Module3: GNU Make Utility and Libraries

GNU Make

Overview of Make

The make utility automatically determines which part of code need to be recompiled and issues commands to recompile them. Make utility can be used with any programming language whose compiler can be run with a shell command. In this module we are using most of the C programs, since they are most common. To use make we must write a file called “makefile” that describes the relationship among files in the program and provides the commands for updating the each file. In program the executable files are updated from the object files, which are compiled from the source files. The make is not limited to programs; it can be used to describe any task where some files must be updated automatically from others whenever the others change. The “make” command uses the “makefile” data base and last modification times of the files to decide which of the files need to be recompiled.

Introduction

The “makefile” tells “make” how to compile and link the program. “make” execute commands in the “makefile” to update the programs. Normally the make looks for makefile and it tries to look by the following names, makefile or Makefile or GNUmakefile.

Writing makefiles

“Make” is a UNIX utility to simplify building program executables from many modules.

1. Create a “Makefile” listing the rules for building the executable. The file should be named as 'Makefile' or 'makefile'. This has to be done only once, except when new modules are added to the program. The “Makefile” must be updated to add new module dependencies to existing rules and to add new rules to build the new modules.
2. After editing program files, rebuild the executables by running “make” command, which will compile the modified source files and creates the executable.

Basic rule:

#comment

Target: **prerequisite** (dependency files)

<Tab> **Command**

Target: The application which need to be build.

Prerequisite: The files needed to build the application (target). These files are dependency files.

Command: The procedure to build the application (Target).

Note: The <Tab> in the command line is necessary for make to work otherwise it will give error message.

Examples:

1. "Makefile" for compiling a program from a single source file.

//Hello.c -- This program display the "Hello World" on console.

```c
#include <stdio.h>

int main()
{
        printf("Hello World\n");
        return 0;

}
```

- Makefile – Make file for compiling the above program.

```
#make file with one source file
Hello:Hello.o
	gcc Hello.c -o Hello

clean:
	$(RM) Hello

~
~
~
```

- Executing make command by giving "$make"

```
bala@bala-virtual-machine:~/Documents/make$ make
cc     -c -o Hello.o Hello.c
gcc Hello.c -o Hello
bala@bala-virtual-machine:~/Documents/make$ ls
Hello  Hello.c  Hello.o  makefile
bala@bala-virtual-machine:~/Documents/make$ ./Hello
Hello World
bala@bala-virtual-machine:~/Documents/make$
```

- $make clean – removes the executable files created by the "makefile"

```
bala@bala-virtual-machine:~/Documents/make$ make clean
rm -f Hello
bala@bala-virtual-machine:~/Documents/make$ ls
Hello.c  Hello.o  makefile
bala@bala-virtual-machine:~/Documents/make$
```

2. "Makefile" for compiling an executable from multiple source files.

In this example we will read two numbers from user input and perform addition and subtraction operation on two numbers using different function calls and display output on console.

//read.c -- This program reads the input integers from user.

```c
#include <stdio.h>

int main()
{
        int a, b;
        scanf("%d%d", &a,&b);
        add(a,b); // funtion call for addition of numbers
        sub(a,b);  // funtion call for subtraction of numbers
        return 0;

}
~
~
```

//add.c – This function performs the addition operation on given integers and gives the sum of two integers.

```c
#include <stdio.h>

add (x,y)
{
        int sum=0;
        sum = x+y;
        printf("sum = %d\n", sum);
        return 0;

}
~
~
~
```

//sub.c -- This function performs the subtraction operation on given integers and gives the difference of two integers.

```c
#include <stdio.h>

int sub(x,y)
{
        int diff=0;
        diff = x-y;
        printf("Diff = %d\n",diff);
        return 0;

}
~
~
~
```

- Makefile -- Make file for compiling the above programs.

```
#Makefile for multiple source files and object files

math:read.o add.o sub.o
        gcc read.c add.c sub.c -o math

clean:
        $(RM) math read.o add.o sub.o
~
~
~
~
~
```

