FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY
Hamdard Institute of Engineering & Technology
Hamdard University

Operating Systems Laboratory Master Manual

Semester 5th

Name: _____

Roll Number: _____

Department of Computing
Hamdard Institute of Engineering & Technology
Hamdard University

_____	_____	_____
Mr. Iqbal Uddin khan	Mr. Iqbal Uddin Khan	Dr. Aqeel ur Rehman
Subject Lab Instructor	Subject Teacher	Sectional Head/Chairman

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY
Hamdard Institute of Engineering & Technology
Hamdard University

Operating Systems Laboratory Manual

Semester 5th

Name: _____

Roll Number: _____

Department of Computing
Hamdard Institute of Engineering & Technology
Hamdard University

General Laboratory Procedure:

While there is no specific document to be submitted at the beginning of the Lab –unless your instructor advises you otherwise-, you are expected to read the experiment fully before you come to the laboratory? Interestingly, you can even try parts of the experiment at home. Here is a list of programs that will equip you with a virtual lab at your home:

Troubleshooting

Things will not always go as expected; this is the nature of the learning process. While conducting the Experiment **think before you do anything.** If you do so you will avoid wasting time going down dead-end streets. Be logical and systematic. First, look for obvious errors that are easy to fix. Is your measuring device correctly set and connected? Are you looking at the proper scale? Is the power supply set for the correct voltage? Is the signal generator correctly set and connected? How are the variables in the code set? Is there a syntax error? And so on. Next, check for obvious misconnections or broken connections, at least in simple circuits.

As you work through your circuit, use your Lab Manual record tests and changes that you make as you go along; don't rely on your memory for what you have tried. Identify some test points in the system at which you know what the signal should be and work your way backwards from the output through the test points until you find a good signal.

Neatness

When you have finished for the day, return all modules to their proper storage bins, return all test leads and probes to their storage racks, return all equipment to its correct location, and clean up the lab station. If appropriate switch off the unneeded equipment. Save your files in the Computer and on any USB device for your records because you might not get the same PC System again for the next experiment. Also email your file contents to your email address as a backup.

Laboratory Safety

Always pay attention to what you are doing and you're surrounding during the experiments and notify the Instructor for any unlikely event or mishap and leave the Laboratory with the permission of Instructor immediately.

All students must read and understand the information in this document with regard to laboratory safety and emergency procedures prior to the first laboratory session.

Your personal laboratory safety depends mostly on YOU.

Effort has been made to address situations that may pose a hazard in the lab but the information and instructions provided cannot be considered all-inclusive.

Students must adhere to written and verbal safety instructions throughout the academic term. Since additional instructions may be given at the beginning of laboratory sessions, it is important that all students arrive at each session on time. With good judgement, the chance of an accident in this course is very small. Nevertheless, research and teaching workplaces (labs, shops, etc.) are full of potential hazards that can cause serious injury and or damage to the equipment. Working alone and unsupervised in laboratories is forbidden if you are working with hazardous substances or equipment. With prior approval, at least two people should be present so that one can shut down equipment and call for help in the event of an emergency. Safety training and/or information should be provided by a faculty member, teaching assistant, lab safety contact, or staff member at the beginning of a new assignment or when a new hazard is introduced into the workplace.

Emergency Response

1. It is your responsibility to read safety and fire alarm posters and follow the instructions during an emergency
2. Know the location of the fire extinguisher, eye wash, and safety shower in your lab and know how to use them.
3. Notify your instructor immediately after any injury, fire or explosion, or spill.
4. Know the building evacuation procedures.

Common Sense

Good common sense is needed for safety in a laboratory. It is expected that each student will work in a responsible manner and exercise good judgement and common sense. If at any time you are not sure how to handle a particular situation, ask your Teaching Assistant or Instructor for advice. **DO NOT TOUCH ANYTHING WITH WHICH YOU ARE NOT COMPLETELY FAMILIAR**!!! It is always better to ask questions than to risk harm to yourself or damage to the equipment.

Personal and General laboratory safety

1. Never eat, drink, or smoke while working in the laboratory.
2. Read labels carefully.
3. Do not use any equipment unless you are trained and approved as a user by your supervisor.
4. Wear safety glasses or face shields when working with hazardous materials and/or equipment.
5. Wear gloves when using any hazardous or toxic agent.
6. Clothing: When handling dangerous substances, wear gloves, laboratory coats, and safety shield or glasses. Shorts and sandals should not be worn in the lab at any time. Shoes are required when working in the machine shops.
7. If you have long hair or loose clothes, make sure it is tied back or confined.

8. Keep the work area clear of all materials except those needed for your work. Coats should be hung in the hall or placed in a locker. Extra books, purses, etc. should be kept away from equipment, that requires air flow or ventilation to prevent overheating.
9. Disposal - Students are responsible for the proper disposal of used material if any in appropriate containers.
10. Equipment Failure - If a piece of equipment fails while being used, report it immediately to your lab assistant or tutor. Never try to fix the problem yourself because you could harm yourself and others.
11. If leaving a lab unattended, turn off all ignition sources and lock the doors.
12. Never pipette anything by mouth.
13. Clean up your work area before leaving.
14. Wash hands before leaving the lab and before eating.
15. Unauthorized person(s) shall not be allowed in a laboratory for any reason

Electrical safety

1. Obtain permission before operating any high voltage equipment.
2. Maintain an unobstructed access to all electrical panels.
3. Wiring or other electrical modifications must be referred to the Electronics Shop or the Building Coordinator.
4. Avoid using extension cords whenever possible. If you must use one, obtain a heavy- duty one that is electrically grounded, with its own fuse, and install it safely. Extension cords should not go under doors, across aisles, be hung from the ceiling, or plugged into other extension cords.
5. Never, ever modify, attach or otherwise change any high voltage equipment.
6. Always make sure all capacitors are discharged (using a grounded cable with an insulating handle) before touching high voltage leads or the "inside" of any equipment even after it has been turned off. Capacitors can hold charge for many hours after the equipment has been turned off.
7. When you are adjusting any high voltage equipment or a laser which is powered with a high voltage supply, USE ONLY ONE HAND. Your other hand is best placed in a pocket or behind your back. This procedure eliminates the possibility of an accident where high voltage current flows up one arm, through your chest, and down the other arm.
8. Discard damaged cords, cords that become hot, or cords with exposed wiring.
9. Before equipment is energized ensure, (1) circuit connections and layout have been checked by a Teaching Assistant (TA) and (2) all colleagues in your group give their assent.
10. Know the correct handling, storage and disposal procedures for batteries, cells, capacitors, inductors and other high energy-storage devices.
11. Experiments left unattended should be isolated from the power supplies. If for a special reason, it must be left on, a barrier and a warning notice are required.
12. Equipment found to be faulty in any way should be reported to the Lab Engineer immediately and taken out of service until inspected and declared safe.
13. Voltages above 50 V RMS AC and 120 V DC are always dangerous. Extra precautions should be considered as voltage levels are increased.

14. Never make any changes to circuits or mechanical layout without first isolating the circuit by switching off and removing connections to power supplies.
15. Know what you must do in an emergency.
16. Emergency Power Off: Every lab is equipped with and Emergency Power Off System.
17. Only authorized personnel are permitted to reset power once the Emergency Power Off system has been engaged.

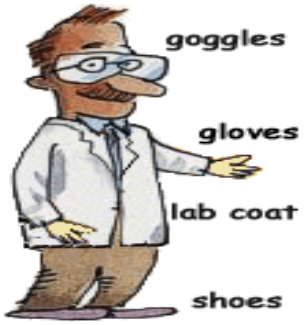
Electrical Emergency Response

The following instructions provide guidelines for handling two types of electrical emergencies:

1. When someone suffers serious electrical shock, he or she may be knocked unconscious. If the victim is still in contact with the electrical current, immediately turn off the electrical power source. If you cannot disconnect the power source, depress the Emergency Power Off switch.
2. Do not touch a victim that is still in contact with a live power source; you could be electrocuted.
3. Have someone call for emergency medical assistance immediately. Administer first-aid, as appropriate.
4. If an electrical fire occurs, try to disconnect the electrical power source, if possible. If the fire is small and you are not in immediate danger; and you have been properly trained in fighting fires, use the correct type of fire extinguisher to extinguish the fire. When in doubt, push in the Emergency Power Off button.
5. NEVER use water to extinguish an electrical fire.

Mechanical safety

1. When using compressed air, use only approved nozzles and never direct the air towards any person.
2. Guards on machinery must be in place during operation.
3. Exercise care when working with or near hydraulically- or pneumatically-driven equipment. Sudden or unexpected motion can inflict serious injury.



Additional Safety Guidelines

1. Never do unauthorized experiments.
2. Never work alone in laboratory.
3. Keep your lab space clean and organized.
4. Do not leave an on-going experiment unattended.
5. Always inform your instructor if you break a thermometer. Do not clean mercury yourself!!
6. Never taste anything. Never pipette by mouth; use a bulb.
7. Never use open flames in laboratory unless instructed by TA.
8. Check your glassware for cracks and chips each time you use it. Cracks could cause the glassware to fail during use and cause serious injury to you or lab mates.
9. Maintain unobstructed access to all exits, fire extinguishers, electrical panels, emergency showers, and eye washes.
10. Do not use corridors for storage or work areas.

11. Do not store heavy items above table height. Any overhead storage of supplies on top of cabinets should be limited to lightweight items only. Also, remember that a 36" diameter area around all fire sprinkler heads must be kept clear at all times.
12. Areas containing lasers, biohazards, radioisotopes, and carcinogens should be posted accordingly. However, do not post areas unnecessarily and be sure that the labels are removed when the hazards are no longer present.
13. Be careful when lifting heavy objects. Only shop staff may operate forklifts or cranes.
14. Clean your lab bench and equipment, and lock the door before you leave the laboratory.

Clothing

1. Dress properly during a laboratory activity.
2. Long hair, dangling jewelry, and loose or baggy
3. Clothing are a hazard in the laboratory.
4. Long hair must be tied back, and dangling jewelry and baggy clothing must be secured.
5. Shoes must completely cover the foot.
6. No sandals allowed on lab days.
7. A lab coat or smock should be worn during laboratory experiments.

Accidents and Injuries

1. Do not panic.
2. Report any accident (spill, breakage, etc.) or injury (cut, burn, etc.) to the teacher immediately, no matter how trivial it seems.
3. If you or your lab partner is hurt, immediately (and loudly) yell out the teacher's name to get the teacher's attention.

General Warning Signs

List of Experiments

S. No	Objective	Page No	Date	Signature
1	Introduction to Operating System	1		
2	Bootloader	5		
3	Introduction to Command Line Interface	11		
4	System Calls	16		
5	Adding Modules in Kernel	23		
6	Inter - Processes Communication	28		
7	Race Condition and Zombie Processes	37		
8	Process Synchronization	42		
9	Thread Libraries	49		
10	Scheduling Schemes	57		
11	Deadlocks	67		
12	Memory Management	71		
13	Virtual Memory	76		
14	File System Access and Control Systems	86		
15	Disk Management Algorithms	99		

Student name: _____

CMS – ID: _____

Roll Number: _____

Lab 01
Introduction to Operating System

Objective

To understand Operating System's Resource Management.

Theory

Operating System

An operating system is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Some operating systems are designed to be convenient, others to be efficient, and others to be some combination of the two. A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and the users. The hardware —the central processing unit (CPU), the memory, and the input/output (I/O) devices —provides the basic computing resources for the system. The application programs —such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

In the end, we can say Operating System is a system software that manages resources like CPU, Memory and Disks and make it available upon requirement / request of any application running.

Resource Monitor

It is a utility in Windows Operating Systems, displays information about the use of hardware like CPU, memory, disk, and network as well as software like file handles and modules consuming resources in real time. You can access resource Monitor by typing ***resmon*** in RUN or Command Prompt (CMD). In figure 1-1 is a running Resource Monitor in Windows 10.

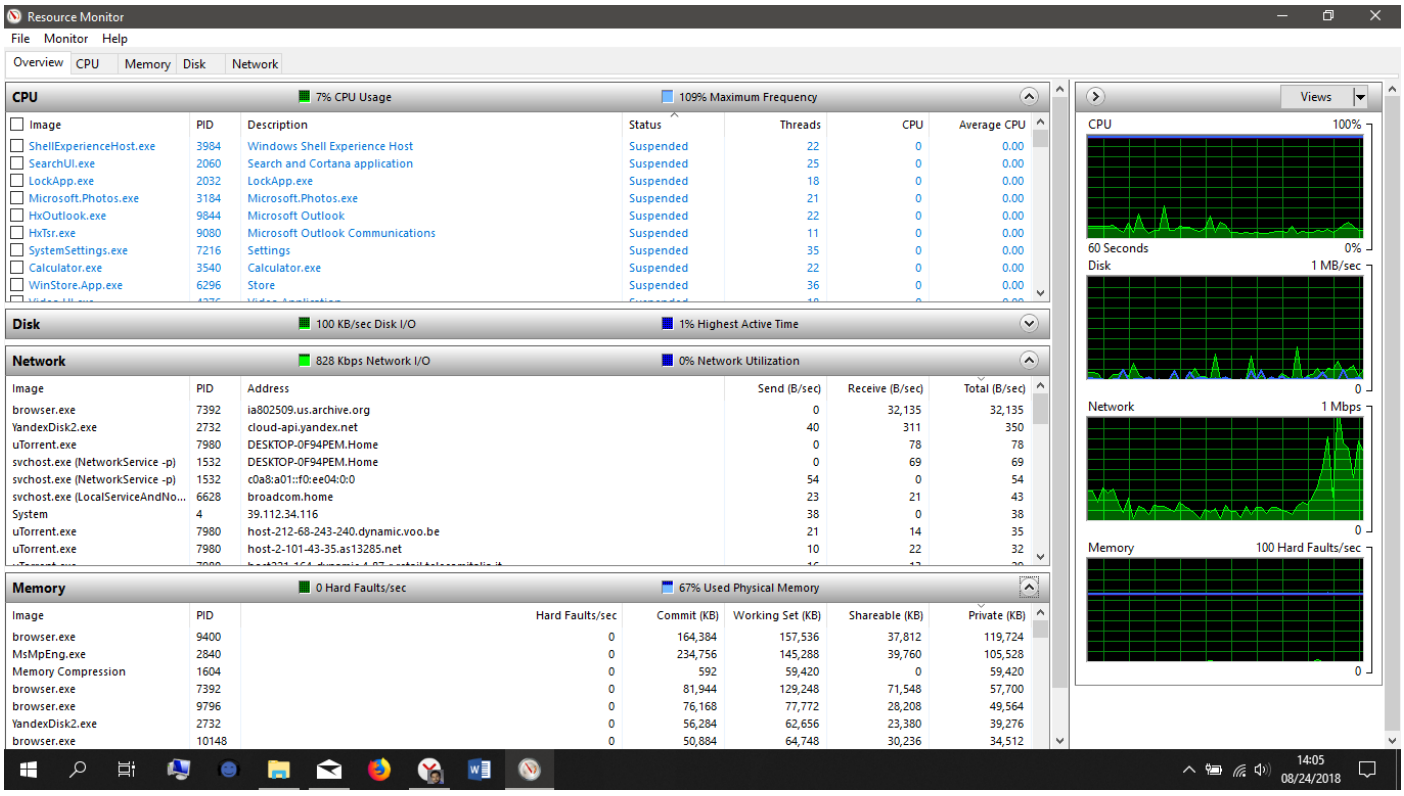


Figure 1-1

Lab Task – 1a

In Resource Monitor Click CPU tab, find mentioned images and fill the following Table – 1:

Table – 1:

S. No	Image	PID	Status	Threads	CPU	Average CPU
1	System					
2	Explorer.exe					
3	Svchost.exe					

Lab Task – 1b

In Resource Monitor Click Memory tab, find mentioned images and fill the following Table – 2:

Table – 2:

S. No	Image	PID	Commit	Working Set	Shareable	Private
1	System					
2	Explorer.exe					
3	Svchost.exe					
4	Physical Memory	Available				
5		Cached				
6		Total				
7		Installed				
8		Hardware Reserved				
9		In Use				
10		Modified				
11		Standby				
12		Free				

Note: As there will be multiple svchost.exe, find with same PID as in Table – 1.

Lab Task – 1c

After filling above tables, give details of **two** images / Applications having most of mentioned attributes:

Table – 3:

S. No	Image	PID	Status	Threads	CPU	CPU %	Read	Write	Working Set
1									
2									

Lab Task – 2a

Compile it by your name and run the following C++ program in CMD window, after running fill the table – 4:

```
#include<iostream>
using namespace std;

main ()
{
WRAP:
     cout << "Operating Systems -- DoC"<< endl;
goto WRAP;
}
```

Table – 4:

S. No	Image	PID	Status	Threads	CPU	CPU %	Read	Write	Working Set
1									

Lab Task – 2b

Compile by some different name and run the following C++ program in CMD window, after running fill the table – 5:

```
#include<iostream>
#include<windows.h>
using namespace std;

main ()
{
WRAP:
     cout << " Operating Systems -- DoC "<< endl;
     Sleep(3000);      // Time in Milli Second
goto WRAP;
}
```

Table – 5:

S. No	Image	PID	Status	Threads	CPU	CPU %	Read	Write	Working Set
1									

Lab Task – 3

Check List:

1. VirtualBox 6.0.4 r128413 or Higher
2. Oracle VM VirtualBox Extension Pack 6.0.4 r128413
    - To check Extension pack, File -> Preferences -> Extensions
3. LUBUNTU1810.ova

Step – 1

Import LUBUNTU VM by double clicking or import options in VirtualBox, and start VM

Step – 2

Login to VM, User: *iqbal*, Password: *123456789* (If required)

Step – 3

- Open **qps** (A Visual Process Manager)
- Process Count is _____
- Sort Processes by CPU% usage
- Click on Linear radio button
- Fill Table – 6 with the date of 10 processes from top
- For ease take Screen Capture / Screen shot before filling Table- 6.

