プログラミング環境 第10回

久野 靖 *

1991.12.10

13 Cによるプログラミング:基本編

13.1 なぜCか?

さて、ここまででは皆様に様々な指令のレベルからUnixというOSを体験していただいてきた。もちろんそれはそれで結構色々なことが分かるのだが、図1のように自分でプログラムを書くことにより、これまで以上に直接的にOSの各種機能を触ってみることができる。そうするときっと、「あ、OSというのはこういう風な機能を提供するのか」「自分たちが使うような指令は実はOSのこういう機能をこう組み合わせるのか」という直観が働くようになるはずである。それをめざして、頑張っていただきたい。

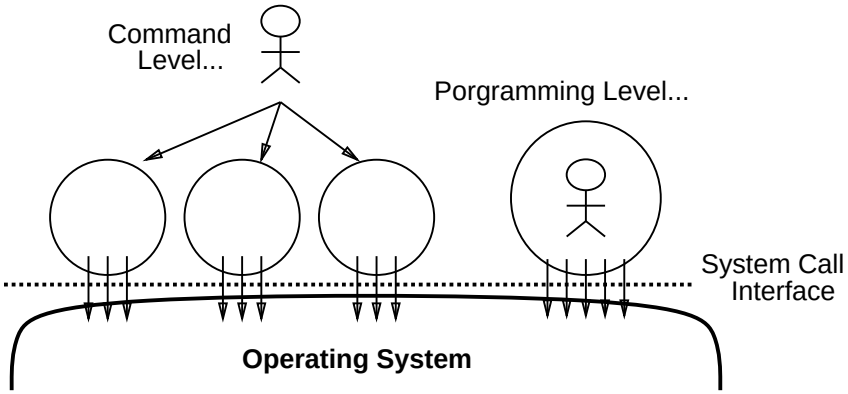

図 1: OSとシステムコール界面

さて、そのためにまずCという言語をとりあえず習っていただくのだが、なんで既に知っている言語、たとえばPascalやSchemeを使ってくれないのか疑問に思う方もいるかも知れない (どうかな?)。それは次のような理由による。もともとUnixはC言語で書かれていて、またその上で主として使われるプログラム言語もCであることが想定されているので、Unixのさまざまな機能や、Unixそのものでなくても Unix上で動く様々な機能 (例えばX-Windowとか) はどれもCから呼び出しやすいように作られている。

例えば、OSの内部でプロセスを管理するのに個々のプロセスごとに構造体 (Pascalのrecordのようなもの) を使ってデータ構造が作られているとしよう。もちろん、これはUnixの中だからCで書かれている。次に、この情報を一般のプログラムがOSからもらってきて調べたいとする。このプログラムがCで書いてあるなら、OSの中の構造体データをそのままの形でコピーしてくれ

*筑波大学経営システム科学専攻

ばそれはCの構造体だからそのまま使うことができる。しかしプログラムがPascalやSchemeで書いてあったら、その情報をPascalのrecordとして正しい形、Schemeのリストとして正しい形に変換しないと使うことができないわけである。一部の機能についてはこういう変換をするコードを書いた人があってそれを使えば受け渡せるが、全部というわけにはいかない。

そういう訳でOSの機能を利用するのにCが有利なため、Unix上のコードはCで書かれたものが多い。するとそれらのコードを (独立なプログラムとしてではなく) サブルーチンとして呼び出して利用するのもやっぱりCからなら簡単だが、PascalやSchemeからCのサブルーチンを呼び出すのは簡単ではない。こういう関係からUnix上でこの種のプログラミングをやるにはC、という環境が自然とできてしまったわけである。

