

Lab 1

Introduction to Linux Programming

Course: Operating Systems

March 7, 2025

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	3
1.1	Linux Environment	3
1.2	GNU C Compiler	3
2	Introduction to Linux Command Line Interface	4
2.1	Command Line Interface	4
2.2	Linux Shell	6
2.3	Linux Commands	6
2.4	Command redirection	10
2.5	Sudo	12
2.6	Permission on Linux	12
3	BASH SCRIPT PROGRAMMING	15
3.1	Introduction to Bash Scripts	15
3.2	How to Create a Bash Script	15
3.3	Bash Script Syntax	16
3.4	Loop	18
3.5	Example	20
3.6	PRACTICE EXERCISE	22
4	INTRODUCTION TO GNU C COMPILER	28
4.1	GCC History	28
4.2	How GCC Works	28
4.3	GNU MAKE	31
5	GNU DEBUGGER	37
5.1	Introduction to GDB	37
5.2	Debugging C++ program with GDB	37
5.3	Exercise	39
6	References	43

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

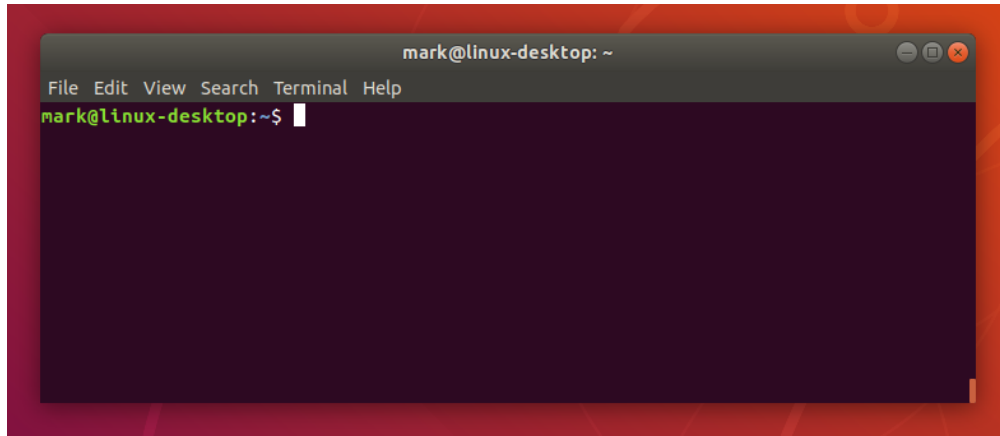


Figure 2.1: Ubuntu Terminal

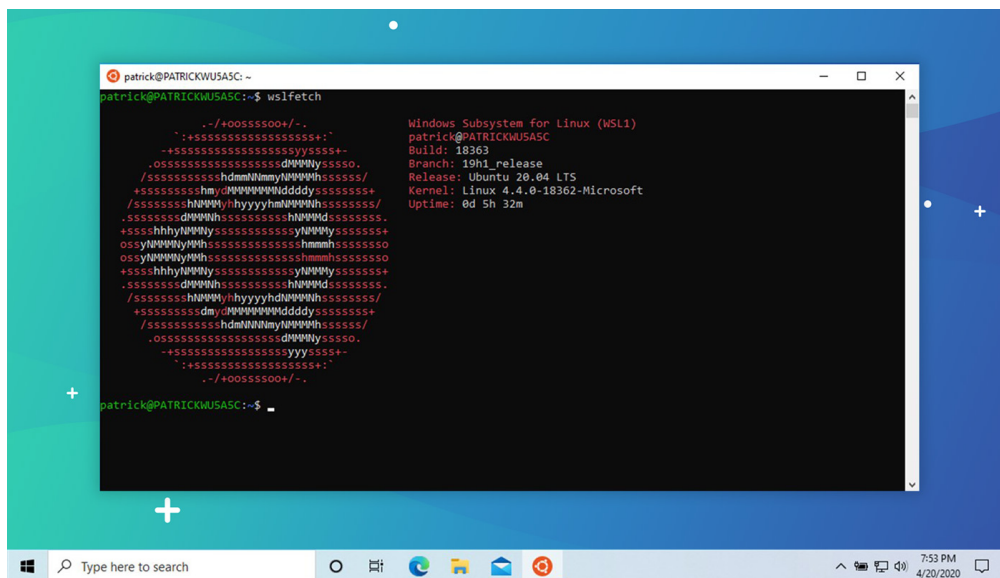


Figure 2.2: Ubuntu on WSL

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...

