ソフトウェア工学　52－12
（1987・2・13）

オブジェクトファイルのシステム間移植

寺田 実 ， 和田 英一

東京大学工学部計数工学科

本稿では，同種CPUを持つ計算機の間で，アセンブリ済（相対形式）オブジェクトを移植する方法を述べる．loadgoという名前の比較的小さなCプログラムを用いて、ターゲット計算機上で別のシステムの環境を一時的に作りだすことにより、オブジェクトを動作させる。

本方式の特長として、loadgoさえ動けば移植そのものはほとんど機械的にできる点がある。また、loadgo自身の移植は、Cで記述されていること、比較的小さいこと、ターゲットの記述とオブジェクト形式の記述が分離していることなどのために容易である。

一般には高級言語による記述が移植性を高める方法とされているが、何らかの事情でそれが困難なものに対して本方式は有効である。具体例として、MC68000のためのUtilisp処理系をUnix系OS間や、CP/M-68K、OS-9などに移植した経験について報告する。

Transportation of Object Files between the Systems with Common CPU

Minoru TERADA and Eiiti WADA

Department of Mathematical Engineering and Instrumentation Physics
Faculty of Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113, Japan

In this paper we propose a method for transporting relocatable object files between the computer systems which have same CPU's. This method uses a small C program called 'loadgo', which simulates the original environment on the top of the target machine.

Once the 'loadgo' works, various object files can be transported without any difficulty. And the 'loadgo' itself is portable because of the following reasons: the 'loadgo' is written in C language; the size of the 'loadgo' is rather small; the description of both machine is separated into 3 files.

We also report the experience of porting the Utilisp system for MC68000 between Unix-like systems, and onto CP/M-68K and OS-9.

1．はじめに

本稿では，同種CPUを持つ計算機の間で，アセンブリ済（相対形式）オブジェクトを機械的に移植する方法を述べる．移植のためにloadgoという名前のプログラムを用いるため，本方式をloadgo方式と呼んでおく．

loadgo方式の利点は次のようなものである．

+ システムXからシステムYへのloadgoがひとたび完成すれば，X上のソフトウェアの多くが機械的にYに移植可能．
+ loadgo自身はC言語によって記述され，３００行程度と比較的小さい．
+ X上のソフトウェアをZで動かすための，loadgo自身の移植が容易．

本方式は，そもそもはUnix上で作成されたソフトウェアの移植を目的としていたため，オブジェクト形式などはUnix系のものを仮定している．したがって，本稿の説明にもUnixの予備知識が多少必要であるかもしれない．

それから用語上の問題であるが，本稿ではシステムコールという言葉をかなり広く使っている．狭義にはカーネルに対するサービス要求を指すものであり，特殊な命令（たとえばchmk，trapなど）によって起動されるものである．しかし本稿では，OSのライブラリに入っていて，手続として（jsr命令で）呼びだせるものを指すことにした．これはUnixのマニュアルの２章と３章をあわせたものに対応する．

本稿の構成は，第２章で同種CPU間の移植の必要性を論じ，第３章でloadgo方式のあらましを述べる．第４章ではCプログラムとしてのloadgoの構成を述べ，第５章はloadgo自身の移植性，すなわち新しい計算機へloadgo方式を適用する作業について論じる．第６章は，実際の移植作業であらわれた問題点とその解決方法を述べ，解決の困難な問題点は第７章に述べる．loadgo方式を用いた移植の経験については第８章に示す．

---

unixはAT&T社の、
unosはCharles River Data Systems社の、
CP/M-68Kは Digital Research社の、
OS-9はマイクロウェアシステムズコーポレーションの商標である。

2．システム間移植

近年マイクロプロセッサが安く入手できるようになって，同種CPUを用いた計算機がいろいろでまわりはじめた．それらの機械は各社が独自性を追求した結果として，完全な互換性を失っていることが多い．たとえばMC68000をCPUとするものをみると，OSとしてUnix，CP/M 68K，OS-9 など様々である．また同じUnixについても，bsd系とSystem-V系との違いがあるし，それぞれの間でも少しずつ異なっている部分がある．

こうした状況のもとでは，たとえ同種CPUのためのソフトウェアであっても，異なるシステム間での移植には困難が生じる．

ここではシステム間移植という言葉で，同種CPUのためのソフトウェアを異なるシステム間で移植することをあらわそう．（これに対し，従来の異種CPU間の移植は，CPU間移植と呼ぶことができる．）

