HCMC University Of Technology
Faculty of Computer Science & Engineering

# Lab 1
# Program and Linux Intro
# Course: Operating Systems

Phuong-Duy Nguyen

March 17, 2024

**GOALS**: The lab helps student to

- Familiar with Linux using the Command Line Interface
- Study how to perform Linux operations using Bash Script
- Familiar with C development on Linux using GNU Toolchain

**OBJECTIVES**:

- Use common Linux commands proficiently
- Know how to write a Linux Bash Script to perform complex tasks
- Know the history and variants of the GNU C Compiler (GCC)
- Understand the fundamental of the C++ compilation process
- Be familiar with GNU Make in modern software development
- Practice debugging C programs using the GNU Debugger

## Contents

# 1. PREREQUISITES

## 1.1. Linux Environment

To practice this class, students must prepare to install Linux operating system on their personal computers (Windows/MacOS) through the following 3 ways (Note: Use only 1 of the 3 following methods) :

- Install Ubuntu virtual machine using VMWare Player on Windows/MacOS (**Recommended**). Required version: Ubuntu 16 or later.
  Instructions: https://www.thegioididong.com/game-app/cach-cai-dat-ubuntu-tren-vmware-chi-tiet-day-du-nhat-1423306
- Install Windows Subsystem for Linux (WSL) on Windows (**Recommended**).
  Instructions: https://docs.hpcc.vn/pages/viewpage.action?pageId=65660
- Install Linux dual-boot with Windows (Not recommended).

## 1.2. GNU C Compiler

Make sure your OS has the GCC installed by using the following command:

```
$ gcc --version
gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
...
```

If GCC is not installed in your environment, please install using this link: https://linuxize.com/post/how-to-install-gcc-compiler-on-ubuntu-18-04/

Basic programming knowledge is also required for this lab.

# 2. Introduction to Linux Command Line Interface

## 2.1. Command Line Interface

Introduction: CLI stands for **Command Line Interface**. The CLI is simply a program that allows users to type text commands to instruct the computer to perform specific tasks.
For example, the `cal` command below tells the computer to display the calendar of the current month:

```
$ cal
   September 2022
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

In the early generations of computers, CLI was used as the basic method of communication between humans and machines, with a keyboard as the input device and a computer monitor that could only display information in the form of text. plain text. After entering the command (the `cal` command above), the result the user receives will be either textual information (returns the current month calendar) or specific actions performed by the computer such as deleting and editing files, shutting down machines,. . .

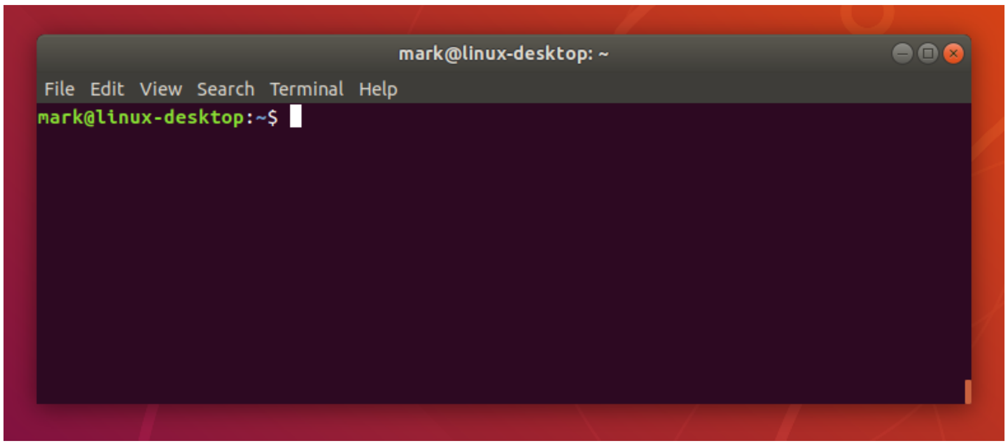
The invention of the computer mouse marked the beginning of the point-and-click method, as a new way to interact with the computer through a graphical user interface (GUI). Nowadays, GUI has become a common way of programming. However, most operating systems still offer a combination of CLI and GUI programming. Although CLI is more confusing and less intuitive than GUI, CLI is still favored by programmers for the following reasons:

- Not all softwares can use GUI
- Most developer tools are CLI in nature
- CLI supports a wide range of operating systems
- The simplicity of CLI
- CLI is very powerful

Start CLI: To use the Linux CLI, we will use the Terminal app. Terminal is a text-based interface/CLI that allows users to interact directly with the UNIX system. Terminal will accept the command lines we enter via the keyboard and execute them, then return the results on the screen.

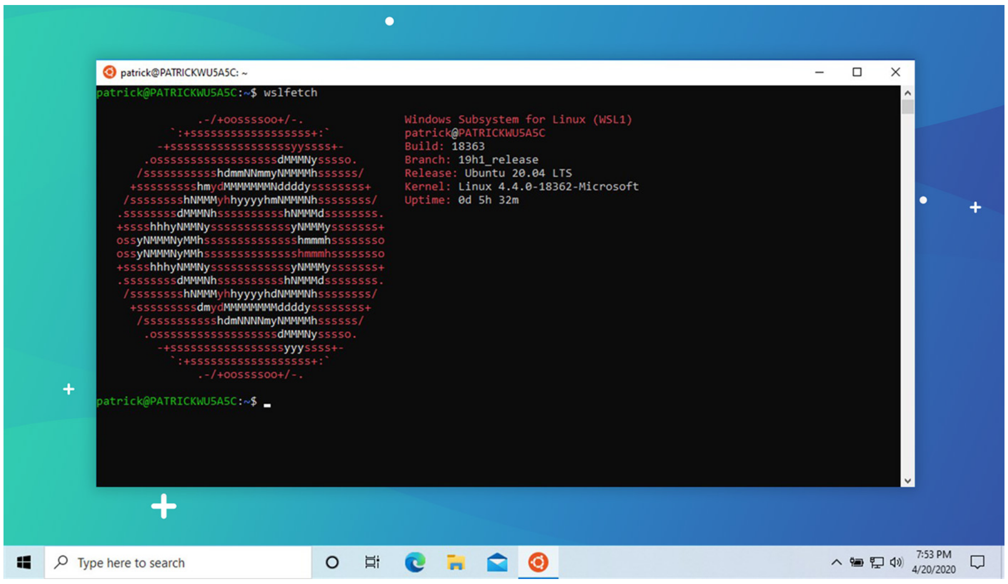
To launch Terminal on Ubuntu (Virtual Machine/Physical Machine), we have two ways:

- **Method 1 (Via GUI)**: Press windows key, type Terminal. Then the Terminal icon appears. Click to open it.
- **Method 2 (Via Shortcut)**: Use the keyboard shortcut Ctrl+Alt+T to open the Terminal



```
mark@linux-desktop:~$
```

Figure 2.1: Ubuntu Terminal



```
patrick@PATRICKWU5A5C:~$ wslfetch

            .-/+oossssoo+/-.                 Windows Subsystem for Linux (WSL1)
        `:+ssssssssssssssssss+:`             patrick@PATRICKWU5A5C
      -+ssssssssssssssssssyyssss+-           Build: 18363
    .ossssssssssssssssssdMMMNysssso.         Branch: 19h1_release
   /ssssssssssshdmmNNmmyNMMMMhssssss/        Release: Ubuntu 20.04 LTS
  +ssssssssshmydMMMMMMMNddddyssssssss+       Kernel: Linux 4.4.0-18362-Microsoft
 /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/      Uptime: 0d 5h 32m
.ssssssssdMMMNhsssssssssshNMMMdssssssss.
+sssshhhyNMMNyssssssssssssyNMMMysssssss+
ossyNMMMNyMMhsssssssssssssshmmmhssssssso
ossyNMMMNyMMhsssssssssssssshmmmhssssssso
+sssshhhyNMMNyssssssssssssyNMMMysssssss+
.ssssssssdMMMNhsssssssssshNMMMdssssssss.
 /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/
  +sssssssssdmydMMMMMMMMddddyssssssss+
   /ssssssssssshdmNNNNmyNMMMMhssssss/
    .ossssssssssssssssssdMMMNysssso.
      -+sssssssssssssssssyyyssss+-
        `:+ssssssssssssssssss+:`
            .-/+oossssoo+/-.