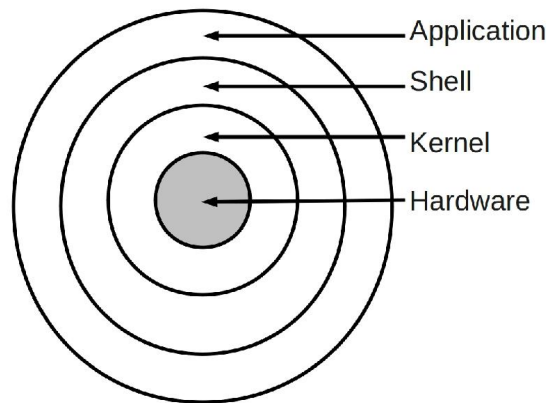


Figure 2.3: OS Layers

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`

POPULAR 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:

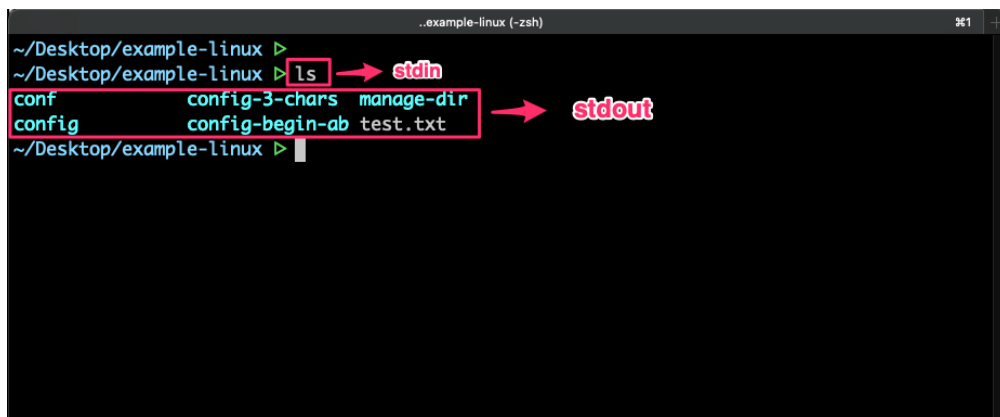


Figure 2.4: stdin and stdout for the `ls` command

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
```

The diagram below the terminal output shows yellow brackets pointing to specific columns of the `ls -l` output:

- MODE**: points to the permission string (e.g., `-rw-r--r--`).
- OWNER**: points to the owner name (e.g., `zaira`).
- GROUP**: points to the group name (e.g., `zaira`).
- SIZE**: points to the file size in bytes (e.g., `89`).
- MODIFICATION DATE**: points to the date and time of last modification (e.g., `Apr 5 20:46`).
- FILE/FOLDER NAME**: points to the file or directory name (e.g., `CODE_OF_CONDUCT.md`).

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>
```

