情報科学演習C 課題3

文責： 内山彰

2017年5月29日 (月)

1 目的

マルチプロセスプログラムを作成することにより，マルチプロセス処理の基本を学ぶ．

最初に，セマフォを用いた複数のプロセス間における同期処理について学習し，次に，パイプを用いたプロセス間通信について学習する．最後に，割り込みを用いたプロセス制御について学習する．

2 プロセスの生成・終了

プログラミングCで学習済みのため，詳細な解説は省略し，ここでは簡単なおさらいに留める．

新しいプロセスの生成・終了に用いる基本的なシステムコールは以下の3つである．

`pid_t fork()`：プロセスの生成

`pid_t wait(int *status)`：子プロセスの終了の待ち合わせ

`void exit(int status)`：プロセスの終了

`fork()` は正常終了すると，子プロセスには0を返し，親プロセスには子プロセスのIDを返す．エラーの場合には子プロセスを生成せず，親プロセスには-1を返す．生成された子プロセスは，親プロセスの情報（プログラムコードやデータ，属性）を引き継ぐ．

子プロセスを生成した親プロセスは，`wait()` で子プロセスの終了（または停止）を待つことが出来る．子プロセスが停止あるいは終了した場合，`wait()` は子プロセスのプロセスIDを返す．そのとき，子プロセスのステータス情報が引数の指す領域に格納される．

`exit()` は引数にプロセスの終了ステータス情報statusを入れて呼び出す．この情報は，`wait()` で待機している親プロセスに渡される．UNIXでは慣習的に正常終了の場合には0，なんらかの異常が起こった場合には0以外の値を入れることになっている．

2.1 サンプルプログラム — fork.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define NUMPROCS 3

void main()
{
  int i, pid, status, c_pid;

  setbuf(stdout, NULL); /* set stdout to be unbufferd */
  for (i=0; i<NUMPROCS; i++) {
    if ((pid=fork())== -1) {
      perror("fork failed.");
      exit(1);
    }
    if (pid == 0) { /* Child process */
      sleep(1);
      printf("Child process i=%d\n", i);
      exit(0);
    }
  }
  /* Parent process */
  for (i=0; i<NUMPROCS; i++) {
    c_pid = wait(&status);
    printf("Parent process: child (%d)\n", c_pid);
  }
}
```

fork() は分身を作るシステムコールなので，親プロセスと子プロセスは全く同じ動作をする．そのままでは意味がないため，fork() の返り値（ここでは pid）によって場合分けをし，親プロセスと子プロセスに異なる処理を実行させる．

仮に fork() せずに単一プロセスで同様のプログラムを書いた場合，1 秒ごとに 1 行ずつ出力されることになる．一方，マルチプロセス処理では全てのプロセスは並行して同時に処理されるため，1 秒後に一斉に結果が出力される．実行して確認すると良い．

なお，setbuf(stdout, NULL); は標準出力へのバッファリングを止めるために実行している．通常，標準出力はバッファリングされているため，printf() を呼び出してもすぐに出力されない．出力のタイミングを正確に把握するためには，バッファリングを止める必要がある．

2.2 マルチプロセス処理における問題 — 競合状態

競合状態（race condition）とは，複数のプロセス[1]が共有データに対して処理を行う場合に，プロセスの処理順によって結果が異なったり，同時に共有データにアクセスすることで予期せぬ結果が起きる状況である．

2.2.1 サンプルプログラム — file-counter.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include <sys/wait.h>
#define NUMPROCS 4

int count1(ct)
FILE *ct;
{
  int count;

  if ((ct=fopen("counter", "r"))==NULL) exit(1);
  fscanf(ct, "%d\n", &count);
  count++;
  fclose(ct);
  if ((ct=fopen("counter", "w"))==NULL) exit(1);
  fprintf(ct, "%d\n", count);
  fclose(ct);
  return count;
}

int main()
{

  int i, count, pid, status;
  FILE *ct;

  setbuf(stdout, NULL); /* set stdout to be unbufferd */

  count = 0;
  if ((ct=fopen("counter", "w"))==NULL) exit(1);
```

