National University of Computer and Emerging Sciences

Laboratory Manual

for

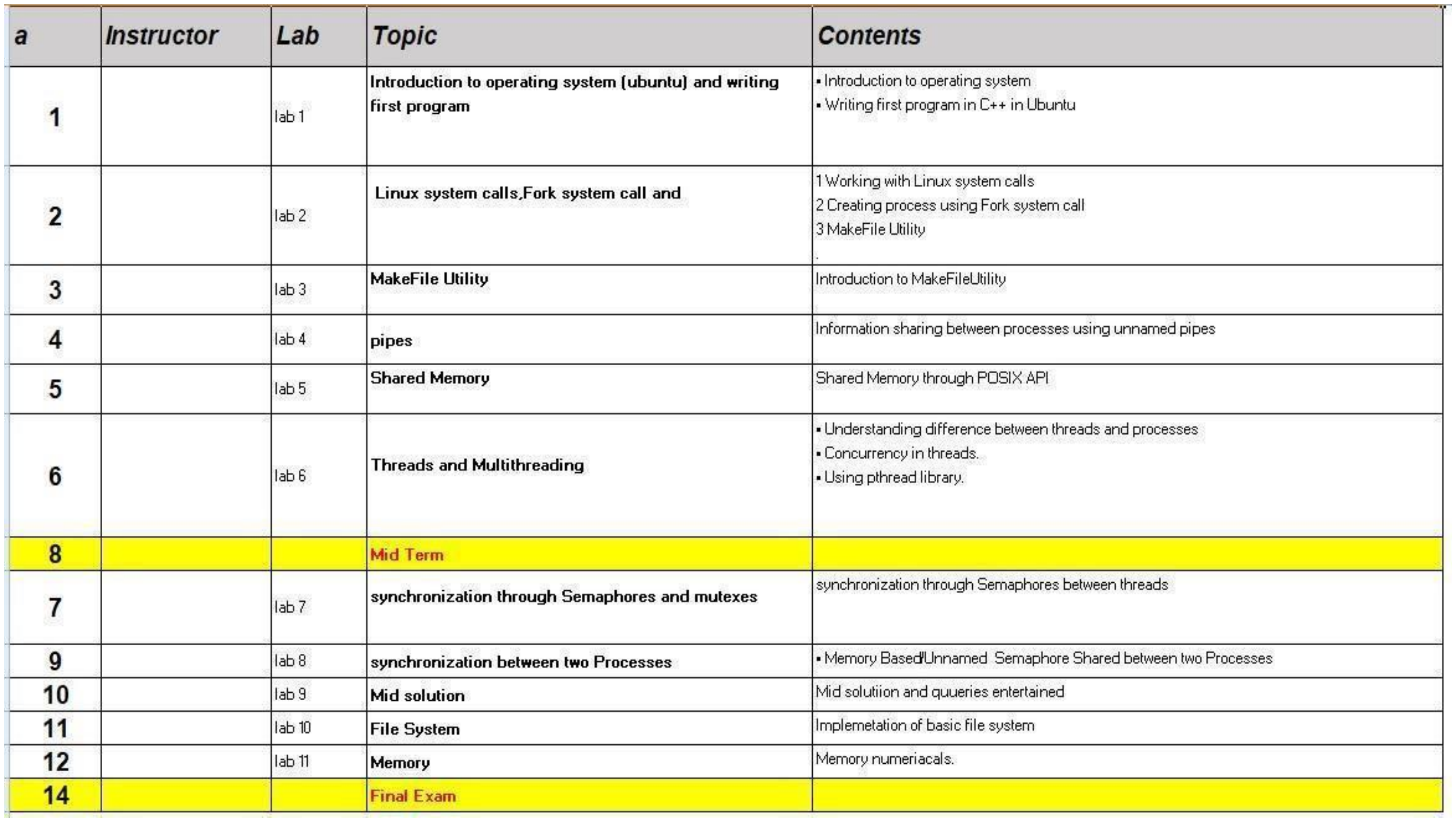
Operating Systems Lab

Course Instructor	Sir Mubashar Hussain
Lab Instructor(s)	Muhammad Hassan Raza
Section	SE-5B
Semester	Fall 2024