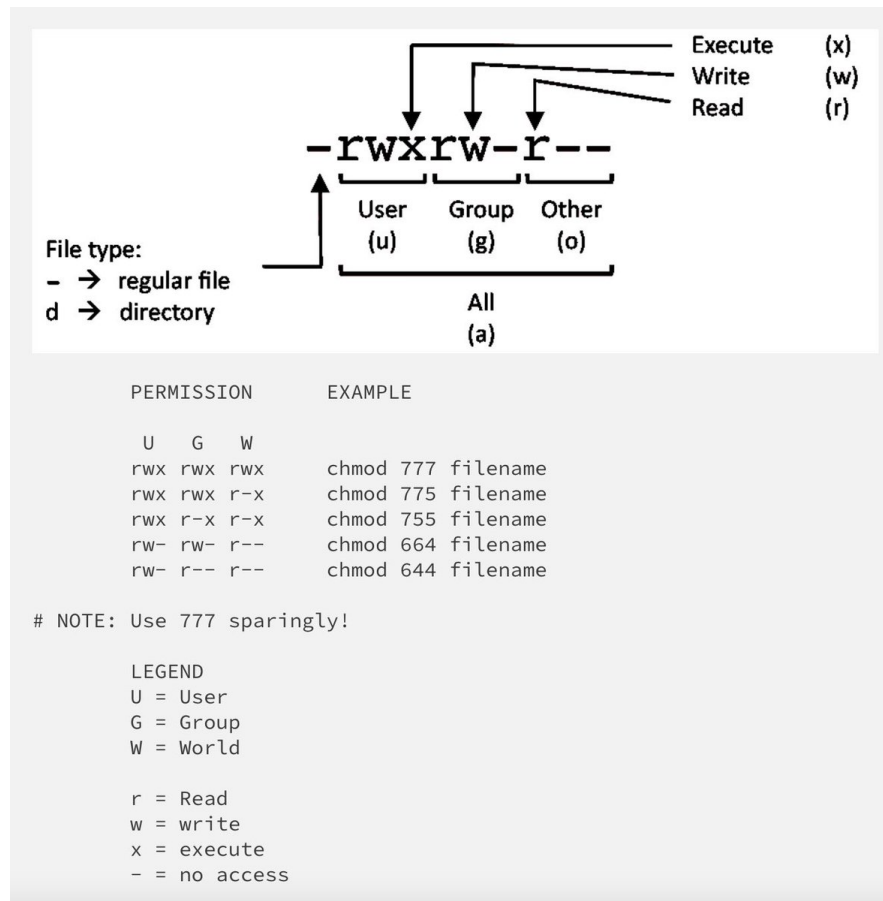


Figure 2.6: Linux file permission presentation

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

3 BASH SCRIPT PROGRAMMING

3.1 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.

3.2 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
```

3.3 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 Comparison 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
```

3.4 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
```

3.5 EXAMPLE

In this example, 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"
```

3.6 PRACTICE EXERCISE

In this exercise, students will implement a **BacKup Utility (BKU)**, a git-inspired backup tool built entirely using bash scripting. Unlike git, which manages all files in a repository, BKU focuses on selectively backing up important source code files within a folder, storing only the differences (diffs) rather than entire file copies. This tool introduces students to file manipulation, process scheduling, and command-line utility design in a Linux environment. Below are the detailed requirements and expected behaviors for each command, including exact inputs and outputs.

REQUIREMENTS

- **Objective:** Create a bash script-based tool called `bku` to back up and manage source code files in a folder.
- **Implementation:** Use only bash scripting. The use of `git` is strictly prohibited and will be checked during evaluation.
- **Root Folder Restriction:** The tool must only function when run from the folder containing the `.bku` directory (the "root folder") after initialization. Running it from any other location (e.g., a subdirectory or outside the folder) will result in an error:

Must be a BKU root folder.

- **Delivery:** Submit two executable files:
 - `bku.sh`: The main BKU tool script.
 - `setup.sh`: A script to check/install dependencies of used Linux tools/packages and add `bku` to the user's `PATH`.

COMMANDS AND SPECIFICATIONS

1. Initialize Backup Repository

- *Purpose:* Initializes the backup system in the current folder by creating a `.bku` subfolder.
- *Behavior:* Checks if `.bku` already exists. If it does, display an error and exit. Otherwise, create `.bku` and initialize any necessary metadata files (e.g., a file to track added files and commit history).
- *Input:* `$ bku init`
- *Output (Success):*

Backup initialized.

- *Output (Error - .bku exists):*

```
Error: Backup already initialized in this folder.
```

- *Notes:* Students can design the .bku folder structure (e.g., a tracked_files list and history.log).

2. Add Files to Tracking

- *Purpose:* Adds a file (or all files in the current folder) to the backup tracking system.
- *Behavior:* If <file location> is provided, track that file. If empty, track all files in the current folder (excluding .bku). Store tracked file info in .bku. This can be considered the first commit of each file, but we do not need to log this action.
- *File Deletion Assumption:* To simplify our requirements, we will assume that all tracked files will not be deleted.
- *Input (Single File):* `$ bku add ./src/main.c`
- *Output (Success):*

```
Added src/main.c to backup tracking.
```

- *Input (All files, including nested folders with unlimited depth):* `$ bku add`
- *Output (Success, assuming files src/main.c and src/utils.c exist):*

```
Added src/main.c to backup tracking.  
Added src/utils.c to backup tracking.
```

- *Output (Error - File doesn't exist):*

```
Error: src/main.c does not exist.
```

- *Output (Error - File is already tracked):*

```
Error: src/main.c is already tracked.
```

3. Check Tracked File Status

- *Purpose:* Shows the status of tracked files, highlighting changes since the last commit using diff.
- *Behavior:* If <file> is specified, show its status. If empty, show all tracked files. Compare current file contents with the last committed version (stored in .bku).
- *Input (Single File):* `$ bku status ./src/main.c`
- *Output (Changed File, e.g., added "printf("Hello");"):*

```
src/main.c:
@@ -1 +1,2 @@
int main() { return 0; }
+printf("Hello");
```

