HCMC University Of Technology
Faculty of Computer Science & Engineering

Course: Operating Systems
Lab 2: Program

Phuong-Duy Nguyen
Email: duynguyen@cse.hcmut.edu.vn

August 29, 2017

Contents

Chapter 1

Introduction

This lab is introduced student the process of compiling a program from source code in Linux.

1.1 Lab objectives

This lab is introduced to these topics:

- The process of compiling a source file into an executable file.
- The experiments of program's internals.

1.2 Lab contents

Students practice the following techniques and skills:

- To use the compiler collection `gcc` to compile and create `binary files` or `executable binary files` from source code.
- To define Makefiles as special format files and use `make` ultility to help you automatically build and manage your projects.
- The practice exercies for executing a program and introduce the concept of process as an instance of program running in a computer.

1.3 Lab outcomes

After completing this laboratory project, a student will be able to:

- Describe how to compile a program source code to assembly code and binary code.

- Identify various kinds of compiled file includes object, library (static and dynamic), executable binaries.
- Understand the binary file's content.
- After completing the self-study section, student may be articulated to organize a 10,000 code line project. The automation scheme support the self installation and removing the project files. For example:

```
$ make clean
$ make
$ make install
$ make uninstall
$ make distclean
```

Chapter 2

Background

2.1 Compiling C programs

2.1.1 Compiling

It is important to understand that while some computer languages (e.g. Scheme or Basic) are normally used with an interactive interpreter (where you type in commands that are immediately executed). C source code files are always compiled into binary code by a program called a "compiler" and then executed. This is actually a multi-step process which we describe in some detail here.

2.1.2 The compiling steps

GCC compiles a C/C++ program into executable in 4 steps as shown in figure 2.1.1.

Preprocessor includes the headers (#include) and expands the macros (#define). The resultant intermediate file "hello.cpp" contains the expanded source code

Compiler compiles the pre-processed source code into assembly code for a specific processor. The -S option specifies to produce assembly code, instead of object code. The resultant assembly file is "hello.S".

Asembler The assembler (as.exe) converts the assembly code into machine code in the object file "hello.o".

Linker links the object code with the library code to produce an executable file "hello.exe".

2.1.3 Compling commands

The `gcc` command program compile a source code files to binary files. The command has multiple options which help the programmer to manipulate the

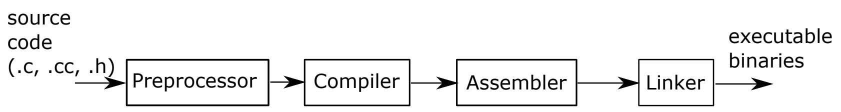

Figure 2.1.1: Program Compiling scheme

compiling progress. The options allow user suspending the compiling at different steps includes preprocessing, compiling, assembling, linking. The following table lists the commands and their corresponding option parameters to change the `gcc` compiler behaviour.

Hint	*This command list is illustrated the usage of compiler option parameter, there is no need to practical on these examples.*

```
% Preprocessed source file
$ gcc -E [-o hello.cpp] hello.c

% Assembly code
$ gcc -S [-o hello.S] hello.c

% Binary file
$ gcc -c [-o hello.o] hello.c

% Executable file
$ gcc [-o hello] hello.c
```

2.2 Makefile

2.2.1 Introduction

Makefile is utility determines which pieces of a large program need to be recompiled. Although you can use bash script as in the previous lab to manage your code, Makefile is a much easier way to organize code compilation. Makefile, which is named 'Makefile' or 'makefile', contains a list of rules. The rule appears in the makefile as follows:

```
# comment
# (note: the <tab> in the command line is necessary for make to
work)
target:  dependency1 dependency2 ...
       <tab> command
```

How Makefiles are remade `Make` program will consider each as a goal target and attempt to update it. If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit

rule applies to it (see Using Implicit Rules), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, make starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.)
`Make` program does its work in two distince phases. During the first phase it reads all the makefiles, included makefiles, etc. and internalizes all the variables and their values, implicit and explicit rules, and constructs a dependency graph of all the targets and their prerequisites. During the second phase, make uses these internal structures to determine what targets will need to be rebuilt and to invoke the rules necessary to do so.

```
$ make
```

It can use target argument to sepcify which part of the makefile to use

```
$ make target-label
$ make clean
$ make all
```

In case the target argument is ommitted, the first-target is set as the goal of makefile.

