プログラミング環境第11回資料(追加ぶん)

久野 靖 *

1991.12.10

16 言語処理系

16.1 機械語プログラムはどこから来るか?

そもそも、計算機は機械語 (つまり、0と1の羅列) しか理解しない。にもかかわらず、我々は

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

のように「わかりやすい言語」(高級言語、とかいう) でプログラムを書くことができる。なんで、そんなことが可能なのか? 言い方を変えれば、こういうプログラムはどのようにして計算機で実行されるのか?

もちろん、答えは「コンパイラが機械語に翻訳してくれるから」である。と教科書には書いてある。しかしそもそも、機械語になっている、という保証はどこにあるか? 自信を持って機械語になっている、と請け合えるか?

もちろん、翻訳した結果は a.out というファイルになっているので、それをよく観察して、確かに機械語であり、しかも自分が書いたプログラムに相当する機能が実現されている、と納得すればそうである。しかしそこまでやってみた人が何人いることか? じゃあやってみよう。まず第一、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いますか?

```
% ls -l t01.c
-rw-r--r--  1 kuno          101 Dec  9 15:36 t01.c
% cc t01.c
% ls -l a.out
-rwxr-xr-x  1 kuno        24576 Dec  9 15:42 a.out
%
```

予想は当たりましたか? また、なぜそうなのでしょう? まあ、その辺の疑問も含めて色々考えてみよう。

それはそうと、**最初の**コンパイラはいったいどうやって作ったんでしょうね....?

16.2 コンパイラの中身

前節でやったように使うだけなら簡単だが、実は cc 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。

*筑波大学経営システム科学専攻

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、アセンブラソースにする (アセンブラソース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いてあるので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、アセンブラ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。

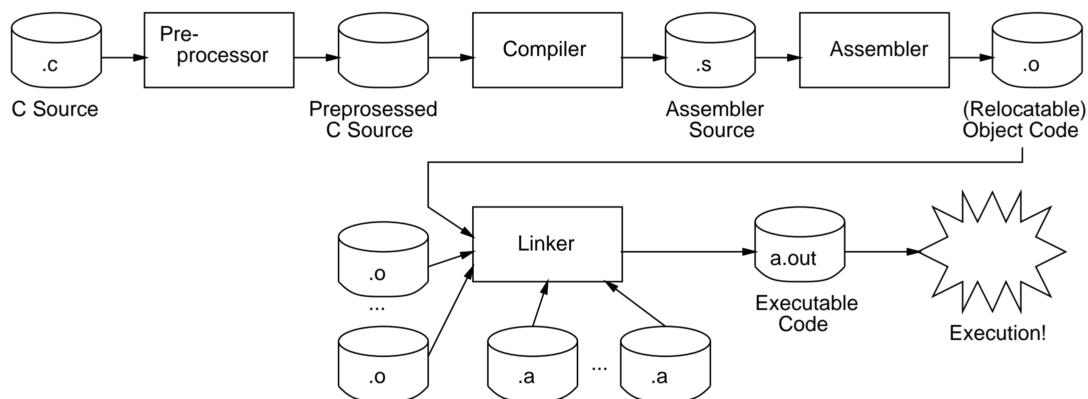

図 1: ソースコードから実行形式までの道のり

この様子を図 1 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。

16.3 アセンブリコード、オブジェクトコード

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 1 の順番を追って見てみよう。まずプリプロセサであるが、`t01.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、`cc` に `-S` というオプションをつけるとアセンブラソースが「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ラソースを見てみよう。

```
% cc -S t01.c
% cat t01.s
LL0:
        .seg    "data"
        .seg    "text"
        .proc 04
        .global _main
_main:                          ←ここが main のはじまりだよ!
        !#PROLOGUE# 0
        sethi   %hi(LF12),%g1
        add     %g1,%lo(LF12),%g1
        save    %sp,%g1,%sp
```

```
        !#PROLOGUE# 1            ←ここまでは手続き入口処理。
        st      %g0,[%fp+-0x4]   ←ほら、i に 0 を入れている。
L14:                             ←ラベルだよ。
        ld      [%fp+-0x4],%o0   ← i の内容をレジスタ o0 に。
        set     0xf4240,%o1      ← 1000000 を o1 に。
        cmp     %o0,%o1          ←くらべる。
        bge     L15              ←分岐してループから抜ける。
        nop                      ←何もしない命令。
        ld      [%fp+-0x4],%o2   ← i の内容を o2 に。
        add     %o2,0x1,%o3      ← 1 足す。
        st      %o3,[%fp+-0x4]   ← i に戻す。
        b       L14              ←ループの頭へ戻る。
        nop                      ←何もしない命令。
L15:
        mov     0,%o0            ←ここから後は戻りしょり。
LE12:
        mov     %o0,%i0
        ret
        restore
       LF12 = -72
        LP12 = 64
        LT12 = 64
        .seg    "data"
%
```