- *Input (All Files):* \$ bku status
- *Output (Multiple Files, e.g., main.c changed, utils.c unchanged):*

```
src/main.c:
@@ -1 +1,2 @@
int main() { return 0; }
+printf("Hello");
src/utils.c: No changes
```

- *Output (Error - File not tracked):*

```
Error: src/main.c is not tracked.
```

- *Output (Error - No file has been tracked):*

```
Error: Nothing has been tracked.
```

4. Commit Changes

- *Purpose:* Commits changes to tracked files with a timestamp-based commit ID.
- *Behavior:* Generates a commit ID (format: hh:mm-DD/MM/YYYY), stores diffs of changed files in .bku (not full files), and updates history. If no file is provided, commit all changed tracked files. A commit message is mandatory.
- *Input (Single File):* \$ bku commit "Added hello print" ./src/main.c
- *Output (Success, e.g., at 14:30 on Feb 24, 2025):*

```
Committed src/main.c with ID 14:30-24/02/2025.
```

- *Input (All Files):* \$ bku commit "Initial commit"
- *Output (Success, e.g., two changed files):*

```
Committed src/main.c with ID 14:30-24/02/2025.
Committed src/utils.c with ID 14:30-24/02/2025.
```

- *Output (Error - No message):*

```
Error: Commit message is required.
```


- *Output (Error - No change or no file is being tracked):*

```
Error: No change to commit.
```

5. Check Backup History

- *Purpose:* Displays the commit history of all tracked files.
- *Behavior:* Lists all commits in chronological order, ending with the init action.
- *Input:* `$ bku history`
- *Output (Example with two commits):*

```
14:30-24/02/2025: Added hello print (src/main.c,src/
utils.c).
15:00-24/02/2025: Fixed utils function (src/utils.c).
14:00-24/02/2025: BKU Init.
```

- *Notes:* **The last line must always be the init action.**

6. Restore Previous Version

- *Purpose:* Reverts a file (or all tracked files) to its previous committed version.
- *Behavior:* Uses the latest diff stored in `.bku` to restore the file. Only supports reverting to the immediate previous version.
- *Input (A single file will be restored to its previous version, although it did not change in the latest commit):* `$ bku restore ./src/main.c`
- *Output (Success):*

```
Restored src/main.c to its previous version.
```

- *Input (All recently changed files in the latest commit will be restored):* `$ bku restore`
- *Output (Success, e.g., two files):*

```
Restored src/main.c to its previous version.
Restored src/utils.c to its previous version.
```

- *Output (Error - No previous version of the input file):*

```
Error: No previous version available for src/main.c
```

- *Output (Error - No file is being tracked or no change since the backup initialization):*

```
Error: No file to be restored.
```

7. Schedule Automatic Backups

- *Purpose*: Schedules automatic commits of all changed tracked files. Students can use arbitrary commit message.
- *Behavior*: Uses cron to schedule commits daily (00:00), hourly (top of the hour), or weekly (00:00 Monday). Message is "Scheduled backup".
- *Input*: `$ bku schedule --daily`
- *Output (Success)*:

```
Scheduled daily backups at daily.
```

- *Input*: `$ bku schedule --off`
- *Output (Success - as always)*:

```
Backup scheduling disabled.
```

8. Remove Backup Repository

- *Purpose*: Removes all BKU backup features from the folder (opposite of init).
- *Behavior*: Deletes the `.bku` folder and disables any scheduled backups.
- *Input*: `$ bku stop`
- *Output (Success)*:

```
Backup system removed.
```

- *Output (Error - No .bku folder)*:

```
Error: No backup system to be removed.
```

9. System Installation

- *Purpose*: Installs BKU by checking dependencies (diff, cron, etc.), installing them if absent using a package manager, copying `bku.sh` to a system PATH (e.g., `/usr/local/bin/bku`), and making it executable.
- *Input*: `$./setup.sh --install`
- *Output (Success)*:

```
Checking dependencies...
All dependencies installed.
BKU installed to /usr/local/bin/bku.
```

- *Output (Error - Installation failed)::*

```
Checking dependencies...
Error: Failed to install packages. Please check your
package manager or install them manually.
```