[1]マルチプロセスに限らず，並列/並行処理において頻繁に起き得る．

```
  fprintf(ct, "%d\n", count);
  fclose(ct);
  for (i=0; i<NUMPROCS; i++) {
    if ((pid=fork())== -1) {
      perror("fork failed.");
      exit(1);
    }
    if (pid == 0) { /* Child process */
      count = count1(ct);
      printf("count = %d\n", count);
      exit(0);
    }
  }
  for (i=0; i<NUMPROCS; i++) {
    wait(&status);
  }
}
```

このプログラムは，ある 1 つのファイル（共有データ）の内容に子プロセスが 1 を加算していくプログラムである．4 つの子プロセスを用いて最終結果が 4 になることを期待しているが，正しく動作しない．何度も実行を繰り返すと，実行するごとに結果が変わったり，あるいは突然不正な値になることがある（実際に動作を確認すると良い）．

この理由は，あるプロセスがファイルの内容を読み出してから書き出すまでの間[2]に，異なるプロセスがファイルの内容を書き換えてしまうためである．

例えば，簡単のためにプロセスが 2 つの場合を考えて，

1. プロセス A がファイルから読み込む（count=0）
2. プロセス A が 1 を加算してファイルへ書き込む（count=1）
3. プロセス B がファイルから読み込む（count=1）
4. プロセス B が 1 を加算してファイルへ書き込む（count=2）

このような処理順であれば正しく動作する．一方で，

1. プロセス A がファイルから読み込む（count=0）
2. プロセス B がファイルから読み込む（count=0）
3. プロセス A が 1 を加算してファイルへ書き込む（count=1）
4. プロセス B が 1 を加算してファイルへ書き込む（count=1）

この場合，処理結果は不正な値となる．マルチプロセス処理では，どちらの実行順になるか保証されていない．

[2] このように，複数の処理が同時に行われると破綻をきたす部分のことをクリティカルセクションと呼ぶ．

3　セマフォによるプロセス間同期

競合状態を回避する方法としては，排他制御が一般的である[3]．排他制御とは，あるプロセスが共有データを独占的に利用できるよう，他のプロセスがそのデータを利用できないようにしておくことである．

排他制御を実現する方法はいくつかある[4]が，ここではセマフォ（semaphore）を用いる方法を採用する．

3.1　セマフォ

セマフォは，複数のプロセス間で同期をとるための古典的な機構[5]である．概念的にはどのプロセスからも見える旗のようなもので，各プロセスが旗の上げ下げをすることができる．

セマフォは値（0 以上の整数）を持ち，原則として wait と signal という二つの操作だけが可能である．wait をしたとき，値が 1 より大きければ値を減らし，0 ならば待機する．signal をしたとき，値が 0 より大きくなるのを待っているプロセスがあればそのプロセスを再開する．なければ値を増やす．

セマフォを操作するための基本的なシステムコールは以下の 3 つ．

`int semget(key_t key, int nsems, int flag)`：セマフォの獲得

`int semop(int semid, struct sembuf *sops, size_t nops)`：セマフォの操作

`int semctl(int semid, int semnum, int cmd, ...)`：セマフォの制御

`semget()` は `key` に対応するセマフォ集合の ID を返す．もしセマフォの獲得に失敗すると-1 を返す．`key` の値が `IPC_PRIVATE` の場合，あるいは `semflg` に `IPC_CREAT` が指定されていて， `key` に対応するセマフォ集合が存在しない場合， `nsems` 個のセマフォからなる新しい集合が作成される（つまり，すでに同じ `key` のセマフォが存在する場合は，新しく作るのではなく既存のセマフォの ID を返す）．

`semop()` は `semid` で指定されたセマフォ集合に対して操作を行う．操作の内容は，`sembuf` で指定する．`sembuf` の構造は以下の通り．

```
struct sembuf {
  unsigned short sem_num;  /* セマフォ番号 */
  short          sem_op;   /* セマフォ操作 */
  short          sem_flg;  /* 操作フラグ */
};
```

操作対象のセマフォを `sem_num` で指定する（`semctl()` の第 2 引数と同じ）．`sem_op` が正の整数の場合 signal 操作となる．その値をセマフォの値に加える．`sem_op` が 0 の場合，セ