patrick@PATRICKWU5A5C:~$ _
```

Figure 2.2: Ubuntu on WSL

For WSL, after we have successfully installed Ubuntu from the Windows Store by following the instructions, we can access the CLI by launching the Ubuntu application on our computer (via Search / Start Menu)

## 2.2. Linux Shell

The shell is a program between you and Linux (or more accurately, between you and the Linux kernel). Each command you type will be interpreted by the shell and passed to the Linux kernel. In a word, Shell is a command language interpreter and it also makes full use of utilities and application programs on the system…

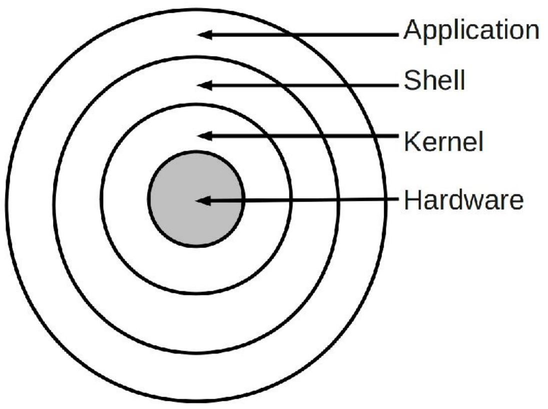


Figure 2.3: OS Layers

Bash (Bourne-Again SHell) is the shell used almost by default on Linux distributions.

**Question:** List some other popular Linux shells and describe their highlighted features.

## 2.3. Linux Commands

CLI interface on Linux: After we successfully open the Terminal, the CLI interface will wait for us to enter the command:

```
Student01@lab-computer:~$
```

All characters to the left of each command line are called the **shell prompt**, displayed in the following structure:

`<username>@<hostname>:<location>$`

Where:

- `username`: Current username (student01)

- `hostname`: The name of the computer ( lab-computer)
- `location`: Working directory (student01's HOME (■) directory)
- `$`: Shell prompt ending separator

When typing a command from the keyboard, the input character will be displayed after the $ sign. Then we can press the Enter key to execute the command. For the example below, we will type the `whoami` command to display the current user who is logging in our system:

```
student01@lab-computer:~$ whoami
student01
```

We can also use the up arrow key to return to previously executed commands.

Command structure: Linux commands can run themselves or accept options and parameters to change their behavior. The typical syntax of commands can be summarized as follows:

`command [options] [parameters]`

Where `options` are usually defined in two ways as follows:

- Short form: Starts with a dash (-) followed by a character. Example: `-l`
- Full form: Starts with 2 dashes (--) followed by a word. Example `--list`

Usually each option will provide both types. For example, `-a` and `--all` both indicate the show all option.
Consider the following example when we run the `ls` command to list the files and subdirectories of a directory:

```
student01@lab-computer:~$ ls
```

When we do not provide any options or parameters, the command will use the default setting as shown here to display a simple list of files and directories at the current working directory (~). To list more detailed information, we use the `-l` option and can add the directory parameter we want to check like `/etc`:

```
student01@lab-computer:~$ ls -l /etc
total 1376
drwxr-xr-x  2 root root     4096 Jun 29  2017 ImageMagick-6
drwxr-xr-x  8 root root     4096 May 18  2017 NetworkManager
drwxr-xr-x  2 root root     4096 Nov 13  2016 Upower
...
```

**Exercise:** Try more options with the `ls` command and analyze its output. You can also try to combine multiple options into one, ex: `-la`.

In case we want to know the detailed information about a command, we use the syntax:

```
man <command to look up>
```

"Man" is an acronym for "manual", is considered a document in Linux that stores all information about supported commands. Ex: `man ls`

Alternatively, we can also use the `-help` or `-h` option to view the help document which is provided by the command itself (if the program supports it).

LINUX FILES AND DIRECTORIES In Linux, everything is considered a file, even a directory is a file type with its own characteristics. Linux has a directory structure that is not divided by drive (C, D, ...) like Windows but has a root directory (usually written as `/`). From this root , Linux filesystem will be splitted into many folders with different purposes. Some commonly used directories are:

- `/home`: contains the content of the user user
- `/root`: contains the contents of the root user
- `/bin`, `/usr/bin`: contains executable programs, most system commands are run. For example ls like we did in the examples above
- `/sbin`, `/usr/sbin`: contains executable programs for admin
- `..` : Parent directory of the working directory
- `.` : Working directory

The path represents how we can refer to the directory structure. The `/` is used in the path to separate each level of this structure. There are 2 types of paths we will encounter:

- **Absolute path**: This path represents the location of the file relative to the `root` directory, so it always begins with `/`. Example: `/usr/bin`
- **Relative path**: This path represents the location of the file from the current directory, thus starting from the current directory. For example: `../../usr/bin` or `./test`

POPLULAR LINUX COMMANDS

- **Show working directory**: We use the `pwd` (print working directory) command to display the absolute path of the current directory (from `/`)
- **Change working directory**: To change the working directory, we use the command: `cd <path of desired directory>`
- **List the contents of the directory**: As introduced in the previous section, we use the `ls` command to list the contents of a directory:
- **Create a new folder**: `mkdir <new directory name>`

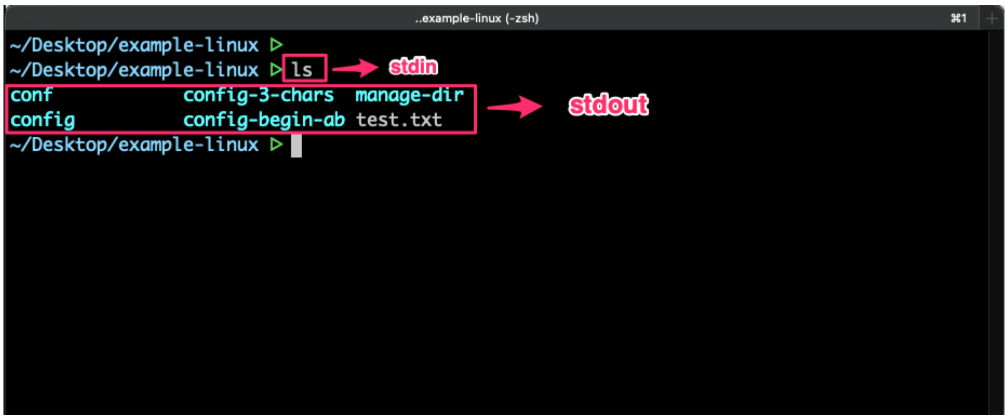
- **Delete folder or file**: `rm <name of folder/file to delete>`. We can combine with these options: `-f` to delete immediately without confirmation and `-r` to delete the entire contents of the directory.
- **Create an empty file:** `touch <new filename>`
- **Copy files**: `cp <file-need-copy> <where-copy-to>`. You can use the `-i` option to enter interactive mode, ie choose whether to overwrite the file with the same name or not.
- **Move files:** `mv <file-need-copy> <place-copy-to>`. If the destination is the same as the file's original location, the move operation becomes a file rename operation
- **Edit file content**: `nano <File name>`
  - GNU nano is an easy-to-use command-line text editor for Unix and Linux operating systems. After the nano editor appears, we can type directly into the CLI and use the navigation keys to navigate.
  - To search for text, press `Ctrl + W`, type the search term, and press Enter. The cursor will move to the word we are looking for. To move to the next position, press `Alt + W`.
  - To save the changes we have made to the file, press `Ctrl + O`. If the file does not exist, it will be created when we save it.
  - To exit nano, press `Ctrl + X`. If there are unsaved changes, nano will be asked if we want to save the changes (type `y` and press `Enter`).
- **View file contents**: The `cat` command helps to view the contents of a file. If we want to display the line number, we can use the `-n <line number>` option. In addition, we can combine with `more` or `less` to view large files by redirecting which will be introduced later.
- **Other useful commands**:
  - `locate`: search for files in an existing database.
  - `wc`: displays statistical information of file content, such as number of lines, number of words, number of characters
  - `clear`: clears the content currently displayed on Terminal
  - `history`: review the history of entered commands
  - `diff`: compare and show the difference between two files
  - `df`: get a report on system disk space usage
  - `du`: check how much space a file or folder takes up

## 2.4. Command redirection

Data Flow   The basic workflow of any command is that they takes input and returns an output. A command will have 3 data streams including:

- **Standard input(stdin)** : The data passed to the command. Stdin is usually from the keyboard, but it can also be from a file or another process
- **Standard out(stdout)**: Is the result returned after successful execution of the statement.
- **Standard error(stderr)**: Is the error returned after executing the command and something went wrong. Stdout is usually output to the screen, but can also be output to a file or another process

For example, when you type `ls` - this is stdin, and stdout is the output you see on the screen:



```
~/Desktop/example-linux ▷
~/Desktop/example-linux ▷ ls
conf              config-3-chars  manage-dir
config            config-begin-ab test.txt
~/Desktop/example-linux ▷
```

Figure 2.4: stdin and stdout for the `ls` command

Redirect command output to file   **Redirection** is a feature in Linux that allows us to change standard input (stdin) and standard output (stdout) when executing a command. Use > or >> to save the output of a command to a file. For example:

```
student01@lab-computer:~$ ls /etc > list.txt
student01@lab-computer:~$ cat list.txt
ImageMagick-6
NetworkManager
UPower
X11
...
```

Note the options:

- `>` : Overwrite the entire file content
- `>>` : Serialize the existing content on the file
- `2>` : Save output to file only if command fails

Command input redirection  Use `<` to redirect input (stdin). The example below passes the input to the lowercase to uppercase conversion statement with the result of the `cal` command in the file `mycal` We will print the calendar to the screen in uppercase format:

```
student01@lab-computer:~$cal > mycal

student01@lab-computer:~$ cat mycal
   September 2022
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 2

student01@lab-computer:~$ tr 'a-z' 'A-Z' < mycal
   SEPTEMBER 2022
SU MO TU WE TH FR SA
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

Piping  We can use `|` to redirect (Piping) the output to another statement. We can also redirect the output of one command to another instead of the file as above. This is useful when there are many statements that need to be concatenated or have too many results. Example, when we type `ls /etc | more` from the keyboard, the output of the `ls` command will be converted to the input of the `more` command and produce the result to the screen.

**Question:** Compare the Output Redirection (`>`/`>>`) with the Piping (`|`) technique.

Filter output with grep:  **Grep** is an acronym for Global Regular Expression Print. The `grep` command in Linux is used to search for a string of characters in a specified file, which is very convenient when searching large log files. Example filtering any files in the `/etc` directory which contains the character `"sys"` :

```
student01@lab-computer:~$ ls -a /etc | grep sys
rsyslog.conf
rsyslog.d
sysctl.conf
...
```

## 2.5. Sudo

**Sudo** is an acronym for "substitute user do", or "super user do". This is a Linux program that allows users to run programs with exclusive rights to secure the information of other users in Linux (usually the `root` user).

For some programs that need privileges to access some protected directories/files, we must use `sudo` to be able to execute programs or actions on that directory/file. For example:

```
student01@lab-computer:~$ mkdir /opt/sample
mkdir: cannot create directory '/opt/sample': Permission denied
student01@lab-computer:~$ sudo mkdir /opt/sample
student01@lab-computer:~$ ls /opt
sample
```

**Question:** Compare the `sudo` and the `su` command.

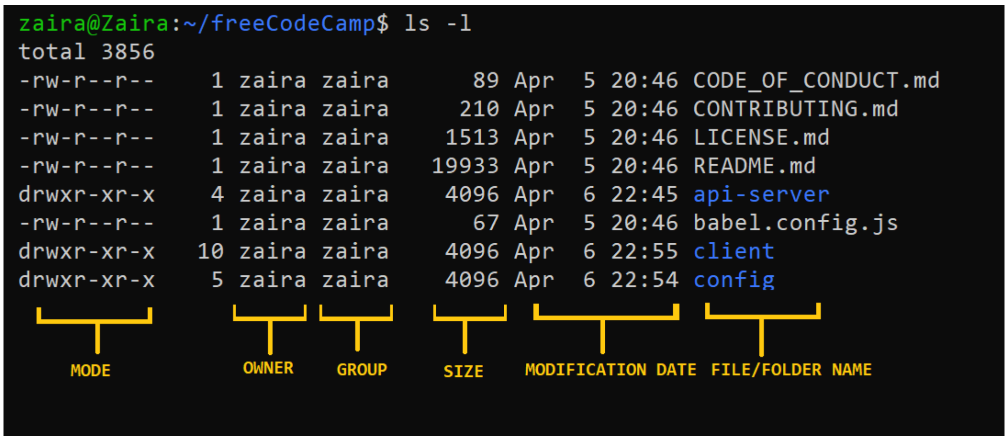
## 2.6. Permission on Linux

In Linux, each file or directory will be granted read, write, and execute permissions for different users. This enhances security to Linux systems. Each file has the following user information:

- **An owner**: identifies the user who owns the file.
- **A group**: which defines the user group of the file.
- **Others**: Any system user who is not the owner and does not belong to the same group

Each user belonging to one of the three groups owner, group, and other will be assigned permissions:

- **read**: the ability to open and view the contents of the file
- **write**: the ability to open and modify the contents of a file
- **execute**: the ability to run the file as an executable program.



```
zaira@Zaira:~/freeCodeCamp$ ls -l
total 3856
-rw-r--r--    1 zaira zaira      89 Apr  5 20:46 CODE_OF_CONDUCT.md
-rw-r--r--    1 zaira zaira     210 Apr  5 20:46 CONTRIBUTING.md
-rw-r--r--    1 zaira zaira    1513 Apr  5 20:46 LICENSE.md
-rw-r--r--    1 zaira zaira   19933 Apr  5 20:46 README.md
drwxr-xr-x    4 zaira zaira    4096 Apr  6 22:45 api-server
-rw-r--r--    1 zaira zaira      67 Apr  5 20:46 babel.config.js
drwxr-xr-x   10 zaira zaira    4096 Apr  6 22:55 client
drwxr-xr-x    5 zaira zaira    4096 Apr  6 22:54 config
```

Figure 2.5: File detail using the `ls` command

To view the permission detail, we can use the `-l` option when calling the `ls` command: Linux file permissions can be displayed in two formats. The first format, as shown above, is called **symbolic notation**, which is a string of 10 characters: One character representing the file type and 9 characters representing the permissions. read (`r`), write (`w`) and execute (`x`) of the file in order of owner, group, and other users. If not allowed, the dash (`-`) symbol will be used.

