第2章　計算機システムのソフトウェア構造

前章で、「ビット列を加工する装置」としての計算機ハードウェアについて、一番下のゲートレベルからシステムのレベルまで一通り説明した。しかし、計算機はハードウェアだけでは用をなさず、ソフトウェアを動かすことではじめて役に立つことができる。そこで今度はふだんあなたが目にしている、ソフトが動いている計算機からはじめて内側へ向かって見ていくことにしよう。

2.1　アプリケーションソフトと基本ソフト

あなたがふだんソフト (一太郎とか 1-2-3 とかが多いかもしれないが、ここではとりあえず Unix でニュースでも読んでいるものとしよう) を使っている時、計算機の上で動いているのはそのプログラム (例えばニュースを読むプログラム — ニュースリーダ) だけだろうか? PC の場合も、新しく入れた機械のためにワープロソフトだけを買おうとしたら「DOS も買ってください」と言われたりするでしょう? つまりソフトウェアには次の 2 種類があるというわけである:

- **アプリケーションソフト:** ユーザがやりたい仕事を実際に実行してくれるソフトをいう。
- **基本ソフト:** アプリケーションソフトを使って仕事をする上で必要な手助けとなるソフトをいう。

具体的には、どのような手助けが必要なのだろうか? それはこれから徐々に見ていただくことにして、ここではとりあえず基本ソフトを次の 3 種類に分類しておく。

- OS(オペレーティングシステム): アプリケーションが動く下ささえとなる機能を提供する。
- 言語処理系: プログラムを作成することを手助けする。
- ユティリティ: ファイルの操作やデータ形式の変換など、汎用的な作業を手助けする。

分類する人の立場によっては言語処理系をユティリティに含めたり、ユティリティの中をさらに細かく分けるかも知れない。以下本章ではまず OS についてもう少し細かく検討し、次に言語処理系を取り上げる。ユティリティについては後の方でそのつど扱う。

2.2　オペレーティングシステム

2.2.1　OS の各種の役割り

上でオペレーティングシステムの役割りは「アプリケーションが動く下ざさえ」と述べたが、それは具体的にはどういうことだろう? もう少し具体的に考えてみよう。

前章で見たように、計算機のハードウェア命令というのはメモリとレジスタの間でデータを転送したり、四則演算をしたりといったごく低いレベルの機能しか提供していない。だから多くのプログラムで必要とするような、キーボードを制御して文字列を読み込んだり、2 進表現のビット列を我々がふだん使っている 10 進表現に変換して画面に表示したりといった機能を実現するのにかなり長い命令列 (プログラムの断片) を必要とする。それを各アプリケーションを作る人が個別に用意するのでは労力の無駄だし、普通のアプリケーションプログラマは入出力機器の制御方法など知らないのが普通である。従って

- 多くのプログラムが必要とする標準的機能を一括して提供する

ことは OS の重要な役割りである (図 2.1)。

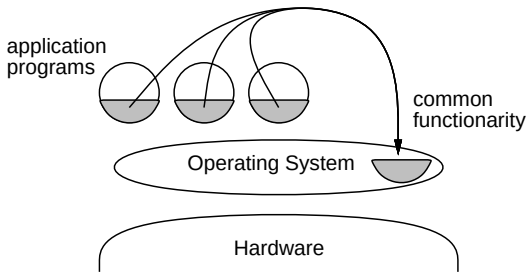

図 2.1: OS による標準的機能の提供

次に、現在の計算機システムでは複数のプログラムを並行して動かす (マルチタスク) 機能が普通に見られる。現に皆様は同じ計算機 (smf とか smb とか) に接続して同時に演習をしていますね? このように

- 複数のプログラムが並行して動作するのを管理する

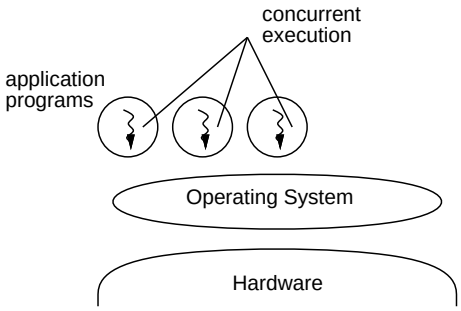

図 2.2: OS によるマルチタスク機能の提供

ことも OS の役割りである (図 2.2)。

そうなってみると、あるプログラムを動かそうと思ったときいきなり計算機を止めてあなたのプログラムをメモリに書き込み始めるわけにはいかない。そんなことをすると他人の仕事がめちゃくちゃになってしまう。(大体どうやって書き込むというのだろう?) だから、