[3]ただし，排他制御を用いることによって別の問題（デッドロック）に陥る恐れがあることに注意．
[4]今回の例（file-counter.c）であれば，ファイルのロック機構を利用することも可能である．
[5]考案したのは，構造化プログラミングで有名なあのエドガー・ダイクストラ．

マフォの値が 0 になるまでプロセスを停止して待機する．sem_op が負の整数の場合 wait 操作となる．その絶対値をセマフォの値から減算する．

semctl() は semid，semnum で指定されたセマフォに対して，cmd で指定された制御を行う．例えば cmd に SETVAL や GETVAL を指定すると，セマフォの値を参照・変更できる．

```
sid=semget(key, 1, 0666 | IPC_CREAT); /* セマフォ集合を獲得 */
semctl(sid, 0, SETVAL, 1); /* 獲得した集合の最初のセマフォの値を 1 に */
```

また，cmd に IPC_RMID を指定すればそのセマフォを破棄できる[6]．

これらのシステムコールについて，より詳細な使い方は Manpage などを参照すること．

3.2 サンプルプログラム — sem-wait.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main()
{
  int i, sid;
  key_t key;
  struct sembuf sb;

  setbuf(stdout, NULL); /* set stdout to be unbufferd */
  if ((key = ftok(".", 1)) == -1){
    fprintf(stderr,"ftok path does not exist.\n");
    exit(1);
  }
  if ((sid=semget(key, 1, 0666 | IPC_CREAT)) == -1) {
    perror("semget error.");
    exit(1);
  }
  printf("waiting for semaphore=1.\n");
  sb.sem_num = 0;
  sb.sem_op  = -1;
  sb.sem_flg = 0;
  if (semop(sid, &sb, 1) == -1) {
    perror("sem_wait semop error.");
    exit(1);
  }
```

[6] ファイルやソケットと同様，必要なくなったセマフォはちゃんと破棄しておくべき．

```
  printf("done.\n");
  exit(0);
}
```

まず，ftok() を用いて key を作成したあと，semget() でセマフォを獲得する．第 3 引数の 0666 はセマフォのパーミッションを表し，全ユーザから読み書きできることを表す．その後，semop() を呼び出すが，このとき sb.sem_op=-1 としているため wait 操作となる．

実際にこのプログラムを実行すると，メッセージを出力したあと即座に終了するか，待ち状態になる．どちらの状態になるかはセマフォの初期値次第であり，演習室の環境では 1 回目は即座に終了するが，もう一度実行すると待ち状態になるはずである．これは，セマフォの初期値が 1 になっており，1 回目はセマフォの値を 1 減らして実行終了するが，2 回目はセマフォが 0 になっているためである．待ち状態から動作を再開するためには，別のプロセスで次の sem-signal を実行する必要がある．

3.3 サンプルプログラム — sem-signal.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main()
{
  int i, sid;
  key_t key;
  struct sembuf sb;

  if ((key = ftok(".", 1)) == -1){
    fprintf(stderr,"ftok path does not exist.\n");
    exit(1);
  }
  if ((sid=semget(key, 1, 0666 | IPC_CREAT)) == -1) {
    perror("semget error.");
    exit(1);
  }
  sb.sem_num = 0;
  sb.sem_op  = 1;
  sb.sem_flg = 0;
  if (semop(sid, &sb, 1) == -1) {
    perror("sem_signal semop error.");
    exit(1);
  }
```

```
  printf("signal: add semaphore by 1.\n");
}
```

sem-wait と同じ key を指定してセマフォを獲得し，signal 操作として semop() を実行する．

試しに，sem-wait を実行して待機状態にした後，別のターミナルで sem-signal を実行してみると良い[7]．sem-signal を実行すると直ちに，sem-wait 側の処理が再開しプログラムが終了することを確認できる．このようにセマフォを用いることで，プロセスの待機・再開をコントロールすることが可能である．

また，逆に sem-signal を先に実行してから，すぐに sem-wait を実行してみると良い．この場合，すでにセマフォの値が 1 になっているため，sem-wait は待機せずにすぐ終了する．

注意

この動作を確認するため，サンプルプログラムではわざとセマフォを破棄していない．本来望ましくないので，きちんと破棄すること．破棄されていないセマフォで一杯になると，それ以上セマフォを獲得できなくなる．(semget error: No space left on device. というエラーが出る．) サンプルプログラム実行後は必ず以下のコマンドを用いて手動でセマフォを削除すること．

