National University of Computer and Emerging Sciences

Laboratory Manual

for

Operating Systems Lab

(BCS-4B)

Course Instructor	Dr. Rana Asif Rehman
Lab Instructor(s)	Mr. Sohaib Ahmed Ms. Haiqa Saman
Section	CS-4B
Semester	Spring 2023

Department of Computer Science

FAST-NU, Lahore, Pakistan

Objectives:

- ✓ Practice commands on terminal
- ✓ MakeFile Utility

Important Note:

- Comment your code intelligently.
- Indent your code properly.
- Use meaningful variable names.
- Use meaningful prompt lines/labels for input/output.
- Use meaningful project and C++ file name

Detailed Description about Makefile Utility:

https://www.gnu.org/savannah-checkouts/gnu/make/manual/make.html

MakeFile

• Make is Unix utility that is designed to start execution of a makefile. A makefile is a special file, containing shell commands that you create and name makefile (or Makefile depending upon the system). While in the directory containing this makefile, you will type *make* and the commands in the makefile will be executed. If you create more than one makefile, be certain you are in the correct directory before typing make.

• Make keeps track of the last time files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the sourcefile up-to-date. If you have a large program with many source and/or header files, when you change a file on which others depend, you must recompile all the dependent files. Without a makefile, this is an extremely time-consuming task.

• As a makefile is a list of shell commands, it must be written for the shell which will process the makefile. A makefile that works well in one shell may not execute properly in another shell.

• However, for a large project where we have thousands of source code files, it becomes difficult to maintain the binary builds. The **make** command allows you to manage large programs or groups of programs.

• The **make** program is an intelligent utility and works based on the changes you do in your source files. If you have four files main.cpp, hello.cpp, factorial.cpp and functions.h, then all the remaining files are dependent on functions.h, and main.cpp is dependent on both hello.cpp and factorial.cpp. Hence if you make any changes in functions.h, then the **make** recompiles all the source files to generate new object files. However, if you make any change in main.cpp, as this is

not dependent of any other file, then only main.cpp file is recompiled, and help.cpp and factorial.cpp are not.

- While compiling a file, the **make** checks its object file and compares the time stamps. If source file has a newer time stamp than the object file, then it generates new object file assuming that the source file has been changed.

Structure of Makefile:

Target: dependencies
    Action/Command

Naming of Makefile:

By default, when make looks for the makefile, it tries the following names, in order: `GNUmakefile', `makefile' and `Makefile'. You can give any of the three names to your makefile. The convention is to us the name "Makefile" (capital M).

Running the Makefile:

Simply run the command "make". The current working directory should be where the intended makefile is placed.

Benefits of Makefile:

Makefile checks the last modified time of both the source file and the output file. If the output file's last modified time is later, then it will not compile the source files since the outputfile is already latest. However, if any of the source files is modified after the creation of output file, then it will run the command since the output file is outdated.

- ✓ By default, make command executes the first target of Makefile.

Example:

Suppose we have two cpp files: main.cpp, lib.cpp, and and a header file lib.h. Suppose the main function in main.cpp makes use of several functions from lib.cpp. In order to compile our program, we will create the makefile as follows:

```
main.out: main.cpp lib.cpp
    g++ main.cpp lib.cpp -o main.out
```

Question 5: (10 marks)

Create 3 files

- main.c
- functions.c
- header.h

header.h file contains following function prototypes

```
void sort(int array[], bool order);
void findHighest(int array[], int postion);
void print(int array[]);
```

functions.c file contains following 3 functions along with their logic

```
void sort(int array[], bool order) {
    > sort in ascending order if order is true
    > sort in descending order if order is false
}

void findHighest(int array[], int nth){
    > find nth highest value
    if nth = 2 find 2nd highest value from the array
}

void print(int array[]){
    > print all elements in the array
}
```

In main.c you will accept command line arguments including 3 things

- an array of integers
- order of sort (1 for ascending order and 0 for descending order)
- nth position to get the nth highest number from the array

Use makefile to execute all these files. Your makefile will look like this.

Example:

```
main: main.o functions.o
    gcc main.o functions.o -o main

main.o: main.c
    gcc -c main.c

functions.o: functions.c
    gcc -c functions.c

clean:
    rm *.o main
```

Input: ./main 11 15 13 12 16 14 18 19 20 17

Output:

```
Array Element:  11 15 13 12 16 14 18 19 20 17
Sorted Elements:  11 12 13 14 15 16 17 18 19 20
The 4 highest value in the array is: 17
```