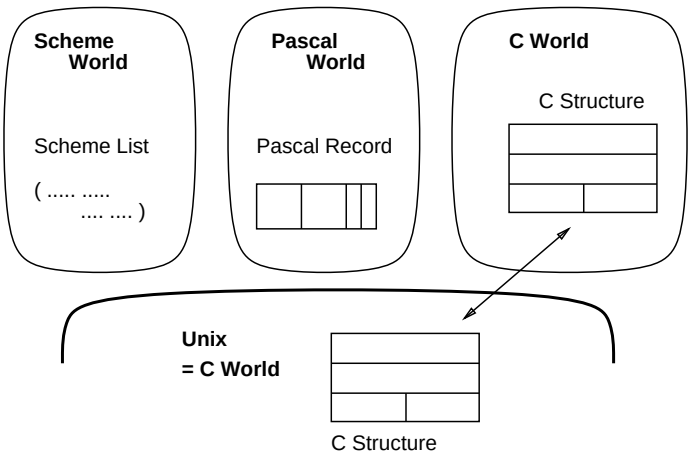

図 2: OS内部との情報の受け渡し

13.2 Cとはどんな言語か?

話が逆になったが、この際だからC言語の由来と特徴についても一応説明しておこう。C言語は、Unixの作者の一人RitcheがUnixを書くために作った言語である。その当時は、最初の高級言語であるFortran(= 数値計算用) に続いて、記号処理言語、文字列処理言語、事務処理言語など様々な言語が作られ使われていた。そのような言語群のなかに、OS開発などの、これまではアセンブラで行われていたプログラミングを置き換えることを目的として設計された「システム記述言語」と呼ばれる類の言語がある。その代表的なものとしてBCPL、BLISSなどが挙げられる。これらの言語は (アセンブラと同様に) アドレス、ポインタ、整数、ビット列などを区別せずにいっしょくたに扱い、その結果OSなどでは不可欠な任意のアドレスの計算やアクセスができるようになっていた。

RitcheはOSを書くためにBCPLを改良した言語「B」(その名前はBCLP、の1文字目だからそうつけたものである) を開発したが、Bは解釈実行がたの言語であり速度に問題があったのでUnixを書くに当たってそれをさらに改良した言語を作り、「C」と名付けたわけである。[1]

Cの特徴は、それ以前のシステム記述言語と異なり、アドレスやポインタを自由に操作することを許しながらそれらを整数などとは一緒にせず、各データに型をきちんとつけて扱うようにした点であるが、一方で言語自体としてはできるだけ簡潔で計算機ハードウェアの機能にほぼ対応したレベルの操作までを提供する点では他のシステム記述言語と同様である。たとえば (今のCはやや違うが、もとのCでは) 代入によって操作できるのは計算機のレジスタに入るようなデータ

[1]その後、「P」とか「L」とかいう言語は作られないまま終わっているようだ。

のみであった。また入出力をはじめとする機能は言語の仕様としてはなく、すべてそのための手続きを呼び出すことで行う。だから言語としてはすごく簡潔である。まあ、ごたくはそれくらいにして、具体的な話に進むことにしよう。

13.3 C プログラムの基本構造

C のプログラムは基本的に次のような構造をしている。まず、プログラムはいくつかのファイルに分けて入れておくことができる (今回はそんなに長いのはないだろうから、全部 1 つのファイルで沢山であるが)。

プログラム =ファイル ...

次に、各ファイルの中には (変数などの) 宣言、および関数定義を好きなだけ書いてよい。プログラムがただの関数の集まり、という辺は Scheme に似ているが、ただし関数定義の中にまた関数定義を書くことは残念ながら許されない。関数の中で、`main` という名前のものは特別で、プログラムの実行はシステムがまず `main` を呼び出すことによって開始される。

ファイル =(宣言 | 関数定義)...

関数定義というのは、Scheme や Pascal の場合と大差ないが、ただしすべての関数や引数に型がないといけない点は Pascal に近い。まず関数の返す値の型を指定し、続いて関数名を書く。引数があるばあいは続くかっこの中にその名前を「,」で区切って書く。それらの型はかっこことじに続いて宣言する (引数がなければこれらは不要だが「()」は必ず書く)。続いて「{」と「}」の間に好きなだけ文を書くが、局所変数などが必要なら文のならびの前に書く。

関数定義 =型指定 関数名 ([名前 ,...]) [宣言 ...]{ [宣言 ...] 文 ...}

さて、「宣言」というのが実は複雑なのだが、詳しい話をやりだすとうんざりすると思うのでとりあえずは簡略化して、次のようなものだということにしておく。

型指定 (名前 [= 式]| 名前 [整定数]),...;

型指定には `int`(整数)、`char`(文字型)、`double` (倍精度実数) などがある。名前は変数の名前である。単独の変数の場合は続いて`=`に続いて初期値をしていしてもよい。`[]` は配列の指定で、その中に配列の大きさを書く。C では配列の添字は必ず 0 から始まる。