The second format is called **numeric notation**, which is a string of three digits, each of which corresponds to user, group, and other permissions. Each digit can be between 0 and 7, and each digit's value is obtained by summing the permissions of that class. The way to convert between symbols and numbers is shown in Figure 2.6

To change the permissions for files and directories, we will use the `chmod` command:

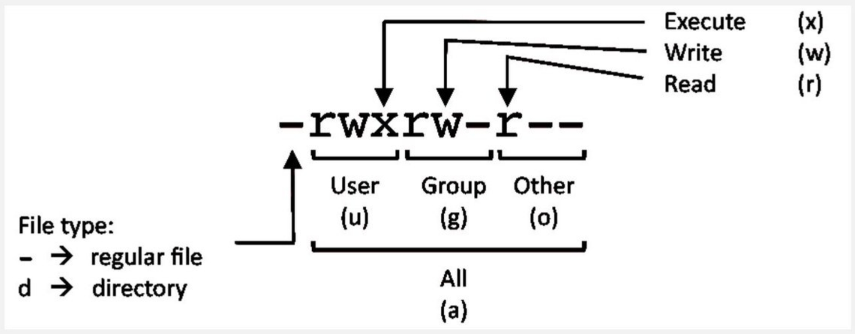
```
chmod <options> <permission index> <file/directory name>
```

The example below shows how to grant the execute permission to a bash script `a.sh`:

```
student01@lab-computer:~$ echo 'echo "Hello World"' > a.sh
student01@lab-computer:~$ ls
a.sh
student01@lab-computer:~$ ./a.sh
-bash: ./a.sh: Permission denied
student01@lab-computer:~$ chmod 744 a.sh
student01@lab-computer:~$ ./a.sh
Hello World
```

In addition, we can also change the ownership of a file. To make changes, we need the sudo privilege:

```
sudo chown <username> <filename>
```



```
        PERMISSION      EXAMPLE

         U   G   W
        rwx rwx rwx     chmod 777 filename
        rwx rwx r-x     chmod 775 filename
        rwx r-x r-x     chmod 755 filename
        rw- rw- r--     chmod 664 filename
        rw- r-- r--     chmod 644 filename

# NOTE: Use 777 sparingly!

        LEGEND
        U = User
        G = Group
        W = World

        r = Read
        w = write
        x = execute
        - = no access
```

Figure 2.6: Linux file permission presentation

**Question:** Discuss about the 777 permission on critical services (web hostings, sensitive databases,...)

## 3. Quick usage of Linux commandline environment

In this section, we will quick coverage the commandline environment with some popular commands. The full detailed command info will be introduced in the appendix.

### 3.1. Remote login

The remote login use ssh tool which can be done with GUI application like PuTTY or ssh command. Due to its practical, we skip it to the next practical section.

### 3.2. Directory/folder and file command

The specific directory/folder command to create a new folder is

```
$ mkdir newfolder
```

The specific directory/folder command to create a new file is

```
$ vi newfile

# if we want to create new file with empty content
$ touch emptyfile

# if we want to create file with some simple text content
$ echo "text content abc..." > text1
```

The common command for directory and file are cp (**c**o**p**y), move (**m**o**v**e), rm (**r**e**m**ove). We use cp to copy the directory/file content to make a copy. If the directory has sub-directory, we do cp recursively with the option "-r"

```
$ cp file1 clonefile1

# if we need to do it recursively
$ cp -r folder1 clonefolder
```

We use the mv to move the directory/file to a new place. If the source place and destination place are the same, then it performs likely rename operation.

```
$ mv folder1/file1 destinationfoler/

# This is actually a rename operation
$ mv file1 newfilename
```

We use the rm to delete the content of the directory/file. If the directory has sub-directory, we do rm recursively with the option "-r"

```
$ rm file1

# If the folder has subfolder, we need to do it recursively
$ rm -r folder1
```

Put all bash commands to a file called script file To reuse and organize commands, we can put it into a file with the file extension ".sh"

```
#! /bin/bash

command1
command2
```

We can put command as listed above, i.e. mkdir, touch, cp, mv, rm. Beside, bash scrip support some structure control as: Variable declare

```
$ vi bashvar.sh
```

```
MyVariable="I will do some math!: "
Number=1

echo $MyVariable $Number + $Number = $((Number + Number))
```

It is important to note that the programming language restrict the assign symbol has no space before and after it. After declaring the variable, we can refer it to future usage with the prefix "$".
By the default, the script file is not executable, we need to grant the execution permission to it:

```
$ chmod +x bashvar.sh

$ ./bashvar.sh
I will do some math!: 1 + 1 = 2
```

- bash for loop: statement which allows code to be repeatedly executed

We can name the iterative variable and later refer it with the prefix "$"

```
for VARIABLE in 1 2 3 4 5 ... N
do
    command1 on $VARIABLE
    command2
    commandN
done
```

## 3.3. Gcc command

To convert the source code file to binary file without address resolution, which means the function has naming but it does not located what is the exact address where the function is located. Then, it is impossible to realize where to jump to execute that function.

```
# option '-c' indicate skipping the address resolution step
$ gcc -c -o hello.o hello.c
```

To convert the source file and mapping the function to it real location address, we use the following command

```
$ gcc -o hello hello.c
```

### 3.4. Makefile rule

Makefile is the input file of make program with exact name "Makefile" (proper capitalization), in which it contains a list of rule, each rule has the syntax

```
<target>: <dependency1> <dependency2>
    <tab>   command1
    <tab>   command2
```

They will perform to the rule as stated below. The make program verifies that all dependencies are present. If any of them already exist, it creates the target by executing all of the commands. An example

```
out: text1 text2
        cat text1 text2 > out
```

This rule implies the verification of file text1 and text2 existence. If they are all existed, the out file is created by executing the command.
To try this example, we create any content of 2 file text1 and text2, then we run the make rule

```
$ vi text1
# type in any text content to file text 1

$ vi text2
# type in any text content to file text 2

$ make out
```

## 4. Practices

In this section, we will work together to complete a project of code management using shell script, compiler and automation tool. In detail, the project includes the following step:

- Remote login using SSH
- Organize folder and file using shell command
- Compose program source code using vi
- Compile program using GNU compiler
- Automation the compile progress using bash scrip or autotool Makefile

Remote login using SSH The ssh command is used to login to remote linux host. Some system does not come with installed SSH service, you need to install it

```
$ sudo apt-get install openssh-server
```

```
$ ipconfig
#or an alternative command depend on the command suport on your
    environment
$ ip addr
$ ssh -l user_account IP_address
```

You can also use the GUI application on Windows named PuTTY.

Organize folder and file using shell command The command ls, cd, cp and mv help you to Organize folder and file in your working folder

```
$ mkdir single-source-file-ws
$ cd single-source-file-ws

# verify our current directory path
$ pwd
/home/user/single-source-file-ws
```

Compose program source code using vi Using vi text editor to compose the source code file

```
$ vi hello.c
```

```
#include <stdio.h>

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

Compile program using GNU compiler Compile the C program

```
$ gcc -o hello.o hello.c
```

Automation the compile progress using bash scrip or autotool Makefile Verify that you are at the parent folder of the workspace foler"single-source-file-ws"

```
$ ls
single-source-file-ws
```

Clone the current workspace folder "single-source-file-ws" to "multiple-source-file-ws"

```
$ cp -r single-source-file-ws multiple-source-file-ws

# verify the successful clone
$ ls
single-source-file-ws    multiple-source-file-ws
```

Change directory to the newly created workspace folder

```
$ cd multiple-source-file-ws
$ ls
hello.c
```

We modify our project from single source file where we directly call printf IO function to multiple source file and move the printf to another separated source file

```
$ vi hello.c
#include <stdio.h>

int main(int argc, char* argv[])
{
   inChoTui();
}

$ vi in.c
#include <stdio.h>

int inChoTui()
{
   printf("Xin chao!\n");
}

$ ls
hello.c      in.c
```

We create shell script file to automation the compile process

```
$ vi build-ws.sh
```

with the following bash script file's contents:

```
#!/bin/bash

SRC_FILE=`ls -R | grep -i ".c"`

echo $SRC_FILE
```

```
rm output/*

for fsrc in $SRC_FILE
do
    nfile="${fsrc%.c}.o"
    gcc -c -o output/$nfile $fsrc
done

cd output
OBJ_FILE=`ls -R | grep -i ".o"`
gcc -o myapp $OBJ_FILE
```

By default, the script file might not have the execution permission. The following command help grant the execution permission

```
$ chmod +x build-ws.sh
$ ./build-ws.sh
```

Then we can verify the output program file

```
$ ls output
hello.o        in.o       myapp
$ output/myapp
```

**An alternative autotool Makefile** For Makefile rules, you must place all of the rule definitions in a fixed-name "Makefile" with proper capitalization.