システム間移植を困難にしている問題点を，ソフトウェアの記述のレベルをおって見てみよう．

a） 高級言語
　処理系がターゲットマシンにも存在する必要がある．
　処理系の細かな相違が問題になることもある．

b） アセンブリ言語
　言語仕様が定まっていない．
　　命令名やアドレッシングモードの記法
　　言語としての機能にも差がある（マクロ機能，ローカルラベル）

c） アセンブリ済（相対形式）オブジェクト
　ファイル形式の違い

d） リンク済（実行可能）オブジェクト
　ファイル形式の違い
　実行開始番地の違い

さらに，OSそのものの違いに由来するシステムコールの違いがある．この違いには大きくわけて３種ある

i） システムコールの機能，パラメータの仕様
　これは上のａ－ｄすべてのレベルで問題になる

ii） システムコールの名称
　これはａ－ｃのレベルで問題になる

iii） システムコールの本当の実現方法
　これはｄのレベルでだけ問題になる

よく知られているように，高級言語による記述は最も問題がすくない．ひろく処理系のゆきわたった高級言語で，

しかもなるべく処理系による差の生じにくい機能のみを用いてソフトウェアを書くのがのぞましいのはもちろんである．

しかし本稿では，アセンブリ済オブジェクトレベルのシステム間移植を対象とする．このレベルの移植が現実的必要性をもっていることを以下でしめそう．

+ 記述言語の機能をフルに利用できる
+ アセンブリ言語による記述で，CPU依存のプログラムの効率を高められる
+ ターゲットマシンに処理系のない言語で書いたプログラムの移植（言語処理系のブートストラップ時などに必要）
+ ソースコードを公開したくない

3．loadgo方式

前節でみたように，アセンブリ済オブジェクトのシステム依存性はファイル形式とシステムコール名称の２点である．loadgo方式とは，loadgoというプログラムによってこの差を吸収しようとするものである（図１）．このプログラムは，Unixにおけるldコマンドとexecシステムコールをあわせたような機能をもっており，具体的には以下の動作をする

1） メモリの確保
2） オブジェクトコードのメモリへのロード
3） 開始番地の決定，リロケーション
4） システムコールの解決
5） 制御の移行

システムコールの解決について少し説明する．システムコールに対応するプログラムコードは，未定義な外部名に対するjsr(Jump SubRoutine)命令になっている．オブジェクトファイルの中には外部名と，それを代入すべきプログラムコード中の場所が何らかの形式ではいっている．この

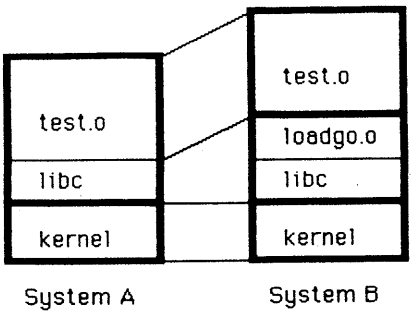

図１．loadgo方式

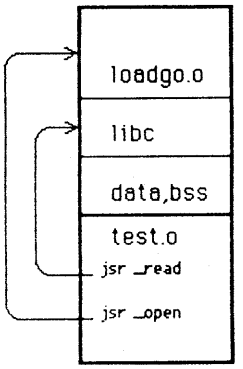

図２．外部参照の解決

情報を利用することで，未定義なjsr命令をどのような名前と結びつけるか（リンクするか）を決定できる．結果として，オブジェクトファイルの中の外部参照はloadgoの内部への jsr になる（図２）．

4．loadgoの構成

loadgo自身は容易に移植可能でなければならない．つまり，どこにでもある言語で，しかも処理系に特有の機能を用いずに記述されなければならない．そのためにc言語を用いることにした．cのもうひとつの特長として，手続の開始番地などを陽に扱うことができる点も有用である．