Table – 6:

S. No	PID	TTY	STAT	USER	MEM	%CPU	START	TIME	COMMAND_LINE
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									

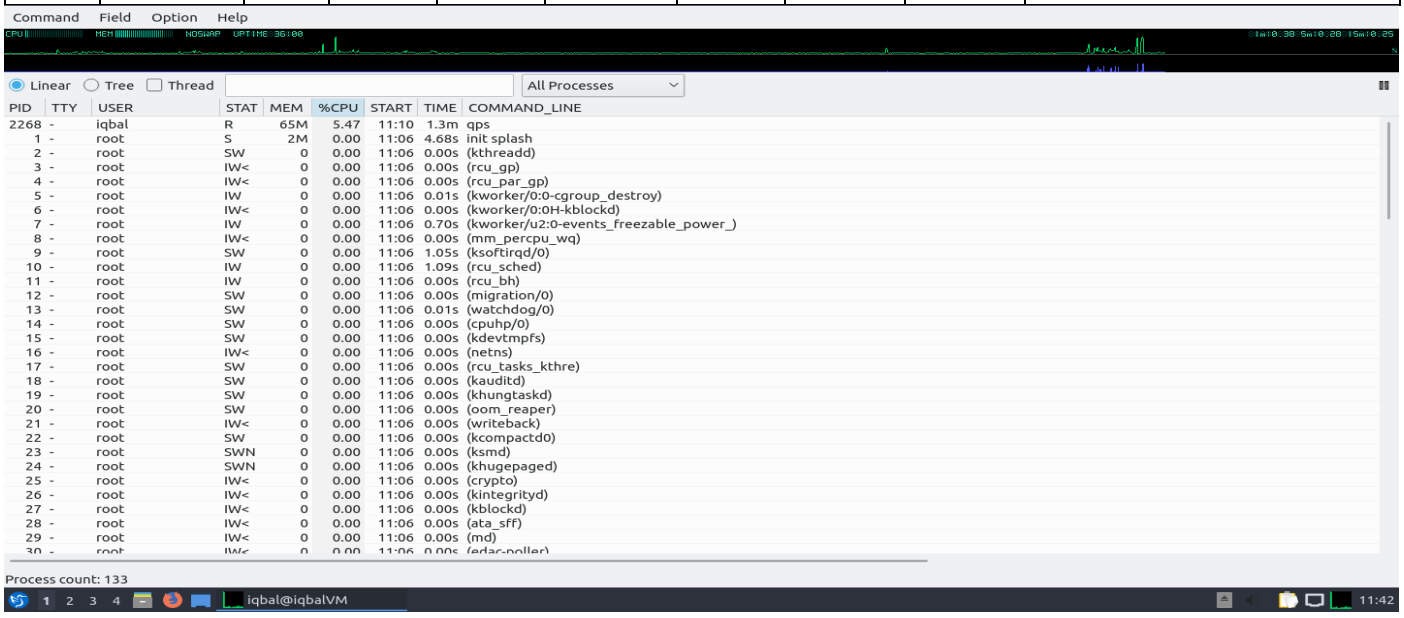


Figure 1-2

Lab 02
Bootloader

Objective

To write a Boot loader using Assembly Language

Theory

What is Boot Loader

Boot-loader is a piece of code that runs before any operating system is running. Boot-loader are used to boot other operating systems, usually each operating system has a set of boot-loaders specific for it. Boot-loaders usually contain several ways to boot the OS kernel and also contain commands for debugging and/or modifying the kernel environment. In other words:

- Stored in the Master Boot Record (MBR), on the first sector of the disk
- Size is exactly 1 sector, or 512 bytes
- Ends with a signature of **0x55** and **0xaa** at bytes **511** and **512**
- Loaded by the BIOS POST interrupt **0x19** at address **0x7c00**
- Runs in 16 – bits mode

Note: Keep in mind that we have no filesystem, no operating system and none of the associated interrupts, libraries and routines.

What is 16 Bits Real Mode

Real mode is an operating mode of all x86-compatible CPUs. The x86 family is backward compatible with the old DOS systems. Because they were 16 bits, all x86 compatible computers boot into 16 bits mode.

- Uses the native segment: offset memory model
- Limited to 1 MB of memory
- Limited to the 16 bits registers
- No virtual memory
- No memory protection
- No multitasking
- No code privileges

Notice that because of the segment: offset model, a given physical address can be represented by many **segment:** offset combinations. They overlap.

The Code

It is pretty simple. Things get difficult when you start to handle a filesystem and want to load a program in memory, like in a dual-stage bootloader, but there's nothing that complex here, you will experience in upcoming labs.

```
org           0x7c00
```

The **org** directive defines where the program expects to be loaded into memory. Not where it is actually loaded - that is controlled by whoever does the loading, here the BIOS POST. Its sole function is to specify one offset which is added to all internal address references within the section.

You can try to modify the address, for instance to **0x7c01**. The program will/may still work but the message will then be truncated.

There is no print function or anything like that. You practically have to point to the first character in the message buffer, load it in a register, print it, then move the pointer to the next character until you reach end

of buffer (represented by 0). So, we can consider ourselves lucky to have the **lodsb** (Load String Byte) instruction:

"**LODSB** transfers the byte pointed to by **DS:SI** into **AL** and increments or decrements **SI** (depending on the state of the Direction Flag) to point to the next byte of the string."

So, it does this:

```
AL <- DS:SI
SI++
```

Given our message:

```
msg:            db      "Hello, World!", 0
```

We can set **SI** to point to the first character:

```
mov     si, msg                 ; SI points to message
```

To print a character at the cursor position, we use service **0x0e** (Write Character in Teletype (TTY) Mode) of BIOS interrupt **0x10** (Video and Screen Services).

```
mov     ah, 0x0e                ; print char service
int     0x10                    ; print char
```

The character is loaded in AL by **lodsb**, while **SI** is incremented for us.

To detect the end of the string, we look for a 0. The obvious way is to do **cmp al, 0** but a common idiom is to use or **al, al**. It's faster or 1 bit shorter, I'm not sure. In a similar fashion, **mov ax, 0** is often replaced by **xor ax, ax**. In any case, after comparing **AL** to **0**, whether with **cmp** or **or**, we test the zero-flag to check if we've reached our **0**. If it's the case, we jump to the halt label. Otherwise, the character is printed and we jump back to the loop.

```
.loop   lodsb                   ; AL <- [DS:SI] && SI++
        or      al, al          ; end of string?
        jz      halt
        int     0x10            ; print char
        jmp     .loop           ; next char
```

No, there is no if-then-else construct. Conditional jumps (like **jz**) and unconditional jumps (**jmp**) are part the trick in code.

When the code is done, we simply halt:

```
halt:           hlt
```

Remember that our bootloader must be exactly 512 bytes, and that the two last bytes must be **0x55** and **0xaa**. We need to fill the space in between, using some handy directives:

- **$** represents the address of the current line
- **$$** is the address of the first instruction
- So, **510 - ($ - $$)** gives us the number of bytes we need for padding
- **times** repeats the given line or instruction

Combining all of above together, we have our padding:

```
times   510 - ($-$$) db 0
```

We still need to add our 2 bytes signature:

```
dw      0xaa55
```

Because the x86 processor family is little endian, the least significant byte (**0x55**) is stored in the smallest address. You might argue that two consecutive **db** would be more explicit and I'd agree.

Compile and check

Linux based Operating Systems usually ships with `as` but its syntax is both backward and cluttered. I opted for **`nasm`**, which is also available on Windows 10 (available on Google Drive), see figure 2-1.

```
C:\Users\Iqbal Uddin Khan> nasm -f bin -o lab02.flp lab02.asm
```

As it is a "pure" binary file due to the **`-f bin`** flag, output can be obtained as floppy image (**`.flp`**).

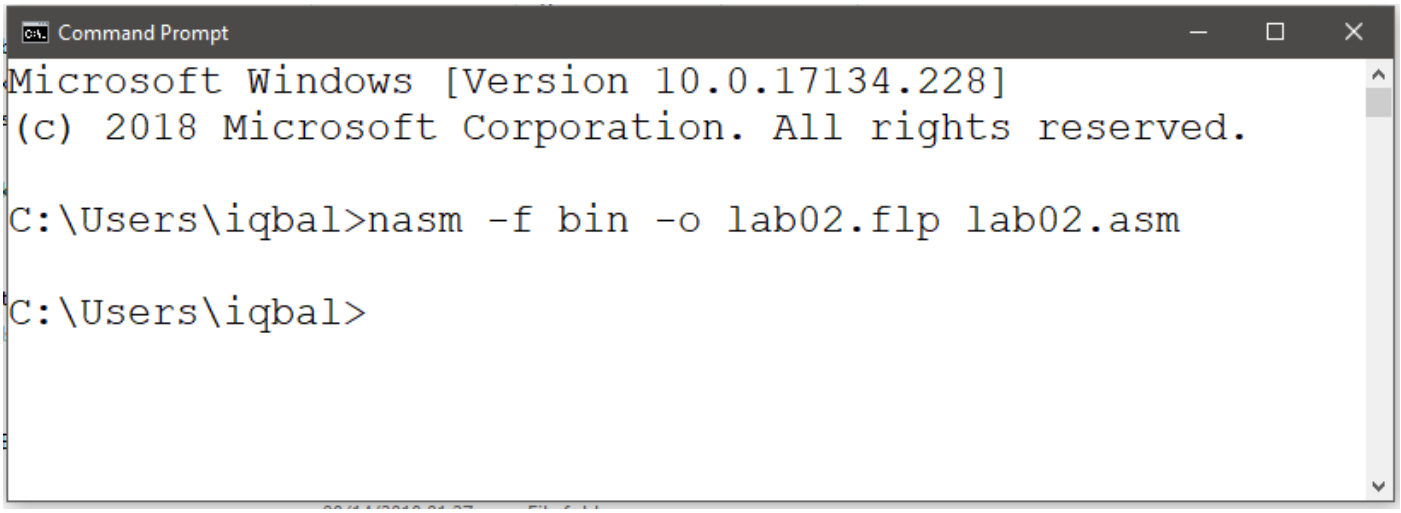


Figure 2 – 1

Make sure the generated file is exactly 512 bytes (**`ls -l`** will do). If you open it with a HEX editor, you will be able to see the hello message, the padding and the signature. Notice **`0x55`** and **`0xAA`** are is the expected order. See figure 2-2, the last bytes are as per requirement.

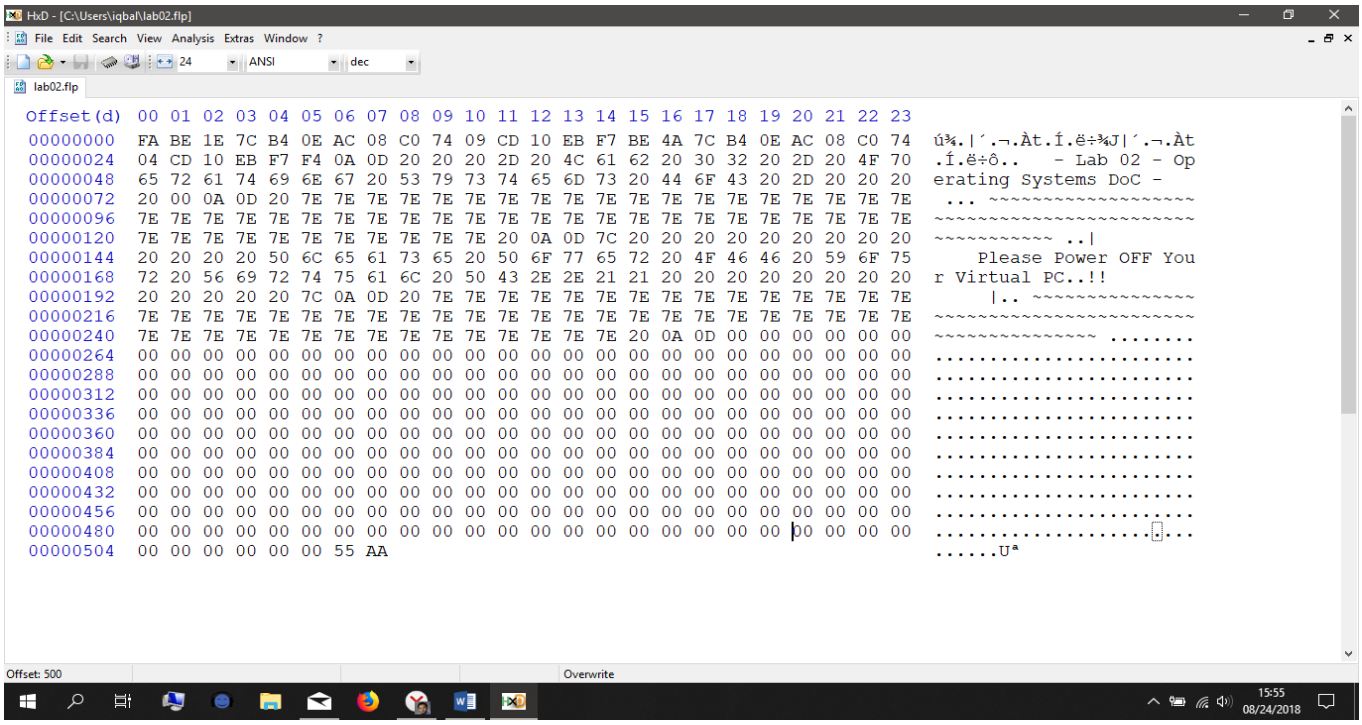


Figure 2 – 2

VirtualBox

To run our boot-loader, we require a PC emulator, for our labs we will use VirtualBox, it is free ware and highly compatible.

We required a Floppy Controller in our Virtual Machine, to add path of virtual floppy or **.flp** image. In figure 2-3, is mounted **.flp** image and in figure 2-4 is complete over view of Virtual Machine.

