National University of Computer and Emerging Sciences

Lab Manual 07
CL461-Artificial Intelligence Lab

Course Instructor	Mubashar Husnain
Lab Instructor (s)	Abdul Rehman Ajmal Sae
Section	BCS-4H
Semester	Spring 2024

Task # 1

Run Basic Terminal Command lines

Command	Purpose
clear	*to clear the terminal screen*
ls	*to list all the files in a directory*
pwd	*print working directory*
cd	*to change the working directory i.e. cd Home, cd root*
mv	*move or rename files or directories*
cp	*copy files or directories*
man	*use to display manual pages of other commands*
touch	*to create a file i.e. touch (name for file).(extension for for)*
*cp dir/**	*to copy all the files of a directory within the current folder*
cp -a tmp/dir1	*to copy a folder within the current folder*
top	*display the system resource usage*
df	*display disk space usage*
sudo	*execute a command with superuser privileges*
ping	*to test the reachability of a host on an internet*
head / tail	*display beginning or end of a file*
less	*display file contents with pagination*
chmod	*change file permissions*
grep	*search for patterns in files*
cat	*concatenate and display file content*

Try on terminal:

```
cd /path/to/directory

mv old_file.txt new_file.txt

cp file.txt /path/to/destination

man ls

touch new_file.txt

cp dir/* .

cp -a tmp/dir1 .

sudo <command>

ping www.example.com

chmod 755 file.txt

grep "pattern" file.txt

cat file.txt
```

Task # 2

Compile and Run a C/C++ program on terminal

Step#1:

write a c or c++ program

Step#2:

check if g/g++ is installed in the terminal.

gcc/g++ --version   *or*   *which gcc/g++*

Step#3 (optional):

Install gcc/g++ in the terminal.

sudo apt-get update

sudo apt-get install gcc / g++

Step#4:

compile the code files using gcc / g++.

gcc/g++ source_files -o output_files

Step#5:

run the code executable or object file.

./output_files

Example:

```
gcc test.c test_lib.c –o run.exe
g++ Hello.cpp -o test
./test
```

Task # 3

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

Command line arguments are parameters supplied to a program when it is invoked.

They allow controlling the program from outside.

In C/C++ programming, command line arguments are passed to the main() function.

The main() function typically takes two arguments: argc and argv[].

- argc holds the number of arguments passed to the program.
- argv[] is an array of pointers to char that points to the arguments passed to the program.
- 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 the command-line shell of operating systems. Each argument is separated by a space.

If an argument contains spaces, it is enclosed in double quotation marks.

Example: ./program arg1 arg2 "argument with spaces"

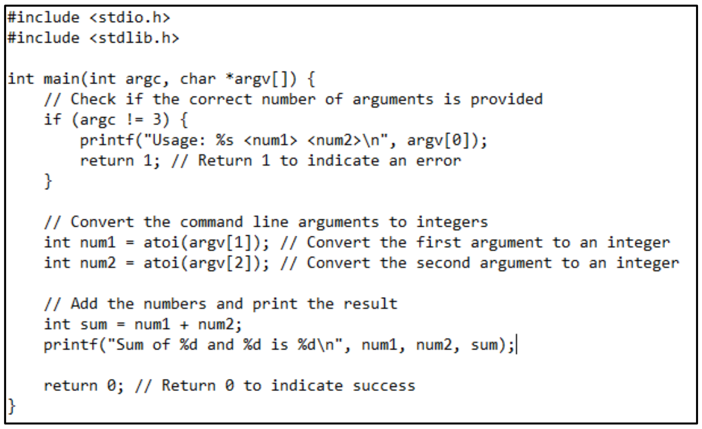
Here, arg are in char so, if we wish to use them in code we must convert them to the required form that is needed in the code.