13.4 C の文

これは基本的にはおなじみの if 文その他、である。面白いのは「式」だけでも「文」になることで、例えば関数呼び出しや代入は「式」である。[2][3]また、「;」だけだと「何もしない文」であるがこれは Pascal の空文に相当する。また複合文は Pascal では begin-end だったが C では {} で囲む。

文 =if 文 |while 文 |... | 式 ; |; |{ [宣言 ...] 文 ...}

あとは awk などでおなじみの文が多い。

[2]例えば `func(1, 2);` などという「文」は `func` という関数を呼び出して値を受けとるが、その値はどこにも入れずに「捨ててしまう」から結果として呼び出すだけ、ということになる。

[3]代入演算子は左辺の場所に右辺の値を入れて、全体としては代入された値をとる。だから `x = y = 0;` などというのも許される。

if 文 =`if`(式) `then` 文 [`else` 文]
while 文 =`while` (式) 文
for 文 =`for` (式 1 ; 式 2 ; 式 3) 文

awk のときと同様、for 文は次の while とほぼ等価である (どこが違うと思うか?)。

式 1 ; `while` (式 2) { 文 式 3 }

return 文で関数のどこからでも値を持って返ることができる。return 文がなく最後の文まで来てしまうと「ごみの値」を持って返ることになるので注意。

return 文 =`return` 式 ;

最後に、break 文はループの途中で抜け出すのに使う。

break 文 =`break` ;

いくらかはしょった文もあるが、まあこれくらいで C のプログラムを普通に書くには十分である。あと、「式」の話も残っているが、そろそろ実例が出ないと退屈なのであとで必要になった時説明することにして先へ進むことにする。

14 CPU 管理、主記憶管理、仮想記憶

14.1 CPU 管理の実験

さて、この講義の最初の方でやったように、CPU の一番基本的な機能は命令を順番に取り出して実行する、ということであった。まずはそれだけ、つまり入出力を伴わないプログラムで遊んでみよう。そんなことができるのかって?ちゃんとできるのである。

```
/* t01.c -- waste CPU by simple loop. */
main() {
  int i = 0;
  while(i < 1000000)
    i = i + 1;
}
```

このプログラムは `main` 関数だけから成り、その中では単に 1,000,000 回足し算を行なうだけである。これをとある計算機 (私が普段使っている) で実行するのにどれくらい掛かると思います?実行してみよう。

```
% cc t01.c
% a.out
% time a.out
1.0u 0.0s 0:01 98% 0+18k 0+0io 0pf+0w
%
```

ちなみに、C のプログラムは「なんとか.c」というファイルに入れる。それをコンパイルするには「cc ファイル名」。すると実行形式に翻訳されたプログラムが a.out というファイルにできるので、これを実行するには「a.out」とファイル名だけをいえばよい。でも、それだけだと腕時計で時間を計らないといけない。「time 指令...」とすれば Unix の方で時間を計測してくれる。これによると、CPU 消費時間約 1.0 秒、システムの CPU 消費時間約 0 秒、経過時間約 1 秒、CPU 使用率約 98%となっている。

さて、それでは同じプログラムを 3 つ同時に走らせたらどうなるだろう (予想は)?結果は次の通り。

```
% time a.out & time a.out & time a.out &
[1] 5687
[2] 5688
[3] 5689
%
[3]  - Done                  a.out
1.0u 0.0s 0:03 33% 0+17k 0+0io 0pf+0w
[2]  - Done                  a.out
1.0u 0.0s 0:03 33% 0+18k 1+0io 0pf+0w
[1]  + Done                  a.out
1.0u 0.0s 0:03 34% 0+17k 0+0io 0pf+0w
%
```

つまり、CPU 消費時間は約 1 秒で同じだが、3 つのプロセスで CPU を取り合うから CPU 使用率は 1/3 ずつになり、経過時間は 3 倍になった。当たり前といえば当たり前である。にしても、Unix はちゃんと 3 つのプロセスに平等に CPU 時間をサービスしていることが分かる。

