計算機プログラミング'95 ＃1

久野　靖*

1995.9.7

0　はじめに

この科目「計算機プログラミング」の位置づけは、「プログラミング基礎」がまったくの初心者を対象としていたのに対し、もう少しadvanced な内容で「基礎」に続いて受講できるが、同時に初心者でない人(計算機を専門にしている人)にも興味深いものを取り上げようということです。

ふつうの情報専門学科でプログラミング入門に続く科目というと「データ構造」「アルゴリズム」などでみっちりしごかれるのですが、それだと専門家には既知の話でつまらないし、非専門家にはハードだと思います。

そこで、今年のこの科目では「システムプログラミング」をテーマにして、OSやその他のライブラリ、ミドルウェアを利用するプログラミングを学んでみていただきます。プログラミング自体の基本原理は「基礎」と同程度から始まりますので、非専門家でも大丈夫だと思いますし、Unixのシステムプログラミングを知らない専門家の人にも楽しめると思います。

Unixシステムプログラミングを知っている人は出てもつまらないでしょうから、単位が必要なら資料のみ入手してレポート課題だけ出してくださって結構です。成績は出席兼小レポートと、最終レポートの両方でつけますが、最終レポートがまっとうなら出席はなくてもAを差し上げます。逆は勘弁してください。

言語はUnixでのシステムプログラミングということでほぼ必然的にCを使用します。(C++でもよいのですが、ちょっと覚えることが多いので…) C言語自体について1から順に教えることはしません(それをやると専門家はつまらない)。毎回、例題ベースで「なんとなく」必要なことだけ覚えていけば済むようにするつもりです。

従って、教科書は特に使用せず、毎回資料を配布します。ただ、Cを知らない人は参照できる本が欲しいと思うかも知れません。その場合は

- カーニハン、リッチー、石田訳、「プログラミング言語C」第2 版、共立、1989

を参考書に指定しますので、各自入手してください。Cを知っている人は買う必要はないでしょう。

1　システムプログラミングとは?

システムプログラミングとは、特定の問題(数値計算とかシミュレーションとか線形計画)を解くためのプログラミング(アプリケーションプログラミング)と対比して使われることばで、OSその他の計算機システムに備わっている機能を駆使するプログラムを作ることをいう。典型的なシステム機能として次のようなものが挙げられる。

- ファイルの操作

*筑波大学大学院経営システム科学専攻

- ネットワーク機能
- ウィンドウシステム
- 資源管理

これらの機能を利用するには、様々なライブラリルーチンを呼び出す。機能の中にはほとんどライブラリサブルーチンで実現されているものも、OS の機能として実現されているものもある。OS 機能の場合には、サブルーチンの中でシステムコールが行われて OS が呼び出されるのだが、プログラム側から見た目はふつうのサブルーチンと同じである。

従って、システムプログラミングについて学ぶということは

- どのような機能があるか
- どんな仕事はどの機能をどう組み合わせて使うか
- そのためのサブルーチン呼び出しの詳細

を学ぶということである。実は詳細はどうでもよくて、一番肝心なのは OS なりウィンドウシステムなりがどんな機能を提供してくれているのか、というモデルを頭の中に獲得することだが、最終的にプログラムを動かすのには詳細まで正しくないといけない。しかし詳細は覚えなくてもマニュアルを参照すれば済むことである。

2 ファイル入出力

計算機科学基礎でやったが、Unix ではプログラムは「チャネル」(小さい番号で表される) に接続された装置 (キーボード、画面、ファイル等) との間で入出力をおこなう。その際には、読み込みは read、書き出しは write というシステムコールを使う。C からの呼び方は:

```
nbytes = read(chan, buf, length)
nbytes = write(chan, buf, length)
```

chan はチャネル番号 (0: 標準入力、1: 標準出力、2: 標準エラー出力)。buf は読み書き内容を格納する場所の先頭番地。length は読み書きしようとするバイト数。nbytes は実際に読み書きできたバイト数。man ページでも確認のこと。たとえばファイルコピーの例題はこうなる。

```
/* t01.c -- file copy */
main() {
  char buf[20];
  int l;
  while((l = read(0, buf, 20)) > 0) {
    write(1, buf, l);
  }
}
```

これを動かすには (打ち込みとコンパイルは覚えていますね??):

```
% a.out <入力ファイル >出力ファイル
```

練習 動かしてみよ。

練習 改行文字 ('\n') 以外の文字をすべて「*」に変換するというプログラムにしてみよ。

なお、配列をアクセスするには

```
... buf[…] ...
buf[…] = ...
```

ここで「…」のところに「何番目」を表す式を書く。基本的な制御構造は:

```
if(条件) {
  ...           ←条件が成り立ったとき実行
}
else {
  ...           ←成り立たなかったとき実行
}
```

else 以下は不要なら書かないでよい。

```
while(条件) {
  ...           ←条件が成り立っている間反復実行
}

for(式 1; 条件; 式 3) {
  ...           ←条件が成り立っている間反復実行
}
```

for では式 1 は繰り返しに入る前に 1 回だけ実行され、式 3 は次の繰り返しに進む直前に実行される。これを使って計数ループを書く。

```
for(i = 0; i < 20; ++i) { ... }
```