```
$ cd ..
$ pwd
/home/user
$ cp -r multiple-source-file-ws make-ws

$ cd make-ws
$ ls
build-ws.sh        hello.c        in.c       output/

$ vi Makefile
myapp: hello.o in.o
        gcc -o myapp in.o hello.o

hello.o: hello.c
        gcc -c -o hello.o hello.c

in.o: in.c
        gcc -c -o in.o  in.c

```

```
clean:
	rm -f *.o
	rm myapp

all: myapp
```

We can execute Makefile rule by directly calling the rule name

```
# by default make without directive rule is the default all
    rule
$ make
$ ls
Makefile     hello.c     in.c    myapp
build-ws.sh  hello.o     in.o    output/

$ make clean
$ make hello.o
$ ls
Makefile   build-ws.sh    hello.c      hello.o in.c     myapp
    output/
```

## 5. Exercise

In the previous section, we have built a Shell Script version of a calculator. Following this idea, we will re-implement a C version.

Requirements:

- Students must create a `Makefile` to build the program with at least these 2 targets: `all` and `clean`
- The executable name is `calc`
- The main program is implemented in the `calc.c` source file, while the calculation logic is held on the other source files.
- Input and output requirements are the same as the Shell Script version

Submission Submission guidelines will be announced by the Instructor.

## 6. References

- Coding style by GNU: `http://www.gnu.org/prep/standards/standards.html`.
- C programming
  - Brian Kernighan, and Dennis Ritchie, `"The C Programming Language"`, Second Edition
  - Randal E. Bryant and David R. O'Hallaron, `"Computer systems: A Programmer‘s Perspective"`, Second Edition
- More information about Vim: `http://vim.wikia.com/wiki/Vim_Tips_Wiki`
- Makefile:
  - A simple Makefile tutorial `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`
  - GNU Make Manual `https://www.gnu.org/software/make/manual/make.html`
  - How to Debug a C or C++ Program on Linux Using gdb. `https://www.maketecheasier.com/debug-program-using-gdb-linux/`
  - GCC and Make: Compiling, Linking and Building C/C++ Applications. `https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html`

*Appendix

# A. BASH SCRIPT PROGRAMMING

## A.1. Introduction to Bash Scripts

**Bash Script** is a series of commands written in a file. They are read and executed by the bash program. The program executes line by line. For example, we can navigate to a certain path, create a directory and create a process inside it using the command line.

We can do the same sequence of steps by saving the commands in a bash script and running it whenever we want, instead of having to retype all commands by hand.

By naming convention, bash scripts end with `.sh`. However, bash scripts can run perfectly fine without the sh extension.

A Bash Script is also defined with a **shebang**. Shebang is the first line of each script. The shebang tells the shell to execute it through the bash shell or other shells we want like zsh. The shebang is simply an absolute path to the bash interpreter, like the example below:

```
#! /bin/bash
```

Note that the user needs to have the execute permission to be able to run a bash script.

## A.2. How to Create a Bash Script

Now we will create a simple script in bash that will output the text Hello World. First, we create a file named `hello_world.sh`:

```
student01@lab-computer:~$ touch hello_world.sh
```

Then we identify the executable path of our system bash to include it in the shebang:

```
student01@lab-computer:~$ which bash
/bin/bash
```

Use `nano` to edit the file `hello_world.sh` that we just created above with the following content:

```
#! /bin/bash
echo "Hello World"
```

Then grant the execution permission:

```
student01@lab-computer:~$ chmod +x hello_world.sh
```

Finally, we can run the above script in one of two ways:

```
student01@lab-computer:~$ ./hello_world.sh
Hello World
student01@lab-computer:~$ bash hello_world.sh
Hello World
```

## A.3. Bash Script Syntax

Variable We can define a variable using the `variable_name=value` syntax (Note: no space between). To get the value of the variable, add the `$` symbol before the variable:

```
#!/bin/bash
# This is comment - A simple variable example - hello.sh
greeting=Hello
name=Tux
echo $greeting $name
```

```
student01@lab-computer:~$ ./hello.sh
Hello Tux
```

Arithmetic operations Bash also supports mathematical operations: `var=$((expression))`. For example:

```
#! /bin/bash
# file sum.sh

var=$((1+2))

echo $var
```

```
student01@lab-computer:~$ ./sum.sh
3
```

Passing arguments to Bash Script When launching a Bash Script, we can pass arguments and use them in the script. Example: `./sum.sh 1 2`
Here is the list of arguments and other system variables:

- `$0` - The name of the Bash script file.
- `$1 - $9` - Arguments passed to the Bash script file, respectively.
- `$#` - The number of arguments we pass to the file the Bash script.
- `$@` - All arguments provided to the Bash script file.

- `$?` - State of the last executed statement ( `0` -> true , `1` -> false )
- $$ - ID of the current script .

```
#! /bin/bash
# file name.sh

echo $0
echo Your name is $1
```

```
student01@lab-computer:~$ ./name.sh student
./name.sh
Your name is student
```

Read in from the keyboard: Sometimes we need to collect user input and perform relevant operations. In bash, we can get user input with the `read` command.

```
read -p "Message" <Read Variable Name>
```

```
#! /bin/bash
# file input.sh

read -p "Enter a number: " a
read -p "Enter a number: " b

var=$((a+b))

echo $var
```

```
student01@lab-computer:~$ ./input.sh
Enter a number: 1
Enter a number: 2
3
```

Comparation Comparation is used to check if statements are being evaluated as true or false. We can use these operators to compare two statements:

- `= :  $a -eq $b`
- `>= :  $a -ge $b`
- `> :  $a -gt $b`
- `<= :  $a -le $b`

- `<` : `$a -lt $b`
- `!=` : `$b -ne $b`

Compare command structure (`if`):

```
if [ conditions ]
then
      commands
fi
```

Example comparing 2 numbers X, Y entered from the keyboard:

```
#! /bin/bash
# file compare.sh


read x
read y

if [ $x -gt $y ]
then
echo X is greater than Y
elif [ $x -lt $y ]
then
echo X is less than Y
elif [ $x -eq $y ]
then
echo X is equal to Y
fi
```

```
student01@lab-computer:~$ ./ compare.sh
3
5
X is less than Y
```

If the comparison structure is more complex, we can use the following conditional forms:
`if...then...else...fi`
`if..elif..else..fi`
`if..then..else..if..then..fi..fi..  (Nested condition)`
with logical operators `AND (&&)` and `OR (||)` to combine multiple conditions

## A.4. Loop

The `for` loop allows us to execute statements for a specific number of times.

```
#!/bin/bash

for i in {1..5}
do
    echo $i
done
```

```
student01@lab-computer:~$ ./script.sh
1
2
3
4
5
```

Or use a `while` loop with a stop condition:

```
#!/bin/bash
i=1
while [[ $i -le 5 ]] ; do
   echo "$i"
  (( i += 1 ))
done
```

```
student01@lab-computer:~$ ./script.sh
1
2
3
4
5
```

Saving results from a Bash command In case we need to save the output of a complex command in Bash Script, we can write the statement inside the backtick: ‘`<command>`‘ or use the dollar sign: `$(command)` :

```
#!/bin/bash

var=`df -h | grep tmpfs`
echo $var
```

```
student01@lab-computer:~$ ./script.sh
tmpfs 201M 22M 179M 11% /run tmpfs 1001M 192K 1000M 1% /dev/shm
    tmpfs 5.0M 4.0K 5.0M 1% /run/lock tmpfs 1001M 0 1001M 0% /
   sys/fs/cgroup tmpfs 201M 0 201M 0% / run/user/1001 tmpfs 201
```

```
M 24K 201M 1% /run/user/112 tmpfs 201M 0 201M 0% /run/user
/1000 tmpfs 201M 0 201M 0% /run/user/1003
```

Function definition A Bash function is basically a set of commands that can be called multiple times. The purpose of a function is to help us make our bash scripts more readable and avoid writing the same code over and over again. Function structure on Bash:

```
functionName(){

    first command

    second command
    ...