またloadgoは２種類の環境の橋渡しをするものであるから，それぞれについて情報を持つ必要がある．オブジェクト形式と，ターゲットのシステムコール環境である．

以上の要請から，loadgoは次の３つのファイルからなる C プログラムとした．

a.out.h

オブジェクトの形式を記述する．Unixシステムに標準的にそなわっているファイルで，/usr/include に置かれている。（他のOSでは名前が違う場合がある．）

loadgo.c

オブジェクトのロード，リロケーション，外部参照の解決を行う．これらの操作はすべてオブジェクト形式に依存するものである．

ext.c

解決されるべき外部手続を記述する．ターゲットの環境に依存する．

Unix 系 OS 用のこれらのファイルの行数は以下の通り

a.out.h	71
loadgo.c	166
ext.c	44

5．loadgo自身の移植作業

loadgo方式をとると，プログラムの移植はすべて単にloadgoを移植するだけになる．しかもこの作業は1回行うだけで，多くのオブジェクトを機械的に移植することが可能になる．以下でその作業について考察する．

ここで，現在システムXからシステムYへのloadgoがすでにあると仮定しよう．（すなわち，X上のオブジェクトがYで動作するために，Y上で動くloadgoがある．Xのための a.out.h，loadgo.c と，Yのためのext.c が存在する．）

5．1　XからZへのloadgo

Z上でXのオブジェクトを動かすためのloadgoを作ることである．Xのための a.out.h，loadgo.c はすでにあるので，Zのための ext.c を作ればよい．ext.c にいれるべきシステムコールの種類は，Yのための ext.c を見ることで容易にわかる．あとはそれらをZの環境のものにおきかえるだけであるから，Zに詳しい人であればそれほど困難ではない．

システムコールの本体をどのように用意するかについてはいろいろな場合がある．以下簡単な順に説明しよう．

a）　ターゲットマシンに同機能の手続が存在する場合

その手続をloadgo自身の中に含め，その番地を呼ぶ．（図3a）

b）　同機能の手続はあるが，たとえばパラメータの順が違うような場合

ただパラメータの順を換えるだけの手続をloadgo内に用意し，それを呼ぶ．（図3b）

c）　同機能の手続がない場合

ターゲットマシンの他の手続を使って，仕様にあったものを作る．（図3c）

5．2　WからYへのloadgo

Y上でWのオブジェクトを動かすためのloadgoを作ることである．Wのための a.out.h を用意し，loadgo.c をつくればよい．それにはWにおけるオブジェクト形式に関する知識が必要であり，1にくらべるとややむずかしい．

```
/* case a:
   both system have the same 'write'
*/
extern int      write();

/* case b:
  'Write' is equivalent to 'write'
  except the order of parameters
*/
write(fd, buf, count)   char *buf;
{
    int result;

    result = Write(buf, fd, count);
    return(result);
}

/* case c:
   'write1' output only one character
*/
write(fd, buf, count)   char *buf;
{
    int n, result;

    n = count;
    while(--n >= 0){
        result = write1(fd, *buf++);
        if(result != 1) break;
    }
    return(count - n);
}
```

図3．システムコールのための手続定義

しかし，1にくらべて2はそう頻繁にする必要がないことに注意されたい．X上で開発したソフトウェアを，いろいろな計算機に移植するには1の作業だけで十分である．

6．loadgoによる移植の問題点とその解決

6．1　システムコール仕様の差

システムコールの仕様に関して誤解（ないし思いこみ）があった．Unixにおいてメモリを確保する手続 malloc は，確保した領域の先頭番地を返すものであるが，これを続けて呼びだした場合，確保される領域の番地は昇順になっていることが多い．（図4）このことは決して malloc の仕様ではないが，そう信じている人は多いと思う．しかるに OS-9 においては MMU を用いたメモリ管理をしていないため，移植したプログラムが正常動作しなかった．

