]) {
  printf("main start\n");

  int c;
  if ((c = setjmp(jmpbuf)) == 0) {
  // Normal routine
    sub();
  } else {
    // something bad happens
    printf("Error happens. no: %d\n", c);
  }
  printf("main end\n");
}
```

このプログラムでは main から sub を呼び出し，特に異常がなければ sub の実行終了で終わるプログラムの流れを想定している．ただし，このプログラムでは sub の呼び出し中

```
printf("do somethin normal\n");
```

の実行後，エラー状況になることを想定している．エラー状況になったとしてその際に呼ぶべき

処理として `longjmp(jmpbuf, 1);`
が記述されている．このプログラムでは無条件にエラーとなったとして longjmp(jmpbuf, 1);+
が実行され，このため sub の実行はここで中断し，呼び出しもとの main に戻り，さらに else 節に実行制御が移る．

この else 節は `if ((c = setjmp(jmpbuf)) == 0) {`
に対応するものであった．setjmp(jmpbuf) は成功すれば jmpbuf に現在の実行コンテキストを保存して 0 を返す．またこの処理は通常すぐ終了する．すなわち，通常は then 節が実行され，その結果として sub が呼びだされる．

sub 内で longjump が呼びされれなければ，こでで終了であり最後の `printf("main end\n")`
が実行されて，終了となる．一方，sub 内で longjmp が呼び出されると，上述のようにここで sub の実行は中断され，あたかも `if ((c = setjmp(jmpbuf)) == 0) {`
で= setjmp(jmpbuf)) が 1 を返したかのように振る舞い (この 1 は loginjump で指定した値である)，結果として else 節が実行されることになる．

演習 5
longjump の行をコメントアウトした場合とそれ以外の場合，両方とも実行し振る舞いの違いを観察せよ．

問 3
観察結果を考察しまとめよ

問 4
jmpbuf に保存される現在の実行コンテキストとは具体的に何か調べよ．

この setjump, longujump は上記のような限られた使い方をする分には有用であり，実際に多くのプログラムで用いられている．ただし，これを超えた使い方をすると非常にそのプログラムの解析が困難になる．

なお上述のプログラムの構文を少し変えると Java における，throws 文と try, catch 節の関係となる．
longjump → throws
setjump → try 節
else 節 → catch 節

先ほどのプログラムを Java 風に書き換えると次のようになる．

```
void sub() {
  System.out.println("sub start");
  System.out.println ("do something normal");
  throws new MyException() ; // some error
  System.out.println ("sub end");
}
```

```
public static void main(String argv) {
  System.out.println ("main start");
  try{
    // Normal routine
    sub();
  } catch(MyException e) {
    // something bad happen
    System.out.println ("Error happens." + e);
  }
  System.out.println ("main end");
}
```

Java では C における setjump, longujump をこのような構文規則に置き換えることにより，その利点を生かしたまま，無規範な記述によるバグの押さえ込みを行っている．

参考文献

[1] 青木峰郎:ふつうの Linux プログラミング Linux の仕組みから学べる gcc プログラミングの王道, ソフトバンククリエイティブ, ISBN-13: 978-4797328356, 2005.