- $Make

```
bala@bala-virtual-machine:~/Documents$ ls
add.c  make  Makefile  read.c  sub.c
bala@bala-virtual-machine:~/Documents$ make
cc    -c -o read.o read.c
cc    -c -o add.o add.c
cc    -c -o sub.o sub.c
gcc read.c add.c sub.c -o math
bala@bala-virtual-machine:~/Documents$ ls
add.c  add.o  make  Makefile  math  read.c  read.o  sub.c  sub.o
bala@bala-virtual-machine:~/Documents$ ./math
3
4
sum = 7
Diff = -1
bala@bala-virtual-machine:~/Documents$ ls
add.c  add.o  make  Makefile  math  read.c  read.o  sub.c  sub.o
```

- Make clean -- – removes the executable files and object files created by the “makefile”.

```
bala@bala-virtual-machine:~/Documents$ make clean
rm -f math read.o add.o sub.o
bala@bala-virtual-machine:~/Documents$ ls
add.c  make  Makefile  read.c  sub.c
bala@bala-virtual-machine:~/Documents$
```

Using make utilities we can run “make” command with target name also. In the below figure we are executing make command with target name.

- $make math

```
bala@bala-virtual-machine:~/Documents$ ls
add.c  make  Makefile  read.c  sub.c
bala@bala-virtual-machine:~/Documents$ make math
cc     -c -o read.o read.c
cc     -c -o add.o add.c
cc     -c -o sub.o sub.c
gcc read.c add.c sub.c -o math
bala@bala-virtual-machine:~/Documents$ ls
add.c  add.o  make  Makefile  math  read.c  read.o  sub.c  sub.o
bala@bala-virtual-machine:~/Documents$ ./ma
make/ math
bala@bala-virtual-machine:~/Documents$ ./math
7
8
sum = 15
Diff = -1
bala@bala-virtual-machine:~/Documents$
```

“Makefile” contain four components and they are

1. Rules – Rules are of two types.
    a. Explicit rule – An explicit rule says when and how to remake one or more files, called targets. It lists the other files that the target depends on called prerequisites of the target, and may also give a recipe to use to create or update the target.
    b. Implicit rule – An implicit rule 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.

2. Variable – A variable definition is a line that specifies a text string value for a variable that can be substituted into the text later. Variables begins with a $ and enclosed within parentheses () or braces {}. Automatic variables are set by make after a rule is matched.
    a. $@: The target filename.
    b. $*: The target filename without the file extension.
    c. $^: The filenames of all the prerequisites, separated by spaces, discard duplicates.
    d. $?: The names of all prerequisites that are newer than the target, separated by spaces.
    e. $<: The first prerequisites file name.

    Examples :$(CC), $(RM), $@, $(CC_FLAGS)

3. Directive – A directive is an instruction for make to do something special while reading the makefile. These may includes:
    a. Reading from another makefile that is the include directive tells make to suspend reading the current makefile and read from other makefiles before continuing.
    b. Deciding whether to use or ignore a part of the makefile.
    c. Defining a variable from a verbatim string containing multiple lines

4. Comments – '#' in the makefile starts a comment that is after whatever written that will be ignored.

Static and Shared libraries

A library is a collection of pre compiled object files that can be linked into programs via linker. There are two types' libraries.

1. **Static Library**

    A static library has file extension of ".a" (archive file) in UNIX operating systems or ".lib" in windows. When your program is linked against a static library, the machine code of external functions used in your program is copied into the executable.

2. **Dynamic library**

    Dynamic library is also called shared library and it has file extension of ".so" (shared objects) in UNIX operating system and ".dll" in windows. When your program is linked against a shared library, only a small table is created in the executable. Before the executable starts running, the operating system loads the machine code needed for the external functions known as dynamic linking. Dynamic linking makes executable files smaller and saves memory, because one copy of a library can be shared between multiple programs.

Differences between static executable and dynamic executable:

- When library is linked statistically to create an executable functions, part of that library are copied as part of that executable image.
- When library is linked dynamically, the functions part of that library is not copied part of executable image. When we execute dynamic executable along with executable image, dynamic libraries are also loaded are referred as load time libraries.
- Static executables are faster when comparing with dynamic executables, since static executable contain all functions.
- Dynamic executables are smaller in size when compare to static libraries, since dynamic executables should not contain all the functions but it has reference to that function in PLT table.