また，CP/M 68K において，標準的な整数が 16bit であるにもかかわらず，それ以上の大きさの領域を malloc しようとしていたこともあった．

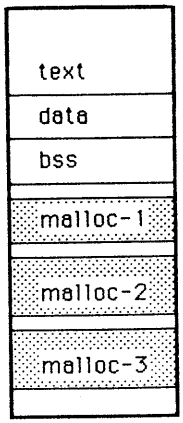

図4．mallocに対する誤解

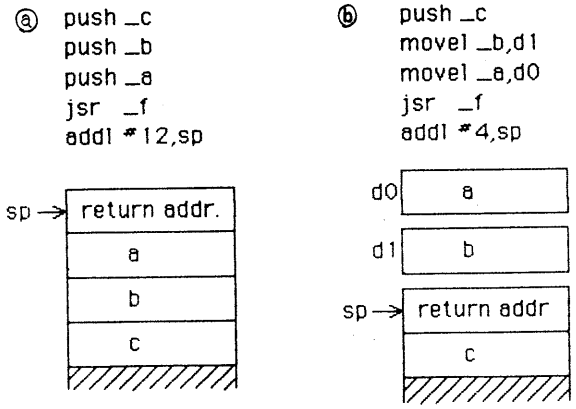

図5．f(a,b,c); に対するコードとスタック

6．2　手続，関数へのパラメータの渡し方

いわゆる calling convention の違いである．多くの C 処理系では，図5a のようにパラメータをスタックに逆順に積むが，処理系によっては図5b のように一部のパラメータをレジスタに持っていくものがある．つまり厳密にいえばオブジェクトファイルは，その形式以外にもこのような形でそれが作られた環境に影響されているわけである．

これはパラメータをつみなおすだけの”つなぎ”手続を用意して解決した．（図6）

6．3　スクラッチレジスタの使い方

手続の内部で，どのレジスタを作業用（スクラッチ）として使うかは，C 処理系によって異なる．したがってオブジェクト側がloadgoを介してシステムコールを呼びだしたところ，本来こわれてはいけないレジスタが破壊されることがありうる．

対策は2通りありうる．呼びだし側，つまりオブジェクト側で必要とするレジスタをすべて退避してからシステムコールをするのが第一の方法．これはどのシステムに移植しても安全であるが，プログラム作成側としては多少わずらわしい．（アセンブリ言語によるプログラミングが必要になるからである．）

もうひとつの方法は，呼ばれる側，すなわちloadgoに作りこむ手続の最初／最後で退避／回復をするものである．これもアセンブリ言語によるプログラミングが必要であるが，loadgo内部にかぎられる．しかしこの方法は，ターゲット側の C 処理系が手続の最初の部分（実行文がはじまる前）ですでにレジスタをこわすことがあれば，使えない．

```
write(d0,d1,fd,buf,count) char *buf;
{
    int result;

    result = Write(fd,buf,count);
    return(result);
}
```

図6．レジスタ渡しのための引数の処理

6．4　大域的領域を指すレジスタ

Unix においてはプログラム領域が固定番地から始まるため，大域的データのアクセスは直接に行うことが可能で，それらの領域を指すためのレジスタは特に必要ではない．

しかしシステムによっては，このような大域的データ領域をレジスタで指す必要がある．このレジスタはプログラムがロードされるときに設定され，その値をユーザが変えることは許されない．たとえば OS-9 では，アドレスレジスタ a6 がその目的に使用されている（図7）．

このようなシステムに普通の（a6 をフレームポインタとして使うような）プログラムをloadgoで移植することを考えよう．プログラムそのものの大域的データは，直接アクセスされるようなコードが出ており，そのリロケーションはloadgoがすでに実行済である．ところがシステムコールのためにloadgoの内部に作りこまれた手続が使う大域的データをアクセスするのに a6 が必要になるのである．

そこで，起動時に設定された a6 の値をどこかに保存しておき，システムコールのための手続の中でプログラム側の a6 と交換してやればよい．ここで問題になるのは，a6 を保存してある場所に対するアクセス方法である．OS-9では，pc相対を用いてテキスト中に保存することにした．（図8）

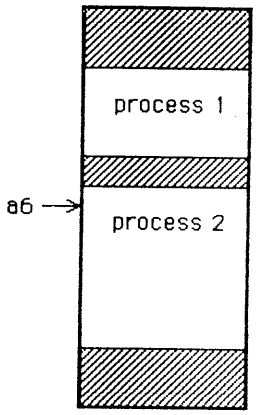

図7．OS-9 のメモリ管理

```
savea6()
{
#asm
        bra     soko
a6save
        dc.l    0
soko
        lea     a6save(pc),a0
        move.l  a6,(a0)
#endasm
}

xchga6()
{
#asm
        lea     a6save(pc),a0
        move.l  (a0),d0
        move.l  a6,(a0)
        move.l  d0,a6
#endasm
}

write(fd,buf,count)  char *buf;
{
    int result;

    xchga6();
    result = Write(fd,buf,count);
    xchga6();
    return(result);
}
```