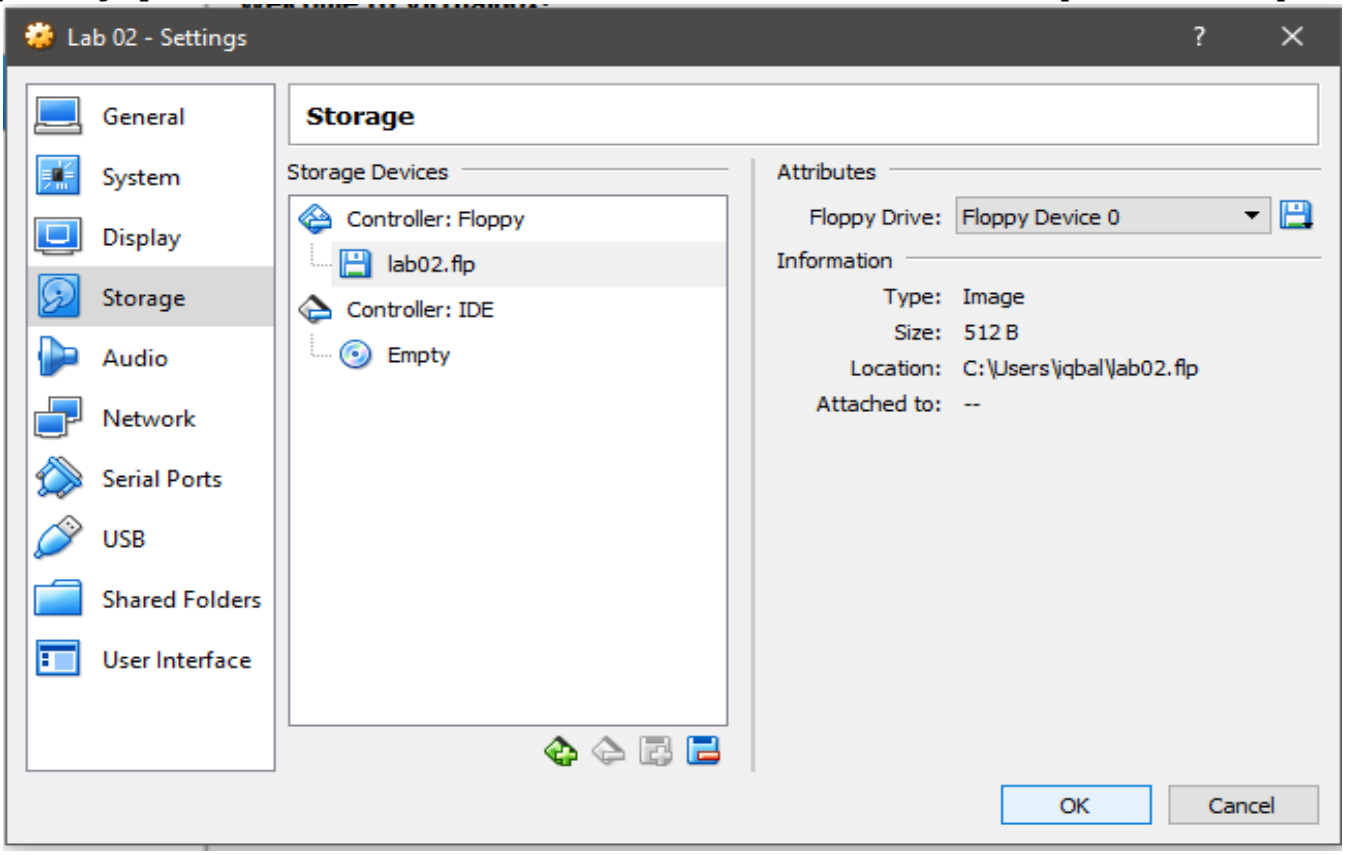


Figure 2 – 3

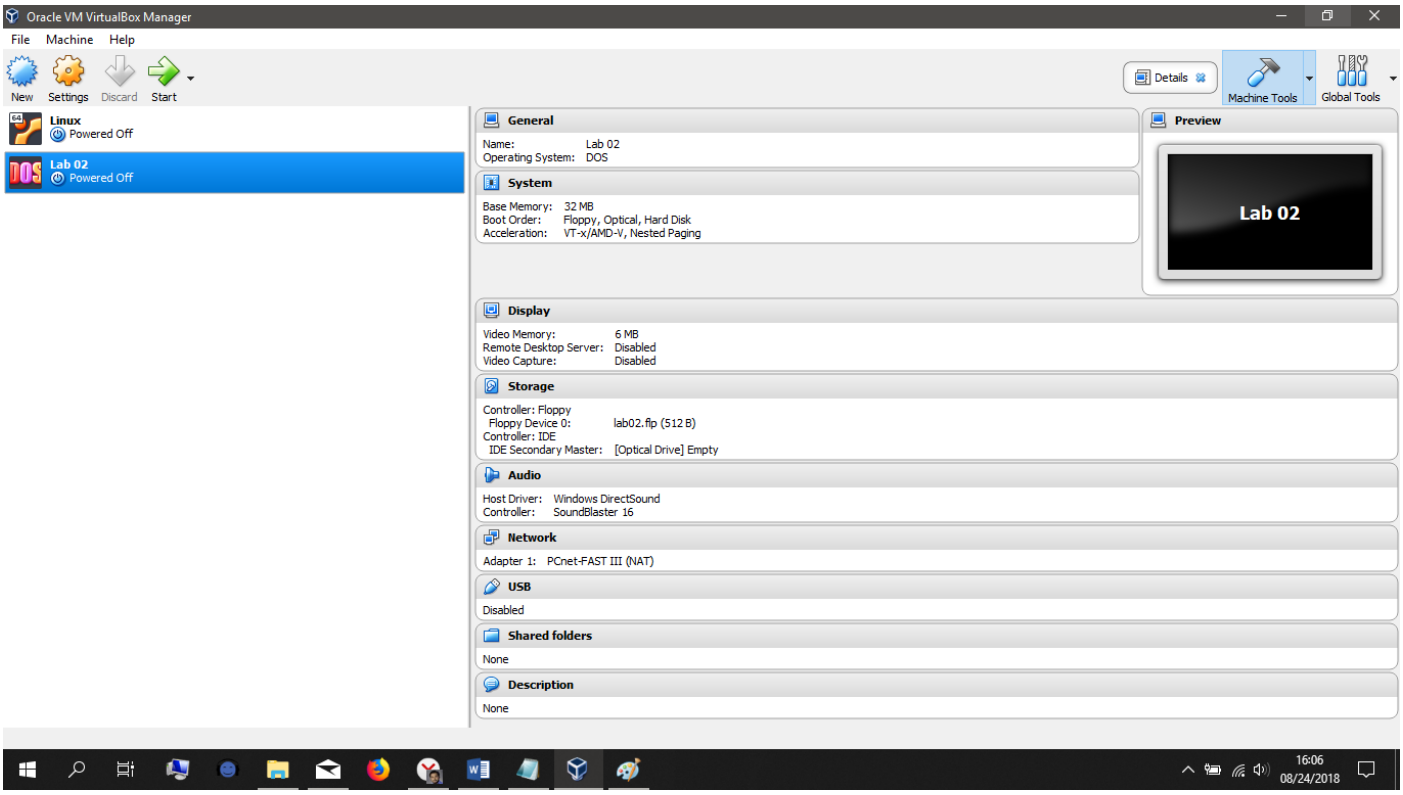


Figure 2 – 4

Lab Task

Write and Assemble following code (Page 8) using NASM and run it on VM on VirtualBox and place screen shot on page 9, use Notepad to write the Assembly code and save at NASM's location.

```
org         0x7c00                  ; BIOS loads at this address

 bits       16                      ; 16 bits real mode
start:
                cli                                ; disable interrupts

                mov     si, msg0                   ; SI points to message
                mov     ah, 0x0e                   ; print char service
.loop0          lodsb                              ; AL <- [DS:SI] && SI++
                or      al, al                     ; end of string?
                jz      .loop1
                int     0x10                       ; print char
                jmp     .loop0                     ; next char


                mov     si, msg1                    ; SI points to message
                mov     ah, 0x0e                   ; print char service
.loop1          lodsb                              ; AL <- [DS:SI] && SI++
                or      al, al                     ; end of string?
                jz      halt
                int     0x10                       ; print char
                jmp     .loop1                     ; next char

halt:           hlt                                ; halt

msg0:  db    0ah, 0dh, "          - Lab 02 - Operating Systems --- DoC -    ",
0
msg1:  db    0ah,                            0dh,                          "
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    ",0ah,   0dh,   "|
Please  Power  OFF  Your  Virtual  PC..!!                    |",0ah,  0dh,   "
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ",0ah, 0dh, 0

 times 510 - ($-$$) db 0
 dw         0xaa55
```

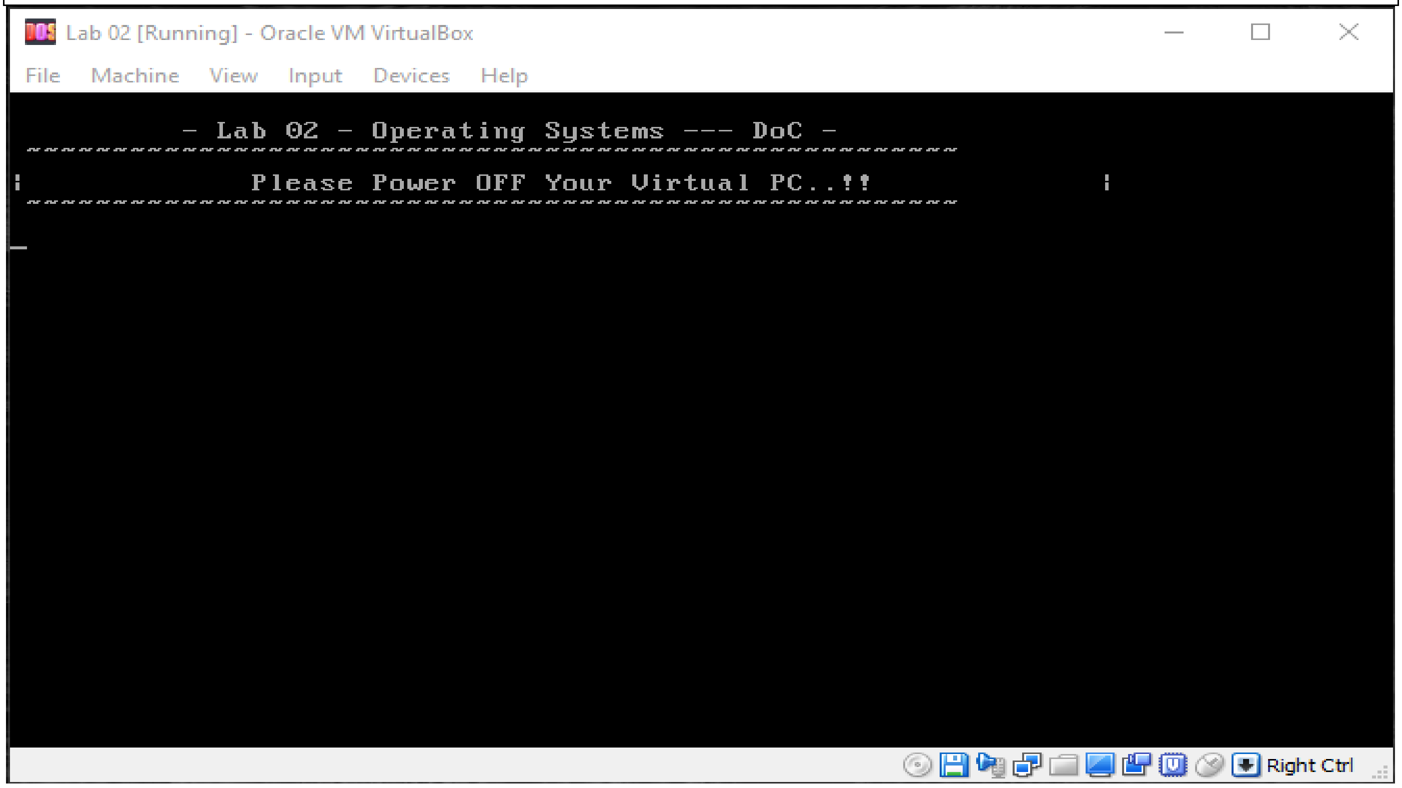


Figure 2 – 5

Paste Screen Shot of running VM

Lab 03
Introduction to Command Line Interface

Objective

To understand Command Prompt (Windows) and Shell (Linux) Interface.

Theory

What is Command Line Interface?

A Command Line Interface (CLI) is a user interface to a computer's operating system or an application in which the user responds to a visual prompt by typing in a command on a specified line, receives a response back from the system, and then enters another command, and so forth.

Command Prompt (CMD)

Command Prompt is a command line interpreter application available in most Windows operating systems. Command Prompt is used to execute entered commands. Most of those commands are used to automate tasks via scripts and batch files, perform advanced administrative functions, and troubleshoot and solve certain kinds of Windows issues. Command Prompt is officially called Windows Command Processor but is also sometimes called the ***command shell*** or ***cmd*** prompt, or even referred to by its filename ***cmd.exe***.

Note: Command Prompt is sometimes incorrectly referred to as "the DOS prompt" or as MS-DOS itself. Command Prompt is a Windows program that emulates many of the command line abilities available in MS-DOS but it is not actually MS-DOS.

Shell

The shell prompt (or command line) is where one types commands. When accessing the system through a text-based terminal, the shell is the main way of accessing programs and doing work on the system. In effect, it is a shell surrounding all other programs being run. When accessing the system through a graphical environment such as X11, it remains possible to open a terminal emulator and do useful work with the shell.

In other words, the **shell** is the program that waits for you to type in a Unix or Unix Type command and then press the return key. Then the shell handles the execution of your command.

Six basic commands

Table – 1:

S. No	Windows (CMD)	Linux based (Shell)	Usage	Example
1	`attrib`	`chmod`	Change file attributes / adds an attribute / removes it	• `ATTRIB -R -A -S -H <VIRUS.EXE>` • `chmod 755 filename`
2	`cls`	`clear`	Clears the screen	• `cls` • `clear`
3	`del`	`rm`	Removes the file	• `DEL <VIRUS.EXE>` • `rm <file.ext>`
4	`dir`	`ls`	List down current directory items	• `dir` • `ls`
5	`md`	`mkdir`	Creates a new folder / Directory	• `md <Folder Name>` • `mkdir <Directory Name>`
6	`cd`	`cd`	Change working folders / Directory	• `cd <Folder Name>` • `cd <Directory Name>`

Command Line Scripting

Scripting allow users to write special functions in a plain text file and pass the name of this file as a command line argument to Windows Command Prompt (CMD) or Shell (Linux). Their kernel runs these functions in a sequential order.

BATCH Scripts (Windows)

Batch Script is incorporated to automate command sequences which are repetitive in nature. Scripting is a way by which one can alleviate this necessity by automating these command sequences in order to make one's life at the shell easier and more productive. In most organizations, Batch Script is incorporated in some way or the other to automate stuff.

Linux Shell

Its environment provided for user interaction as shown in figure 3-1. Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Linux may use one of the following most popular shells:

Table – 2:

Shell Name	Developed By	Location	Comments
BASH (Bourne Again Shell)	Brian Fox and Chet Ramey	FSF Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C Shell)	Billy Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn Shell)	David Korn	AT&T Bell Labs	Many features of the C shell, inspired by the requests of Bell Labs users

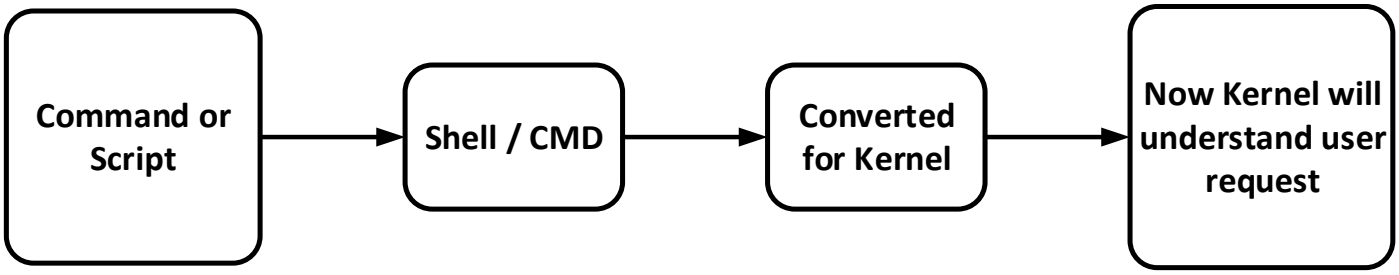


Figure 3 - 1

Lab Task – 1a

Writing a simple BATCH File

1. Open Notepad
2. Type the below given commands
3. Save it as **`Bat0.bat`**
4. Open CMD
5. Change your folder where **`.bat`** file is saved
6. Execute BATCH file by typing **`bat0`**
7. Write the output in Block – 1

Commands for bat0.bat

```
@echo off
cls
md Batch
cd Batch
cls
echo "Batch Folder Created ...!!"
echo "It is now your, working Directory...!!"
```

Block – 1

Write Complete output after execution of bat0.bat

Lab Task – 1b

Writing a simple SHELL script

1. Open Terminal
2. Type **`nano shell0.sh`** (Nano is a command line test editor)
3. Type the command list in following box
4. Execute file by typing **`bash shell0.sh`**
5. Write the output in Block – 2

Commands for shell0.sh

```
clear
mkdir shell-Dic
cd shell-Dic
clear
echo "Directory Created …...!!!"
echo $PWD is complete path of shell-Dic directory!
```

Block – 2

Write Complete output after execution of *bash shell0*

Command Line Programming

Batch File Programming

The conditional statement enriches the features of the batch file programming. The conditional statements are widely used for deciding, and in accordance to the decision taken, the result or the output is produced. The primary decision-making statements used in batch programs are, IF and IF NOT. The decision-making statements deals with error levels, string matching and files. The following piece of code will help you in a better understanding of how it deals with files and folders:

Script named bat0.bat

```
@echo off
if exist C:\users\iqbal Uddin khan\desktop\test. (
echo Found
) else (
echo Not found
)
Pause
```

Output of bat0.bat

```
C:\Users\Iqbal Uddin Khan>bat0
Not found
Press any key to continue . . .
```

Shell Programming

Programming with the Bourne shell is similar to programming in a conventional language. If you've ever written code in C or C++, or even BASIC or JAVA, you'll recognize many common features. For instance, the shell has variables, conditional and looping constructs, functions, and more. Shell programming is also different from conventional programming languages. For example, the shell itself doesn't provide much useful functionality; instead, most work must be done by invoking external programs. As a result, the shell has powerful features for using programs together in sequence to get work done.

Scripted named shell1.sh

```
#------ Simple Arithmetic Calculator --------
echo 'enter the value for a'
read a
echo 'enter the value for b'
read b
echo 'enter operator'
read op
echo "-----"
case $op in
     +) expr  $a + $b;;
     -) expr $a - $b;;
     \*) expr $a \* $b;;
     /) expr $a / $b ;;
esac
echo "-----"
```

Output of shell1.sh

```
iqbal@Iqbal-PC:~$ bash shell1
bash: shell1: No such file or directory
iqbal@Iqbal-PC:~$ bash shell1.sh
enter the value for a
3
enter the value for b
4
enter operator
+
-----
7
-----
iqbal@Iqbal-PC:~$
```

Lab Task – 2

Write a script to generated table of your choice up to **12** counts and save it in a file.

Write the shell script here

Lab 04
System Calls

Objective

To understand Windows and Linux based System Calls.

Theory

What is a System Call?

A system call is a request for the operating system to do something on behalf of the user's program. The system calls are functions used in the kernel itself. To the programmer, the system call appears as a normal C function call. However, since a system call executes code in the kernel, there must be a mechanism to change the mode of a process from user mode to kernel mode.

Use of System Calls

UNIX type system calls are used to:

1. Manage the file system
2. Control processes
3. Provide inter-process communication (Will be discussed in Lab 06)

Major System Calls

The UNIX system interface consists of about 80 system calls (as UNIX evolves this number will increase). The table below lists about 40 of the most important system calls.

The file structure related system calls in UNIX let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices.