Suppose you have ./add 35 59

So first, convert them to integers by using a function atoi. and print the sum on the screen:

cout<< atoi(argv[1]) + atoi(argv[2]);

Sample:



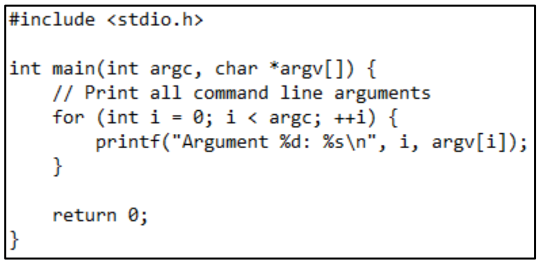
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    // Check if the correct number of arguments is provided
    if (argc != 3) {
        printf("Usage: %s <num1> <num2>\n", argv[0]);
        return 1; // Return 1 to indicate an error
    }

    // Convert the command line arguments to integers
    int num1 = atoi(argv[1]); // Convert the first argument to an integer
    int num2 = atoi(argv[2]); // Convert the second argument to an integer

    // Add the numbers and print the result
    int sum = num1 + num2;
    printf("Sum of %d and %d is %d\n", num1, num2, sum);

    return 0; // Return 0 to indicate success
}
```



```
#include <stdio.h>

int main(int argc, char *argv[]) {
    // Print all command line arguments
    for (int i = 0; i < argc; ++i) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

Command for terminal:

./add 5 7   ,   ./print "Hey there (^_^)/ , my name is Anato."

Task # 4

MakeFile

What is a MakeFile?

The **make** utility is a tool used to manage the compilation and building of software projects, particularly those with many source code files. It operates based on a set of rules defined in a special file called a makefile. Here's a summary of its key features and functionality:

- **Execution of Makefiles:** make initiates the execution of a makefile, which contains shell commands for building the project. By typing make in the directory containing the makefile, the commands specified in the makefile are executed.
- **Dependency Tracking:** Make keeps track of the last modification times of files, typically object files. It updates only those files that are necessary to keep the project up-to-date. When a source file is changed, it recompiles only the dependent files, which saves time, especially in large projects with many source files.
- **Management of Large Projects:** For large projects with numerous source code files, maintaining binary builds manually becomes challenging. make simplifies this process by intelligently managing the compilation process based on the changes made to the source files.
- **Dependency Resolution:** make understands the dependencies between source files. If a source file is modified, it recompiles only the files directly or indirectly dependent on it. This reduces unnecessary recompilation, making the build process more efficient.
- **Time Stamp Comparison:** When compiling a file, make compares the time stamps of the source file and its corresponding object file. If the source file is newer, indicating that it has been modified, make generates a new object file.

In essence, **make** automates the compilation process, ensuring that only the necessary files are recompiled when changes are made, thereby improving efficiency and productivity in software development.

Structure of Makefile:

Each rule in a Makefile consists of a target, dependencies, and actions.

Syntax: Target: dependencies

Action

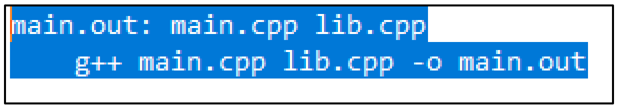
Benefits of Makefile:

Makefile checks the last modified times of source files and output files to determine whether compilation is necessary. If the output file is up-to-date with respect to the source files, no compilation is performed. If any source file is modified after the creation of the output file, the Makefile runs the compilation command to update the output file.

Example:

Suppose we have two C++ files: main.cpp and lib.cpp, along with a header file lib.h.
main.cpp utilizes functions from lib.cpp.

The Makefile for compiling this program would look like:

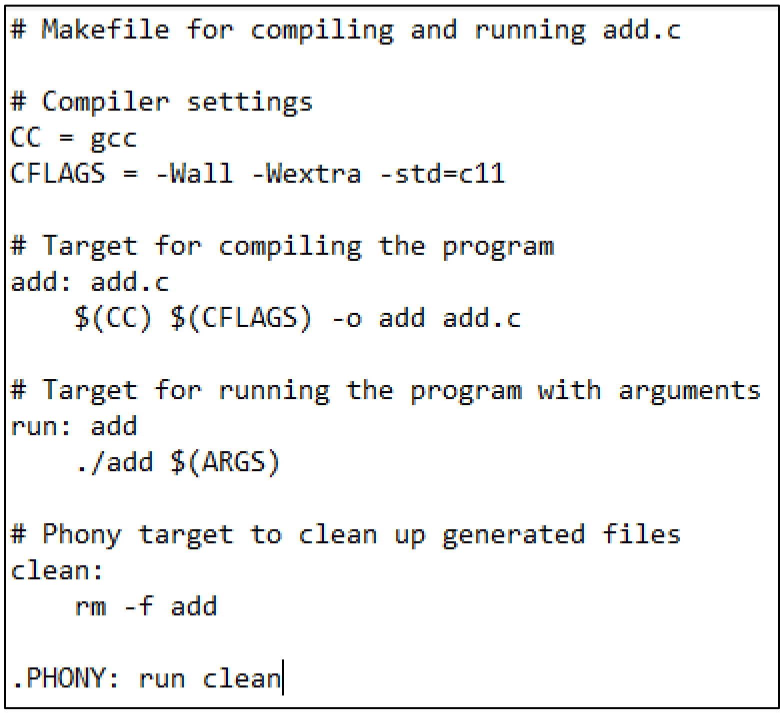


```
main.out: main.cpp lib.cpp
    g++ main.cpp lib.cpp -o main.out
```

Now, one might think, can I pass arg to makefile 🤔

Yes, you can create a Makefile to compile and run a C program while taking Arguments.

Lets make a Makefile to add two numbers:



```
# Makefile for compiling and running add.c

# Compiler settings
CC = gcc
CFLAGS = -Wall -Wextra -std=c11

# Target for compiling the program
add: add.c
    $(CC) $(CFLAGS) -o add add.c

# Target for running the program with arguments
run: add
    ./add $(ARGS)

# Phony target to clean up generated files
clean:
    rm -f add

.PHONY: run clean
```

In this Makefile:

- **CC** specifies the compiler to use (in this case, gcc).
- **CFLAGS** specifies compiler flags, such as enabling warnings and specifying the C standard.
- The **add** target compiles the add.c program into an executable named add.
- The **run** target depends on the add target, ensuring that the program is compiled before it is executed. It runs the add program with the arguments provided through the ARGS variable.
- The **clean** target removes the generated executable file.
- .**PHONY** is used to specify that run and clean are not actual files but rather phony targets, meaning they do not represent actual files in the directory.

To Compile and run this Makefile:

- Compile the program:
  make

- Run the program with arguments (e.g., add 23 45):
  make ARGS="23 45" run

Task # 5

System Calls

There are 5 different categories of system calls

- Process control
- File management
- Device management
- Information maintenance
- Communication

What is a Process?

Process refers to an instance of a running program on a computer system.
It is the basic unit of execution in an operating system

How to check or see them on terminal?

*The **ps** command displays information about active processes.*

*The **top** command provides a dynamic, real-time view of running processes, system resource usage, and more. It continuously updates the display until you manually exit it*

*Similar to top, **htop** is an interactive process viewer that provides a more user-friendly interface with more features and customization options.*

*The **pgrep** command searches for processes by name and prints their process IDs (PIDs).*