- `ipcs -s`：セマフォの一覧表示（ID と key を確認）
- `ipcrm -s [id]`：ID を指定してセマフォを削除

3.4 課題 3-1

セマフォを用いて排他制御を実現し，2.2.1 節の file-counter プログラムが確実に正しく動作するようにせよ．正しい動作とは，

1. counter ファイルの内容が最終的に 4 になること
2. 何度実行しても同じ結果になること（実行ごとに結果が変化してはならない）
3. file-counter プログラムが単体で動作すること（外部で別のプログラムを動かしてセマフォを操作してはならない）

以上の条件を満たすこと[8]である．

排他制御を実現するための手順は，以下を参考にすると良い．

1. プログラム内のクリティカルセクション（CS）を探す．

[7] 同じ端末（expXXX）上で実行すること．
[8] もちろんちゃんと 4 プロセスで協調して動作すること．1 プロセスで 4 回インクリメントするのは論外．

2. 複数のプロセスが CS を同時に実行しないよう，CS の直前で排他制御をはじめる（ロック）．セマフォを利用して，最初の 1 プロセスだけが CS を実行し，他のプロセスは待機するようにする．

3. CS の最後で排他制御を終える（アンロック）．セマフォを利用して，待機中の他のプロセスを再開させる．

4 パイプを用いたプロセス間通信

課題 2 でソケットを利用したプロセス間通信について学習した．`fork()` で作成した親子のプロセス間では，パイプを用いてより簡単にプロセス間通信を実現できる．

パイプとはメモリ内に設けられたバッファ領域のことで，これを 2 つのプロセス間の通信経路として利用する．UNIX ではこの領域を特別なファイルと見なして扱うため，通常のファイルやソケットと同様に `read()` や `write()` でアクセスが可能である．

パイプを作成するシステムコールを以下に示す．

`int pipe(int fildes[2])`：パイプの作成

`pipe()` は，引数で渡された 2 個の配列の各要素にファイル記述子を格納して返す．

- `fildes[0]`：読み出しモードでオープンされたファイル記述子（read 専用）
- `fildes[1]`：書き込みモードでオープンされたファイル記述子（write 専用）

これらは同じパイプの出口と入口を表している．つまり `fildes[1]` にデータを書き込むと，そのデータは `fildes[0]` から読み出される．

なお，パイプに一度にバッファリングできるデータサイズには上限がある[9]．それを超えて書き込みしようとすると，ブロックされて待ち状態になってしまう．そのパイプからデータが読み出されて余裕が出来ると，新たに書き込むことが可能になる．

4.1 サンプルプログラム — pipe.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define BUFSIZE 256

int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
```

[9]環境に依存するが，およそ 4KB〜5KB．

```
  int fd[2];
  int pid, msglen, status;

  if (argc != 2) {
    printf("bad argument.\n");
    exit(1);
  }

  if (pipe(fd) == -1) {
    perror("pipe failed.");
    exit(1);
  }

  if ((pid=fork())== -1) {
    perror("fork failed.");
    exit(1);
  }
  if (pid == 0) { /* Child process */
    close(fd[0]);
    msglen = strlen(argv[1]) + 1;
    if (write(fd[1], argv[1], msglen) == -1) {
      perror("pipe write.");
      exit(1);
    }
    exit(0);
  } else { /* Parent process */
    close(fd[1]);
    if (read(fd[0], buf, BUFSIZE) == -1) {
      perror("pipe read.");
      exit(1);
    }
    printf("Message from child process: \n");
    printf("\t%s\n",buf);
    wait(&status);
  }
}
```

引数で渡した文字列を，子プロセスからパイプを介して親プロセスに送信し，親プロセスが出力するプログラムである．

プロセス間通信にパイプを用いる場合，複数のプロセスでパイプのファイル記述子を共有する必要がある[10]．そのため，必ず `fork()` でプロセスを作成する前に，`pipe()` を実行

[10]逆に言えば，課題 2 のようなクライアント・サーバプログラムのプロセス間通信にはパイプを利用でき

しなければならない．

また，一般に 1 つのパイプは単方向の通信のみに利用する．言い換えると，あるプロセスから見たとき，1 つのパイプを読み書き両方に利用することは望ましくない．このプログラムではパイプを子プロセスから親プロセスへ向きの通信に利用する．そこで安全のため，利用しないパイプのファイル記述子は `close()` しておく（親プロセスでは `fd[1]` を，子プロセスでは `fd[0]` を閉じておく）．このようなプログラムを単方向パイプと呼ぶ．

4.2 課題 3-2-1

pipe.c を拡張し，双方向パイプのプログラム two-way-pipe を作成せよ．