10. System Uninstallation

- *Purpose:* Uninstalls BKU by removing bku from the system PATH (e.g., /usr/local/bin/bku), deleting any scheduled cron jobs, and optionally cleaning up residual files, while leaving dependencies intact unless explicitly requested.
- *Input:* `$./setup.sh --uninstall`
- *Output (Success):*

```
Checking BKU installation...
Removing BKU from /usr/local/bin/bku...
Removing scheduled backups...
BKU successfully uninstalled.
```

- *Output (Error - BKU not installed):*

```
Checking BKU installation...
Error: BKU is not installed in /usr/local/bin/bku.
Nothing to uninstall.
```

ADDITIONAL NOTES

- **Backup Process:** Store diffs in .bku using a structure of your choice (e.g., <commit-id>-<file>.diff). Use diff to generate changes and patch to apply them for restoration.
- **Evaluation:** A public test script will be provided to check functionality with sample cases. Final grades will use private test cases.
- **Submission:** Submit bku.sh and setup.sh as instructed by the instructor.

4 INTRODUCTION TO GNU C COMPILER

4.1 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.

4.2 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.

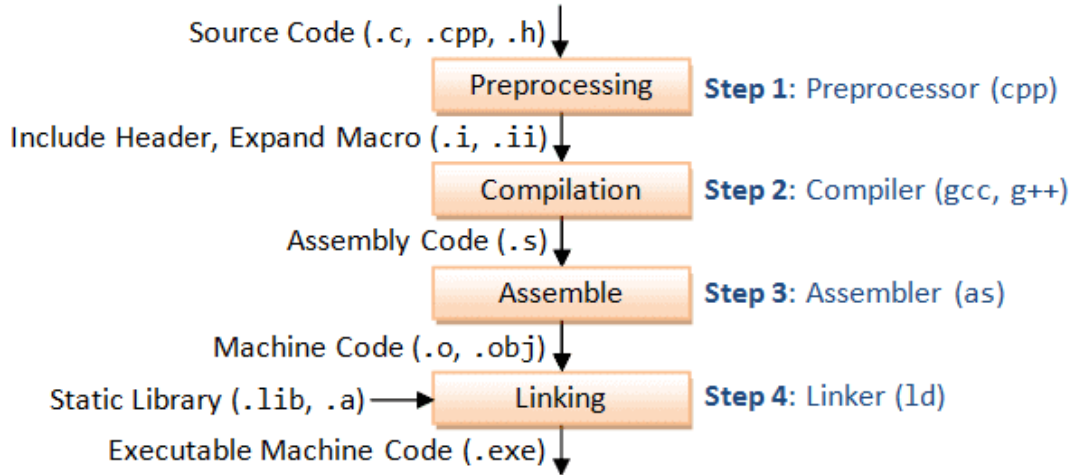


Figure 4.1: C Program Compilation Process

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

```

4.3 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.

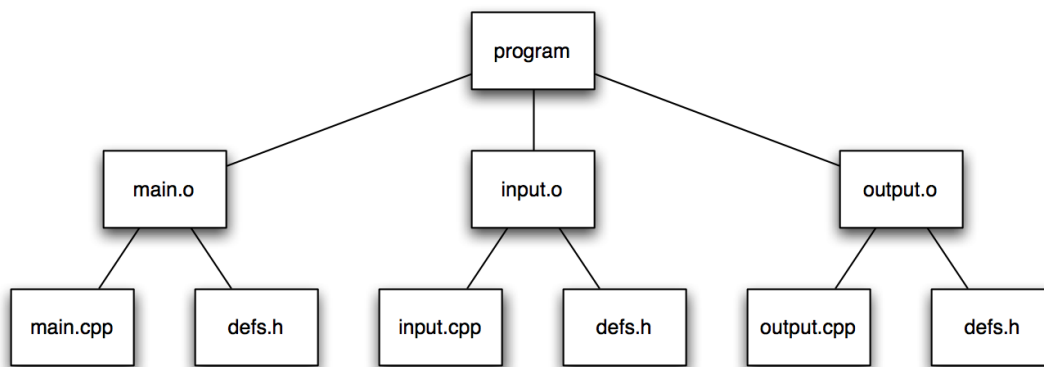


Figure 4.2: Source File Dependency Graph in Make

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

5 GNU DEBUGGER

5.1 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
```

5.2 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.

5.3 EXERCISE

In this exercise, students will develop a **CNew** application, a command-line tool written in C that generates a *boilerplate* structure for simple C projects. This tool automates the setup of a standard directory layout, including a Makefile, source files, and optional Git initialization, streamlining project creation. Students will also implement an installation/uninstallation process via a Makefile. Below are the detailed requirements and expected behaviors (with exact inputs and outputs).