}
```

For example:

```
#!/bin/bash

hello_world() {
  echo 'hello␣everyone'
}

hello_world
```

```
student01@lab-computer:~$ ./script.sh
hello everyone
```

## A.5. Example

In this exercise, we will create a simple calculator application that can perform basic arithmetic operations like addition, subtraction, multiplication, or division depending on the number the user enters in Bash. For example:

```
Enter two numbers:
5.6
3.4
Enter Choice:
1. Addition
2. Subtraction
3. Multiplication
4. Division
3
```

```
5.6 * 3.4 = 19.0
```

Requirements analysis We will perform the following steps:

1. Read two numbers `a` and `b`
2. Read the `choice` of math operation (1-addition, 2-subtraction, 3-multiplication, 4-division):
   - If the `choice` equals 1: Calculate `res = a + b`
   - If the `choice` is equal to 2: Calculate `res = a - b`
   - If the `choice` equals 3: Calculate `res = a * b`
   - If the `choice` equals 4: Calculate `res = a / b`
3. Output: `res`

Complete program Here is the sample code of our program. You can use another approach to complete this task.

```
# !/bin/bash

# Take user Input
echo "Enter two numbers : "
read a
read b

# Input type of operation
echo "Enter choice :"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read ch

# Switch Case to perform
# calculator operations
case $ch in
  1)res=`echo $a + $b | bc`
  ;;
  2)res=`echo $a - $b | bc`
  ;;
  3)res=`echo $a \* $b | bc`
  ;;
  4)res=`echo "scale=2; $a / $b" | bc`
```

```
    ;;
esac
echo "Result : $res"
```

## A.6. PRACTICE EXERCISE

From the basic example using Shell Script that we have practiced above, each student is required to build an advanced calculator application that meets the following requirements:

### Input Requirements

1. Allow direct input of any binary operation, separated by space.

   `<number> <operator> <number>`

   Example: `2 + 5`

2. Supports 5 basic calculations: Add (+), Subtract (-), Multiply (*), Divide (/), Interger Divide (%). Store the result of the last calculation in the variable `ANS`, and it can be accessed again at the next calculation. For example:

```
~$ ./calc.sh
>> 2 + 5
7


~$ ./calc.sh
>> ANS + 3
10
```

3. The `ANS` variable is initialized with a value of 0 and **will not be lost** after restarting the program

4. Store the history of the last 5 successful calculations and use the `HIST` input command to review:

```
~$ ./calc.sh
>> HIST
1 + 2 = 3
ANS - 4 = -1
3 * 5 = 15
9 % 6 = 1
25 / 6 = 4.17
```

5. When waiting to enter any math operations, only `>>` is displayed (Note: There is one trailing space):

```
~$ ./calc.sh
>>
```

6. After entering a valid calculation (with a space between the operators), press `ENTER` to calculate the result (Note: Print the result with a carriage return):

```
~$ ./calc.sh
>> 1 + 2
3
```

Display requirements:

1. For any non-integer (decimal) results, round to 2 decimal digits:

```
~$ ./calc.sh
>> 5 / 3
1.67
```

2. After completing a calculation, press any key to start a new calculation with the display information is cleared:

```
>>
```

3. For any invalid operations (Divide by 0), show `MATH ERROR`:

```
~$ ./calc.sh
>> 1 / 0
MATH ERROR
```

4. For any incorrect inputs, show `SYNTAX ERROR`:

```
~$ ./calc.sh
>> 1 @ 2
SYNTAX ERROR
```

5. Enter `EXIT` to exit the program

```
~$ ./calc.sh
>> EXIT
~$
```

Submission Submit only one executable file: `calc.sh` as instructed by the Instructor

# B. INTRODUCTION TO GNU C COMPILER

## B.1. GCC History

The **GNU Compiler Collection**, commonly known as GCC, is a set of compilers and development tools available for Linux, Windows, various BSDs, and a wide assortment of other operating systems.

As its name, GCC primarily supports C and C++ and includes other modern languages like Objective-C, Ada, Go, and Fortran. The Free Software Foundation (FSF) wrote GCC and released it as free software.

GCC is a key component of the so-called "GNU Toolchain", for developing applications and writing operating systems. The GNU Toolchain includes:

- GNU Compiler Collection (GCC): a compiler suite that supports many languages, such as C/C++ and Objective-C/C++.
- GNU Make: an automation tool for compiling and building applications.
- GNU Binutils: a suite of binary utility tools, including linker and assembler.
- GNU Debugger (GDB).
- GNU Autotools: A build system including Autoconf, Autoheader, Automake, and Libtool.
- GNU Bison: a parser generator (similar to lex and yacc).

GCC is portable and runs on many operating platforms. GCC (and GNU Toolchain) is currently available on all Unixes. They are also ported to Windows (by Cygwin, MinGW, and MinGW-W64).

GCC is also a cross-compiler, for producing executables on different platforms. GNU Toolchain, including GCC, is included in all Unixes. It is the standard compiler for most Unix-like operating systems.

Microsoft Visual C++, GNU Compiler Collection (GCC), and Clang/Low-Level Virtual Machine (LLVM) are three mainstream C/C++ compilers in the industry. In this course, we will use the standard GCC as it is the default compiler on Linux.

## B.2. How GCC Works

GCC is a toolchain that compiles code, links it with any library dependencies, converts that code to assembly, and then prepares executable files. It follows the standard UNIX design philosophy of using simple tools that perform individual tasks well.

First of all, what even is an **executable**? An executable is a special type of file that contains machine instructions (ones and zeros), and running this file causes the computer to perform those instructions. Compiling is the process of turning our C++ program files into an executable.

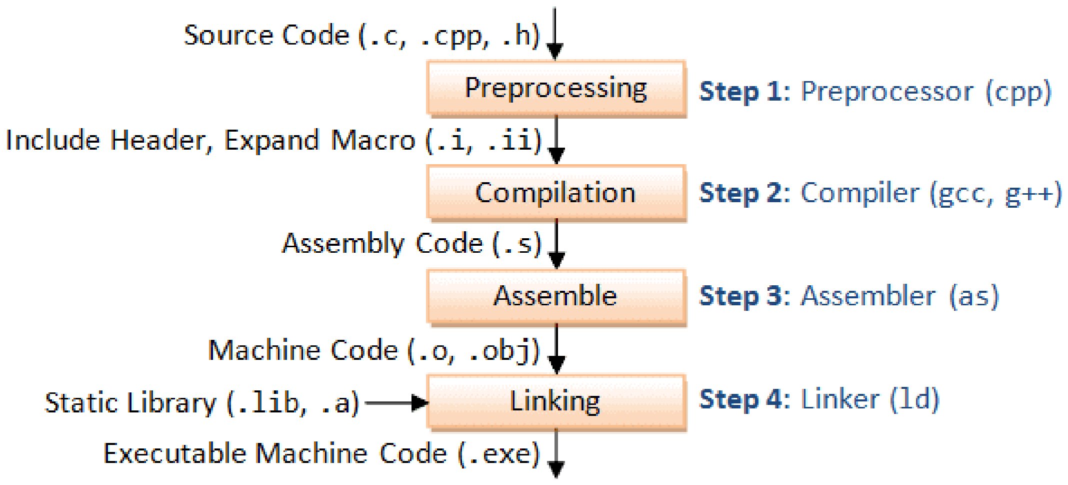

Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Figure B.1: C Program Compilation Process

When we run GCC on a source code file, there are four steps before an executable file is made:

1. **Preprocessing** is the first pass of any C compilation. It removes comments, expands `#include` files and macros, and processes conditional compilation instructions. This can be output as a `.i` file.

2. **Compilation** is the second pass. It takes the preprocessor's output and source code's output and generates assembler source code. Assembly language is a low-level programming language (even lower than C) that is still human-readable but consists of mnemonic instructions that have strong correspondence to machine instructions.

3. **Assembly** is the third stage of compilation. It takes the assembly source code and produces an object file, which contains basic machine instructions and symbols (e.g., function names) that are no longer human-readable since they are in bits.

4. **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols and assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

Let's consider the following basic Hello World C program:

```
#include<stdio.h>

#define YEAR 2022

// File helloworld.c
// This is the main program
int main() {
        printf("Hello %d\n", YEAR);
        return 0;
}
```

Normally, we will compile and run this file directly using the following command:

```
~$ gcc -o helloworld helloworld.c
```

```
~$ ./helloworld
Hello World
```

Moreover, we can stop at any of four above compilation stage using corresponding options. Let's stop at the first step - preprocessing and discuss its output.