- ユーザが指定したプログラムを読み込んで実行開始させる

というのも OS の基本的な役割りである (図 2.3)。

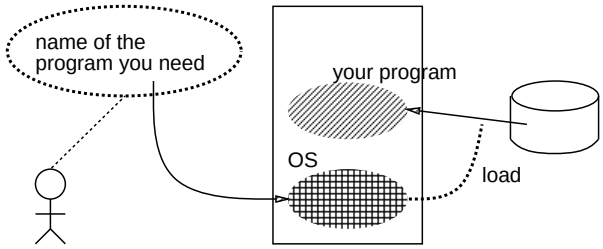

図 2.3: OS によるプログラムの実行開始

さて次に、そうやって並行して動いている複数のプログラムがたまたま同時に同じメモリ番地やディスク上の領域を使おうとしたらやっぱり大混乱になる。また、それらが一斉にプリンタに出力しようとして、混ざった出力が出てきても困るだろう。だから

- 計算機のメモリを各プログラムにうまく割り当てて調整する
- 入出力装置に対するアクセスを管理する

というのも重要である。

ところで、見かたを変えると計算機に備わっている入出力装置、メモリ、そして CPU などは数が限られた貴重な「資源」である。だから、OS の機能のうち、マルチタスク管理、メモリ管理、入出力管理などは統一して

- 計算機内部の各種資源を管理する

ものと考えることができる。言い替えれば、あなたが動いている計算機を目にする時、そこには常に OS が動いていて、ハードウェアと混然一体となってすべてを管理している。そしてあなたやあなたのプログラムが計算機を使おうとする時、必要な資源のすべては OS が管理してくれていて、OS に頼むことによってはじめてそれらを利用して仕事ができるわけである (図 2.4)。

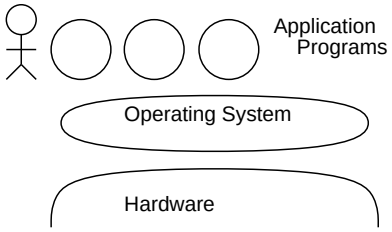

図 2.4: OS とユーザ、アプリケーションの関係

2.2.2　マルチタスクとプロセス

さて、OS には上に述べたように様々な機能が備わっているが、まずはその中で一番目だつ機能であるマルチタスク、つまり複数のプログラムを並行して走らせる機能について見てみることにする。そもそも、どのようにしてそんなことが可能になるのだと思いますか?

まず、計算機の機能をごく簡単化して復習してみよう。メモリの上には命令の列 (プログラム) が置かれていて、CPU はプログラムカウンタが指している番地から順に命令を取り出しては実行して行く (図 2.5)。

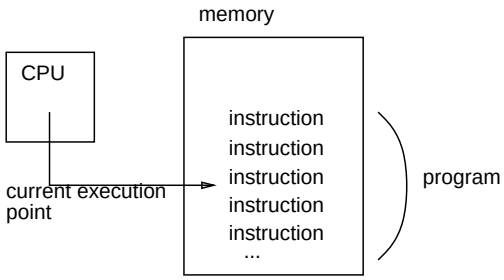

図 2.5: CPU による命令の実行

そこで、複数のプログラムを並行して動かすには、それらをメモリに一緒に入れておく。一方、CPU に「時計」(前章で述べた、ゲートを制御するク

ロックとは別のもの) をつけておく (図 2.6)。そして、時計を動かした状態でまずはプログラム A の実行を始める。時計は一定時間たつと CPU に信号を送り、CPU は信号を受け取るとプログラム A の実行を一時中断してプログラム B の実行に切り替わる。またしばらくするとプログラム C に、そして次は A に戻る。このようにして、複数のプログラムが実は「小刻みに切り替わりながら」実行されるのだが、CPU は非常に高速なのでユーザにとってはすべてのプログラムが同時に動いているようにしか見えない。

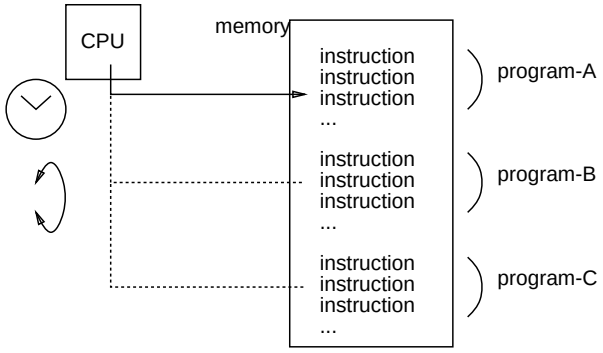

図 2.6: マルチタスク機能の実現

より厳密には、「プログラム」のうち 1 つは OS であり、時計から信号が来るとまず OS の実行に切り替わる。OS は各プログラムの使用時間の割り当てや優先順位を調べて次に実行すべきプログラムを選択し、それの実行を開始させる。だから OS によって各プログラムの CPU 割り当ては自由に制御できるわけである。

Unix 用語ではこの「動いている状態のプログラム」のことを「プロセス」と呼ぶ。(IBM 用語ではタスク。またアクティビティなどと呼ぶ文明もある。) たとえば 10 人の人が同時にある計算機で Mule を使っていれば、プログラムは 1 つだがプロセスは 10 あることになる。

ところで、計算機によっては CPU が複数ついているもの (マルチプロセサという) もある。たとえば smb は 4CPU のマルチプロセサである。しかしそのようなシステムでも、動かしたいプログラムの数 (プロセス数) は CPU の数よりずっと多いのが普通だから、上で述べたような小刻みな切り替わりがあることに変わりはない。

最後になったが、このように沢山プロセスが作れることはどういう利点があると思うか?