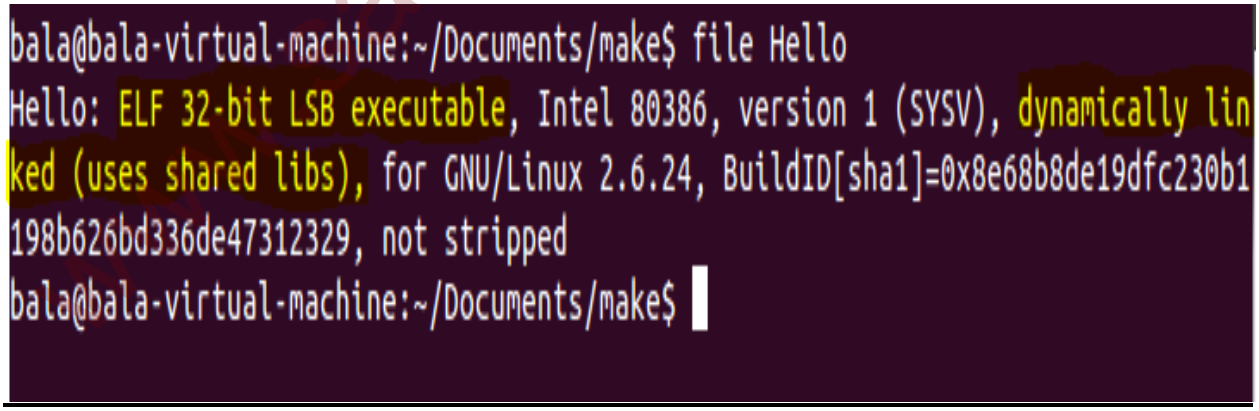
Utilities for Examining the Compiled files

File:

File option is used to display the type of object files and executable files.

Example: gcc –c Hello.c –o Hello

File Hello



objdump (object dump)

Binary dissembler tool, this tool is used to display information of the object files.

$objdump add.o

```
bala@ubuntu:~/Documents$ objdump -D add.o

add.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 f4 07 00 00 00    movl   $0x7,-0xc(%rbp)
   f:   c7 45 f8 07 00 00 00    movl   $0x7,-0x8(%rbp)
  16:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  1d:   8b 45 f8                mov    -0x8(%rbp),%eax
  20:   8b 55 f4                mov    -0xc(%rbp),%edx
  23:   01 d0                   add    %edx,%eax
  25:   89 45 fc                mov    %eax,-0x4(%rbp)
  28:   b8 00 00 00 00          mov    $0x0,%eax
  2d:   8b 55 fc                mov    -0x4(%rbp),%edx
  30:   89 d6                   mov    %edx,%esi
  32:   48 89 c7                mov    %rax,%rdi
  35:   b8 00 00 00 00          mov    $0x0,%eax
  3a:   e8 00 00 00 00          callq  3f <main+0x3f>
  3f:   b8 00 00 00 00          mov    $0x0,%eax
  44:   c9                      leaveq
  45:   c3                      retq

Disassembly of section .rodata:

0000000000000000 <.rodata>:
   0:   73 75                   jae    77 <main+0x77>
   2:   6d                      insl   (%dx),%es:(%rdi)
   3:   20 69 73                and    %ch,0x73(%rcx)
   6:   20 3d 20 25 64 0a       and    %bh,0xa642520(%rip)        # a64252c <main+0xa64252c>
        ...

Disassembly of section .comment:

0000000000000000 <.comment>:
   0:   00 47 43                add    %al,0x43(%rdi)
   3:   43 3a 20                rex.XB cmp (%r8),%spl
   6:   28 55 62                sub    %dl,0x62(%rbp)
   9:   75 6e                   jne    79 <main+0x79>
   b:   74 75                   je     82 <main+0x82>
   d:   2f                      (bad)
```

- Objdump –D add.o

-D → dissemble the content