Table – 1

File Structure Related Calls			
Creating Channel	**creat()**	Aliasing and Removing	**link()**
	open()		**unlink()**
	close()	File Status	**stat()**
Input / Output	**read()**		**fstat()**
	write()	Access Control	**access()**
Random Access	**lseek()**		**chmod()**
Channel Duplication	**dup()**		**chown()**
Device Control	**Ioctl()**		**unmask()**

Table – 2

Processes Related Calls			
Creating and Termination	**exec()**	Process ID	**getpid()**
	fork()		**getppid()**
	wait()	Process Control	**signal()**
	exit()		**kill()**
Ownership and Group	**getuid()**		**alarm()**
	getgid()	Change Working Directory	**chdir()**
	getegid()		

Lab Task – 1a

Use **creat()** system call to create a file.

creat( ), creates an empty file with the specified mode permissions, if the file named by file_name does not exist. However, if the file does exist, its contents are discarded and the mode value is ignored. The permissions of the existing file are retained.

The mode is usually specified as an octal number such as 0666 that would mean read/write permission for owner, group, and others or the mode may also be entered using manifest constants defined in the "/usr/include/sys/stat.h" file. e.g. S_IREAD,S_IWRITE Header files needed for this system call in which the actual prototype appears, & in which useful constants are defined are:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
```

All input and output operations start by opening a file using either the "**creat()**" or "**open()**" system calls. These calls return a file descriptor that identifies the I/O channel. Recall that file descriptors 0, 1, and 2 refer to standard input, standard output, and standard error files respectively, and that file descriptor 0 is a channel to your terminal's keyboard and file descriptors 1 and 2 are channels to your terminal's display screen.

How to use **creat()**

Create a file newfile with read & write permission to owner and read only for the group and others.

Hint 4 – 1:

```
fd = creat("/tmp/newfile", 0644);
```

Write Your code here

Lab Task – 1b

Use **open()** system call to set permissions on a file.

open( ), It opens a file for reading or writing, or creates an empty file:

- file_name is a pointer to the character string that names the file,
- option_flags represent the type of channel, and
- mode defines the file's access permissions if the file is being created. It is only used with the O_CREAT option_flag and it is concerned with the security permissions.

Header files needed for this system call in which the actual prototype appears, & in which useful constants are defined are:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
```

Multiple values are combined using the | operator (i.e. bitwise OR). Note: some combinations are mutually exclusive such as: **O_RDONLY | O_WRONLY** and will cause **open()** to fail. If the **O_CREAT** flag is used, then a mode argument is required. The mode argument may be specified in the same manner as in the **creat( )** system call.

Example#3:

This causes the file data, in the current working directory, to be opened as read only for the use by the program.

```
fd = open("data", O_RDONLY);
```

Hint 4 – 2:
The open call can also be used to create a file from scratch, as follows:

```
fd = open ("/tmp/newfile", O_WRONLY | O_CREAT, 0644);
```

Hint 4 – 3:
means, if file lock does not exist, then create it with permission **0644**. If it does exist, then fail open call, returning –1 in **fd**.

```
fd = open("lock", O_WRONLY | O_CREAT | O_EXCL, 0644);
```

Hint 4 – 4:
O_TRUNC when used with **O_CREAT** it will force a file to be truncated to zero bytes if it exists and its access permissions allow.

```
fd = open("file", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

Hint 4 – 5:

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

Write your code in the below given:

Write Your code here

Lab Task – 1c

Use **lseek()** system call to access file.

The UNIX system file system treats an ordinary file as a sequence of bytes. No internal structure is imposed on a file by the operating system. Generally, a file is read or written sequentially -- that is, from beginning to the end of the file. Sometimes sequential reading and writing is not appropriate. It may be inefficient, for instance, to read an entire file just to move to the end of the file to add characters. Fortunately, the UNIX system lets you read and write anywhere in the file. Known as "random access", this capability is made possible with the **lseek()** system call. During file I/O, the UNIX system uses a long integer, also called a File Pointer, to keep track of the next byte to read or write. This long integer represents the number of bytes from the beginning of the file to that next character. Random access I/O is achieved by changing the value of this file pointer using the **lseek()** system call. **lseek()** enables random access into a file. To use it we need header files:

```
#include<sys/types.h>
#include<unistd.h>
```

Hint 4 – 6:
A program fragment gives a position of 16 bytes before the end of the file. From this example it is clear that offset can be negative i.e. it is possible to move backwards from the starting point indicated by whence.

```
newpos = lseek(fd, -16, SEEK_END)
```

Hint 4 – 7:
A program fragment that will append to the end of an existing file by opening the file, moving to the end with **lseek()**, and starting to write.

```
fd = open(filename, O_RDWR);
lseek(fd, 0, SEEK_END);
write(fd, outbuf, OBSIZE);
```

Now write a code to create, access and write into a file.

Write Your code here

Write Your code here

Lab Task – 2

Write a program creates a separate process using the **`CreateProcess()`** system call in **Win32**.

Hint 4 – 8:

Use following Header files:

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

Hint 4 – 9:

```
    if( !CreateProcess( NULL,   // No module name (use command line).
        "<path to the executable file>", // Command line.
        NULL,             // Process handle not inheritable.
        NULL,             // Thread handle not inheritable.
        FALSE,            // Set handle inheritance to FALSE.
        0,                // No creation flags.
        NULL,             // Use parent's environment block.
        NULL,             // Use parent's starting directory.
        &si,              // Pointer to STARTUPINFO structure.
        &pi )            // Pointer to PROCESS_INFORMATION structure.
    )
```

Write the code in the box on next page:

Write Your code here

Lab 05
Adding Modules to Kernel

Objective

To add new modules in Kernel (Linux).

Theory

What Is a Kernel Module?

So, you want to write a kernel module. You know C, you've written a few normal programs (in theory assignment) to run as processes, and now you want to get to where the real action is, to where a single wild pointer can wipe out your file system and a core dump means a reboot.

What exactly is a kernel module? Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

Modules vs Programs

A program usually begins with a **`main()`** function, executes a bunch of instructions and terminates upon completion of those instructions. Kernel modules work a bit differently. A module always begins with either the **`init_module`** or the function you specify with **`module_init`** call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they're needed. Once it does this, entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides. All modules end by calling either **`cleanup_module`** or the function you specify with the **`module_exit`** call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered.

Every module must have an entry function and an exit function. Since there's more than one way to specify entry and exit functions, I'll try my best to use the terms `**entry function**' and `**exit function'**, but if I slip and simply refer to them as **`init_module`** and **`cleanup_module`**, you will know the meaning.

Makefile

A **makefile** used to tell make what to do. Most often, the **makefile** tells make how to compile and link a program. A simple **makefile** that describes how to compile and link a text editor which consists of eight C source files and three header files. The **makefile** can also tell make how to run miscellaneous commands when explicitly asked (for example, to remove certain files as a clean-up operation).

When make recompiles the editor, each changed C source fie must be recompiled. If a header fie has changed, each C source fie that includes the header fie must be recompiled to be safe. Each compilation produces an object fie corresponding to the source fie. Finally, if any source fie has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable editor.

Lab Task – 1

Compile the following codes as directed and write output in each box:

1. Open a text editor, write following code and save as **mod-1.c**

```
#include <linux/module.h>
#include <linux/kernel.h
int init_module(void)
{
        printk(KERN_INFO "Module Mod-1 added...\n");
        return 0;
}
void cleanup_module(void)
{
        printk(KERN_INFO "Module mod-1 is exiting...\n");
}
```

2. To write a make file, open text editor, write following lines and save it as **Makefile**.

```
obj-m += mod-1.o
all:
     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

3. Now type **make**, write its output in below given box

Write output after typing 'make'

4. To insert module type **`sudo insmod mod-1.c`**
5. Was there any warning, if **Yes**, mention below?

_____

_____

_____

6. Now type **lsmod**, what is your module size _____ and used by _____
7. Now type **dmesg**, write its output related to your module, in below given box:

Write output after typing 'dmesg'

8. Was there any **License Warring**, if **Yes** type specify line below?

9. Now type **sudo rmmod mod-1**, to remove module
10. Type **dmesg -c**, write its output in below given box

Write output after typing 'dmesg -c'

Lab Task – 2

Write two kernel modules and a single **Makefile** to compile them. Insert them in kernel and check messages and warnings.

1. Contents of **Makefile**

Write contents of make file

2. Contents of **`mod-2.c`** file (Non-Licensed)

3. Contents of **`mod-4.c`** file (License Defined)

Hint 5 – 1:

As, **MODULE_DESCRIPTION()** is used to describe what the module does, **MODULE_AUTHOR()** declares the module's author, and **MODULE_SUPPORTED_DEVICE()** declares what types of devices the module supports.

Observation and Conclusion

Write your observations and **dmesg** output after adding modules

Observations and Conclusion

Lab 06
Inter – Process Communication

Objective

To understand inter-process communication via Pipes, Signals and Shared Memory.

Theory

There are many mechanisms through which the processes communicate, in this lab, we will discuss two such mechanisms: Pipes, Signals and Shared Memory. A pipe is used for one-way communication of a stream of bytes. Signals inform processes of the occurrence of asynchronous events. In this lab we will learn how to create pipes and how processes communicate by reading or writing to the pipe and also how to have a two-way communication between processes. This lab also discusses how user-defined handlers for particular signals can replace the default signals handlers and also how the processes can ignore the signals. Shared Memory is an efficient means of passing data between programs. One program will create a memory portion, which other processes (if permitted) can access. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.

Pipes

Pipes are familiar to most UNIX users as a shell facility. For instance, to print a sorted list of who is logged on, you can enter this command line:

```
who | sort | lpr
```

There are three processes here, connected with two pipes. Data flows in one direction only, from who to sort to **lpr**. It is also possible to set up bidirectional pipelines (from process A to B, and from B back to A) and pipelines in a ring (from A to B to C to A) using system calls. The shell, however, provides no notation for these more elaborate arrangements, so they are unknown to most Unix users. We'll begin by showing some simple examples of processes connected by a one-directional pipeline.

Pipe System Call

```
int pfd[2];
int pipe (pfd); /* Returns 0 on success or -1 on error */
```

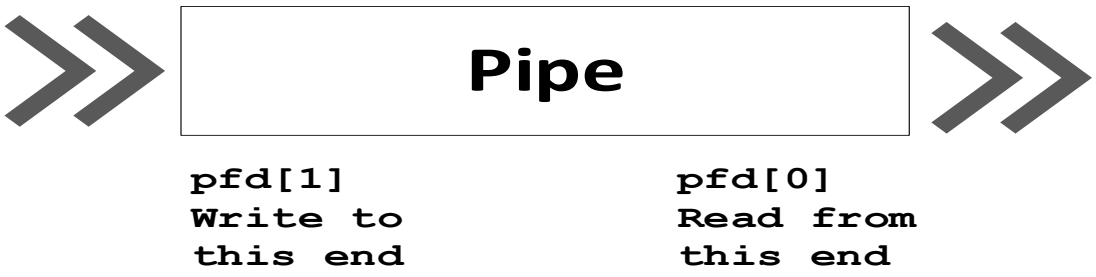


Figure 6 – 1

I/O with a Pipe

These two file descriptors can be used for block I/O

```
write(pfd[1], buf, SIZE);
read(pfd[0], buf, SIZE);
```

Fork and a Pipe

A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using **`fork()`**. A pipe opened before the fork becomes shared between the two processes.

Figure 6 – 2: Before Fork ()

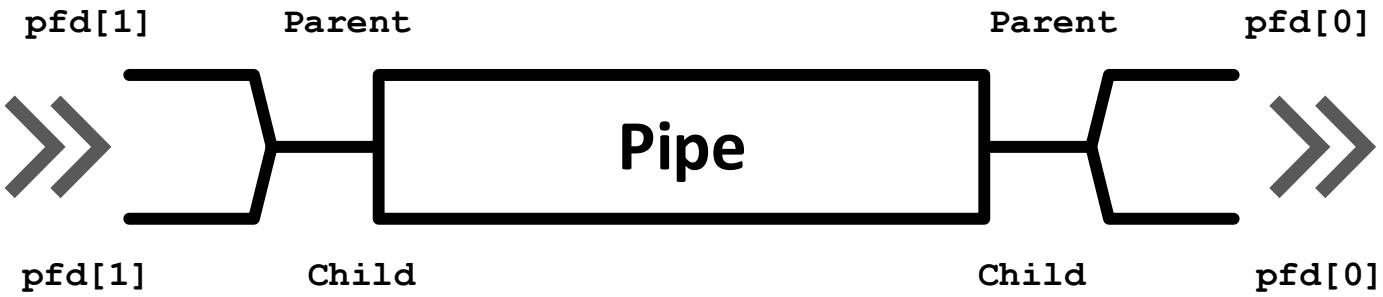


Figure 6 – 3: After Fork ()

This gives two read ends and two write ends. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed. Either process can write into the pipe, and either can read from it. Which process will get what is not known?

For predictable behavior, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again. Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end.

The child closes its write end and reads from the other end. When the processes have ceased communication, the parent closes its write end. This means that the child gets **`eof`** on its next read, and it can close its read end. Pipes use the buffer cache just as ordinary files do. Therefore, the benefits of writing and reading pipes in units of a block (usually 512 bytes) are just as great. A single write execution is atomic, so if 512 bytes are written with a single system call, the corresponding read will return with 512 bytes (if it requests that many). It will not return with less than the full block.

However, if the writer is not writing complete blocks, but the reader is trying to read complete blocks, the reader may keep getting partial blocks anyhow. This won't happen if the writer is faster than the reader, since then the writer will be able to fill the pipe with a complete block before the reader gets around to reading anything. Still, it's best to buffer writes and reads on pipes, and this is what the Standard I/O Library does automatically.

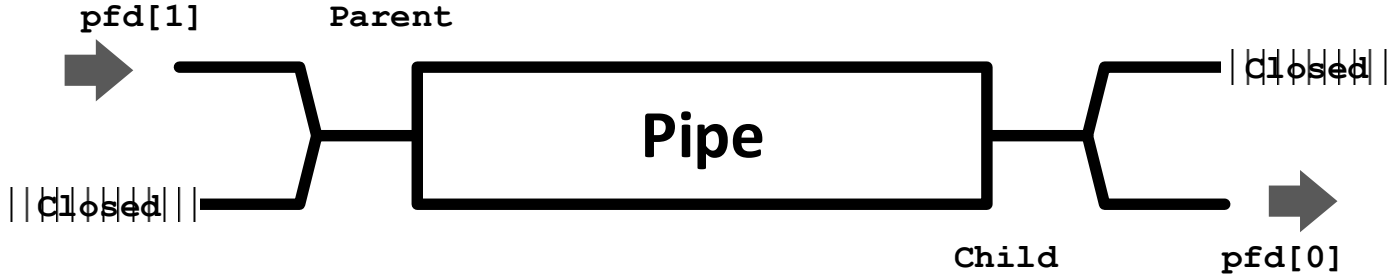


Figure 6 – 4: Parent to Child Write

Example 6 – 1:

```
#include <stdio.h>
#define SIZE 1024
int main( )
{int pfd[2];
 int nread;
 int pid;
 char buf[SIZE];
 if (pipe(pfd) == -1)
 {perror("pipe failed");
 exit(1);  }
 if ((pid = fork()) < 0)
 { perror("fork failed");
 exit(2); }
 if (pid == 0)
 { /* child */
 close(pfd[1]);
 while ((nread = read(pfd[0], buf, SIZE)) != 0)
 printf("child read %s\n", buf);
 close(pfd[0]);  }
 else
 { /* parent */
 close(pfd[0]);
 strcpy(buf, "hello...");
 /* include null terminator in write */
 write(pfd[1], buf, strlen(buf)+1);
 close(pfd[1]);
 } }
```

Write observation / output of the **exam1.c**

Observations and Conclusion

Given that we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't. Once the processes are created, they can't be connected, because there's no way for the process that makes the pipe to pass a file descriptor to the other process. It can pass the file descriptor number, of course, but that number won't be valid in the other process. But if we make a pipe in one process before creating the other process, it will inherit the pipe file descriptors, and they will be valid in both processes. Thus, two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on, but they must be related, and the pipe must be passed on at birth. In practice, this may be a severe limitation, because if a process dies there's no way to recreate it and reconnect it to its pipes -- the survivors must be killed too, and then the whole family has to be recreated.

In general, then, here is how to connect two processes with a pipe:
Make the pipe.

1. Fork to create the reading child.
2. In the child close the writing end of the pipe, and do any other preparations that are needed.
3. In the child execute the child program.
4. In the parent close the reading end of the pipe.
5. If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.

Here's a small program that uses a pipe to allow the parent to read a message from its child:

Example 6 – 2:
In general, then, here is how to connect two processes with a pipe:

1. Make the pipe.
2. Fork to create the reading child.
3. In the child close the writing end of the pipe, and do any other preparations that are needed.
4. In the child execute the child program.
5. In the parent close the reading end of the pipe.
6. If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.