- 複数の端末をつないで、多人数で同時に使える
- 一人で複数の仕事を並行してこなせる
- 自分に代わって何かを監視するプログラムが動かせる

- 決まった時間になったらあることをする、というのができる
- あることをするために別のことをやめなくてもいい

もちろん、一つのプログラムに（聖徳太子みたいに）沢山のことをやらせるのはがんばれば可能である。しかしそんなことで苦労するより、沢山プロセスを使ってそれぞれに簡単な仕事をするプログラムを走らせる方が作るのも管理するのも楽である。

2.2.3　ps — Unix でのプロセス観察

Unix では、「ps」というプログラムによって現在動いているプロセスを観察することができる。ps に与えるパラメタによって、表示するプロセスの範囲や詳しさを制御できる。

ps　　--- 現在使っている端末 (窓) から起動した自分のプロセスの表示
ps x　--- 〃、OS の版によっては自分のプロセスすべての表示になる
ps ax　--- 他人のものも含めてすべてのプロセスを表示
ps lax --- 〃、ただしより詳しい表示
ps uax --- 〃、ただし CPU 使用量の多い順に、ユーザ名つきで表示
ps vax --- 〃、ただしメモリ使用量の多い順に表示

たとえば smb で ps ax を実行した結果を次に示す。

```
  PID TT       S  TIME COMMAND
    0 ?        T  0:00 sched
    1 ?        S 17:32 /etc/init -
    2 ?        S  0:10 pageout
    3 ?        S 200:59 fsflush
  135 ?        S  0:49 /usr/sbin/rpcbind
  137 ?        S  0:02 /usr/sbin/keyserv
  142 ?        S  0:00 /usr/sbin/kerbd
  151 ?        S  0:27 /usr/sbin/inetd -s
  158 ?        S  2:28 /usr/lib/autofs/automountd
  162 ?        S  0:00 /usr/lib/nfs/statd
  164 ?        S  0:01 /usr/lib/nfs/lockd
  177 ?        S  0:03 /usr/sbin/syslogd
  187 ?        S  0:13 /usr/sbin/cron
  197 ?        S  0:08 /usr/lib/lpsched
  205 ?        S  0:00 lpNet
  206 ?        S  0:05 /usr/lib/sendmail -bd -q1h
  214 ?        S  0:00 /usr/sbin/cssd
  215 ?        S  0:00 /usr/bin/sh /usr/lib/css.d/atok7.sh
  217 ?        S  0:00 /usr/bin/sh /usr/lib/css.d/ccv.sh
  218 ?        S  0:00 /usr/bin/sh /usr/lib/css.d/kkcv.sh
  219 ?        S  0:00 /usr/bin/sh /usr/lib/css.d/cs00.sh
  220 ?        S  0:00 /usr/sbin/atok7
  221 ?        S  0:00 /usr/sbin/ccv -f
```

```
  223 ?        S  0:00 /usr/sbin/kkcv -f
  224 ?        S  0:00 /usr/sbin/cs00
  226 ?        S  0:12 lpNet
  232 ?        S  0:10 /usr/sbin/vold
  237 ?        S  0:31 /etc/local/replpasswd
  262 ?        S  0:00 /usr/lib/nfs/nfsd -a 16
  264 ?        S  0:30 /usr/lib/nfs/mountd
  265 ?        S  0:07 /usr/lib/saf/sac -t 300
  266 console  S  0:00 /usr/lib/saf/ttymon -g -h -p smb console login:  -T sun
  268 ?        S  0:09 /usr/lib/saf/ttymon
 5045 ?        S  0:00 /usr/local/X11R6/bin/xdm -config /usr/local/X11R6/lib/X
 5053 ?        S  0:05 kterm -geom 80x48+300+100 -T console -n console -e bash
 5060 ?        S  0:10 oclock -transp -geom 100x100-0+0
 5061 ?        S  0:09 twm
 5069 pts/8    S  0:00 bash
 7779 ?        S  0:00 /usr/local/X11R6/bin/xdm -config /usr/local/X11R6/lib/X
 8219 ?        S  0:09 /usr/local/X11R6/bin/xdm -config /usr/local/X11R6/lib/X
 8822 ?        S  0:12 /usr/local/X11R6/bin/kterm -T 2:smb -n 2:smb -d smr03:0
 8823 pts/11   S  0:01 /usr/local/bin/bash
 9077 pts/11   O  0:00 ps ax
 9485 ?        S  0:06 /sbin/ewhod utogw
14998 ?        S  0:01 lpNet
19042 ?        S  0:12 lpNet
26192 ?        S  1:56 /usr/local/canna2.2/bin/cannaserver
```