REQUIREMENTS

- **Objective:** Create a C program named `cnew` that generates a boilerplate C project structure.
- **Implementation:** Write the tool in C, using a Makefile for compilation and installation. External tools like `git` may only be invoked via system calls for the `--with-git` option.
- **Submission:** Submit two files:
 - `main.c`: The C source code for the `cnew` tool.
 - Makefile: Compiles `main.c` into the `cnew` executable and handles installation/uninstallation.

COMMANDS AND SPECIFICATIONS

1. Project Creation

- *Purpose:* Generates a new C project with a standard directory layout in a folder named `<project-name>`.
- *Behavior:* Creates a directory `<project-name>` with subdirectories `src` (for source files) and `include` (for header files), a Makefile, `src/main.c`, and `README.md`. Checks if `<project-name>` already exists to avoid overwriting. `<project-name>` must contain only letters, numbers, and hyphens (no spaces).
- *Input:* `$ cnew --name my-project`

- *Output (Success):*

```
Created C project 'my-project' with standard layout.
Running 'make all' to verify...
Build successfully. Binary 'program' created.
Output check successfully.
Project setup complete.
```

- *Output (Error - Directory exists):*

```
Error: Directory 'my-project' already exists.
```

- *Output (Error - Invalid project name):*

```
Error: Project name 'my project' is invalid. Use only
      letters, numbers, and hyphens (e.g., my-project).
```

2. Project Creation with Git

- *Purpose:* Generates a C project and initializes it as a Git repository with a .gitignore file.
- *Behavior:* Same as above, plus initializes a Git repository in <project-name> and creates a .gitignore with C-specific entries (e.g., *.o, program).
- *Input:* `$ cnew --name my-project --with-git`
- *Output (Success):*

```
Created C project 'my-project' with standard layout.
Initialized Git repository.
Added .gitignore for C projects.
Running 'make all' to verify...
Build successfully. Binary 'program' created.
Output check successfully.
Project setup complete.
```

- *Output (Error - Git not installed):*

```
Error: Git is not installed. Install it to use --with-
      git.
```

3. Help Display

- *Purpose:* Displays usage information for the cnew tool.
- *Behavior:* Prints a help message detailing available options and exits.
- *Input:* `$ cnew --help`

- *Output:*

```
Usage: cnew [options]
Options:
  --name <project-name>    Create a new C project with
                           the given name (required).
                           Name must contain only
                           letters, numbers, and
                           hyphens.
  --with-git                Initialize the project as a
                           Git repository with a .gitignore.
  --help                    Display this help message.
Example:
  cnew --name my-project --with-git
```

4. Project Structure and Makefile

- *Purpose:* The generated project must have a standard layout and a functional Makefile.
- *Behavior:* Creates:
 - <project-name>/src/main.c: A simple C file (students define its content, e.g., a "Hello, World!" program).
 - <project-name>/Makefile: Includes all (builds program) and clean (removes program and object files).
 - <project-name>/README.md: Contains the project name (e.g., "# my-project").
- *Output of make all:* Produces a program binary in <project-name>/.

5. Program Installation

- *Purpose:* Installs cnew by compiling main.c into an executable and adding it to the system PATH.
- *Behavior:* The submission Makefile includes install and uninstall targets. The install target compiles main.c into cnew and moves it to /usr/local/bin, while the uninstall target removes it and cleans up.
- *Input:* \$ make install
- *Output (Success - as always):*

```
cnew installed successfully.
```

- *Input:* \$ make uninstall
- *Output (Success - as always):*

```
cnew uninstalled successfully.
```

ADDITIONAL NOTES

- **Verification:** After generating the project, `cnew` runs `make all` in the new directory and executes `./program` to check output. Students define `main.c` content, but it must compile and run.
- **Error Handling:** Check for directory existence, valid project names, and Git availability (if `--with-git` is used). Exit with appropriate error codes (e.g., `exit(1)`).
- **Evaluation:** A public test script will verify functionality (e.g., project creation, Git setup, Makefile targets, installation). Private test cases will determine final grades.
- **Submission:** Submit `main.c` and `Makefile` as instructed 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

Makefile 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 -I moduleA.
#   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
3.0	09.23	PD Nguyen	Update the exercise content
4.0	26.02.25	Thanh HLH	Update exercise with new content