```
two-way-pipe MessageToParent MessageToChild
```

two-way-pipe は 2 つの引数をとる．最初の引数で渡した文字列を子プロセスから親プロセスへ送信し，次の引数で渡した文字列を親プロセスから子プロセスへ送信する．

双方向パイプを実現するためには，2 つのパイプが必要になる．“親プロセスから子プロセス向けのパイプ”と “子プロセスから親プロセス向けのパイプ”を作成し，読み書きすれば良い．

2 つの送信は同時に行うこと．つまり，各プロセスは最初にパイプへデータを `write()` し，次に `read()` する．

4.3 課題 3-2-2

CLE の課題 3 に添付したマージソートを行うプログラム mergesort.c を拡張し，並列版マージソートを作成せよ．

マージソートは，未整列配列を部分配列に分割し，部分配列ごとにソートした後マージすることで全体をソートするアルゴリズムである．各部分配列のソートは互いに独立な（依存関係がない）ため，同時に実行することができる．

そこで，2 つのプロセスで手分けしてソートするよう，以下のようにプログラムを拡張する[11]．

1. 最初に配列を中央で 2 分割する．
2. プロセスを 2 つ用意し，1 つの部分配列を 1 つのプロセスでそれぞれソートする．各プロセスでのソートには（再帰的に）マージソートを用いる．
3. 2 つのプロセスのソート結果（整列済み部分配列）を，パイプを使って送信し，1 つのプロセスに集める．
4. 集めた整列済み部分配列をマージし，結果を得る．

ない．

[11] この並列化手法を分割統治法（Divide and conquer）と呼ぶ．

元の mergesort.c は配列を 2 つに分割してソートとマージを再帰的に繰り返すプログラムである．`fork()` によるプロセスの作成まで再帰的に繰り返す必要はないことに留意せよ[12]．

5 割り込みを用いたプロセス制御

プロセスに対するソフトウェア割り込みの方法として，シグナル[13]がある．シグナルを用いると，プロセスあるいはユーザから，ある事象の発生を他のプロセスに非同期に伝えることができる．

割り込みとは，プロセスが明示的に受信を待っていない（なんらかの処理を実行している）状態でも，外部から働きかけて現在の処理を中断させ，ある決まった処理を実行させることである．割り込み処理が終わると，中断前の状態に戻ってそれまでの処理を再開する．例えば，既によく使用している割り込みとして CTRL-C によるシグナル `SIGINT` の送信がある．プログラムの実行中にキーボードの CTRL-C を押すと，プログラムの実行を中断して強制終了させる（この場合はもちろん再開しない）．

シグナルは 20 種類以上あり，シグナルごとに意味が異なる．そのシステムで利用可能なシグナルはヘッダファイル signal.h[14] で確認できる．各シグナルの詳しい意味は，各自で調べてみると良い．

5.1 シグナル受信時の処理

シグナルごとに受信時のデフォルト処理が決まっているが，これをユーザの設定した別の処理に変更することができる．これによって，CTRL-C を押した場合にすぐに終了するのではなく任意の動作をできるようになる．

シグナル受信時の処理を変更するシステムコール `signal()` の仕様を以下に示す．

- void (*signal(int sig, void (*func)(int)))(int)

あまり見かけない書き方なので分かりづらいと思う．まず，`signal()` の戻り値は関数へのポインタである（そしてその関数の戻り値が void ＝何も返さない）．したがって，他のシステムコールのように戻り値が-1 かどうかでエラーチェックができない．エラーチェックのためには `SIG_ERR` [15] と比較する．

次に，第 1 引数は設定したいシグナルの種類（`SIGINT` など）を指定する．第 2 引数は，そのシグナルを受信したときに実行させたい関数名（正確には関数へのポインタ）を指定する．この関数のことをシグナルハンドラと呼ぶ．サンプルで説明したほうが早いので，次のサンプルプログラムを見て欲しい．

[12] このようなプログラムを書くと，あっというまにプロセス数が増えすぎて資源が枯渇する．
[13] セマフォの signal 処理とはまた別の意味なので注意．
[14] 演習室ならば /usr/include/asm-generic/signal.h
[15] これも signal.h で定義されている．

5.1.1 サンプルプログラム — signal.c

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#define BUFSIZE 256

void sigmsg()
{
  printf("\n!! CTRL-C is pressed !!\n\n");
   /*
  if(signal(SIGINT,sigmsg) == SIG_ERR) {
    perror("signal failed.");
    exit(1);
  }
  */
}

int main()
{
  char buf[BUFSIZE];

  if(signal(SIGINT,sigmsg) == SIG_ERR) {
    perror("signal failed.");
    exit(1);
  }

  while (fgets(buf, BUFSIZE, stdin) != NULL) {
    printf("echo: %s",buf);
  }
}
```