これらのうち、pts/8 などのように端末番号が記されているプロセスがおもにユーザが使っているもので、他のプロセスは大部分、システムのさまざまな作業を担っている。このように、Unix では複数のユーザが自分のために複数のプロセスを駆使しているのに加え、システム自体の運用のために多数のシステムプロセスが動いているのが普通である。

2.2.4 プロセスの新規生成

ではさっそく、プロセスを 1 つ作って見よう。

```
% ps x
   PID TT       S  TIME COMMAND
  9091 pts/11   O  0:00 ps x
  8823 pts/11   S  0:01 /usr/local/bin/bash
% xclock -a -u 1 &
[1] 9092
% ps x
   PID TT       S  TIME COMMAND
  9093 pts/11   O  0:00 ps x
  9092 pts/11   S  0:00 xclock -a -u 1
  8823 pts/11   S  0:01 /usr/local/bin/bash
%
```

「xclock…」を実行すると秒針つきの時計が画面に現われ、秒針が動いているのが見える。「ps x」で見ると確かに xclock というプロセスが増えているのが分かる。

実はふだんお世話になっている mule やコマンドの窓も同様にして作ることができる。

```
% mule &
[2] 9101
% kterm -e bash &
[3] 9102
% ps x
   PID TT       S  TIME COMMAND
  9092 pts/11   S  0:00 xclock -a -u 1
  8823 pts/11   S  0:02 /usr/local/bin/bash
  9102 pts/11   S  0:00 kterm -e bash
  9110 pts/11   O  0:00 ps x
```

ここで^X^C で mule を終了させたり、exit でコマンドの窓を終わらせたりすれば対応するプロセスも消滅する。このように、Unix ではこれまでの仕事と並行してなにかをさせるには、新しいプロセスを作ってそれにまかせるのが自然かつ簡単な方法である。

2.2.5　kill — プロセスの操作

ps の表示には必ず PID(プロセス ID) と呼ばれる番号が含まれる。これはプロセスの固有番号で、これを指定することによって自分のプロセスをいろいろに操作することができる。操作は kill 指令によってプロセスに各種のシグナルを送ることで行える。

kill -STOP プロセス ID	プロセスの実行を一時凍結する
kill -CONT プロセス ID	凍結したプロセスの実行を再開する
kill -TERM プロセス ID	プロセスに「終わってほしい」と信号する
kill -KILL プロセス ID	プロセスを強制終了させる。

なお、2 番目のパラメタを省略すると-TERM が送られる。また、標準設定ではコマンドを実行中に^C と^Z を押すとそのコマンドを実行しているプロセスに-TERM および-STOP シグナルがそれぞれ送られる。

2.2.6　プロセスの生成、コマンドインタープリタ

実はプロセスには「親子関係」がある。これは、どのプロセスがどのプロセスを生成したか、という関係のことである。例えば先の例で今度は「ps lx」を実行させてみる：

```
% ps lx
 F UID   PID PPID CP PRI NI    SZ RSS    WCHAN S TT       TIME COMMAND
 8  20 9092 8823 24  51 20   437 318 pollwait S pts/11   0:00 xclock -a -u
 8  20 8823 8822 80   0 20   265 223  ptm_dip S pts/11   0:02 /usr/local/bi
 8  20 9151 8823 14   0 20   206 173          O pts/11   0:00 ps lx
 8  20 9102 8823 30  20 20   494 400 pollwait S pts/11   0:00 kterm -e bash
```

この中の PID と PPID(Parent PID) を見てみると、あとから作った 3 つのプロセスの親は最初からある bash のプロセスになっている。言い替えれば bash のプロセスが ps その他のプロセスを生成している。

これは何を意味するだろう。実はあなたや私がキーボードから指令を打ち込むとそれは bash というプログラムによって読みとられる。bash はその文字のならびを見て、その内容に応じて求められているプログラムを走らせる (ということは、プロセスを生成する)。この様子を図 2.7 に示す。このように、利用者から指令を表す文字列を受けとってその内容に応じて内部の動作を起動するプログラムをコマンドインタープリタと呼ぶ。

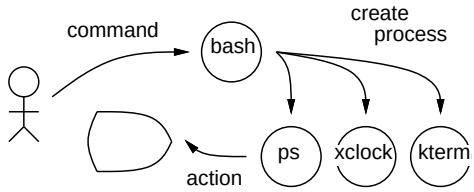

図 2.7: コマンドインタープリタ

コマンドインタープリタは Unix を使う上で欠かせない OS の一部ではあるが、一方でこれまでに見てきた、計算機が動き始める時まずメモリに読み込まれる OS の中核部分 (カーネル) とは違って、普通のユーザプログラムと同様にプロセスとして実行される。このように、メモリに常駐しなくてもよい部分はカーネルとは分けてプロセスとして実行させることでシステム全体の見通しをよくしているのも Unix の特徴である。(ps ax を実行したとき表示された多数のシステムプロセスも同様に Unix の機能の一部を担っているわけである。)