*The **pidof** command returns the PID of a running program*

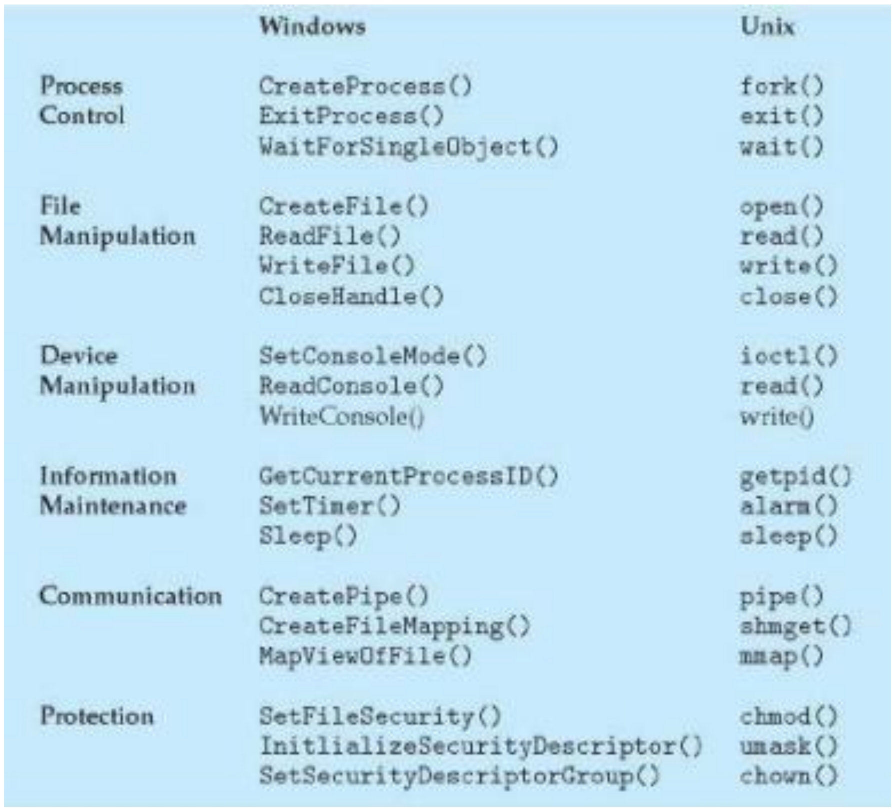
*The **ps aux** command displays a detailed list of all running processes on the system.*

How many types of Processes?

- *Foreground Processes:*
  They run on the screen and need input from the user.
  For example Office Programs

- *Background Processes: They run in the background and usually do not need user input.*
  For example Antivirus.

Init is the parent of all Linux processes. It is the first process to start when a computer boots up, and it runs until the system shuts down. It is the ancestor of all other processes.



	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Examples of Windows and Unix System Calls

1. Process control

fork()

It is a system call in Unix-like operating systems that creates a new process, known as the child process, which is a duplicate of the calling process, known as the parent process.

Return Value:
- In the parent process, fork() returns the process ID (PID) of the child process.
- In the child process, fork() returns 0 to indicate that it is the child process.

Parent Process:
- Invokes fork() system call to create a child process.
- Continues execution from the next line after fork().
- Has its own copy of any data.
- Receives a return value from fork() greater than 0, which is the PID of the child process.

Child Process:
- Created by the fork() system call.
- A duplicate/copy of the parent process on Linux.
- Has a separate address space from the parent process.
- Shares the same code segments as the parent process.
- Executes independently of the parent process.
- Continues execution from the next line after fork().
- Has its own copy of any data.
- Receives a return value of 0 from fork() to indicate that it is the child process.

<u>wait()</u>
It is a function is used by the parent process in Unix-like operating systems to wait for the termination of its child process

- `wait()` is used by the parent process.
- The parent's execution is suspended.
- The child process continues its execution.
- Upon termination of the child, `wait()` returns an exit status to the operating system.
- The exit status is then returned to the waiting parent process.
- The parent process resumes execution.
- Required header files: `<sys/wait.h>` and `<sys/types.h>`.

<u>exit()</u>
It is a system call used by a process to terminate its execution. It returns an exit status, retrieved by the parent process using wait().

- Common exit status values include EXIT_SUCCESS (0) and EXIT_FAILURE (1).
- OS reclaims resources allocated by the terminated process and performs cleanup operations.
- exit() ensures proper termination of the process.
- _exit(): Terminates the current process without additional cleanup. Typically used by child processes to avoid releasing resources belonging to the parent process.

<u>exec()</u>
The exec system call is used to replace the current program in an active process with a new program.

- It replaces the previous executable file with a new one, effectively loading a new program into the process.
- The entire content of the process is replaced with the new program.
- The user data segment executing the exec() system call is replaced with the data file specified as an argument.
- The new program is loaded into the same process space, so the process ID (PID) remains unchanged.
- However, the data, code, stack, heap, etc., of the process are replaced with those of the newly loaded program.
- The new program starts execution from its entry point.
- exec system calls are a collection of functions, with common names like execl, execle, execlp, execv, execve, and execvp.
- The suffixes in the function names denote different features:
    l: Indicates command line arguments passed as a list to the function.
    p: Indicates usage of the path environment variable to find the file to be loaded into the process.
    v: Indicates command line arguments passed as an array of pointers to the function.