Department of Software Engineering

FAST-NU, Lahore, Pakistan

Tentative Course Outline:



a	Instructor	Lab	Topic	Contents
1		lab 1	**Introduction to operating system (ubuntu) and writing first program**	▪ Introduction to operating system ▪ Writing first program in C++ in Ubuntu
2		lab 2	**Linux system calls,Fork system call and**	1 Working with Linux system calls 2 Creating process using Fork system call 3 MakeFile Utility .
3		lab 3	**MakeFile Utility**	Introduction to MakeFileUtility
4		lab 4	**pipes**	Information sharing between processes using unnamed pipes
5		lab 5	**Shared Memory**	Shared Memory through POSIX API
6		lab 6	**Threads and Multithreading**	▪ Understanding difference between threads and processes ▪ Concurrency in threads. ▪ Using pthread library.
8			Mid Term	
7		lab 7	**synchronization through Semaphores and mutexes**	synchronization through Semaphores between threads
9		lab 8	**synchronization between two Processes**	▪ Memory Based/Unnamed Semaphore Shared between two Processes
10		lab 9	**Mid solution**	Mid solutiion and quueries entertained
11		lab 10	**File System**	Implemetation of basic file system
12		lab 11	**Memory**	Memory numeriacals.
14			Final Exam	

Objectives

In this lab, students will:

1. Practice Basic commands on terminal
2. Develop a small program in C for reading/writing files
3. Learn about Makefile

Basic Commands

• Clear the console: **clear**

• Changing working Directory: **cd Desktop**   **cd Home**

• List all files in directory: **ls**

• Copy all files of a directory within the current work directory: **cp dir/***

• Copy a directory within the current work directory: **cp -a tmp/dir1**

• Look what these commands do **cp -a dir1 dir2 cp filename1 filename2**

Compiling C and C++ Programs on the Terminal:

For C++:
Command: g++ source_files… -o output_file

For C:
Command: gcc source_files… -o outputfiles

Example:
gcc main.c lib.c –o run.exe

Passing Command Line Arguments to a C/C++ Program

- Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C/C++ programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

- To pass command line arguments, we typically define main() with two arguments: **first argument counts the number of arguments** on the command line and **the second is a pointer array which holds pointers of type char which points to the arguments** passed to the program. The syntax to define the main method is **int main (int argc, char *argv[])**.

- Here, **argc** variable will hold the number of arguments pass to the program while the argv will contain pointers to those variables. **argv[0] holds the name of the program while argv[1] to argv[argc] hold the arguments**.

- Command-line arguments are given after the name of the program in command-line shell of Operating Systems. Each argument separated by a space. If a space is included in the argument, then it is written in “”.

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
```

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.

Example:

Suppose we have two cpp files: main.cpp, lib.cpp, and and a heade 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:

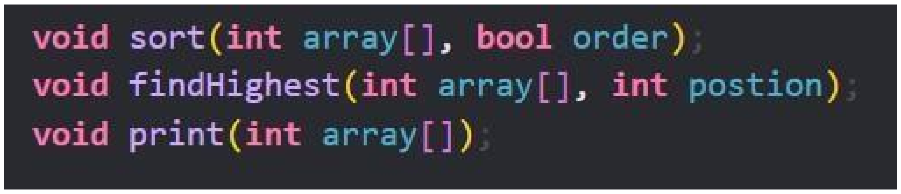
```
main.out: main.cpp lib.cpp     g++