Here's a program that uses a pipe to allow the parent to read a message from its child:

```
#include <stdio.h>
#include <string.h>
#define READ 0
#define WRITE 1
char* phrase = "OS - F17 Lab 06 Eamaple - 2" ;
main ( )
{  int fd [2], bytesread ;
 char message [100] ;
 pipe ( fd ) ;
 if ( fork ( ) == 0 ) /* child, writer */
 {  close ( fd [READ] ) ; /* close unused end */
 write ( fd [WRITE], phrase, strlen (phrase) + 1) ;
 close ( fd [WRITE] ) ; /* close used end */  }
 else /* parent, reader */
 {  close ( fd [WRITE] ) ; /* close unused end */
```

```
bytesread = read (fd [READ], message, 100) ;
printf ("Read %d bytes : %s\n", bytesread, message) ;
close ( fd [READ] ) ; /* close used end */
} }
```

Write observation / output of the **exam2.c**

Observations and Conclusion

Signals

Programs must sometimes deal with unexpected or unpredictable events, such as:

1. A floating point errors
2. A power failure
3. An alarm clock "ring"
4. The death of a child process
5. A termination request from a user (i.e., a Control-C)
6. A suspend request from a user (i.e., a Control-Z)

These kinds of events are sometimes called interrupts, as they must interrupt the regular flow of a program in order to be processed. When UNIX recognizes that such an event has occurred, it sends the corresponding process a signal. The kernel isn't the only one that can send a signal; any process can send any other process a signal, as long as it has permissions. A programmer may arrange for a particular signal to be ignored or to be processed by a special piece of code called a signal handler. In the latter case, the process that receives the signal suspends its current flow of control, executes the signal handler, and then resumes the original flow of control when the signal handler finishes.

Signals inform processes of the occurrence of asynchronous events. Every type of signal has a HANDLER which is a function. All signals have default handlers which may be replaced with user-defined handlers. The default signal handlers for each process usually terminate the process or ignore the signal, but this is not always the case. Signals may be sent to a process from another process, from the kernel, or from devices such as terminals. The ^C, ^Z, ^S and ^Q terminal commands all generate signals which are sent to the foreground process when pressed. The kernel handles the delivery of signals to a process. Signals are checked for whenever a process is being rescheduled, put to sleep, or re-executing in user mode after a system call.

Pause System Call

int pause (), **pause()** suspends the calling process and returns when the calling process receives a signal. It is most often used to wait efficiently for an alarm signal. **pause()** doesn't return anything useful.

The following example catches and processes the SIGALRM signal efficiently by having user written signal handler, **alarmHandler ()**, **by using signal ()**.

Example 6 – 3:

```
#include <stdio.h>
#include <signal.h>
int alarmFlag = 0 ;
void alarmHandler ( ) ;
main ( )
{  signal(SIGALRM, alarmHandler) ; /*Install signal Handler*/
 alarm (5) ;
 printf ("Looping ...\n") ;
 while (!alarmFlag)
 {  pause ( ) ; /* wait for a signal */  }
 printf ("Loop ends due to alarm signal\n") ; }
void alarmHandler ( )
{ printf ("An ALARM clock signal was received\n") ;
 alarmFlag = 1 ; }
```

Write observation / output of the **exam3.c**

Observations and Conclusion

Protecting Critical Code and Chaining Interrupt Handlers:

The same techniques described previously may be used to protect critical pieces of code against Control-C attacks and other signals. In these cases, it's common to save the previous value of the handler so that it can be restored after the critical code has executed. Here's the source code of the program that protects itself against SIGINT signals:

Now run the Example 6 – 4, by pressing Control-C twice while the program sleeps.

Example 6 – 4:

```
#include <stdio.h>
#include <signal.h>
main ( )
{  int (*oldHandler) ( ) ; /* holds old handler value */
 printf ("I can be Control-C'ed \n") ;
 sleep (5) ;
 oldHandler = signal(SIGINT, SIG_IGN) ; /* Ignore Ctrl-C */
```

```
printf ("I am protected from Control-C now \n") ;
sleep (5) ;
signal (SIGINT, oldHandler) ; /* Restore old handler */
printf ("I can be Control-C'ed again \n") ;
sleep (5) ;
printf ("Bye!!!!!!!\n") ; }
```

Write observation / output of the **exam4.c**

Observations and Conclusion

Shared Memory

System call **shmdt()** is used to detach a shared memory. After a shared memory is detached, it cannot be used. However, it is still there and can be re-attached back to a process's address space, perhaps at a different address. To remove a shared memory, use **shmctl()**.

The only argument to **shmdt()** is the shared memory address returned by **shmat()**. Thus, the following code detaches the shared memory from a program:

```
shmdt (shm_ptr);
```

where **shm_ptr** is the pointer to the shared memory. This pointer is returned by **shmat()** when the shared memory is attached. If the detach operation fails, the returned function value is non-zero. To remove a shared memory segment, use the following code:

```
shmctl (shm_id, IPC_RMID, NULL);
```

where shm_id is the shared memory ID. **IPC_RMID** indicates this is a remove operation. Note that after the removal of a shared memory segment, if you want to use it again, you should use **shmget()** followed by **shmat()**.

Example 6 – 5:

Two different processes communicating via shared memory we develop two programs here that illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously:

Shared Memory Server Code

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSIZE 27
```

```
main()
{  char c;
 int shmid;
 key_t key;
 char *shm, *s;
key = 5678;
if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0)
 {  perror("shmget");
 exit(1);  }
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
 {  perror("shmat");
 exit(1); }
s = shm;
 for (c = 'a'; c <= 'z'; c++)
 *s++ = c;
 *s = NULL;
while (*shm != '*')
 sleep(1);
 exit(0); }
```

Shared Memory Client Code

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSIZE 27
main()
{  int shmid;
 key_t key;
 char *shm, *s;
key = 5678;
if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
 perror("shmget");
 exit(1);
 }
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
 perror("shmat");
 exit(1);
 }
for (s = shm; *s != NULL; s++)
```

```
      putchar(*s);
     putchar('\n');
     *shm = '*';
      printf ("\nIts done from client.\n\n\n");
      exit(0);}
```

Write observation / output of the **Shared Memory Codes**

Observations and Conclusion

Lab 07
Race Condition and Zombie Process

Objective

To Processes' Race Condition and Zombie Processes.

Theory

Race Condition

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. It is also defined as; an execution ordering of concurrent flows that results in undesired behavior is called a race condition-a software defect and frequent source of vulnerabilities.

Race condition is possible in runtime environments, including operating systems that must control access to shared resources, especially through process scheduling.

Avoid Race Condition

If a process wants to wait for a child to terminate, it must call one of the wait functions. If a process wants to wait for its parent to terminate, a loop of the following form could be used

```
while( getppid() != 1 )
  sleep(1);
```

The problem with this type of loop (called polling) is that it wastes CPU time, since the caller is woken up every second to test the condition. To avoid race conditions and to avoid polling, some form of signaling is required between multi processes.

fork() function

An existing process can create a new one by calling the fork function. Returns: 0 in child, process ID of child in parent, 1 on error.

```
#include <unistd.h>
pid_t fork(void);
```

- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call **`getppid`** to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent. Example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment.

Zombie Process

A zombie process or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status. The term zombie process derives from the common definition of zombie - an undead person 😊.

System() function

It is convenient to execute a command string from within a program. ANSI C defines the system function, but its operation is strongly system dependent. The system function is not defined by POSIX, because it is not an interface to the operating system, but really an interface to a shell. The prototype of system function is:

```
#include <stdlib.h>
int system(const char *cmdstring);
```

It executes a command specified in string by calling **/bin/sh -c** string, and returns after the command has been completed. During execution of the command, SIGCHLD will be blocked, and **SIGINT** and **SIGQUIT** will be ignored. If **cmdstring** is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, system is always available. Because system is implemented by calling **fork**, **exec**, and **waitpid**, there are three types of return values.

- If either the fork fails or **waitpid** returns an error other than **EINTR**, system returns 1 with **errno** set to indicate the error.
- If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed **exit**(127).
- Otherwise, all three functions fork, exec, and **waitpid** succeed, and the return value from system is the termination status of the shell, in the format specified for **waitpid**.

Lab Task – 1

Write a C/C++ program to illustrate the race condition

Code for Task – 1

Code for Task – 1

Write the output appeared after running the **`race.c`**

Output of race.c

Lab Task – 2

Write a C/C++ program that creates a zombie and then calls system to execute the **`ps`** command to verify that the process is zombie.

Write the output appeared after running the **`zombie.c`**

Output of zombie.c

Lab 08
Process Synchronization

Objective

To understand Processes' Synchronization by simulating Producer Consumer and Dining Philosophers Problem

Theory

When two or more threads need to access to a shared resource, they need some way to ensure that the resource will be used only one thread at a time. The process by which this is achieved is called synchronization. Semaphores allow process to synchronize execution.

Producer – Consumer Problem

Producer – Consumer problem is also known as the bounded-buffer problem. It is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Semaphore

The semaphore system calls allow processes to synchronize execution by doing a set of operation's atomically on a set of semaphores. The implementation of Semaphores defines two integers – valued objects that have two atomic operations define for them. The P operation decrements value of a semaphore if its value greater than 0, and V operation increments its value. Because the operations are atomic, at most one P or V operation can succeed on a semaphore at any time. The semaphore system calls are **`scmget`** to create and gain access to a set of semaphores, **`scmctl`** to do various control operations on the set, and **`scmop`** to manipulate the value of semaphores.

Dining Philosophers'' Problem

The dining philosophers' problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them (As discussed in Lecture).

Lab Task – 1

Write a C program to simulate Producer – Consumer problem using semaphores. Write details of program line by line (You may also comment on each line). Sample output is presented on next page.

Description

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.

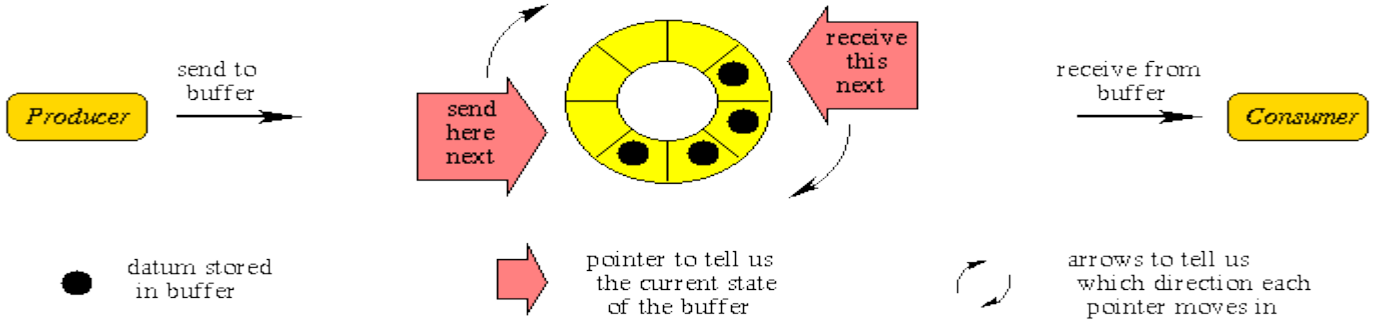


Figure 8-1

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

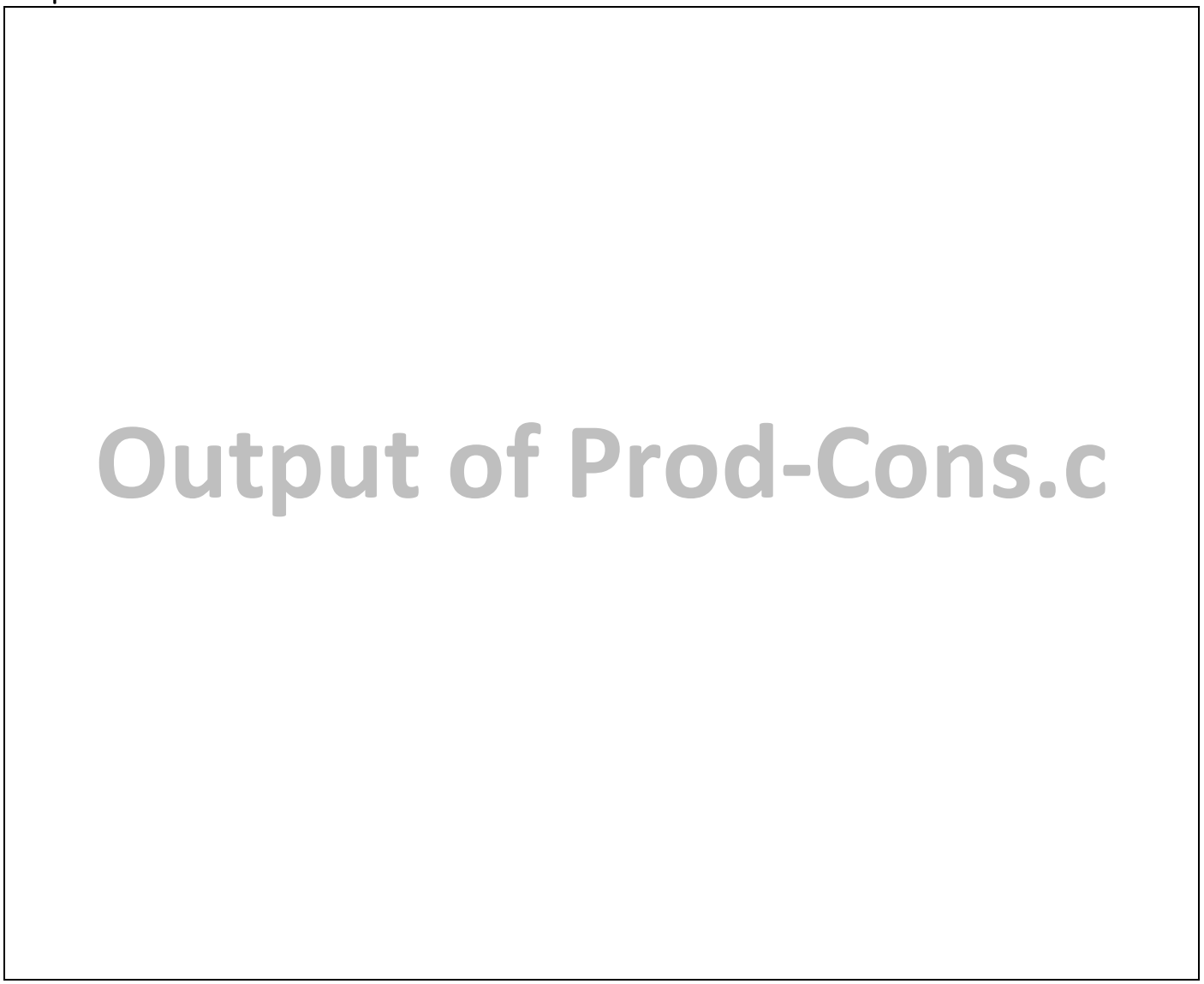
Sample output

```
1. Produce 2. Consume 3. Exit
Enter your choice: 2
Buffer is Empty
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce 2. Consume 3. Exit
Enter your choice: 2
The consumed value is 100
1. Produce 2. Consume 3. Exit
Enter your choice: 3
```

Write Commented Code in following box:

Prod-Cons.c

Output of Prod-Cons.c



Lab Task – 2

Write a C program to simulate the concept of Dining-Philosophers problem. Write details of program line by line (You may also comment on each line).

Description

The dining - philosophers' problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining - philosophers' problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

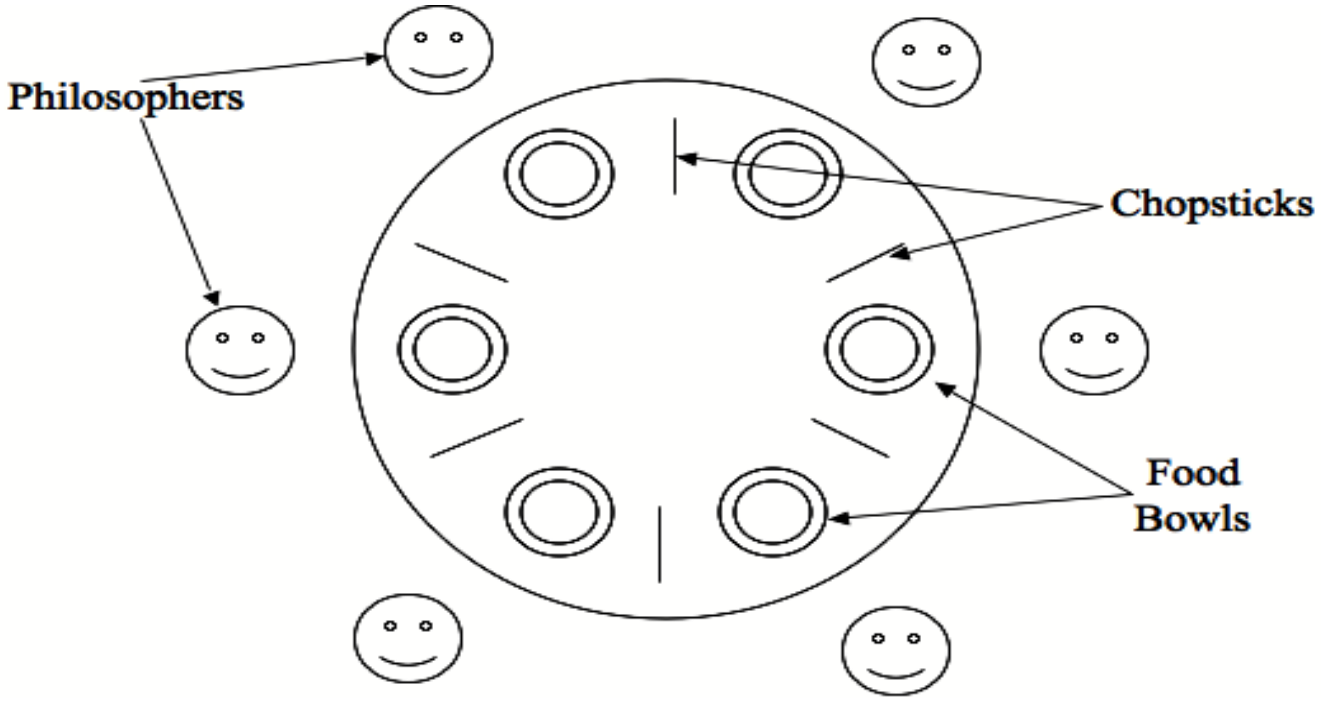


Figure 8-2

Write Commented Code in following box:

Output of Di-Ph.c

Lab 09
Thread Libraries

Objective

To understand Threads Creation and Execution

Theory

What is Thread?