ところで、例えば「1 秒だけ待つ」のにこういう風にループして待つプログラムを使うのは正しくない。(なぜか?)CPU の無駄使いだし、第一他のプロセスが動いていると所要時間が変わってしまう。そういう時には自分でじたばたするのでなく、OS に頼む。

```
/* t02.c -- wait 1sec by sleep. */
main() {
  sleep(1);
}
```

sleep というのいうのは (sleep 指令と同様)、n 秒間だけ実行を中断して待つ、ということを OS に頼む。こういう風に OS に頼むのを「システムコール」という。さてこれだと。

```
% cc t02.c
% time a.out
0.0u 0.0s 0:01 9% 0+31k 0+0io 0pf+0w
% time a.out & time a.out & time a.out &
[1] 5707
[2] 5708
[3] 5709
%
[3]    Done                  a.out
0.0u 0.0s 0:01 7% 0+18k 0+0io 0pf+0w
[2]  + Done                  a.out
0.0u 0.0s 0:01 7% 0+18k 1+1io 0pf+0w
[1]  + Done                  a.out
0.0u 0.0s 0:01 7% 0+17k 0+0io 0pf+0w
%
```

OS に頼むのだから、CPU 消費時間はゼロ。経過時間はちゃんと 1 秒で、3 つ同時に走らせてもこれは変わらない。

14.2 主記憶管理の実験

ところで、最初のプログラムの「変形版」として、単に数を数えるだけでなく同時に配列を操作する、というのを作ってみた:

```
/* t03.c -- fill an array with '1'. */
int a[1000000];
main() {
  int i = 0;
  while(i < 1000000) {
    a[i] = 1; i = i + 1; }
}
```

このプログラムは単に大きさ 1000000 の a という配列を用意して、そこに 1 をどんどん入れているだけである。これを実行するとどのくらい時間が掛かると思うか?(ヒント:配列のアクセス 1 回はたぶん足し算 1 回とおなじくらいの CPU 時間を消費すると思うよ。)

```
% cc t03.c
% time a.out
1.8u 1.0s 0:03 100% 0+254k 1+0io 2pf+0w
%
```

当たりましたか?実はユーザプログラムの中で CPU を消費している時間は確かにさっきの倍程度なのだが、大きさ 1000000 の配列を含むほどおおきな主記憶領域を用意するのにシステムの中で 1 秒よけいに!掛かっているわけなのである。

このことをもう少し詳しく考えてみる。複数のプログラムを並行して実行するためには、それらのプログラムが主記憶に入っていないといけない。つまり、主記憶を複数のプロセスが並行して利用できるように管理するのが主記憶管理ということになる。(昔のシステムではプロセスが切り替わるたびにその内容をディスクに書き出し、次のプログラムをディスクから読み込む、ということをやったものもあったが、これはひどく遅いので今はやらない。) 主記憶に複数のプログラムが詰め込まれている様子を図 3 に示す。

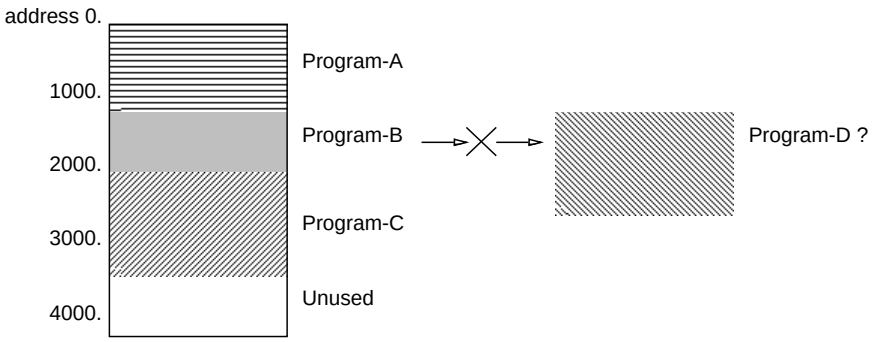

図 3: 主記憶管理の問題

ここで 2 つ問題がある。まず、主記憶には 0 番地から順に番地が振ってあって、プログラムの命令には「何番地の内容をアクセスする」とか「次は何番地の命令を実行する」などと書かれている。ところがプログラム A は 0 番地から始まっているからいいが、B と C はそれぞれ前のやつの次から始まっている。だから動かし始める時に例えば「全ての番地を 1200 ずつずらす」といった修正が必要になってしまう。