2. File management & Device management
   open()
   read()
   write()
   close()

3. Information maintenance
   <u>sleep()</u>
   <u>getpid()</u>

4. Communication
   <u>pipe()</u>
   <u>shmget()</u>
   <u>mmap()</u>

Syntaxes of exec family functions:

The following are the syntaxes for each function of exec:

```
int execlp(const char* file, const char* arg, …)
int execv(const char* path, const char* argv[])
```

Execv ()

In execv() ,system call doesn’t search the PATH. Instead, the full path to the new executable must be specified.

Path:The path to the new program executable.

Argv:Argument vector. The argv argument is an array of character pointers to null-terminated strings. The last member of this array must be a null pointer. These strings constitute the argument list available to the new process image. The value in argv[0] should point to the filename of the executable for the new program.

```
#include &lt;unistd.h&gt;
int main(void) {
char*binaryPath =&quot;/bin/ls&quot;;
char*args[]= {binaryPath,&quot;-lh&quot;,&quot;/home&quot;, NULL};
execv(binaryPath, args);
return 0;
}
```

Execlp ()

execlp() uses the PATH environment variable. So, if an executable file or command is available in the PATH, then the command or the filename is enough to run it, the full path is not needed.

```
#include &lt;unistd.h&gt;
int main(void) {
char char*programName =
*arg1 = &quot;-lh&quot;;

&quot;ls&quot;;
char*arg2 =&quot;/home&quot;;
execlp(programName, programName, arg1, arg2, NULL);
return 0;
}
```

Open System call for Filing

Open system call is used for opening a file.

int open(const char *pathname, int flags, mode_t mode);

1. pathname is a file name
2. The argument flags must include one of the following access modes

O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file in read-only, write-only, or read/write modes, respectively. Apart from above, flags can also have any of the following:

(A) O_APPEND (file is opened in append mode)

(B) O_CREAT (If pathname does not exist, create it as a regular file.)

(C) O_EXCL Ensure that this call creates the file: if this flag is specified in conjunction with O_CREAT, and pathname already exists, then open() fails.

Note: to use two flags at once use bitwise OR operator, i.e., O_WRONLY | O_CREAT

3. Mode is only required when a new file is created and is used to set permissions on the new file

4 Read/Write System Call

The read () system call receives the user’s input from the file that is the file descriptor and puts it in the buffer “buff”, which is just a character array. It can only read up to specified number of bytes at a time.

read (int fd, void* buf, bytes);

write (int fd, void* buf, bytes);

Seek() System Call

Read system call reads from start. What if we want to read or write from a specific portion of file? Lseek() is used to read or write at specific portion of file.

Lseek(fd,bytes, SEEK_CURR)

Instead of SEEK_CURR, which is used to move pointer from current position upto specified bytes, you can also use SEEK_SET to set pointer from beginning of file and SEEK_END from end of file.

Inlab Question (10 marks)

Question 1

Create 2 processes using fork system call such that all the processes have same parent. In first process, use array of alphabets and special characters entered through command line. This process will print array and its size and then call exec system to pass the previous array and remove all special characters. So that, only alphabets remain in that array. This file further calls exec and pass updated array to this. In this program, you have to find number of elements in array and its size in bytes. In second fork you need to print PID of this process. PIDs of both exec and fork should be printed on your screen. Parent should wait until child has finished its execution.

Question 2

Design a program using lseek() to read alternative byte from a file. You have to create a txt file and write" OperatingSystem" in it and then add any alphabet after each letter in previously mentioned text. Using lseek() read alternative alphabets and write it on screen.

Note: Use only read, write and open system calls. Use of Cin, cout, prinf, ofstream,

ifstream etc. will result in zero marks.

Help:

man 2 open

man 2 read

man 2 write