A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

PTHREADS

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the *IEEE POSIX 1003.1c standard (1995)*. Implementations which adhere to this standard are referred to as POSIX threads, or **`pthreads`**. Most hardware vendors now offer **`pthreads`** in addition to their proprietary API's.

`pthreads` are defined as a set of C language programming types and procedure calls. Vendors usually provide a **`pthreads`** implementation in the form of a header/include file and a library which you link with your program.

Why PTHREAD?

1. The primary motivation for using **`pthreads`** is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
2. All threads within a process share the same address space. Inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication.
3. Threaded applications offer potential performance gains and practical advantages over nonthreaded applications in several other ways:

    3.1 Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
    3.2 Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
    3.3 Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

4. Multi-threaded applications will work on a uniprocessor system, yet naturally take advantage of a multiprocessor system, without recompiling.
5. In a multiprocessor environment, the most important reason for using **pthreads** is to take advantage of potential parallelism. This will be the focus of the remainder of this session.

The **pthreads** API

The subroutines which comprise the **pthreads** API can be informally grouped into three major classes:

Thread management

The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes

The second class of functions deal with a coarse type of synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables

The third class of functions deal with a finer type of synchronization – based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Naming conventions

All identifiers in the thread's library begin with **pthread_**

pthread_	- Threads themselves and miscellaneous subroutines
pthread_t	- Thread objects
pthread_attr	- Thread attributes objects
pthread_mutex	- Mutexes
pthread_mutexattr	- Mutex attributes objects.
pthread_cond	- Condition variables
pthread_condattr	- Condition attributes objects
pthread_key	- Thread-specific data keys

Thread Management Functions

The function **pthread_create** is used to create a new thread, and a thread to terminate itself uses the function **pthread_exit**. A thread to wait for termination of another thread uses the function **pthread_join**.

int pthread_create

```
int pthread_create
(
pthread_t * threadhandle, /* Thread handle returned by reference */
pthread_attr_t *attribute, /* Special Attribute for starting thread,
may be
NULL */
void *(*start_routine)(void *), /* Main Function which thread
executes */
```

```
void *arg /* An extra argument passed as a pointer */ );
```

Description

Request the PThread library for creation of a new thread. The return value is 0 on success. The return value is negative on failure. The pthread_t is an abstract datatype that is used as a handle to reference the thread.

void pthread_exit

```
void pthread_exit
(
void *retval /* return value passed as a pointer */
);
```

Description

This Function is used by a thread to terminate. The return value is passed as a pointer. This pointer value can be anything so long as it does not exceed the size of (void *). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large.

int pthread_join

```
(
pthread_t threadhandle, /* Pass threadhandle */
void **returnvalue /* Return value is returned by ref. */
);
```

Description

Return **0** on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.

Thread Initialization

Include the pthread.h library

```
#include <pthread.h>
```

Declare a variable of type pthread_t

```
pthread_t the_thread
```

When you compile, add -lpthread to the linker flags

```
~ $ gcc -o thread1 thread1.c -lpthread
```

Initially, threads are created from within a process. Once created, threads are peers, and may create other threads. Note that an "initial thread" exists by default and is the thread, which runs main.

Terminating Thread Execution

```
int pthread_cancel (pthread_t thread)
```

pthread_cancel sends a cancellation request to the thread denoted by the thread argument. If there is no such thread, **pthread_cancel** fails. Otherwise it returns 0. A cancel is a mechanism by which a calling thread informs either itself or the called thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread receives or handles the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested. The programmer may specify a termination status, which is stored as a void pointer for any thread that may join the calling thread. The thread returns from its starting routine (the main routine for the initial thread). By

default, the **pthreads** library will reclaim any system resources used by the thread. This is similar to a process terminating when it reaches the end of main.

The thread makes a call to the **pthread_exit** subroutine (covered below). The thread receives a signal that terminates it the entire process is terminated due to a call to either the exec or exit subroutines.

Lab Task – 1

Run the following code and write its output:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
// Cearting external function to be called as a thread
void *child_func(void *p) {
 printf ("The created Child ID is --->  %d\n", getpid( ));
}
main ( ) {
 pthread_t child ;       // Creating thread
 // Calling child_func()
 pthread_create (&child, NULL, child_func, NULL) ;
 printf ("Parent ID is --->  %d\n", getpid( )) ;
 pthread_join (child, NULL) ;
 printf ("No more child process alive...!!!\n") ;
}
```

Answer:

Why the process IDs of parent and child threads are same?

_____

_____

_____

Lab Task – 2

Run the following code and write its output:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
int global_data = 5 ;
void *child_func(void *p) {
printf ("Now the Child is here. Global data was %d.\n", global_data);
 global_data = 15 ;
printf ("Its Child Again. Global data was now %d.\n", global_data);
}
int main ( ) {
  pthread_t child ;
  pthread_create (&child, NULL, child_func, NULL) ;
 printf ("The Parent is here. Global data = %d\n", global_data) ;
 global_data = 10 ;
 pthread_join (child, NULL) ;
 printf ("End of program. Global data = %d\n", global_data) ;
 return 0;
}
```

Output of thread2.c

Answer:

Do the threads have separate copies of **global_data**, explain your answer?

_____

_____

_____

Lab Task – 3

Write a C program including following function:

```
void *runner(void *param)
{
int upper = atoi(param);
int i;
sum=0;
if(upper > 0)
{
for(i=1; i <= upper;i++)
sum += i;
}
pthread_exit(0);
}
```

Description

Write a Program that creates a separate thread that determines the summation of a non-negative integer. In asked thread program, separate thread begin execution in a specified function. In above part of program, it is the runner function. When this program begins, a single thread of control begins in main. After some initialization, main creates a second thread that begins control in the summer function.

Program multhread.c

Program multhread.c

Output of multhread.c
Without input integer

Output of multhread.c
With Integer

Reference text

All **Pthread** programs must include the **pthread.h** header file. The statement **pthread_t tid** declares the identifier for the thread we will create. Each thread has a set of attributes including stack size and scheduling information. The **pthread_attr_t attr** declaration represents the attributes for the thread. We will set the attributes in the function call **pthread_attr_init(&attr)**.

Because we did not explicitly set any attributes, we will use the default attribute provided. A separate thread is created with the **pthread_create** function call. In addition to passing the thread identifier and the attributes for the thread. We also pass the name of the function where the new thread will execution, in this case runner function. Lastly, we pass the integer parameter that was provided on the command line, **argv[1]**.

At this point, the program has two threads: the initial thread in main and the thread performing the summation in the runner function. After creating the second thread, the main thread will wait for the runner thread to complete by calling the **pthread_join** function. The runner thread will complete when it calls the function **pthread_exit**.

Lab 10
Scheduling Schemes

Objective

Write C programs to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time for each below mentioned scheme, assume all the processes arrive at the same time:

1 – FCFS 2 – SJF 3 – Round Robin 4 – Priority

Theory

FCFS – First Come First Serve Scheduling

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

SJF – Shortest Job First Scheduling

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

RR – Round Robin Scheduling

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

Priority Based Scheduling Algorithm

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

Lab Task – 1a

Run the following C program and write the output in the output box, enter at-least **5** processes and at-least **5** burst time for each process.

Note: You might need to install extra libraries in LINUX based OS, if you use ***ncurses*** in LINUX based OSes. ***ncurses*** is equivalent of ***conio***:

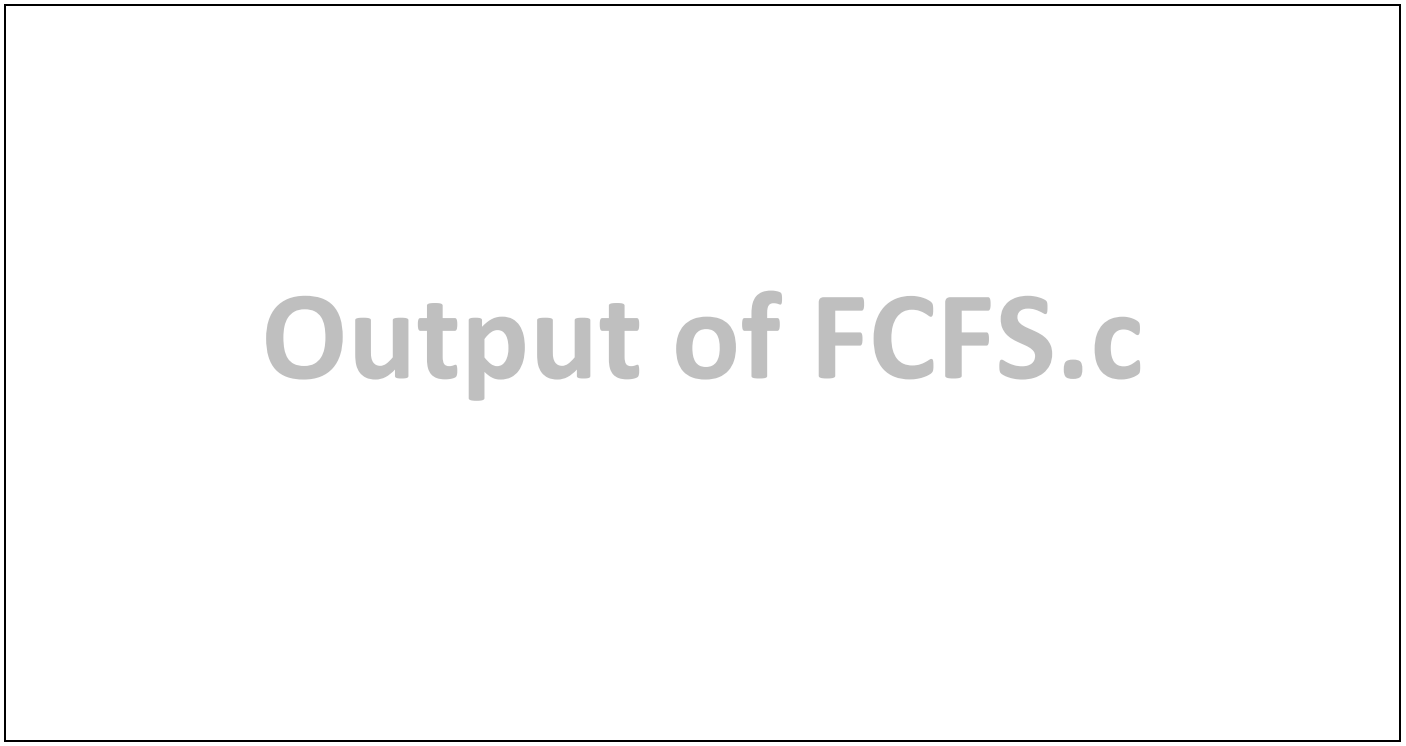
```
apt-get install libncurses5-dev libncursesw5-dev
```

For non-root login type with **sudo**:

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

FCFS.c

```
#include<stdio.h>
main() {
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
system("clear");
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{    printf("\nEnter Burst Time for Process %d -- ", i);
     scanf("%d", &bt[i]);     }
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{    wt[i] = wt[i-1] +bt[i-1];
     tat[i] = tat[i-1] +bt[i];
     wtavg = wtavg + wt[i];
     tatavg = tatavg + tat[i];     }
printf("\t  PROCESS  \tBURST  TIME  \t  WAITING  TIME\t  TURNAROUND
TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f\n", tatavg/n);
}
```



Lab Task – 1b

Run the following C program and write the output in the output box, enter at-least **5** processes and at-least **5** burst time for each process.

SJF.c

```
#include<stdio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
system("clear");
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
     for(i=0;i<n;i++)    {
     p[i]=i;
     printf("Enter Burst Time for Process %d -- ", i);
     scanf("%d", &bt[i]);     }
     for(i=0;i<n;i++)
          for(k=i+1;k<n;k++)
          if(bt[i]>bt[k])     {
          temp=bt[i];
          bt[i]=bt[k];
          bt[k]=temp;
          temp=p[i];
          p[i]=p[k];
          p[k]=temp;     }
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
     for(i=1;i<n;i++)    {
     wt[i] = wt[i-1] +bt[i-1];
     tat[i] = tat[i-1] +bt[i];
     wtavg = wtavg + wt[i];
     tatavg = tatavg + tat[i];     }
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f\n", tatavg/n); }
```

Output of SJF.c

Lab Task – 1c

Run the following C program and write the output in the output box, enter at-least **5** processes and select odd number for burst time for each process. Time slice should be **2**.

RR.c

```
#include<stdio.h>
main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
system("clear");
printf("Enter the no of processes -- ");
scanf("%d",&n);
     for(i=0;i<n;i++)    {
          printf("\nEnter Burst Time for process %d -- ", i+1);
          scanf("%d",&bu[i]);
          ct[i]=bu[i];   }
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
     for(i=1;i<n;i++)
          if(max<bu[i])
          max=bu[i];
          for(j=0;j<(max/t)+1;j++)
               for(i=0;i<n;i++)
```

```
                    if(bu[i]!=0)
                              if(bu[i]<=t)    {
                              tat[i]=temp+bu[i];
                              temp=temp+bu[i];
                              bu[i]=0;   }
                    else  {
                    bu[i]=bu[i]-t;
                    temp=temp+t;    }
          for(i=0;i<n;i++)     {
          wa[i]=tat[i]-ct[i];
          att+=tat[i];
          awt+=wa[i];     }
     printf("\nThe Average Turnaround time is -- %f",att/n);
     printf("\nThe Average Waiting time is -- %f ",awt/n);
     printf("\n\tPROCESS\t  BURST  TIME  \t  WAITING  TIME\tTURNAROUND
TIME\n");
     for(i=0;i<n;i++)
     printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
     }
```

Output of SJF.c

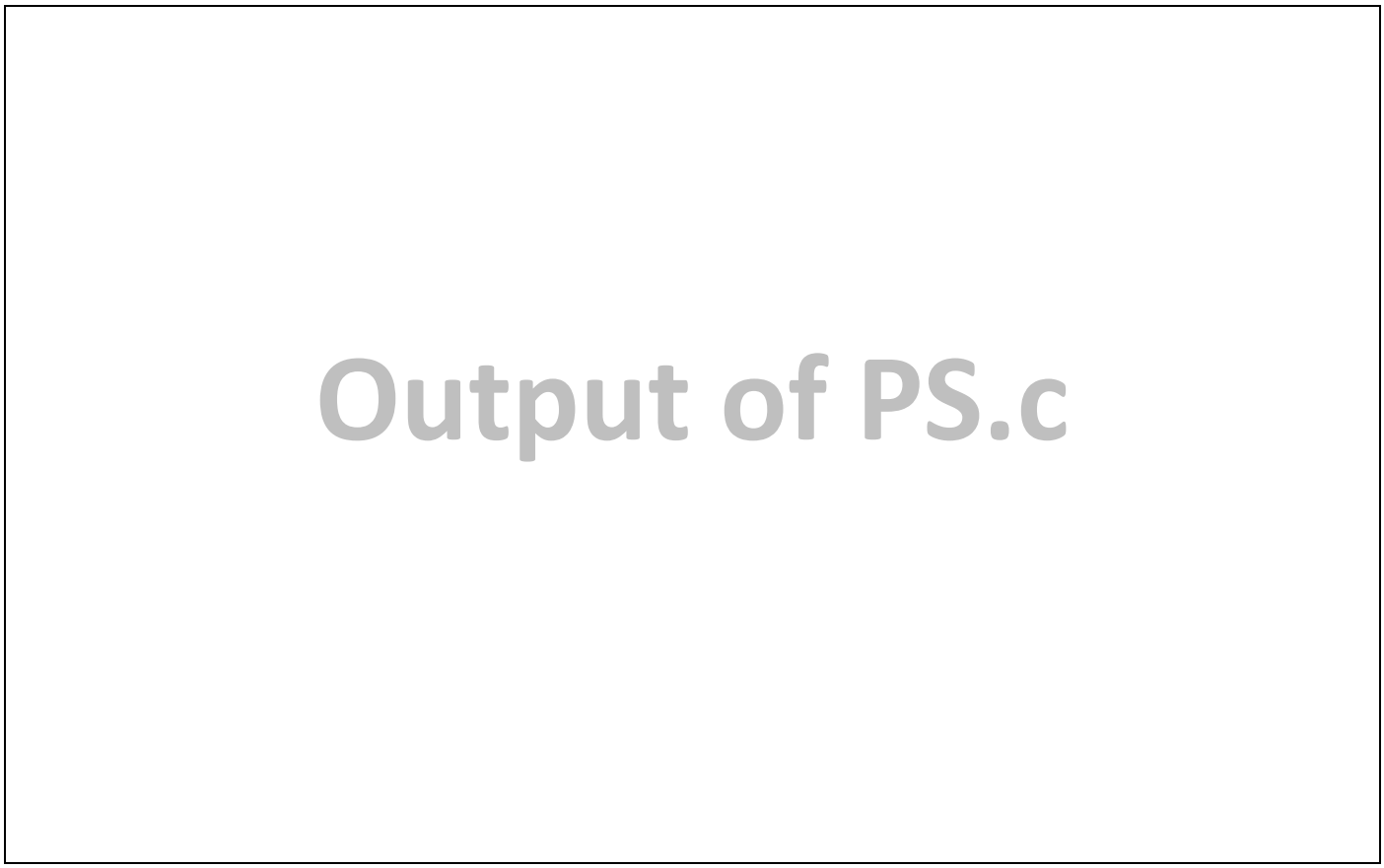
Lab Task – 1d

Run the following C program and write the output in the output box, enter at-least **5** processes and at-least **5** burst time for each process. Select priority between 1 – 10, remember lowest integer highest priority.