ところでさっきから毎回 ps を実行するごとに、その PID が違っていることにお気づきだろうか? つまりさっきの ps のプロセスは毎回実行が終れば消えてしまい、必要のつど新たに作られる。一方 bash そのものは同じままなわけである。この様子を図 11 に示す。つまり、bash はずっと動いたままで、利用者が指令を打ち込むたびに新しいプロセスが bash によって作られる。bash が終るのは、利用者が exit という特別な指令を打ち込んだ時だけである。(ということは、exit というのは他の指令のように新しいプロセスとして実行されるのではなく bash 自身によって実行されることになる。このような「特別な」指令がいくつか存在する。)

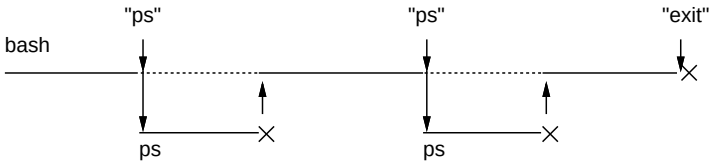

図 2.8: bash によるコマンドの発行と待ち合わせ

ときに、図で点線のところは、bash が子供のプロセスの完了を待っていることを意味する。これは、普通利用者は一つの指令を打ち込んだらそれが終るのを待ってから次の指令を打ち込むだろうから、正しいあり方だといえる。でも指令がとても時間が掛かるようなものの場合には待っていたくないかも知れない。

そういうときは指令の最後に「&」をつけることで、「待たずにすぐ次の指令をやるよ」という指定ができる。さっき時計や mule などの窓を作るとき最後に&のついたコマンド行を使ったのはそういう意味だったのである。逆にいえば、&をつけるから新しいプロセスができるのではなく、いつでも新しいプロセスはできるのだが&をつけないとそのプロセスが終わるまで待つので次の指令を打つときにはもうそのプロセスはなくなっている、というだけのことだったのである。

2.3　言語処理系と機械語

2.3.1　プログラムはどこから来るか?

前節で新しくプログラムを実行開始させる (つまりプロセスを作る) ことについて説明したので、今度はそのプログラムをどうやって作るかについて考えてみることにする。

前章で述べたように、メモリに入っていて CPU が実行するプログラムは CPU の構造に合わせたビットの列であるが、人間がいきなりこのビットの列 (機械語と呼ばれる) を書き下して作るというのは相当無理がある。(1 箇所でも 0 と 1 を間違うととんでもないことになったりする。)

それでも計算機が遅くてメモリも貴重だった頃には一生懸命命令の使い方を工夫して機械語プログラミングをしたこともあったが、現在では Fortran とか Pascal とか C などの、人間に読めるような書き方の言語 (高水準言語) でプログラムを書くのが普通である。

例えば次は単に画面に 10 回、「Hello.」と打ち出す C のプログラムである。

```
/* t00.c -- say 'Hello.' 10 times. */

main() {
```

```
    int i = 0;
    while(i < 10) {
      printf("Hello.\n");
      i = i + 1; }
}
```

さて、この C のプログラムを計算機で実際に走らせるにはどういう過程を経ればいいのだろうか?

一般に計算機システムには高水準言語のプログラムを「翻訳」 (コンパイル、ともいう) して計算機が実行可能な機械語のプログラムに変換するようなプログラム (コンパイラ、という) があらかじめ用意されている。そして、低水準言語 (アセンブリ言語 — 後述 — または機械語) で書かれたプログラムが特定の CPU でしか動作しないのに対し、高水準言語で書かれたプログラムはコンパイラさえあれば様々な CPU の上で動かすことができる。

我々のシステムでは、C のソースプログラム (翻訳する前のもとのプログラムという意味) を「なんとか.c」という名前のファイルに入れておき、gcc という指令をこのファイル名を指定して起動すると、翻訳結果の実行可能な機械語プログラムを a.out というファイルに入れてくれる。

この機械語プログラムを実行させるには単にそのファイル名を言えばよい。例えば上のプログラムが t00.c というファイルに入っていたとすると次のような具合である。

```
% gcc t00.c
% a.out
Hello.
Hello.
...
Hello.
%
```

ときに、ソースプログラムとその翻訳結果である実行形式プログラムはどっちが大きいと思うか?

```
% ls -l t00.c a.out
-rwxr-xr-x  1 kuno   7512 Apr 25 13:06 a.out
-rw-r--r--  1 kuno    122 Apr 25 11:42 t00.c
%
```