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。

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。

```
% cc -c t01.s
% ls -l t01.o
-rw-r--r--  1 kuno          134 Dec  9 15:49 t01.o
% od -x t01.o
0000000  0103 0107 0000 0050 0000 0000 0000 0000
0000020  0000 000c 0000 0000 0000 0000 0000 0000
0000040  033f ffff 8200 63b8 9de3 8001 c027 bffc
0000060  d007 bffc 1300 03d0 9212 6240 80a2 0009
0000100  1680 0007 0100 0000 d407 bffc 9602 a001
0000120  d627 bffc 10bf fff7 0100 0000 9010 2000
0000140  b010 0008 81c7 e008 81e8 0000 0000 0000
0000160  0000 0004 0500 0000 0000 0000 0000 000a
0000200  5f6d 6169 6e00
0000206
%
```

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 の意味だもんで)、その右のが中身である。最初の 32 バイトはヘッダ (ファイルの中のどんな部分がどのくらいの大きさか、などが入った場所) で、40_8 バイト目からが命令である。何がなんだか分からない、という人もじっと見ていれば分かってくる。例えば変数 i は fp レジスタ起点で-4 番地だから、何回も出てくる bffc というのはそのオフセットくさい、とわかる。実はこのマシンは SPARC アーキテクチャといい、命令は全部 32 ビットで、その中にオフセットも何も押し込めるので結構ビット数が苦しく、多くの命令は下 13 ビットがオフセットを表す。32 ビットのリテラルも 1 命令ではロードできず、上半分と下半分に分けてロードする (sethi、setlo という命令がそうである)。なお set というのは実はアセンブラ命令で sethi/lo の 2 命令に展開される。

さて、これは関数 main のみを含むオブジェクトファイルだったが、そんなに大きくはない。ところが a.out の方はこれにライブラリ関数とかが沢山ついて巨大になるわけである。でもその中にはちゃんと今のコードが埋まっている。.o ファイルから a.out にするのも同じ cc 指令でできるのでやってみよう。

```
% cc t01.o
% cc t01.o
% od -x a.out
....
0001200  726f 0a00 0000 0000 033f ffff 8200 63b8
0001220  9de3 8001 c027 bffc d007 bffc 1300 03d0
0001240  9212 6240 80a2 0009 1680 0007 0100 0000
0001260  d407 bffc 9602 a001 d627 bffc 10bf fff7
0001300  0100 0000 9010 2000 b010 0008 81c7 e008
....
```

巨大ではあっても、ちゃんとその中に問題の部分が埋まっている。というわけで、確かに自分が書いたコードは翻訳されて機械語になっていることが納得... できましたか?

16.4 最適化

さて、コンパイラの主要な仕事の 1 つに最適化、というのがある。これはつまり、単に動く機械語コードを生成するだけでなくて、なるべく高速に実行できるようにコードを改良する、ということである。[1]

現在では最適化技術の進歩によって、アセンブラで書かなくても高級言語から十分速いコードが生成できる、というのが常識になっている。実は、先ほどのコードは全然最適化を掛けていなかったので無駄がいっぱいある。(どこどどこかお分かりですか?) では、最適化を掛けるとどのくらい実行時間が短縮されると思うか? やってみよう。

```
% cc -O t01.c
% time a.out
0.2u 0.0s 0:00 96% 0+17k 1+1io 0pf+0w
%
```

見ての通り、実行時間は 5 分の 1! になってしまった。さっそくコードを見てみよう。

```
% cc -O3 -S t01.c
% cat t01.s
        .seg    "text"                  ! [internal]
        .proc   4
        .global _main
_main:
        sethi   %hi(0xf4240),%o0
        add     %o0,%lo(0xf4240),%o4  ← o4 に 1000000。
        mov     0,%o5   ← i の代わりにレジスタ o5 を 0 に。
        inc     %o5     ← 1 増やす
LY3:                                    ! [internal]
        cmp     %o5,%o4  ←レジスタ比較
        bl,a    LY3      ←分岐
        inc     %o5      ←ここにも「1 増やす」が。
        retl
        add     %g0,0,%o0
        .seg    "data"                  ! [internal]
%
```

[1] 実はできたコードが「最適」であるという保証など何もないのだけど、歴史的に「最適化」と呼んでいる。

すなわち、変数をいちいちメモリからロードして計算し格納する代わりに、レジスタを変数だと思って処理している。従って、ループの本体は`inc`命令、`cmp`命令、条件分岐命令の3つだけで、メモリアクセスがない。このためこんなに高速化できるのである。

なお、`inc`命令が分岐の後にあることについて説明が必要であろう。SPARCでは遅延分岐といって分岐命令の後にある命令は分岐が起きた場合にも必ず実行される。これは全体として命令の平均実行速度を上げるためにこうなっているものである。先の最適化されてない命令列で分岐命令の後に`nop`があったのはこのためなのであった。[2]

[2]なお、このシステムでは遅延分岐の処理をコンパイラの最適化フェーズがやっているので、その効果はアセンブリコード上で見ることができたが、システムによってはアセンブラが遅延分岐の処理をやるものもある。その場合にはアセンブリコード上の命令列と機械語ファイルの命令列が異なることになる。皆様が使っているシステムもそうかも知れないので、練習問題をやるとき注意されたい。