もう一の問題は、例えば B が早く終ったので次は D を入れようと思っても、B のいたすき間には入らないのでどうするか、というものである。C が終るまで待つ、というのも無駄な話だし、かといって動いている C を止めて場所を移すのも大変だし、動き始めてしまったプログラムの場所を移動するのは困難である。昔の OS がこれらの問題に悩んでいた、というのは本当の話である。

そこで今ではどうしているかというと、図 4 のように主記憶を決まった大きさのページに分割し、プログラムを読み込む時は空いているページを取っては適当に入れる。従って場所的にはばらばらに入ることになるが、別にプロセスごとそのプロセスの何番地から何番地はどのページか、を記録したページマップというものを設ける。こうすればどのプロセスもめでたく 0 番地から始まることができるし、主記憶がばらばらに空いてもページ単位で新しいプログラムもばらばらに入れられる。このようなやり方を「ページ方式」と呼ぶ。そうすると、大きなプログラムを動かすためにはまずそのプログラムが必要とするだけのあきページをさがしてきて、次にページごとの「矢印」をこれらのページを指すようにすべて設定する、という仕事が必要である。OS はそのために時間を食っていたのである。だから必要とする主記憶が倍になると、OS の仕事も倍の時間が掛かると予想される。

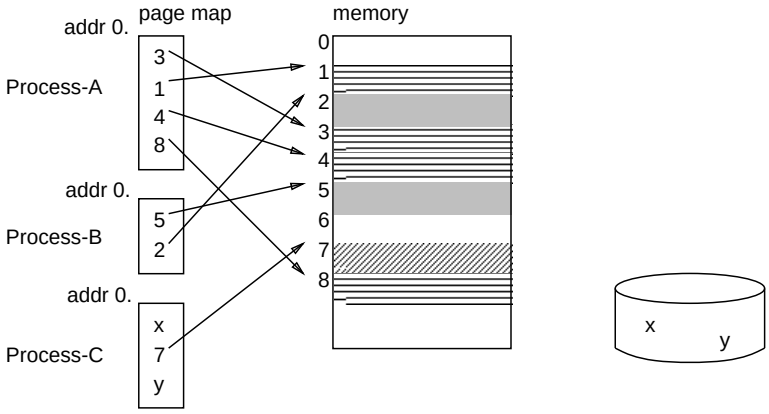

図 4: ページ方式

14.3 仮想記憶の実験

ところで、さっきのプログラムを、最初のプログラムと同様に 3 つ同時に走らせたときの所要時間はどのくらいだと思うか?

```
% time a.out & time a.out & time a.out &
[1] 19001
[2] 19002
[3] 19003
%
[3]  + Done                    a.out
2.1u 0.9s 0:14 21% 0+200k 0+0io 4pf+0w
[2]  + Done                    a.out
2.0u 1.1s 0:14 21% 0+198k 0+0io 5pf+0w
[1]  + Done                    a.out
1.8u 1.2s 0:14 21% 0+195k 0+0io 3pf+0w
%
```

CPU 時間とシステム時間は同じだが、経過時間が 3 倍でなく 5 倍になっている。これはなぜだろう?

ところで、図 4 をよく見ると、プロセス C のページが 2 つほどディスクに入っているように描いてある。これはどういうことかというと、このプロセスのプログラムが 0 番地付近を触ると、そのプロセスは一時中断されて OS に実行が渡り、OS はディスクからページ x を読み出して主記憶のどこかに置き、ページマップもそこを指すように直す。その後再びプロセス C が再開されると、無事その番地がアクセスできる。

このように「触った時なければ持ってくる」やり方を「デマンドページング」という。そして、この方法によれば実は一度に走っているプログラムの大きさの合計が主記憶より大きくてもいいし、それどころかそれ自身だけで主記憶より大きなプログラムも実行させられる!こういうのを「本当は主記憶がそんなにあるわけではないのに、仮想的にあるかのように働かせられる」ので「仮想記憶」と呼ぶ。しかし注意しなければならないのは、本当に主記憶で足りない部分はディスクが使われるわけだから、そうなると一気に実行速度が遅くなってしまうことである。