標準入力からの入力をそのまま標準出力に書き出すだけの単純なプログラムである．

fgets() のループを実行する前に，signal() で SIGINT に対する受信処理を sigmsg() に変更している．このため，ループ実行中に CTRL-C を押しても，メッセージが出力されるだけでプログラムは終了しない．シグナルハンドラ（sigmsg()）の実行を終えると，プログラムはシグナル受信前の状態に戻る．今回の例では（たいていの場合）fgets() の入力待ちに戻るので，ループが継続される．プログラムを終了するためには CTRL-D[16]を入力して EOF を与える．

シグナルハンドラはユーザ定義の関数で，名前も処理内容も自由に決めてよい．ただし，戻り値は void，引数はなし[17]と決まっている．ここでは単にメッセージを標準出力に書き

[16]余談だが，CTRL-D はシグナルを送るわけではなく，あくまで EOF を標準入力に与えるだけである．
[17]正確にはシステム側で暗黙的にシグナルの種類が渡される．

出すだけの単純な関数 sigmsg() である．また，システムによっては 1 度シグナルを受信するとシグナルハンドラがデフォルトにリセットされてしまうので，シグナルハンドラ内で再度 signal() を呼び出して再設定する必要がある．演習室ではこの必要は無い．

また，特別なシグナルハンドラとして SIG_DFL と SIG_IGN があらかじめ定義されている．SIG_DFL を指定すると，そのシグナル受信時の動作をデフォルト設定に戻す．SIG_IGN を指定すると，そのシグナルを無視[18]するように設定できる．実際にサンプルプログラムを少し変更して，SIGINT を無視するように試してみると良い．

注意

実はこのサンプルプログラムには問題がある．シグナルハンドラ内で “非同期安全でない関数（printf）” を呼び出すためである．printf は内部でグローバル変数（stdout や stderr）にアクセスするが，シグナルを受信したときにグローバル変数の値がどうなっているかは保証がない．そのため，予期せぬ動作をする可能性がある．

一般に，シグナルハンドラ内では “非同期安全な関数 ”[19]のみを利用すべきである．究極的には単にフラグをセットするだけのシンプルな処理に留めるのが良い．

5.2 プロセス間でのシグナル送受信

CTRL-C 以外にもプロセスにシグナルを送る方法はいくつかある[20]．一般に任意のプロセスから任意のプロセスへシグナルを送るためには，kill() システムコールを利用する．

- int kill(pid_t pid, int sig)：シグナルの送信

第 1 引数にシグナルを送りたいプロセスのプロセス ID（PID），第 2 引数にシグナルの種類を指定する．たまに名前から勘違いする人がいるが，必ず SIGKILL を送るわけではないし，必ずプロセスを殺す（強制終了させる）わけでもない[21]．原則として，kill() を呼び出すプロセスのユーザ ID と，シグナルを受け取るプロセスのユーザ ID は一致しなければならない．

