cコンパイラと実行コード

岡野浩三

2023年5月23日

1　はじめに —この資料の目的—

普段何気なく使っているccが何をしているかみていきます.

2　ファイルの解凍

まずE-alpsで配布するcctest.tarファイルをホームディレクトリにおく．この実行のためには，ホストOSとのファイルシェア機能を用いるか，Ubuntuの WEBブラウザを経由して直接ダウンロードするかなどの方法がある．どの方法でもよいのでこのファイルを指定位置に置く.

tarファイルを解凍するには以下のようにコマンドを実行する.

```
$cd
$tar xvf cctest.tar
$ls
```

ここでcctestというディレクトリができていることを確認すること.

次にホームディレクトリ下にlibディレクトリ, includeディレクトリを作成しておく.

```
$mkdir lib include
$ls
```

ここでlib, includeの2つのディレクトリができていることを確認すること.

演習 1

ここまで実行せよ.

3　コンパイラの実行

まずcctestに移動し，以降ここで作業をすることにする.

```
$cd ~/cctest
$ls
```

ls コマンドで複数の.c ファイルと.h ファイルがあることを確認する．

以降の作業はこの c ファイルをコンパイルして myLib.a というライブラリをコンパイルし $HOME-/lib 下に置くこと，および，mycommand という実行ファイルを作成することを意図している．

通常ライブラリは複数のプログラムで共通に用いることができる関数や諸定義をまとめて保管したものである．

.a の拡張子を持つファイルは静的ライブラリとして用いられる．

3.1 ライブラリの作成

```
$gcc -c mycode.c
$ls
```

を実行し mycode.o の生成を確認しよう．-c オプションは実行形式ではなくバイナリのコンパイルまでを行う．

c のコンパイラは伝統的には cc である．gcc は gnus 版の cc である．ここでは gcc を使う．

```
$more mycode.c
```

でプログラムの中身を確認しよう．関数が 2 つほど定義されているだけである．そのうちの 1 つはは引数の文字列を改行付きで表示するだけの関数である．

当然ながら実用的なライブラリではもっと意味のある関数を複数定義しておくのが普通である．

```
$ar rsv libmyLib.a mycode.o
$ls
```

ar で libmyLib.a を生成する．ls で libmyLib.a が新規に作成されたことを確認せよ．静的ライブラリはライブラリ名の先頭に lib が付く．すなわちこのライブラリは myLib というライブラリ名であることを意図している．

なお ar は tar と同様にアーカイバを作成するコマンドである．したがって複数のオブジェクトファイル (*.o) をまとめることができるがこの例では簡単のため 1 つのオブジェクトファイルをアーカイブしている．

```
$mv libmyLib.a ~/lib
$mv myLib.h ~/include
```

で今作成した libmyLib.a をホーム下の lib に移動する．また myLib.h をホーム下の include に移動する．この lib ディレクトリはユーザ個人が作成したライブラリをまとめて置いておくディレクトリを意図している．同様に，ヘッダーファイルをまとめておいておく置いておくディレクトリとして include を意図している．なお，システムで広範使われるライブラリは/lib, /usr/lib などに置かれている．また，それらのライブラリのヘッダファイルが/usr/include などに置かれている．

演習 2

ここまで実行せよ．

3.2 mycommand のコンパイル

次にこのライブラリを活用?して mycommand という実行ファイルをコンパイルしよう.
まず

```
$more main.c
$more sub.c
$more ~/include/myLib.h
```

で各ファイルの中身を確認しよう.

問 1
include している *header* ファイルは何だろうか？それぞれどこにファイルの実体があるだろうか？<> と ""によるファイル指定の違いを調べよ. *include* の各文を削除するとココンパイル時にどのような不具合が起こるだろうか？実際に実行して調べてみよ.

大きなプログラムを作成するときはこのように複数の.c ファイルに分割してプログラムを記述し，コンパイルも分割コンパイルするのが定石である.
分割コンパイルは-c オプションを用いて以下のように各 C ファイルごとまとめて行う.

```
$gcc -I ~/include -c main.c
$gcc -c sub.c
```

こうすることで，何か修正を行うときに，修正のあったファイルだけ再コンパイルすればよいようになり時間を節約できる.
なおここで -I は標準でないヘッダーファイルの置き場所であるディレクトリを指定する.