PS.c

```
#include<stdio.h>
main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
float wtavg, tatavg;
system("clear");
printf("Enter the number of processes --- ");
scanf("%d",&n);
     for(i=0;i<n;i++)    {
          p[i] = i;
          printf("Enter the Burst Time & Priority of Process %d --- ",i);
          scanf("%d %d",&bt[i], &pri[i]);}
     for(i=0;i<n;i++)
          for(k=i+1;k<n;k++)
               if(pri[i] > pri[k]) {
               temp=p[i];
               p[i]=p[k];
               p[k]=temp;
               temp=bt[i];
               bt[i]=bt[k];
               bt[k]=temp;
               temp=pri[i];
               pri[i]=pri[k];
               pri[k]=temp;   }
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
     for(i=1;i<n;i++)    {
          wt[i] = wt[i-1] + bt[i-1];
          tat[i] = tat[i-1] + bt[i];
          wtavg = wtavg + wt[i];
          tatavg = tatavg + tat[i];     }
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
     for(i=0;i<n;i++)
```

```
printf("\n%d    \t\t    %d    \t\t    %d    \t\t    %d    \t\t    %d
",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n);
printf("\nAverage Turnaround Time is --- %f\n",tatavg/n);
}
```



Lab Task – 2

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. The priority of each process ranges from **1** to **3**. Use fixed priority scheduling for all the processes.

Description

Multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm like round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues.

Write program with explanation in box on next page.

MLQS.c

MLQS.c

MLQS.c

Output of MLQS.c

Lab 11
Deadlocks

Objective

Write a C program to simulate Bankers Agorithm for the purpose of deadlock avoidance.

Theory

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

Lab Task – 1

Write C program to simulate Deadlock Avoidance

DL-A.c

Description of DL-A.c

Output of DL-A.c

Lab 12
Memory Management

Objective

Write a C program to simulate the following contiguous memory allocation techniques:

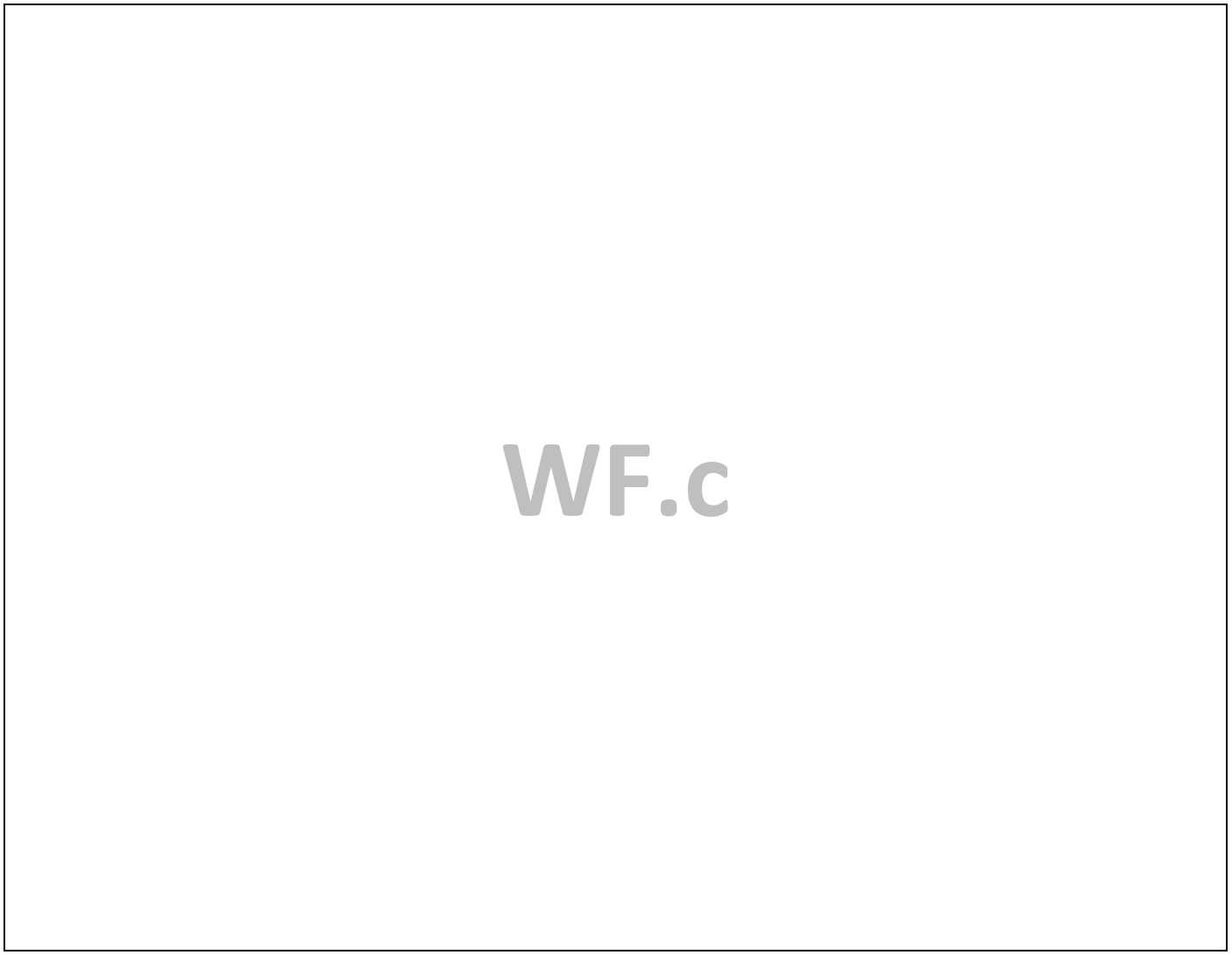
1) Worst – Fit 2) Best – Fit 3) First – Fit

Theory

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

Lab Task – 1a

Write C Program to simulate Worst – Fit Memory Management Algorithm



WF.c

Output of WF.c

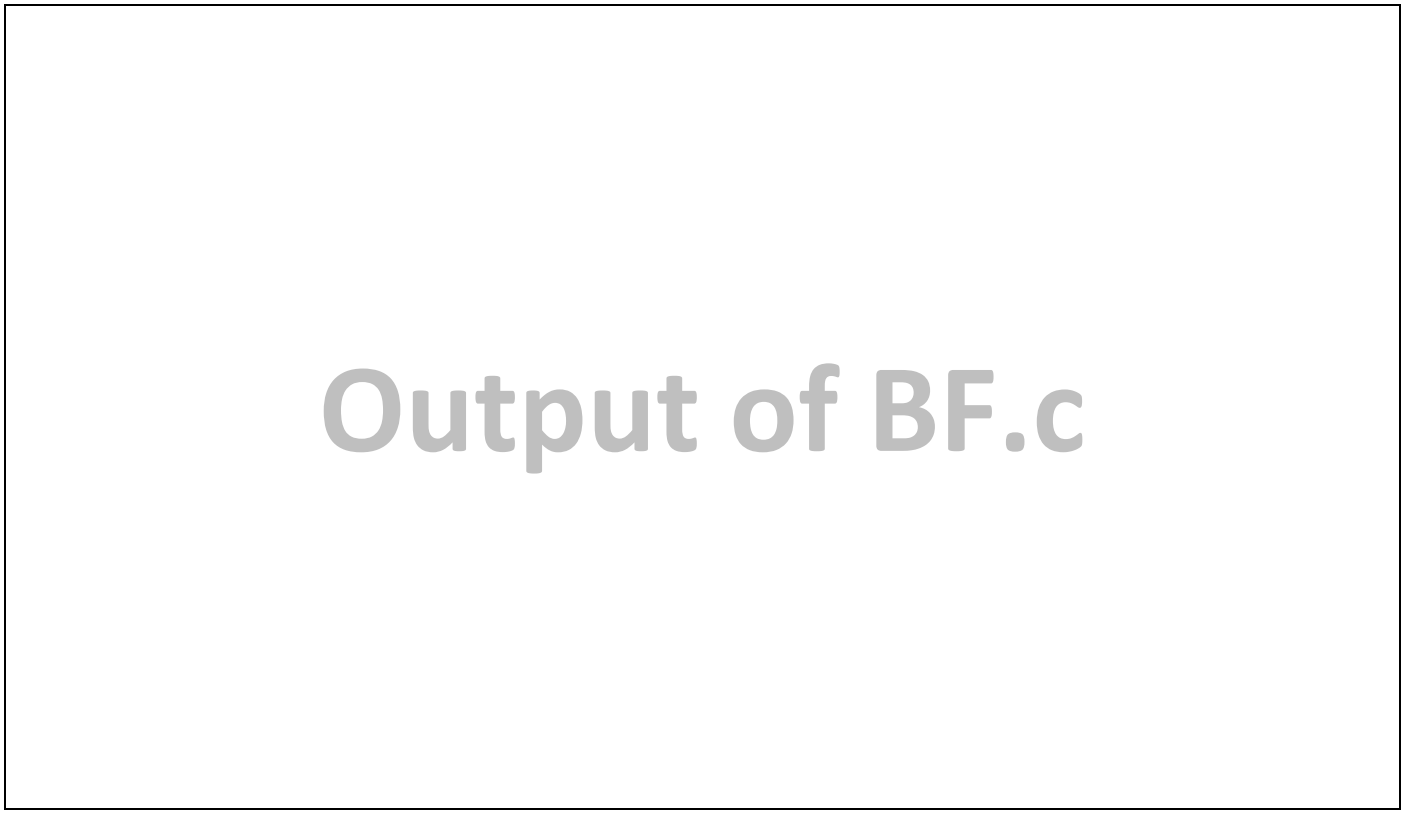
Output of WF.c

Lab Task – 1b

Write C Program to simulate Best – Fit Memory Management Algorithm

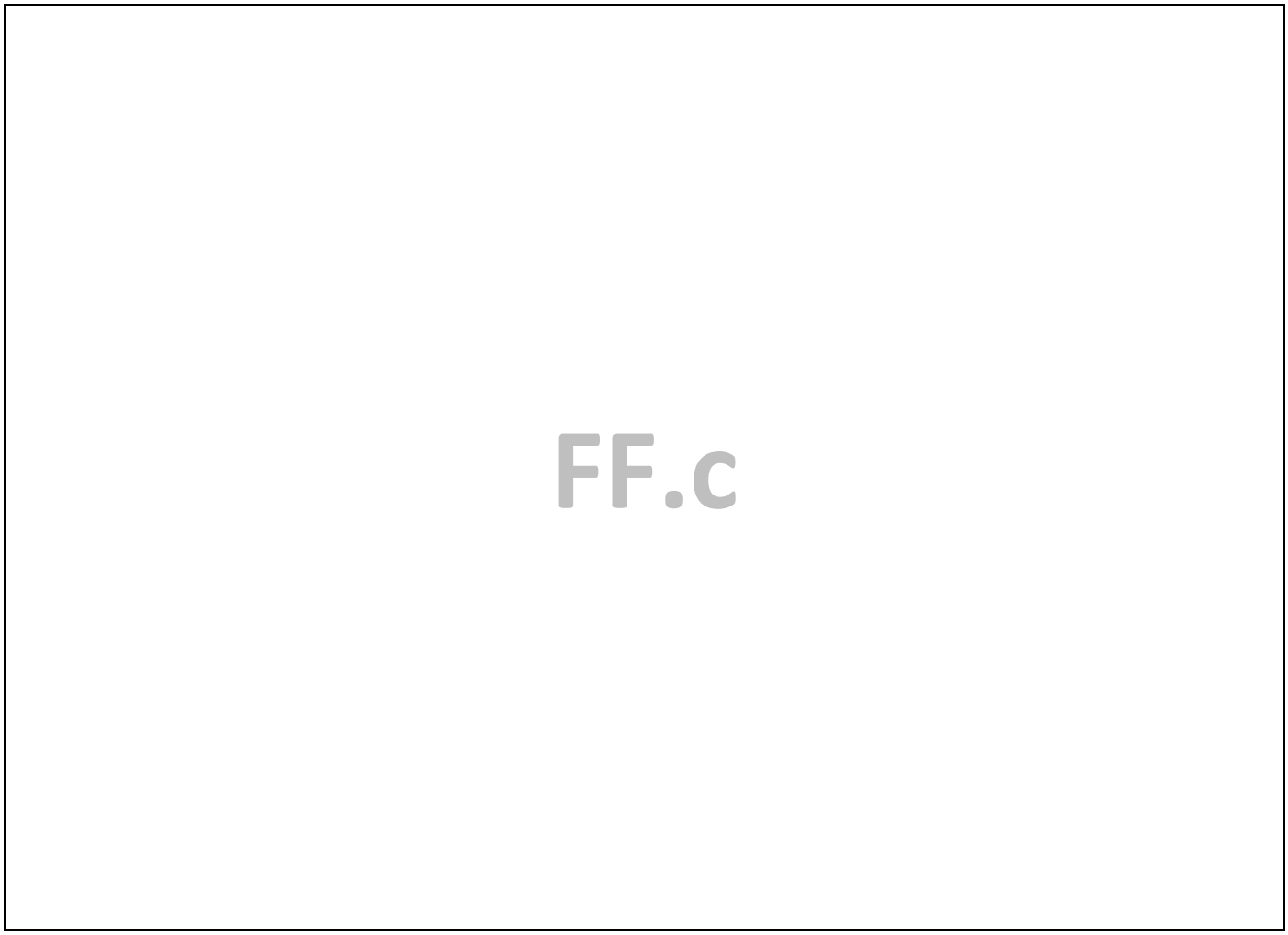
BF.c

Output of BF.c



Lab Task – 1c

Write C Program to simulate First – Fit Memory Management Algorithm



FF.c

Output of FF.c

Output of FF.c

Lab 13
Virtual Memory

Objective

To simulate FIFO, LRU and LFU page replacement algorithm in Virtual Memory Management

Theory

FIFO

FIFO (First in First Out) algorithm: FIFO is the simplest page replacement algorithm, the idea behind this is, “Replace a page that page is oldest page of main memory” or “Replace the page that has been in memory longest”. FIFO focuses on the length of time a page has been in the memory rather than how much the page is being used.

LRU

LRU (Least Recently Used): the criteria of this algorithm are “Replace a page that has been used for the longest period of time”. This strategy is the page replacement algorithm looking backward in time, rather than forward.

LFU

LFU (Least Frequently Used): The least frequently used algorithm “select a page for replacement, if the page has not been used for the often in the past” or “Replace page that page has smallest count” for this algorithm each page maintains as counter which counter value shows the least count, replace that page. The frequency counter is reset each time is page is loaded.

Lab Task – 1

Run the following C program to simulate **FIFO** Page Replacement. Run it at-least 3 time with variable inputs. Also explain the outputs in the explaining area.