5.2.1 サンプルプログラム — kill.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
#define NUMPROCS 3
```

[18] ただし一部のシグナルは無視できない．例えば強制終了の SIGKILL はこの設定が無効である．誰も終了できないプロセスを生み出さないため．

[19] 何が非同期安全な関数かは環境によって異なる．自分で調べること．

[20] 例えば，alarm() を利用して一定時間後に SIGARM を送るなど．

[21] これも余談だが，同じ事は UNIX コマンドの kill にも言える．

```
int main()
{
  int i, pid, status;
  int childlen[NUMPROCS];

  for (i=0; i<NUMPROCS; i++) {
    if ((pid=fork())== -1) {
      perror("fork failed.");
      exit(1);
    }
    if (pid == 0) { /* Child process */
      while (1) {
        printf("Child process: i=%d\n", i);
        sleep(1);
      }
      exit(0);
    } else {
      childlen[i] = pid;
    }
  }
  /* Parent process */
  for (i=0; i<NUMPROCS; i++) {
    sleep(3);
    if (kill(childlen[i],SIGTERM) == -1) {
      perror("kill failed.");
      exit(1);
    }
  }
  printf("Parent process: all childlen are terminated.\n");
  for (i=0; i<NUMPROCS; i++)
    wait(&status);
}
```

親プロセスから子プロセスへシグナルを送るプログラムである．親プロセスは fork() したときに子プロセスの PID が分かるので，簡単にシグナルを送ることができる．

各子プロセスは無限ループを実行している．そこへ親プロセスが 3 秒おきに SIGTERM（ソフトウェア終了シグナル）を送ってくるため，シグナルを受けた子プロセスから順番に終了していく．実際に動作してみると，時間の経過とともに子プロセスの出力が減っていくことが確認できる．

なお，kill() を用いたシグナルの送信は，親子プロセス間でなくても可能である．ただし，何らかの手段でシグナルを送りたいプロセスの PID を取得しなければならない．

5.3 課題 3-3-1

alarm() を使用せずに，同等の機能を実現する myalarm() をマルチプロセスを用いて実装せよ．

- unsigned int alarm(unsigned int sec)：シグナルタイマー

alarm() は実行した sec 秒後に，呼び出し元のプロセスへシグナル SIGALRM を送信するライブラリ関数である．使い方の例としてサンプルプログラム alarm.c を載せる．標準入力をそのまま出力するプログラムだが，10 秒間なにも入力がなければメッセージを表示してプログラムを終了する（タイムアウト）．もし入力があればタイムアウト時間をリセットする．

alarm.c の関数 myalarm() の定義を書き換えて，alarm() を使わずに同等の機能を実現すること．その際，以下の条件を満たす必要がある．

1. 書き換えて良いのは myalarm() の中だけである．
2. 呼び出し元のプロセスの処理は，myalarm() を呼び出したらすぐに戻ってくること（ブロックしてはならない）．
3. マルチプロセスを利用して，指定した秒数が経過したら呼び出し元のプロセスに SIGALRM を送信する．その際，子プロセスがゾンビプロセス[22]となって残らないように注意する．
4. すでにタイマーがセットされていた場合，そのタイマーを削除して新しいタイマーを設定する（myalarm() を呼び出すたびにプロセスが増えていってはならない）．

1〜3 の条件を満たそうとすると，今までの例のように単純に wait() できないことに気づくであろう．wait() を使わずにゾンビプロセスを防ぐ方法があるので，各自で調べて実装すること[23]．

5.3.1 サンプルプログラム — alarm.c

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define BUFSIZE 256
#define TIMEOUT 10

void myalarm(int sec) {
```

[22]親プロセスに wait されずに放置された子プロセスはゾンビプロセスになってしまう．
[23]ヒント：プロセス制御に関する特殊なシグナルについて調べてみよ．

```
  alarm(sec);
}

void timeout()
{
  printf("This program is timeout.\n");
  exit(0);
}

int main()
{
  char buf[BUFSIZE];

  if(signal(SIGALRM,timeout) == SIG_ERR) {
    perror("signal failed.");
    exit(1);
  }

  myalarm(TIMEOUT);
  while (fgets(buf, BUFSIZE, stdin) != NULL) {
    printf("echo: %s",buf);
    myalarm(TIMEOUT);
  }

}
```