問 2
sub.c をコンパイルするときに *-I* オプションの指定が不要なのはなぜだろうか?

```
$ls
```

で main.o, sub.o の生成を確認する.
できたオブジェクトファイルをまとめ，外部ライブラリとともにリンクして実行ファイルを作成するには以下のようにおこなう. 同じ cc コマンドではあるが，この場合はコンパイル作業はほとんどされず，オブジェクトファイルやライブラリのリンク作業を主にしている.

```
$gcc -o mycommand main.o sub.o -L ~/lib -l myLib
```

- -o オプションは最終生成される実行ファイルの名前を指定するのに用いる.
- -L オプションは外部ライブラリの位置をコンパイラに知らせるために用いる.
- -l オプションで具体的なライブラリ名を指定する.

```
$ ./mycommand
```

で実行できる．

演習 3
ここまで実行せよ．

問 3
コンパイルするたびにこのようなコマンドを正確にタイプするのは面倒である．*Shell script* にするという解決法もあるが，古くからは　*Makefle* と *make* コマンドを用いる方法が知られている．*Makefile* について調べて簡単にまとめよ．また他に良い方法はないだろうか？

4　バイナリ解析

```
$file ~/lib/libmyLib.a
$file main.o
$file sub.o
$file mycommand
```

を実行せよ．file コマンドは引数で指定されたファイルがどのようなファイルであるかを主にファイル先頭のマジックナンバー等の情報などを用いて判定する．mycommand と *.o での出力の違いが何であるか確認せよ．

Linux ではバイナリ実行ファイルも*.o ファイルも共通の elf 形式で表される．elf 形式ではコンパイルされた機械語だけではなく，様々なメタ情報を持つ．メタ情報の中で一番大事なのがシンボルテーブルであろう．

通常 1 つのファイルがコンパイルされただけでは (他のファイルで定義されているため) 番地情報が分からない関数名や変数名が存在する．逆にそのファイルが他のファイルに公開する関数名や変数名の情報もいる．

例えば以下の stings sub.o で得られた stirngs の結果として buff, subcal, sprintf が読み取れる．buff, subcal は sub.c が外部に公開する情報であるし，sprintf は sub.c が使っている関数でこの時点ではどこで定義されているか不明な関数名となる．

```
$strings ~/lib/libmyLib.a
$strings main.o
$strings sub.o
$strings mycommand
```

を実行せよ．strings コマンドはバイナリ中に存在する文字列を抜き出して表示する．

. で始まる文字列は elf 形式のセクション名を表す．例えば，.text にはプログラムの機械語，.rodata には読込専用のデータ領域，.data は読書きできるデータ領域を意味する．elf では非常に多くのセ

クションが使われている．すべてを理解する必要はない．ただ，単なる実行ファイルやバイナリファイルといっても単純に機械語が入っているだけでなく，実に多くのメタ情報が併せて記載されていることを実感してほしい．

strings コマンドでは文字列の「あたり」は付けられるがどれが正確にシンボル情報なのかはわからない．シンボル情報を正確に取り出すには nm コマンドを用いる．

```
$nm  ~/lib/libmyLib.a
$nm  main.o
$nm  sub.o
$nm  mycommand
```

を実行せよ．各ファイルの情報を注意深く観察せよ．U がついているシンボルは「未定義」を意味する．

nm main.o の実行結果では myprint, subcal, exit などが未定義となっている．これらのうち myprint, subcal はそれぞれ libmyLib.a, sub.o で定義されているので最終的には「シンボルが解決される」すなわち，そのシンボルの番地情報が確定できるはずである．nm mycommand でそれらのシンボルがどのようになっているか観察せよ．

mycommand で未定義のシンボルはあるだろうか？

```
$nm  mycommand | grep U
```

を実行せよ．この実行の意図するところは何だろうか? 何がわかるだろうか? U がついたシンボルがまだあるとすれば，それらは何だろうか?

readelf は elf ファイルのメタ情報を表示する．下記以外にも様々なオプションがある．

```
$readelf -h  ~/lib/libmyLib.a
$readelf -h main.o
$readelf -h sub.o
$readelf -h mycommand
```

```
$strace ./mycommand
```

を実行せよ．

strace は引数のコマンドを実行してそこで用いられたシステムコールを表示する．大部分は (共有) 動的ライブラリの呼び出しに関わるシステムコールである．通常動的ライブラリは .so のファイル名がついている (shared object). mmap は動的ライブラリの中のオブジェクトをメモリにマップしているシステムコールである．brk はメモリの操作に関わるシステムコールである．

これらのシステムコールを除くと

execev, write, exit_group などのシステムコールが浮かび上がる．

write がまさに標準出力への出力をしていることが読み取れる．プロラムでは write を使わず printf を使っていたことに注意．printf の関数の本体の中で write が呼び出されているという推測ができる．

演習 4

バイナリ解析の節の内容を一通り実行せよ．また自分なりにソースファイルを改変しさまざまな解析をしてみよ．わかったことをまとめよ．