だから上で経過時間が増えたのは、配列 1000000 個ぶんのプログラム 1 個なら主記憶に入り切ったのに 3 つだと入り切らなくなり、ディスクが使われ始めたということを示しているわけである。その様子をもっとよく観察するには、配列の数をいくつか変えながらプログラムを動かし時間を計ってグラフを描いてみればよい。その時毎回プログラムを編集するのでは煩わしいので、次のようにプログラムを直す。

```
/* t03.c -- fill an array with '1'. */
int a[SIZE];
```

```
main() {
  int i = 0;
  while(i < SIZE) {
    a[i] = 1; i = i + 1; }
}
```

この SIZE という所をいろいろ変えればいいのだが、それは翻訳のときに指定することができる。実際やってみよう。

```
% cc t03.c -DSIZE=100000
% time a.out
0.1u 0.2s 0:00 115% 0+38k 0+0io 0pf+0w
% cc t03.c -DSIZE=500000
% time a.out
0.9u 0.5s 0:01 108% 0+131k 4+0io 4pf+0w
% cc t03.c -DSIZE=1000000
% time a.out
1.8u 1.1s 0:03 97% 0+254k 0+0io 0pf+0w
% cc t03.c -DSIZE=2000000
% time a.out
3.7u 2.0s 0:05 98% 0+507k 1+0io 1pf+0w
% cc t03.c -DSIZE=3000000
% time a.out
5.7u 2.8s 0:13 62% 0+709k 0+0io 2pf+0w
% cc t03.c -DSIZE=4000000
% time a.out
7.4u 4.0s 0:23 49% 0+821k 3+0io 11pf+0w
%
```

この様子をグラフに描いたのが図 5 である。これを見ると、配列の大きさが 2000000 のところが折れ目になっていて、そこから先では経過時間が急激に増えるのがよく分かる。実はこのマシンは主記憶が 8MB ついているのであった。整数 1 個はこのマシンでは 4 バイトであるから、この 2000000 というのはちょうど主記憶に入る上限くらいということになり、よくつじつまが合う。もちろん、こうやってシステムを「いじめる」のはあくまでも「実験」のためであり、現実の仕事の時にはけっしてこの領域に立ち入らないように注意を払う必要がある (さもないと、いつまでもプログラムの実行が終らない、という事態になる)。もう一つ注意すべきことは、自分一人で計算機を使っているのでない限り、自分の仕事だけなら入り切る場合でも 2 名以上の合計では「いじめ」の領域に入ってしまうこともある点である。そして最後に、当たり前であるが、この実験は他人が仕事をしているマシンでは絶対やらないこと (あなたが「いじめ」られるはめになりますよ!)。