5.4 課題 3-3-2

前節で作成した myalarm() を利用して，課題 2 で作成した simple-talk-client プログラムにタイムアウト機能を実装せよ．

1. 一定時間入出力がない場合，メッセージを出力してプログラムを終了すること．
2. シグナルハンドラ内では，“非同期安全な関数”のみを利用する．これまでのサンプルプログラムは printf() を利用しているのでダメな例である．真似しないこと．
3. プログラムを終了する前にきちんと後始末すること．SIGALRM を受け取ったからといっていきなり exit() してはならない．ソケットの close など，必要な後始末をしてからプログラムを終了する[24]．

ただし，myalarm() をどうしても仕様通りに作れなかった場合は，システムに用意されている alarm() を利用しても構わない．

[24] ヒント：全ての後始末をシグナルハンドラ内でやろうとするとかなり無理がある．シグナルハンドラ内でフラグをたて，後始末は外でやるほうが楽であろう．

6　発展課題

余力のあるものは取り組んでみて欲しい．

1. セマフォを用いたバリア同期
   セマフォを利用して，バリア同期（プロセスの待ち合わせ）を実現せよ．

   課題 3-1 の排他制御は，“あるプロセスが特定の処理している間，他のプロセスは待機する ”という機能である．バリア同期は，“全てのプロセスが特定の処理に到達するまで，先に到達したプロセスは待機する ”という機能である．任意のプロセス数に対してバリア同期を実現するプログラムを作成すること．

2. マージソートの更なる並列化
   課題 3-2-2 のマージソートは，2 つのプロセスに処理を分割して並列実行する．これを任意の P 個のプロセスで並列実行できるように拡張せよ．

   これにはいくつかの方法が考えられる．例えば，最初に配列を P 個に分割してそれぞれプロセスを割り当てる方法がある．他には，2 分割してプロセスを割り当てる処理を（制限つきで）再帰的に繰り返す方法もある．自由な方法で実装して構わない．

   なお，並列化しても必ずしもプログラムが高速化されるとは限らない．演習室の仮想環境では端末の CPU 数が 1 となっているため，実際には並列処理が行われない．また，CPU 数が複数であっても，プロセス間通信（データのコピー）に時間がかかるため，並列化しないほうが早い場合もありえる．自宅など複数の CPU が利用可能な環境があれば，実際に配列のサイズ N とプロセス数 P を色々と変化させながらソート時間を測定し[25]，並列化しない元のプログラムと比較して，並列化によって高速化効果が得られる条件を調べてみよ．

3. 交互実行
   セマフォやシグナルを用いて，2 つのプロセスが交互に動作するプログラムを作成せよ．

   例えば以下のような実装が考えられる．プログラムが実行されたら 2 つのプロセス (A と B とする) に fork する．A は “A[1]〜A[100] まで”，B は “B[1]〜B[100] まで” を printf などで出力する．ただし，A[n] と B[n] が交互に出力されるよう，プロセス間の調停を行う．

7　レポート提出について

課題 3 に関しても PDF で作成したレポートと作成したプログラムのソースコードのファイルを CLE へ提出してもらう．この資料に示した課題をまとめて記載し，1 つのレポートとして提出すること．

レポートのスタイルは特に問わないが，TeX の使用を推奨する．手書きのレポートは認めない．

レポートの表紙は，1 ページを使って，

[25] 配列の入出力など，本質的でない処理を時間計測に含めないよう注意せよ．

- 授業名
- 課題名
- 学籍番号
- 名前

を書くこと．レポートの内容に関しては，少なくとも，

- プログラムの仕様 (使い方，制限事項など)
- 実行結果
- 考察
- 感想，意見，疑問等

を盛り込むこと．もちろんその他にもレポートに書くべきことはあるが，演習のガイダンス時に配布した資料等を参考にして作成すること．

〆切は，

7/3(月) 12:59

とする．CLE に記録された時刻を提出時刻とし，演習開始の時点で提出が終わっていない分は提出遅れとして扱う．なお，いくらかの誤差が考えられるため，十分な余裕を持って提出すること．

参考文献

- login:UNIX C システムプログラミング入門，河野 清尊，オーム社，1992