図8．大域レジスタ a6 の処理

7．残る問題点

7．1　テキスト領域の保護

Unixではプログラムのテキスト領域は書きこみができないようになっていて，プログラムの暴走による被害を最小にとどめることができる．しかしloadgo方式ではオブジェクトは（mallocによって確保した）ユーザ領域にロードされるため，こうした保護機構を利用できない．

7．2　OSの本質的な相違による問題

エディタなど複雑な入出力を要するプログラムは，それぞれのシステムに特有のシステムコールを使うことが多い．このようなプログラムはloadgo方式の対象にはなじまない．

また，割込みに対する処理など，システムコール以外でのOS依存性はもちろん対応できない．たとえばユーザからのアテンションによって，スタック上につくられるフレームの形式を利用したプログラムなど．

8．loadgoによる移植の経験

われわれの研究室では，Utilisp[1] 処理系の開発を行ってきた．これを MC68000 に移植する際，lap 言語をもちいて記述し，それを Utilisp を用いてアセンブリ言語に展開することにした．[2]

このような方法をとると，言語処理系の移植につきもののブートストラップの問題が生じる．つまり lap の展開のための Utilisp が必要になるのである．ひとつの方法としてクロスシステムの利用がある．実際，初期にはこの方法で sun1 から M685 へ移植した．しかしこの方法は通信速度によるネックが大きく，作業効率が良くない．

そこで，loadgo方式によって M685 から OA-90 への移植を行ってみた．どちらも Unix 系の OS を持つため，作業は非常に容易であった．システムコールがほとんど同じだったためである．また，Utilisp のオブジェクトファイル（約１５０Ｋバイト）の転送をくりかえす必要がないので，作業効率もたいへん良かった．

次に，PC-9801 の 68000 ボード上の CP/M 68K への移植を院生の人にやってもらった．この作業によって，いろいろな問題点があきらかになったが，移植そのものは完成した．問題点はほとんど C 処理系の違いに起因するもので，整数型の語長の違いなどであった．

さらに，今年度の卒論の一部として，FM-16β の 68000 ボード上の OS-9 への移植をおこなった．これも4年生1人の作業で移植できた．こちらではパラメータの渡し方の違いや，大域領域を指すレジスタなどの問題があきらかになった．

9．おわりに

簡単な C プログラムを用いて，アセンブリ済オブジェクトファイルをシステム間で移植する方法について述べた．この方法は，Unix 以外の OS にも適用可能であるが，Unix 系 OS の間の移植は特に容易である．

本来はオブジェクト形式を統一し，（実行形式）オブジェクトレベルでの互換性を保証するのが最も望ましいわけだが，ますます多様化する計算機の世界ではそれはかなりむずかしい．むしろこのような形で移植を容易にするほうが現実的であろう．

参考文献

[1] 近山 隆：Utilispシステムの開発，情報処理学会論文誌，Vol.24，No.5，pp.599-604 (1983)．

[2] 和田 英一，富岡 豊：Utilispの68000への移植，情報処理学会記号処理研究会資料 84-29 (1984)．

付録．Utilisp のために必要なシステムコール

以下に示すのは，MC68000 用の Utilisp を loadgo 方式で移植する際必要となるシステムコールである．言いかえれば，これだけを用意することができれば Utilisp の移植が可能になるということである．

このうちの fork，exec，wait などは，外部コマンドの実行のために必要なだけで，Lisp としての機能には関係がない．

```
struct symtabent exttab[] = {
    0,  0,          "_etext",
    0,  0,          "_edata",
    0,  0,          "_end",
    0,  atoi,       "_atoi",
    0,  chdir,      "_chdir",
    0,  close,      "_close",
    0,  creat,      "_creat",
    0,  execl,      "_execl",
    0,  exit,       "_exit",
    0,  fork,       "_fork",
    0,  getenv,     "_getenv",
    0,  localtime,"_localtime",
    0,  lseek,      "_lseek",
    0,  malloc,     "_malloc",
    0,  open,       "_open",
    0,  pipe,       "_pipe",
    0,  read,       "_read",
    0,  signal,     "_signal",
    0,  time,       "_time",
    0,  times,      "_times",
    0,  ttyname,    "_ttyname",
    0,  wait,       "_wait",
    0,  write,      "_write",
    0,  recover,    "_recover",
    0,  0,       0,
};
```