なお、いずれも「...」の部分に文が 1 つしかなければ「{」と「}」で囲まなくてもよい。しかし必要なときに忘れるとよくないので、いつもつける方がよい。

練習 上のプログラムを改造して、入力ファイルに現れる文字「a」の個数を数える、というプログラムにしてみよ。

練習 バッファの大きさを 1 にしたらプログラムは簡潔になる。やってみよ。

なお、数えた数値を打ち出すのにはたとえば次のように printf を使う。

```
printf("count = %d\n", count);
```

ところで、バッファの大きさを変えるたびに「20」のところをすべて書き換えるのは面倒だし、まちがいのもと。そこで、

```
#define BUFSIZE 20
```

というのを最初に入れておき、あとは「20」のかわりに「BUFSIZE」と書くようにしよう。

3 バッファリング

システムコールは OS を呼び出すのにかなりの命令数を必要とするので、1 文字読むごとに read システムコールを使うようなプログラムは効率が悪い。しかし、プログラム上は 1 文字ずつ読めた方が都合がよい。そこで、自分で「1 文字ずつ読む read」を作ってしまえばよい。

```
char r1buf[BUFSIZE];
int  r1ptr = 0;
int  r1len = 0;

read1(char buf[]) {
  if(r1ptr >= r1len) {
    r1len = read(0, r1buf, BUFSIZE); r1ptr = 0;
    if(r1len == 0) return 0;
  }
  buf[0] = r1buf[r1ptr++]; return 1;
}
```

練習 先のプログラム上の read1 を利用するように直せ。

練習 BUFSIZE をいくつにするのがよいか実験してみよ。

4 標準入力ライブラリ

システムコールレベルでは大きなブロックで読むのがよくて、プログラム側では 1 文字ずつ処理できると嬉しい→そのための標準ライブラリが用意されている。

```
/* t03b.c -- count 'a' using stream */
#include <stdio.h>

main() {
  int c;
  int count = 0;
  while((c = getchar()) != EOF) {
    if(c == 'a') ++count;
  }
  printf("count = %d\n", count);
}
```

練習 このプログラムを利用して、いくつかの典型的な tr の動作を行わせてみよ。たとえば「tr a-z A-Z」はどうか。

練習 sed だとどうか。たとえば「sed 's/a/AA/'」では。

練習 「sed 's/tion/TION/'」ではどうか。

5 一度に全部読む?

しかしそもそも、たくさんまとめて読んだ方がよいのだったら、ファイル全部をまとめて読むというのはどうだろう? そうすれば読み込みのループがなくて済む (読んだ内容のうちで「a」を数えるのにはループが必要。

```
/* t04.c -- count 'a' with one read */
#define BUFSIZE 250000
char buf[BUFSIZE];  ←巨大な配列は外でとる (要説明)

main() {
  int l, i;
  int count = 0;
  l = read(0, buf, BUFSIZE);
  for(i = 0; i < l; ++i) {
    if(buf[i] == 'a') ++count;
  }
  printf("count = %d\n", count);
}
```

なぜ 25000 かというと、「ls -l /usr/dict/words」で調べると大きさが 20 万ちょっとだから。しかし、このプログラムだと 250000 バイトより大きいファイルでは処理できない。

ではどうしたらいいか? プログラムの内部でファイルの大きさを調べて、その大きさの配列を自分で用意する。どうやって大きさを調べるか? それは、fstat というシステムコールを使えばよい。

```
/* t05.c -- count 'a' with one read, allocate by myself */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

main() {
  int l, i, count = 0;
  struct stat st;
  char *buf;
  fstat(0, &st);
  buf = malloc(st.st_size);
  l = read(0, buf, st.st_size);
  for(i = 0; i < l; ++i) {
    if(buf[i] == 'a') ++count;
  }
  printf("count = %d\n", count);
}
```

6 ファイルマッピング

しかし実は、ファイルを「メモリに貼りつけて」しまうことができる。これだと、malloc がいらない分だけ速い。ほかに、ファイルが更新できたりプログラム間でデータが共有できるなどの機能もある。

```
/* t06.c -- count 'a' with file mapping */
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <sys/mman.h>

main() {
  int l, i, count = 0;
  struct stat st;
  char *buf;
  fstat(0, &st);
  l = st.st_size;
  buf = mmap(0, l, PROT_READ, MAP_PRIVATE, 0, 0);
  for(i = 0; i < l; ++i) {
    if(buf[i] == 'a') ++count;
  }
  printf("count = %d\n", count);
}
```

練習 このプログラムを利用して、さっきの tr や sed の例をやってみよ。ただし/usr/dict/words を自分のところへコピーしてやらないといけない。