main.cpp lib.cpp -o main.out
```

Question 1:

Create 3 files

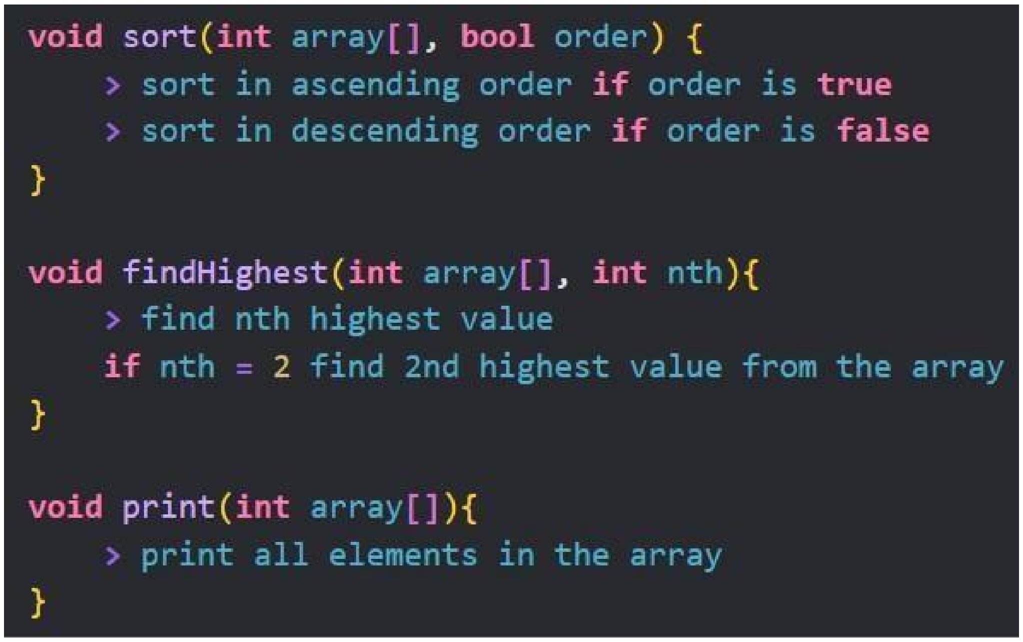
- main.c
- functions.c
- header.h

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

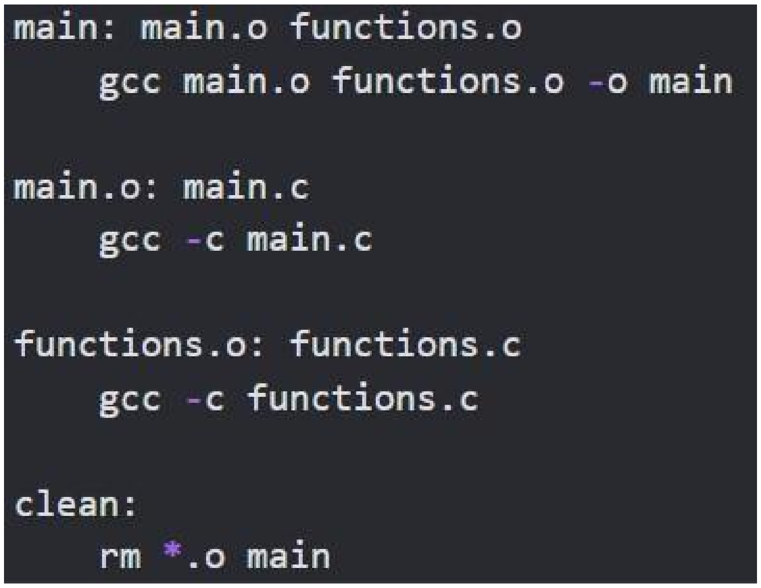


```
void sort(int array[], bool order) {
    > sort in ascending order if order is true
    > sort in descending order if order is false
}

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

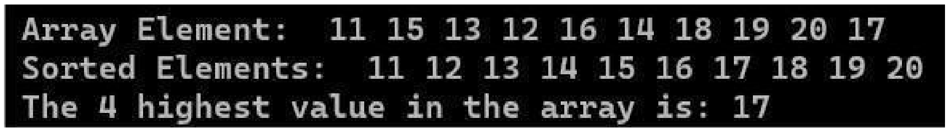
Use makefile to execute all these files. Your makefile will look like this.



```
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
```

Example:

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
```

Question No 2

Write a program which creates a process using fork() system call. A child process will be created. In child process, read from a input file and write all numbers in an output file. All alphabets in input file should be ignored. In parent process, read numbers from output file and take sum and average of those numbers and print it. **Names of file should be entered through command line. (**recall command line arguments)

NOTE: Parent should wait for child to finish its task first
You will need to create 2 text files before initiating this task and write some random number and alphabets in it.
You will consider each single number as a separate number i.e range of numbers will be 0-9