予想は当たりましたか? また、なぜそうなのだろう? 答はすこし後で。

2.3.2 コンパイラの中身

実は gcc という指令はより正確にはコンパイラドライバと呼ばれ、次のような順で仕事を進める。

1. まず、指定された C ソースをまず、定数やライブラリ宣言のための前処理プログラム (プリプロセサ) に通す。

2. その出力を「ほんものの」コンパイラで翻訳し、アセンブリ言語ソースにする (アセンブリ言語とは機械語とほぼ対応していて、ただし命令や番地などを 2 進や 16 進のかわりに名前で書けるため人間にとっては機械語より読みやすい形式である)。

3. アセンブラを起動して、アセンブリ言語ソースを再配置可能な (つまり具体的にはまだ何番地に置かれるか決まっていない) 機械語ファイル (オブジェクトコードと呼ぶ) に変換する。

4. リンカを起動して、複数のオブジェクトファイルをまとめ、必要なライブラリルーチンがあればそれもいっしょに入れて、それぞれの番地を決定し、最終出力である実行可能な機械語ファイルにする。

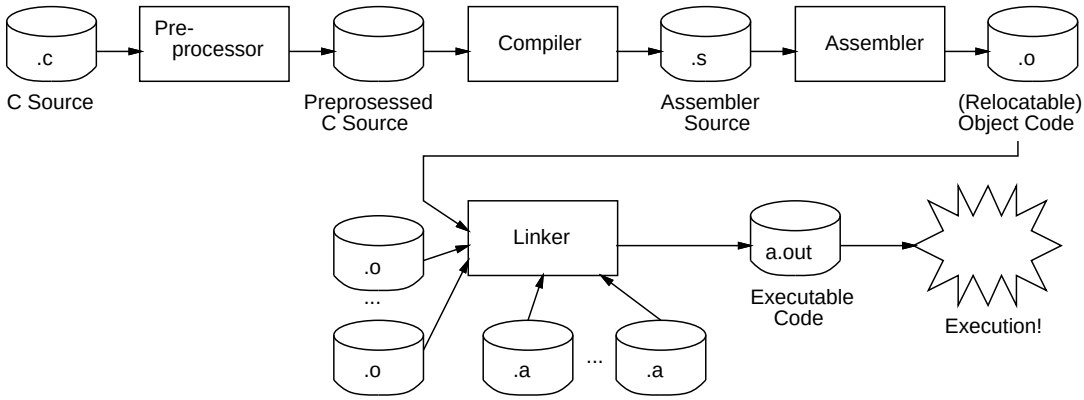

図 2.9: ソースコードから実行形式までの道のり

この様子を図 2.9 に示した。ここでいくつかの疑問があると思うのだが、特に「なんでこんなに複雑なことをするのか? なんで、いきなりコンパイラが実行可能ファイルにしないのか?」というあたりが典型的かと思う。皆様はどう思われますか。

2.3.3　アセンブリコード

さて、それではさっきのように「一気に」機械語にしてしまう代わりに、図 2.9 の順番を追って見てみよう。マシンとしては、CPU が同じ SPARC アーキテクチャ(というのは「方式」のことだと思っていただければよい) である smb/smf/smg/smd を対象とする。

まずプリプロセサであるが、 `t00.c` には特に定数定義やライブラリのための宣言がないのでプリプロセサを通しても何も変化がないため、略す。次はコンパイラだが、 `gcc` に `-S` というオプションをつけるとアセンブリ言語ソース「なんとか`.s`」ができたところで止まるようになっているので、これを利用してアセンブリコードを見てみよう。

```
% gcc -S t00.c
% cat t00.s
        .file   "t00.c"
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "Hello.\n" ← 引数文字列
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:                       ← ここが main 先頭
        !#PROLOGUE# 0
        save %sp,-120,%sp   ← 入口処理
        !#PROLOGUE# 1
        st %g0,[%fp-20]     ← i = 0
        nop
.LL2:
        ld [%fp-20],%o0     ← i をレジスタ o0 に
        cmp %o0,9           ← 9 と比較
        bg .LL3             ← より大なら LL3 へ
        nop
        sethi %hi(.LLC0),%o1  ← 引数セット
        or %o1,%lo(.LLC0),%o0
        call printf,0       ← printf を呼ぶ
        nop
        ld [%fp-20],%o0     ← i を o0 に
        add %o0,1,%o1       ← 1 足す
        st %o1,[%fp-20]     ← o0 を i に戻す
        b .LL2              ← LL2 へ戻る
        nop
.LL3:
.LL1:
        ret                 ←
        restore             ← 終り処理
.LLfe1:
        .size    main,.LLfe1-main
        .ident  "GCC: (GNU) 2.5.7"
%
```