```
~$ gcc -E helloworld.c -o helloworld.i
```

```
~$ cat helloworld.i
# 1 "helloworld.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 370 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
...

int main() {
 printf("Hello %d\n", 2022);
 return 0;
}
```

We can see that the preprocessor handles all the preprocessor directives, like `#include` and `#define`. It is agnostic of the syntax of C, which is why it must be used with care.

Next, let's check the output of the compilation step, which is the assembly format:

```
~$ gcc -S helloworld.c
~$ cat hello-world.s
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 12, 0     sdk_version 12, 3
        .globl  _main                           ## -- Begin
           function main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
...
```

Then we will use the assembler (`as`) to convert the assembly code into machine code:

```
~$ as helloworld.s -o helloworld.o
```

```
~$ file helloworld.o
helloworld.o: ELF 64-bit LSB relocatable, x86-64, version 1 (
   SYSV), not stripped
```

We can also see the detailed compilation process by enabling the -v (verbose) option:

```
~$ gcc -v -o helloworld helloworld.c
```

## B.3. GNU MAKE

What is Makefiles? Makefiles are used to help decide which parts of a large program need to be recompiled. In the vast majority of cases, C or C++ files are compiled.

Since building large C/C++ programs usually involves multiple steps, a tool like Make is needed to ensure all source files are compiled and linked. Make also lets the developer control how ancillary files like documentation, man pages, systemd profiles, init scripts, and configuration templates are packaged and installed.

Make is not limited to languages like C/C++. Web developers can use GNU Make to do repetitive tasks like minifying CSS and JS, and system administrators can automate maintenance tasks. Additionally, end-users can use Make to compile and install software without being programmers or experts on the software they are installing.

Here's an example dependency graph that you might build with Make. If any file's dependencies change, then the file will get recompiled.

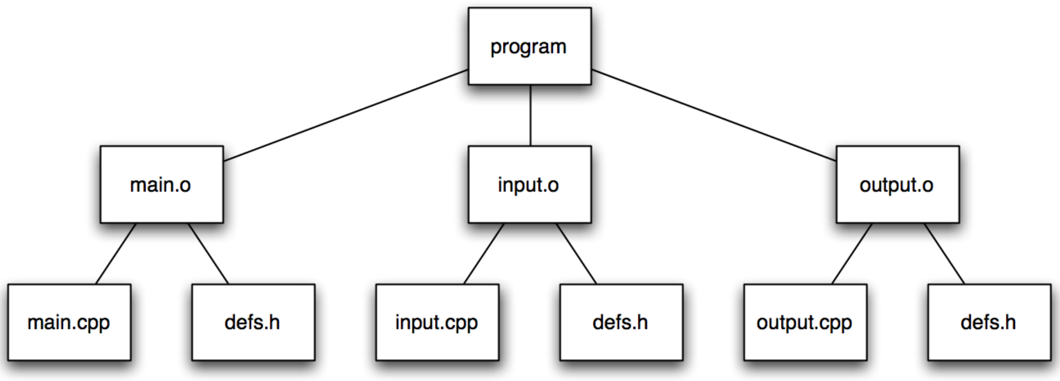


Figure B.2: Source File Dependency Graph in Make

Make was initially developed by Stuart Feldman in April of 1976, but GNU released its free software version of the tool in the late 1980s. Version 3.56 is still available (via diffs) from the GNU FTP server, dating back to September 23, 1989.

GNU Make is a valuable tool for compiling software projects, especially those in the GNU/Linux landscape. Its simple Makefile syntax and intelligent processing of target files make it an excellent choice for development.

Makefile Syntax  A makefile is simply a text file (commonly named `Makefile`) that consists of a set of rules. A rule generally looks like this:

```
targets: prerequisites
        command
        command
        command
```

- The `targets` are file names, separated by spaces. Typically, there is only one per rule.
- The `commands` are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.
- The `prerequisites` are also filenames, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies

Getting Started with Makefile  Let's start with a formal Hello World example. Use `nano` or another of your favorite text editors to compose a file named `Makefile` with the following content:

```
hello:
        echo "Hello, World"
        echo "This line will always print because the file
           hello does not exist."
```

Let's break it down:

- We have one target called `hello`
- This target has two commands
- This target has no prerequisites

We'll then run `make hello`. As long as the `hello` file does not exist, the commands will run:

```
~$ make hello
echo "Hello, World"
Hello, World
echo "This line will always print because the file hello does
   not exist."
This line will always print because the file hello does not
   exist.
```

It's important to realize that `hello` is both a target and a file. That's because the two are directly tied together. Typically, when a target is run (aka when the commands of a target are run), the commands will create a file with the same name as the target. In this case, the `hello` target does not create the `hello` file.

Let's create a more typical Makefile - one that compiles our previous C file:

```
hello: helloworld.c
        gcc -o hello helloworld.c
```

When we run make again and again, the following set of steps happens:

1. The first target `hello` is selected because the first target is the default target
2. This has a prerequisite of `helloworld.c`
3. Make decides if it should run the `hello` target. It will only run if hello doesn't exist, or `helloworld.c` is newer than the previous build.

This last step is critical and is the essence of our Make. What it's attempting to do is decide if the prerequisites of blah have changed since blah was last compiled.

That is, if `helloworld.c` is modified, running make should recompile the file. And conversely, if `helloworld.c` has not changed, then it should not be recompiled.

To make this happen, it uses the filesystem timestamps as a proxy to determine if something has changed. This is a reasonable heuristic because file timestamps typically will only change if the files are modified. But it's important to realize that this isn't always the case.

**Question:**

1. What are the advantages of Makefile? Give examples?
2. Compiling a program in the first time usually takes a longer time in comparison with the next re-compiling. What is the reason?
3. Is there any Makefile mechanism for other programming languages? If it has, give an example?

Make clean  The `clean` is often used as a target that removes the output of other targets, but it is not a special word in Make. You can run `make` and `make clean` on this to create and delete some file.
Note that `clean` is doing two new things here:

- It's a target that is not first (the default) and not a prerequisite. That means it should never run unless you explicitly call make clean
- It's not intended to be a filename and should be marked as `PHONY`. If you happen to have a file named clean, this target won't run, which is not what we want.

A phony target is one that is not really the name of a file. It is just a name for some commands to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name and to improve performance.

```
hello: helloworld.cpp
	g++ -o hello helloworld.cpp

.PHONY: clean
clean:
	rm -f hello
```

Makefile Variables  We can define variables to reuse them across the Makefile. However, variables can only be strings. We can reference variables using either `$` or `$()`:

```
input := helloworld.cpp
output := helloworld

${output}: $(input)
	g++ -o ${output} ${input}
```

```
clean:
	rm -f ${output}
```

Please notice that single or double quotes have no meaning to Make. They are simply characters that are assigned to the variable. Quotes are only useful to shell/bash command,

Targets You can define an `all` target. Since this is the first rule listed, it will run by default if `make` is called without specifying a target.