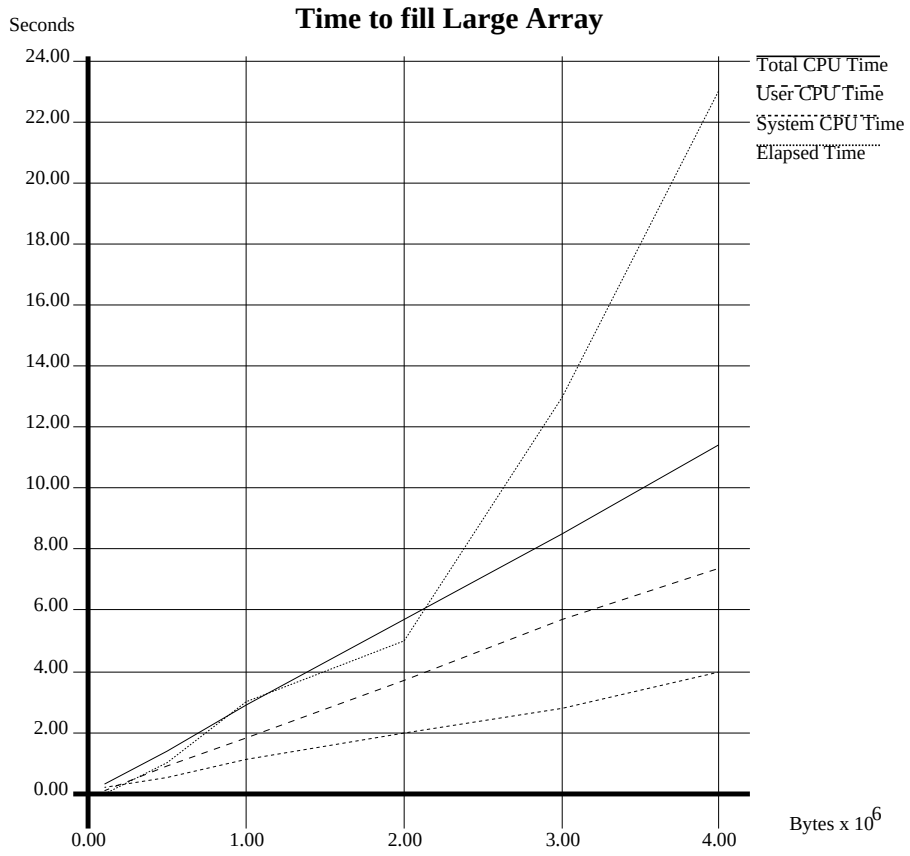

図 5: 仮想記憶いじめのグラフ

A　演習および課題

A.1　12 節の演習

といっても、12 節は C のイントロばっかりなので演習になる材料が全くない。なので代わりに、(本当は本文で述べるべきだったと思う) 参考書の紹介をしておこう (気が向いたら買って読むのが演習というわけ)。

C に関する本の「定番」はなんといっても

カーニハン他、石田訳、「プログラム言語 C」第 1 版/第 2 版、共立出版

である。これは C を作った本人である Ritche が共著者ということで定番なわけである。なお、第 1 版は Unix の C、第 2 版は ANSI 規格の C に準拠している。しかしやはり第 2 版の方が新しい分だけよく書けていると思うので、多少の言語仕様の違いががまんできれば第 2 版をおすすめする。

もう 1 つの推薦は

椋田、「はじめての C」、技術評論社

であるが、これは一人でどんどん読んで自習するのに向いていると思う。それ以外でも気に入った本があれば C そのものがそう違う訳ではないので、お好きなのをどうぞ。

A.2　13 節の演習

本文でも書いたが、この演習の内容には他の人が同時に同じマシンを使っているとうまく計測できないもの、それどころかその人に迷惑を掛けてしまうものが含まれている。その点を注意深く配慮しながらやること。

13-1. あなたが使っているマシンで、最初の 3 つの例題を動かしてみよ。私が実行例を作成したマシンと比較して速度はどうか? [4]

13-2. この節で取り上げた例題は整数の加算と配列のアクセスばっかりであった。ところで、減算、乗算、除算などは加算とくらべて、あるいは配列アクセスとくらべてどのくらい速い/遅いか、実測してみよ。また整数でなく実数の場合だとどうか? [5]

13-3. C コンパイラのオプションとして「`-S`」を指定すると、実行ファイルを作る代わりにコンパイル結果のアセンブラ語出力を作らせられる。これで今回取り上げたようなプログラムがどのように翻訳されるかを観察し、それを確認するためそのアセンブラ出力をエディタでちょっと手直ししてから実行形式にして動かし、その効果を見てみよ。[6]

13-5. C コンパイラのオプションとして「最適化」を指定できることが多い。マニュアルでオプションの指定方法を調べ、上で計ったようなプログラムに関して、最適化を掛け、その場合プログラムの実行速度はどのくらい変化するかを調べ、それがどのような最適化によるものか推測せよ。できれば最適化した/しない場合両方のアセンブラ語出力を比較してそれを検証せよ。

13-6. あなたが普段使っているマシンではどの位の主記憶が利用可能であるかをまず予測し、続いて `t03.c` を使って実際に調べてみよ。必ずグラフを描いてみせること。

---

[4] マシンによっては 1000000、というのは変更しないと動かないことがあるかも知れない。

[5] 本当はループの中には比較命令と条件分岐命令も含まれていることをも考慮すべきですよねえ?

[6] アセンブラファイルは最後に`.s` がついた名前となる。これを実行形式にするにはまた「`cc t01.c`」のように cc を使えばよい。

13-7. `t03.c` のプログラムを配列の大きさ n で m 個同時に走らせた場合どのくらいの経過時間を要するか予測する「公式」を作ってみよ。いくつかの例によってその公式を検証してみよ。[7]

A.3 本日の課題

本日の課題は、13-1.〜13-7. のうちから 2 問選択とします。くれぐれも善良な他ユーザに迷惑を掛けないように。この講義の聴講者の間で戦うぶんには構いませんが。

[7] たぶん、前問で作ったグラフが役に立つはずである。