どうですか、結構読めるでしょう? でも、やっぱりこんなのを書くよりは C で書いた方がずっと楽だし分かりやすいのも確かである。

2.3.4 オブジェクトコード

さて、次はこれをオブジェクト形式にする。それには cc に-c というオプションをつけると、「なんとか.o」というファイルにオブジェクト形式を書き出した段階で止まる。なお、入力の方も「なんとか.c」だとプリプロセサから始まるが、「なんとか.s」だとアセンブラから始まるようになっている。ところで、オブジェクト形式ファイルは機械語 (バイナリファイル) だから cat などで打ち出すわけに行かない。そういう時は od -x で中身を 16 進で打ち出させることにする。

```
% gcc -c t00.s
% od -x t00.o
0000000 7f45 4c46 0102 0100 0000 0000 0000 0000
0000020 0001 0002 0000 0001 0000 0000 0000 0000
0000040 0000 0198 0000 0000 0034 0000 0000 0028
0000060 0008 0001 002e 7368 7374 7274 6162 002e
0000100 726f 6461 7461 002e 7465 7874 002e 7379
0000120 6d74 6162 002e 7374 7274 6162 002e 7265
0000140 6c61 2e74 6578 7400 2e63 6f6d 6d65 6e74
0000160 0000 0000 0000 0000 4865 6c6c 6f2e 0a00
0000200 9de3 bf88 c027 bfec 0100 0000 d007 bfec
0000220 80a2 2009 1480 000b 0100 0000 1300 0000
0000240 9012 6000 4000 0000 0100 0000 d007 bfec
0000260 9202 2001 d227 bfec 10bf fff5 0100 0000
0000300 81c7 e008 81e8 0000 0000 0001 0000 0000
以下略…
%
```

左側の 7 桁がファイル先頭からの位置 (8 進! 不便だが `od` はもともと Octal Dump の意味)、その右のが中身である。最初の方にヘッダ (このファイルのどこに何が書かれているかの記述) があり、0200 の行から命令が入っている (smb の場合。smf/smg/smd ではヘッダがもっとずっと短い)。ほとんどちんぷんかんぷんだが、「0100 0000」というのが目につく。実はこれが nop 命令 (何もしない命令) なので、先のアセンブリ言語ソースで nop のある場所とつき合わせていくとどれがどの命令か判る (なお SPARC では命令はすべて 32 ビットである)。

nop 命令がこれほど多い理由は、SPARC では分岐命令の直後の命令は分岐に先立って実行されるので、そこに nop を入れておかないと思わぬ命令が実行されてしまうからである。実は nop でなくもっと役に立つ命令を入れるようにすれば CPU をより効率的に使うプログラムができる。コンパイラに最適化 (とは生成コードの改良のこと) を行うよう指定すれば、そのようなコードが出る。

2.3.5　実行可能形式とライブラリとリンカ

上で述べたように、再配置可能オブジェクトはそのままではまだ番地が確定していないので、これをリンカに通して番地を割り当ててはじめて実行できる機械語になる。なぜそんな面倒なことをするのだろうか?

それは、たとえば非常に大きなプログラムを作成する場合、それをいくつかの主要なファイルに分けて別々に翻訳するためである。実行形式は各ファイルに対応する再配置可能オブジェクトを集めて組み立てるだけなので、どれか 1 つのファイルを修正した場合、そのファイルだけ翻訳し直して再配置可能オブジェクトを作り直せばよい。もしコンパイラがいきなり実行形式を出すようになっていたなら、全部のファイルを毎回コンパイルしなければならないだろう。

もう 1 つ重要なのは、よく使われる汎用的なサブルーチンなどは予め翻訳して再配置可能オブジェクトの形で共通の場所に蓄えておき利用するようにできるということである。このようなものを「ライブラリ」と呼ぶ。Unix では複数の「.o」ファイルをまとめた「.a」ファイルという形式が用意されていて、リンカはプログラム中に見あたらないサブルーチンを.a ファイルから自動的に探してきて組み込む機能を持っている。たとえば t00.c で使っていた printf というサブルーチンも標準ライブラリに含まれている。

これでようやく、小さな C プログラムでも翻訳した実行形式がかなり大きなものになる理由が説明できる。つまり、小さなプログラムでも標準入出力などのライブラリサブルーチンは標準で組み込まれるため、どうしても大きくなってしまうわけなのであった。

2.3.6 機械語命令の実行時間

さて、皆様は最近の計算機が機械語 1 命令を実行するのにどのくらいの時間を必要とするかご存知だろうか? たとえば、次のプログラムを見ていただきたい。

```
/* t01.c --- simple loop */

main() {
  int i = 0;
  printf("start.\n");
  while(i < 1000000) {
    i = i + 1; }
  printf("end.\n");
}
```

このプログラムは見ての通り、足し算を 1000000 回行うもので、ほとんどループ部分だけで実行時間が決まる。これのアセンブリ言語コードを出力して、問題のループ部分を抜き出したものを示す。

```
.LL2:
        ld [%fp-20],%o0
        sethi %hi(999999),%o2
        or %o2,%lo(999999),%o1
        cmp %o0,%o1
        bg .LL3
        nop
        ld [%fp-20],%o0
        add %o0,1,%o1          ←※
```

```
        st %o1,[%fp-20]
        b .LL2
        nop
.LL3:
```