2.2.2 Makefile example

The following Makefile use to compile a project includes 3 source code file: `main.c`, `hello.c` and `hello.h`

- hello.h declares prototype of the function print_hello()
- hello.c implements the body of the function print_hello() to print to the screen the string "Hello World!"
- main.c implements the body of the function main() includes the call of the function print_hello()

To manage described project, a Makefile managing these source code files has the following contents:

```
all: myapp
myapp: main.o hello.o
        gcc -o myapp main.o hello.o
main.o: main.c hello.h
        gcc -o main.o -c main.c
hello.o: hello.c hello.h
        gcc -o hello.o -c hello.c
clean:
        rm myapp
        rm *.o
```

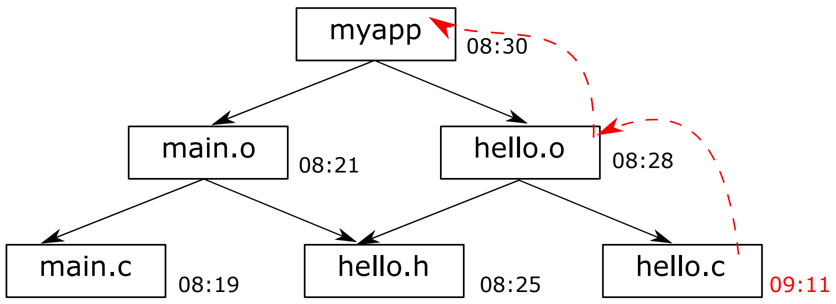

Figure 2.2.1: The illustration of running and validating Makefile

The content of the Makefile is illustrated as an undirected acyclic graph in 2.2.1. Each Makefile rule is represented as a part of the tree with the parent node acting as rule's target and its' child nodes act as rules dependencies.

Hint	*By default, the make call with omitted target will result in the execution of the first rule, the* ***`target all`*** *in our example. The* ***`target clean`*** *is a special case where the dependencies are omitted, so the rule will be executed at every call of* ***`make clean`***.

2.2.3 Makefile contents

Makefile contains five kinds of things: explicit rules, implicit rules, variable definition, directives, and comments.

Explicit rules	says when and how to remake one or more files, called the ruleóÀẼs targets. It lists the other files that the targets depend on, called the prerequisites of the target, and may also give a recipe to use to create or update the targets.
Implicit rules	says when and how to remake a class of files based on their names. It describes how a target may depend on a file with a name similar to the target and gives a recipe to create or update such a target.
Variable definition	is a line that specifies a text string value.
Directive	is an instruction for make to do something special while reading the makefile such as include other Makefile, conditional branching control (if-else) etc.
Comments	starts by '#' character.

2.2.4 Other compiler automation tools

Some IDE such as Microsoft Visual Studio, Eclipse etc. may define some other tools to manage the source code. However, these tools share the same principle

with Makefile. Under the circumstances, the IDE allow generating an Makefile file instead of equivalent tools.
An example of description file used in Excel to define all used folders in xml language:

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
<classpathentry kind="src" path="src"/>
<classpathentry kind="src" path="ext_src"/>
<classpathentry kind="output" path="bin"/>
</classpath>
```

In this example, if we reconfigure compiler parameter in menu Project->Properties->C/C++ Build to "GNU Make" then Eclipse will automatically generate the corresponding Makefile. This test is considered as an extra-homework.

Chapter 3

Practical procedure

3.1 Compile and execute a C program

In general, the compiling progress includes these steps:

1. **Step 1:** Create source code file hello.c

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  printf("Hello, World!\n");
  return 0;
}
```

2. **Step 2:** Create object file:

```
$ gcc -c souce_code_file.c
# Example:
$ gcc -c hello.c
# or
$ gcc -c -o hello.o hello.c
```

3. **Step 3:** Create executable file:

```
$ gcc -o executable_file file1object1.o fileobject2.o ...
# Example:
$ gcc -o hello hello.o
```

- To combile directedly the program from source code, we can use the command `gcc` without -E/-S/-c argument. Although the ttep reduction, this approach may cause some aggregated errors which make it more difficult determine root causes. Create executable file:

```
$ gcc -o executable_file src_file1.c src_file2.c ...
# Example:
$ gcc -o hello hello.c
```

4. `Step 4:` Run the program: after completing the compilation, we create an executable file with the name defined by the argument of option -o. The file is already come up with execution permission so that it does need to change mode `chmod`. We can list the created file by `ls` command.

```
$ ./executable_file
# Example: to list the crated executable binary file
$ ls
hello    hello.c          hello.o
# To execute the binary file
$ ./hello
```

- During compilation, the source code may caused some errors. The compiler will assist in debugging simple errors through displaying compiled results. Bug fixes may performed on the source code where we do not implement, so we have to rely on the display information to find the error. Consider the following error examples:
- Error code will be display at the beginning of the compiler directive `<error_file>:<error_row>:<error_column>:<error_type>:<detailed_info>`
- Error example 1:

```
$ gcc -o hello.o -c hello.c
hello.c:1:18: fatal error: stdo.h: No such file or directory
compilation terminated.
```

- Analyse the example:
  - error_file: hello.c
  - error_row: 1
  - error_column: 18
  - error_type: error
  - detailed_info: stdo.h: No such file or directory
- Error example 2:

```
$ gcc -o hello.o -c hello.c
hello.c: In function 'main':
hello.c:6:11: warning: initialization makes integer ...
hello.c:7:2: error: 'b' undeclared (first use in ...
hello.c:7:2: note: each undeclared identifier is...
hello.c:10:3: error: expected ';' before 'return'
```

- The directive at line 3, this is just a `warning` and do not break the compile process, so we can temporatory ignore such a warning for the higher priority errors.
- The directive at line 4, this is an error caused by the source code at line 7 column 2 in file `hello.c`.
- The directive at line 5, this is an error caused by the source code at line 7 column 2 in file `hello.c`.
- During the error fixing, due to the effect of previous errors caused to the later one, it is recommended to fix the errors in order. Afer fixing a number of errors, we may recompile the source code to excluse the side effects.