```
all: one two three

one:
	touch one
two:
	touch two
three:
	touch three

clean:
	rm -f one two three
```

When there are multiple targets for a rule, the commands will be run for each target. We will use `$@` which is an automatic variable that contains the target name.

```
all: f1.o f2.o

f1.o f2.o:
	echo $@
# Equivalent to:
# f1.o:
#	echo f1.o
# f2.o:
#	echo f2.o
```

Automatic variables They are variables that have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule:

```
hey: one two
	# Outputs "hey", since this is the target name
	echo $@

	# Outputs all prerequisites newer than the target
	echo $?
	# Outputs all prerequisites
	echo $^

	touch hey

one:
	touch one

two:
	touch two

clean:
	rm -f hey one two
```

Implicit Rules  Let's see how we can now build a C program without ever explicitly telling Make how to do the compilation, by using the implicit rules:

- `CC`: Program for compiling C programs; default `cc`
- `CXX`: Program for compiling C++ programs; default `g++`
- `CFLAGS`: Extra flags to give to the C compiler
- `CXXFLAGS`: Extra flags to give to the C++ compiler
- `CPPFLAGS`: Extra flags to give to the C preprocessor
- `LDFLAGS`: Extra flags to give to compilers when they are supposed to invoke the linker

# C. GNU DEBUGGER

## C.1. Introduction to GDB

The normal process for developing computer programs goes something like this: write some code, compile the code, and run the program. If the program does not work as expected, then you go back to the code to look for errors (bugs) and repeat the cycle again.

Depending on the complexity of the program and the nature of the bugs, there are times when you can do with some additional help in tracking down the errors. This is what a "debugger" does. It allows you to examine a computer program while it is running. You can see the values of the different variables, you can examine the contents of memory and you can halt the program at a specified point and step through the code one line at a time.

The primary debugger on Linux is the GNU debugger (gdb). It might already be installed on your system (or a slimmed-down version called gdb-minimal), but to be sure type the following `gdb` command in a terminal:

```
~$ gdb --version
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
```

## C.2. Debugging C++ program with GDB

Let's modify our following basic Hello World C program:

```
#include<stdio.h>

int main() {
    int i;
        for (i=0; i < 10; i++) {
          printf("Hello %d\n", i);
    }
        return 0;
}
```

Then compile with the `-g` option:

```
~$ gcc -g -o helloworld helloworld.c
```

To start debugging the program, we type:

```
~$ gdb helloworld
```

At this point if you just start running the program (using the `run` command), then the program will execute and finish before you get a chance to do anything. To stop that, you need to create a "breakpoint" that will halt the program at a specified point. The easiest way to do this is to tell the debugger to stop in the function `main()`:

```
break main
```

Now start the program:

```
run
```

The debugger will stop at the first executable line of the main function, e.g. the `for` loop. To step to the next line, type `next` or `n` for short. Keep using `next` to repeat the loop a couple of times.

To inspect the value of a variable, we use the `print` command. In our example program, we can examine the contents of the variable `i`:

```
print i
```

Repeat around the loop and few more times and see how `i` changes:

```
next

next

next

next

print i
```

The loop will continue while `i` is less than 10. You can change the value of a variable using `set var`. Type the following in gdb to set `i` to 10.

```
set var i = 10

print i

next
```

You may need to do another `next` (depending on where the program was halted when you set `i` to 10), but when the `for` loop line is next executed, the loop will exit, because `i` is no longer less than 10.

The `next` command doesn't drill down into functions but rather the function is executed, and the debugger stops again at the next line after the function. If you want to step into a function, use the `step` command, or `s` for short.

Another way to debug your program is to set a watch on a variable. What this does is halt the program whenever the variable changes. Restart the program again by typing `run`. Since the program is already running, the debugger will ask if you want to start it again from the beginning. The program will stop in the main (as we didn't remove the breakpoint). Now set a watch on `i`:

```
watch i
continue
```

To stop debugging, just use the `quit` command.

# D. Signal Programming

```
//signal.c

/** This example illustrate the using of IPC signal() and
  * kill() routines to suspend child process until its
  * parent process is finished.
  */
#include<stdlib.h>
#include <stdio.h>
#include <unistd.h>      /*defines fork(),and pid_t*/

int main(int argc, char** argv) {
  pid_t child_pid;
  sigset_t mask, oldmask;

  /*lets fork off a child process...*/
  child_pid = fork();

  /*check what the fork() call actually did*/
  if(child_pid == -1) {
    perror("fork");
   /*print a system-defined error message*/
    exit(1);
  }

  if(child_pid == 0) {
    /*fork() succeeded , we're inside  the child  process*/
    signal(SIGUSR1, parentdone);
    /* set up a signal*/ /
    sigemptyset( & mask);
    sigaddset( & mask, SIGUSR1);

    /*Wait for a signal to arrive*/
    sigprocmask(SIG_BLOCK, & mask, & oldmask);
    while (!usr_interrupt)
      sigsuspend(&oldmask);
    sigprocmask(SIG_UNBLOCK, & mask, NULL);

    printf("World!\n");
    fflush(stdout);
    36
  }
  else {
    /*fork() succeeded, we're inside the parent process*/
    printf("Hello␣,␣");

    fflush(stdout);
    kill(child_pid, SIGUSR1);
    wait(NULL);

  }

  return 0;
}
```

# E. Makefile full directives example

```
TARGET_EXEC := final_program

BUILD_DIR := ./build
SRC_DIRS := ./src

# Find all the C and C++ files we want to compile
# Note the single quotes around the * expressions. Make will
    incorrectly expand these otherwise.
SRCS := $(shell find $(SRC_DIRS) -name '*.cpp' -or -name '*.c'
    -or -name '*.s')

# String substitution for every C/C++ file.
# As an example, hello.cpp turns into ./build/hello.cpp.o
OBJS := $(SRCS:%=$(BUILD_DIR)/%.o)

# String substitution (suffix version without %).
# As an example, ./build/hello.cpp.o turns into ./build/hello.
    cpp.d
DEPS := $(OBJS:.o=.d)

# Every folder in ./src will need to be passed to GCC so that
    it can find header files
INC_DIRS := $(shell find $(SRC_DIRS) -type d)
# Add a prefix to INC_DIRS. So moduleA would become -ImoduleA.
    GCC understands this -I flag
INC_FLAGS := $(addprefix -I,$(INC_DIRS))

# The -MMD and -MP flags together generate Makefiles for us!
# These files will have .d instead of .o as the output.
CPPFLAGS := $(INC_FLAGS) -MMD -MP

# The final build step.
$(BUILD_DIR)/$(TARGET_EXEC): $(OBJS)
	$(CXX) $(OBJS) -o $@ $(LDFLAGS)

# Build step for C source
$(BUILD_DIR)/%.c.o: %.c
	mkdir -p $(dir $@)
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

# Build step for C++ source
```

```
$(BUILD_DIR)/%.cpp.o: %.cpp
	mkdir -p $(dir $@)
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@


.PHONY: clean
clean:
	rm -r $(BUILD_DIR)

# Include the .d makefiles. The - at the front suppresses the
    errors of missing
# Makefiles. Initially, all the .d files will be missing, and
    we don't want those
# errors to show up.
-include $(DEPS)
```

# Revision History

| Revision | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | 11.03.15 | PD Nguyen | created |
| 1.1 | 11.09.15 | PD Nguyen | add introduction and exercise section |
| 2.0 | 25.02.16 | PD Nguyen | Restructure the content to form an tutorial |
| 2.1 | 20.08.16 | DH Nguyen | Update C and Vim to Appendix |
| 2.2 | 03.04.20 | Thanh HLH | Update exercise |
| 3.0 | 01.10.22 | Thanh HLH | Merge old materials and add new content |