単純に命令数を数えると、11 命令あるので、このプログラムはおよそ 11×10^6 個の命令を実行する。従って、実行に要する時間をこの値で割るとおおざっぱな 1 命令あたりの実行時間がわかることになる。プログラムの実行時間を計るには `time` コマンドを使ってプログラム実行を行えばよい。

```
% time a.out
start.
end.
        1.0 real         0.8 user         0.1 sys
%
```

ここで real はプログラム開始から実行までの所要時間、user はこのプログラムが CPU を実際に消費していた時間、sys はこのプログラムのために OS が実行していた時間を意味する。命令の所要時間を計るのだから、この場合は user と記されている時間を使用する。

ただし、注意しなければならないのは、この計測をやっている間に他の人が CPU を消費するような作業をしていると (CPU の取り合いになるので) 正しく計れないという点である。このため、計測の前に「`vmstat 1`」を実行する。このコマンドは毎秒ごとに CPU の使用状況を報告してくれる。一番最後の (id と記された) 欄が CPU のあき比率で、これが 100% に近ければ計測しても大丈夫である。(smb では「`mpstat 1`」を使ってやると 4 つの CPU ごとの状況が出る。あいている CPU が 1 つ以上あれば計っても大丈夫である。) `vmstat` も `mpstat` も終らせるには`^C` を打てばよい。

話を戻すと、user と記された時間 0.8 秒を先の数で割ると 1 命令の実行に約 72ns、逆数を取って 1 秒間に約 14,000,000 命令実行できることになる。これは私の部屋に置いてある smr03 というホストで機種は SparcStation SLC、Sun の公称では 12MIPS(million instructions per sec) となっている。だから、このおおざっぱな計測でもまあ 1 桁半くらいは合っていると言ってよい。(なお、皆様が使っているマシンははもっと高速な CPU を持っている。) [1]

試みに、※の直後に「`add %o0,1,%o2`」という命令を 10 個入れて同様に計って見た。

```
% time a.out
start.
```

[1] MIPS 値というのはわかり易くはあるけれど、実用的なプログラムの実行時間をあらわす値としては相当いい加減である。本当に計算機の機種選定をやるような場合には、もっと大規模なベンチマークテストを行って評価するのが普通である。

```
end.
        1.5 real         1.3 user         0.1 sys
%
```

加算命令を 10^7 回余計に実行し CPU 時間は 0.5 秒増えた。従って加算命令の所要時間は 50ns となり、先の平均よりだいぶ小さいことになる。

2.4 まとめと演習問題

この章では計算機のソフトウェアの構造について、特にオペレーティングシステム (OS) の各種機能と言語処理系の構成について学んだ。「計算機システムの中では多数のプロセスが並行動作していること」「高水準言語のプログラムが段々に翻訳されて機械語になる過程」について納得して頂けただろうか。

演習 2-1. どれかの窓で「ps lx」を実行し、どのようなプロセスがあるかを観察せよ。また自分の UID が何番かも調べよ。次に「ps lax」で全プロセスを表示させ、そのうち自分のプロセスはどれとどれかも調べよ。さらに、mule や端末の窓を増やして、対応するプロセスが増えたことを確認せよ。(注意! 別のマシンの窓を開いた場合には、それは別のマシンのプロセスになってしまう。この演習をやる場合にはたとえば Console のマシンで全部やるようにする。)

演習 2-2. 「`xclock -a -u 1 &[RET]`」により秒針付きの時計の窓を作り、このコマンドを実行したのと同じ窓で ps を使ってこの時計のプロセス番号を調べよ。また、このプロセスを凍結したり再開するとどうなるか調べよ。このプロセスを強制終了させるとどうなるか? 先の mule や端末の窓だとどうか?

演習 2-3. `t00.c` を打ち込んで動かせ。また `t00.s` や `t00.o` の内容を自分でも確認してみよ。さらに、`t00.s` を mule で編集し、メッセージの文字列やループの回数を修正した上で再配置可能オブジェクトを作り、どこが変化したか確認せよ。また実行形式を作り動かしてみよ。[2]

演習 2-4. `t01.c` を smb、smf/smd/smg、smeiXX/smriXX で動かし、本文で実行例に用いたマシンと速度を比較せよ。さらに、加算命令の実行時間も計測してみよ。全部 OS が違うのでそのつど `gcc -S` からやる必要がある。[3]

---

[2] `t00.s` からオブジェクトを作るのは「`gcc -c t00.s`」で、また実行形式を作るのは「`gcc t00.s`」でできます。

[3] 他の人と同時に測ると正しくない結果になる。vmstat や mpstat で CPU が空いていることを確認してから測るように。

演習 2-5. `t01.c` を翻訳したアセンブリ言語コードを見ると無駄な部分があり、手で直すともっと高速にできそうである。実際に手で直して高速化してみよ。それと `gcc` に`-O4` オプション (最適化) を指定して作ったコードとを (アセンブリ言語コード、実行時間とも) 比較してみよ。