3.2 Practice on managing source code using Makefile

Step 1 Implement the source code file: `main.c`, `hello.c` vÕà `hello.h` include:

- hello.h declares prototype of the function print_hello()
- hello.c implements the body of the function print_hello() to print to the screen the string "Hello World!"
- main.c implements the body of the function main() includes the call of the function print_hello()

Step 2 Use text editor to compose the content of Makefile (or makefile):

```
all: myapp
myapp: main.o hello.o
        gcc -o myapp main.o hello.o
main.o: main.c hello.h
        gcc -o main.o -c main.c
hello.o: hello.c hello.h
        gcc -o hello.o -c hello.c
clean:
        rm myapp
        rm *.o
```

Step 3 Run makefile with the default target:

```
# To compile project folder
$ make all

# After run make all, use ls command to check the new generated file
# To clean up project folder (after running use ls to verify the effect)
$ make clean
```

Step 4 Expand the experiment with other makefile target. Student can make a change to any source code file and recall the make to verify whether the modified file is recompile. As the results of the extended steps, student may have some conclusion about the mechanism of makefile working which is based on dependency graph.

```
# Clean up project to clean all generated file
$ make all

# Run specific target makefile rule
$ make hello.o

# Run make all and check which files is compile.
$ make all
```

3.3 Input File Processing

Input file includes a matrix, a program used to get information of each line in the input file is shown in the sample code.

0	10	3	2
2	20	4	1
2	10	4	0
4	15	1	5

Hint: To manipulate the content of file, the function `fopen()` is called to open a file first. Then, use the `fscanf()` function to read or the `fprintf()` function to write each line in the input file.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    FILE* fp;
    int n1, n2, n3, n4;
    char s[10];

    /* Open file named "input.txt" using mode read ("r") */
    fp = fopen ("input.txt", "r");
    fscanf(fp, "%d %d %d %d", &n1, &n2, &n3, &n4);

    /* Close stream; skip error-checking for sample code */
    fclose(f);

    /* Print out the data input get from file
       Notice that the input get from file using fscan
                  the output print to screen using printf
    */
    printf("%d %d %d %d", n1, n2, n3, n4);
}
```

In the case of using this command: `fscanf(f, "%d %d %d %s", &n1, &n2, &n3, &s);`, what is the value of the `s` variable. Note: When file cursor reach the end of the file, the `fscanf` function return a EOF value (End Of File).
By using format strings (% c, % i, % s, % p, % x ...), we can perform file read operations with texttt fscanf () and print the contents of the file using the texttt fprintf () function. The format strings are quite similar to the texttt scanf () and texttt printf () functions.
Generally, the file operations are seem to be an I/O operation which has the file description initialized by `fopen()` function. The passing argument is the path to the target file. On the other hand, the default I/O operations are look like the interacting with the default file descriptions includes stdin, stdout and stderror.

Chapter 4

Exercise

Guideline Student follow the tutorial to do Ex1 and Ex2 in the lab. Ex3 is the homework and the result will be referenced in the next lab. The last two exercises are optional. The Ex5 will be used as and extra practical and reused in later lab.

Ex1 (4 marks) Student do 4 steps of compiling a program `Hello World` in 3.1.

Ex2 (4 marks) Student practice to manage a small project as in 3.3 using Makefile

BT3 (2 marks) Student stop the compile progress at step 2 with option -S (refer the 2.1.3 section). Read and compare the content of main function between the source code and the assembler code.

BT4 (Advance) By using the tools introduced in this lab, student read the binary code and mapping it into machine instruction.

BT5 (Expansion exercise) Based on the file operation in the tutorial, student implement a program to read all lines of file "input.txt".

Appendix A

Makefile example

```
FC=gfortran
CC=gcc
CP=cp

.PHONY: all clean

OBJS = mylib.o mylib_c.o

# Compiler flags
FFLAGS = -g -traceback -heap-arrays 10 \
         -I. -L/usr/lib64 -lGL -lGLU -lX11 -lXext

CFLAGS = -g -traceback -heap-arrays 10 \
         -I. -lGL -lGLU -lX11 -lXext

MAKEFLAGS = -W -w

PRJ_BINS=hello
PRJ_OBJS = $(addsuffix .o,$(PRJ_BINS))

objects := $(PRJ_OBJS) $(OBJS)

all: myapp

%.o: %.f90
	$(FC) -D_MACHTYPE_LINUX $< -c -o $@

%.o: %.F
	$(FC) -D_MACHTYPE_LINUX $< -c -o $@

```

```
%.o: %.c
	$(CC) -D_MACHTYPE_LINUX  $< -c -o $@

myapp: objects
	$(CC) $(CFLAGS) $^ $(objects) -o $@

clean:
	@echo "Cleaning up.."
	rm -f *.o
	rm -f $(PRJ_BINS)
```