FIFO.c

```
#include<stdio.h>
int fr[3],m;
void display();
void main(){
 int i,j,page[20];
 int flag1=0,flag2=0,pf=0;
 int n, top=0;
 float pr;
 system("clear");
 printf("Enter length of the reference string: ");
 scanf("%d",&n);
 printf("Enter the reference string: ");
     for(i=0;i<n;i++)
          scanf("%d",&page[i]);
 printf("Enter no of frames: ");
 scanf("%d",&m);
```

```
          for(i=0;i<m;i++)
               fr[i]=-1;
               for(j=0;j<n;j++) {
                flag1=0;
                flag2=0;
                     for(i=0;i<m;i++) {
                     if(fr[i]==page[j]) {
                     flag1=1;
                     flag2=1;
                      break; } }
                     if(flag1==0) {
                          for(i=0;i<m;i++) {
                          if(fr[i]==-1)  {
                          fr[i]=page[i];
                          flag2=1;
                          break;  } } }
     if(flag2==0)  {
     fr[top]=page[j];
     top++;
     pf++;
     if(top>=m)
     top=0;  }
     display(); }
    pf+=m;
    printf("Number of page faults : %d\n", pf);
    pr=(float)pf/n*100;
    printf("Page fault rate = %f \n", pr); }
    void display() {
     int i;
          for(i=0;i<m;i++)
               printf("%d\t", fr[i]);
          printf("\n"); }
```

Output – 1 FIFO.c

Output – 2 FIFO.c

Output – 3 FIFO.c

Explanation of Outputs FIFO.c

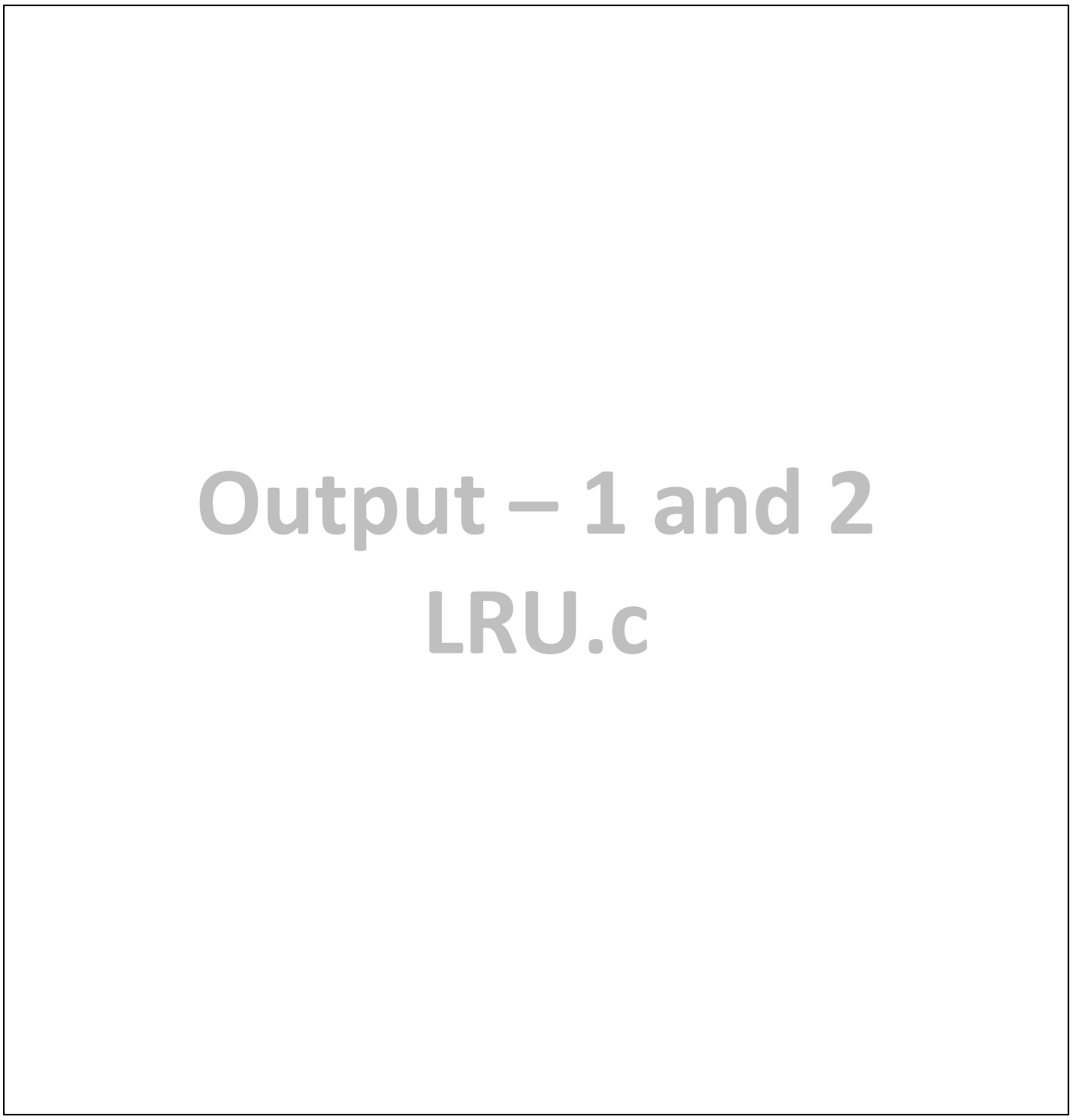
Lab Task – 2

Run the following C program to simulate **LRU** Page Replacement. Run it at-least 3 time with variable inputs. Also explain the outputs in the explaining area.

LRU.c

```
#include<stdio.h>
main()
{
int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
system("clear");
printf("Enter the length of reference string -- ");
scanf("%d",&n);
printf("Enter the reference string -- ");
     for(i=0;i<n;i++) {
          scanf("%d",&rs[i]);
          flag[i]=0;}
printf("Enter the number of frames -- ");
scanf("%d",&f);
     for(i=0;i<f;i++) {
          count[i]=0;
          m[i]=-1; }
printf("\nThe Page Replacement process is -- \n");
     for(i=0;i<n;i++) {
          for(j=0;j<f;j++){
               if(m[j]==rs[i]) {
               flag[i]=1;
               count[j]=next;
               next++; } }
          if(flag[i]==0)      {
          if(i<f)        {
               m[i]=rs[i];
               count[i]=next;
               next++;        }
               else           {
               min=0;
          for(j=1;j<f;j++)
               if(count[min] > count[j])
               min=j;
               m[min]=rs[i];
```

```
                    count[min]=next;
                    next++; }
               pf++; }
          for(j=0;j<f;j++)
               printf("%d\t", m[j]);
               if(flag[i]==0)
               printf("PF No. -- %d" , pf);
               printf("\n");
               }
     printf("\nThe number of page faults using LRU are %d\n",pf);}
```



Output – 3 LRU.c

Explanation of Outputs LRU.c

Lab Task – 3

Run the following C program to simulate **LFU** Page Replacement. Run it at-least 3 time with variable inputs. Also explain the outputs in the explaining area.

LFU.c

```
#include<stdio.h>
main() {
int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0;
system("clear");
printf("\nEnter number of page references -- ");
scanf("%d",&m);
printf("\nEnter the reference string -- ");
     for(i=0;i<m;i++)
     scanf("%d",&rs[i]);
printf("\nEnter the available no. of frames -- ");
scanf("%d",&f);
     for(i=0;i<f;i++) {
     cntr[i]=0;
     a[i]=-1; }
printf("\nThe Page Replacement Process is - \n");
     for(i=0;i<m;i++) {
          for(j=0;j<f;j++)
          if(rs[i]==a[j]){
          cntr[j]++;
          break;}
     if(j==f){
     min = 0;
          for(k=1;k<f;k++)
               if(cntr[k]<cntr[min])
               min=k;
               a[min]=rs[i];
               cntr[min]=1;
               pf++; }
printf("\n");
     for(j=0;j<f;j++)
          printf("\t%d",a[j]);
          if(j==f)
          printf("\tPF No. %d",pf);          }
printf("\n\n Total number of page faults -- %d\n",pf);}
```

Output – 1 and 2
LFU.c

Output – 3 LFU.c

Explanation of Outputs FIFO.c

Lab 14
File System Access and Control System

Objective

To understand File System Access and Control, Systems' Calls in LINUX based Operating Systems

Theory

Testing File Permissions

access

The **access** system call determines whether the calling process has access permission to a file. It can check any combination of read, write, and execute permission, and it can also check for a file's existence. The access call takes two arguments. The first is the path to the file to check. The second is a bitwise or of **R_OK**, **W_OK**, and **X_OK**, corresponding to read, write, and execute permission. The return value is **0** if the process has all the specified permissions. If the file exists but the calling process does not have the specified permissions, access returns –1 and sets **errno** to **EACCES** (or **EROFS**, if write permission was requested for a (file on a read-only file system).

If the second argument is **F_OK**, access simply checks for the file's existence. If the file exists, the return value is **0**; if not, the return value is **–1** and **errno** is set to **ENOENT**. Note that **errno** may instead be set to **EACCES** if a directory in the file path is inaccessible. The program will be discussed in Lab Task – 1

Locking File

fcntl

The **fcntl** system call is the access point for several advanced operations on file descriptors. The first argument to **fcntl** is an open file descriptor, and the second is a value that indicates which operation is to be performed. For some operations, **fcntl** takes an additional argument. We'll describe here one of the most useful **fcntl** operations, file locking. The **fcntl** system call allows a program to place a read lock or a write lock on a file, somewhat analogous to the mutex locks discussed in theory sessions. A read lock is placed on a readable file descriptor, and a write lock is placed on a writable file descriptor. More than one process may hold a read lock on the same file at the same time, but only one process may hold a write lock, and the same file may not be both locked for read and locked for write. Note that placing a lock does not actually prevent other processes from opening the file, reading from it, or writing to it, unless they acquire locks with **fcntl** as well.

To place a lock on a file, first create and zero out a struct flock variable. Set the **l_type** field of the structure to **F_RDLCK** for a read lock or **F_WRLCK** for a write lock. Then call **fcntl**, passing a file descriptor to the file, the **F_SETLCKW** operation code, and a pointer to the struct flock variable. If another process holds a lock that prevents a new lock from being acquired, **fcntl** blocks until that lock is released.

Flushing Disk Buffers

fsync and fdatasync

On most operating systems, when you write to a file, the data is not immediately written to disk. Instead, the operating system caches the written data in a memory buffer, to reduce the number of required disk writes and improve program responsiveness. When the buffer fills or some other condition occurs (for instance, enough time elapses), the system writes the cached data to disk all at one time. Linux provides caching of this type as well. Normally, this is a great boon to performance. However, this behavior can make programs that depend on the integrity of disk-based records unreliable. If the system goes down suddenly—for instance, due to a kernel crash or power outage—any data written by a program that is in the memory cache but has not yet been written to disk is lost.

For example, suppose that you are writing a transaction-processing program that keeps a journal file. The journal file contains records of all transactions that have been processed so that if a system failure occurs, the state of the transaction data can be reconstructed. It is obviously important to preserve the integrity of the journal file— whenever a transaction is processed, its journal entry should be sent to the disk drive immediately.

To help you implement this, Linux provides the **fsync** system call. It takes one argument, a writable file descriptor, and flushes to disk any data written to this file. The **fsync** call doesn't return until the data has physically been written.

Resource Limits

getrlimit and setrlimit

The **getrlimit** and **setrlimit** system calls allow a process to read and set limits on the system resources that it can consume. You may be familiar with the **ulimit** shell command, which enables you to restrict the resource usage of programs you run; these system calls allow a program to do this programmatically. For each resource there are two limits, the hard limit and the soft limit. The soft limit may never exceed the hard limit, and only processes with superuser privilege may change the hard limit. Typically, an application program will reduce the soft limit to place a throttle on the resources it uses.

Both **getrlimit** and **setrlimit** take as arguments a code specifying the resource limit type and a pointer to a **structrlimit** variable. The **getrlimit** call fills the fields of this structure, while the **setrlimit** call changes the limit based on its contents. The **rlimit** structure has two fields: **rlim_cur** is the soft limit, and **rlim_max** is the hard limit.

Setting Memory Permissions

mprotect

If a program attempts to perform an operation on a memory location that is not allowed by these permissions, it is terminated with a **SIGSEGV** (segmentation violation) signal. After memory has been mapped, these permissions can be modified with the **mprotect** system call. The arguments to **mprotect** are an address of a memory region, the size of the region, and a set of protection flags. The memory region must consist of entire pages: The address of the region must be aligned to the system's page size, and the length of the region must be a page size multiple. The protection flags for these pages are replaced with the specified value. An advanced technique to monitor memory access is to protect the region of memory using **mmap** or **mprotect** and then handle the **SIGSEGV** signal that Linux sends to the program when it tries to access that memory.

Fast Data Transfer

sendfile

The **sendfile** system call provides an efficient mechanism for copying data from one file descriptor to another. The file descriptors may be open to disk files, sockets, or other devices. Typically, to copy from one file descriptor to another, a program allocates a fixed size buffer, copies some data from one descriptor into the buffer, writes the buffer out to the other descriptor, and repeats until all the data has been copied. This is inefficient in both time and space because it requires additional memory for the buffer and performs an extra copy of the data into that buffer.

Using **sendfile**, the intermediate buffer can be eliminated. Call **sendfile**, passing the file descriptor to write to; the descriptor to read from; a pointer to an offset variable; and the number of bytes to transfer. The offset variable contains the offset in the input file from which the read should start (0 indicates the beginning of the file) and is updated to the position in the file after the transfer. The return value is the number of bytes transferred. Include **<sys/sendfile.h>** in your program if it uses **sendfile**.

Wall Clock Time

gettimeofday

The **gettimeofday** system call gets the system's wall-clock time. It takes a pointer to a struct **timeval** variable. This structure represents a time, in seconds, split into two fields. The **tv_sec** field contains the integral number of seconds, and the **tv_usec** field contains an additional number of microseconds. This **struct timeval** value represents the number of seconds elapsed since the start of the UNIX epoch, on midnight UTC on January 1, 1970.The **gettimeofday** call also takes a second argument, which should be **NULL**. Include **<sys/time.h>** if you use this system call.

The number of seconds in the UNIX epoch isn't usually a very handy way of representing dates. The **localtime** and **strftime** library functions help manipulate the return value of **gettimeofday**. The **localtime** function takes a pointer to the number of seconds (the **tv_sec** field of **struct timeval**) and returns a pointer to a **struct tm** object. The structure contains more useful fields, which are filled according to the local time zone.

Lab Task – 1

- Make a small text file, then compile following C program.
- Run it as **./access <your text file name with extension>**
- Write output in output box

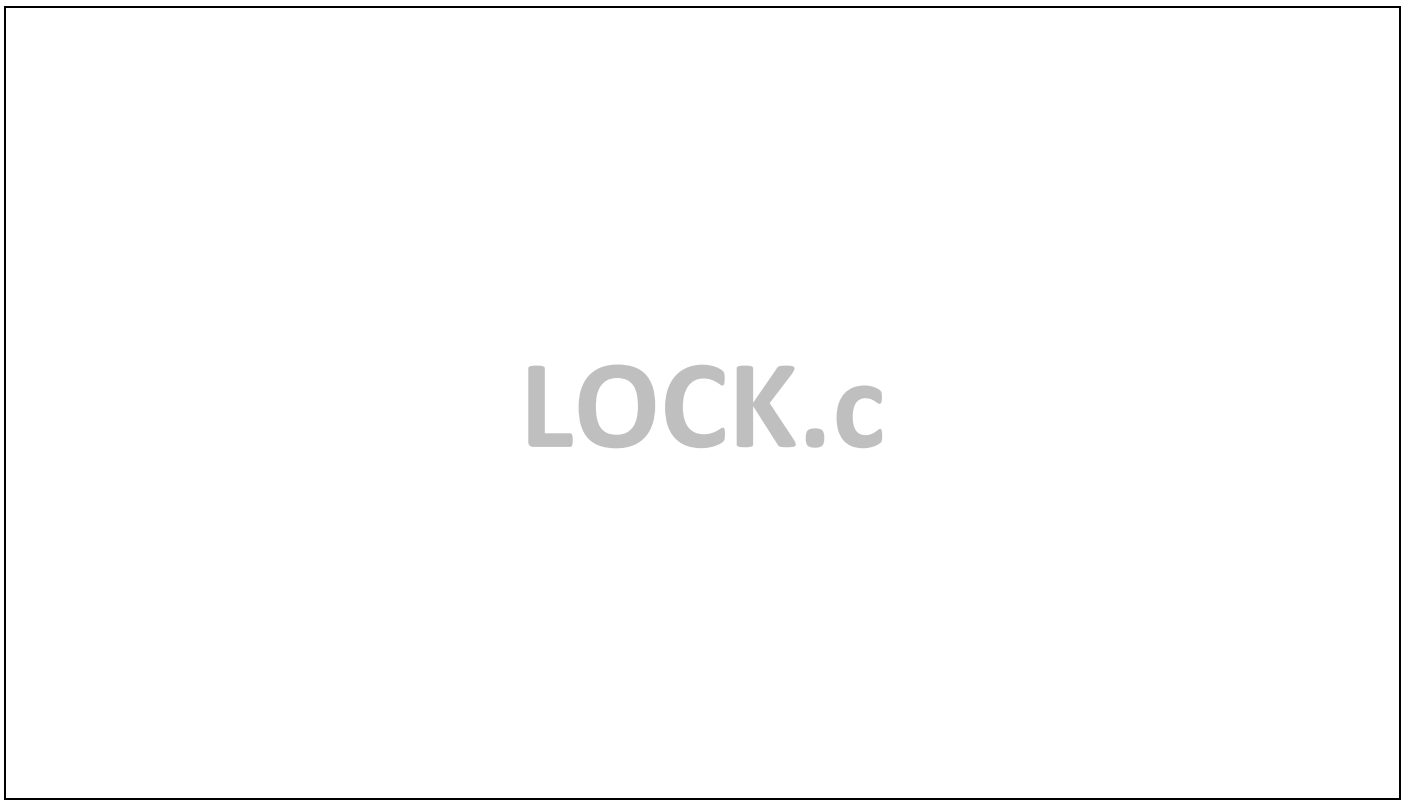
ACCESS.c

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
int main (int argc, char* argv[]) {
char* path = argv[1];
int rval;
/* Check file existence. */
rval = access (path, F_OK);
if (rval == 0)
printf ("%s exists\n", path);
else {
if (errno == ENOENT)
printf ("%s does not exist\n", path);
else if (errno == EACCES)
printf ("%s is not accessible\n", path);
return 0; }
/* Check read access. */
rval = access (path, R_OK);
if (rval == 0)
printf ("%s is readable\n", path);
else
printf ("%s is not readable (access denied)\n", path);
/* Check write access. */
```

```
rval = access (path, W_OK);
if (rval == 0)
printf ("%s is writable\n", path);
else if (errno == EACCES)
printf ("%s is not writable (access denied)\n", path);
else if (errno == EROFS)
printf ("%s is not writable (read-only filesystem)\n", path);
return 0;}
```

Lab Task – 2

Write a C program that opens a file (using **`fcntl`**) for writing whose name is provided on the command line, and then places a write lock on it. The program waits for the user to hit Enter and then unlocks and closes the file.



LOCK.c

Output

Output LOCK.c

Lab Task – 3

Complete and compile the following C program and explain its execution

DISK.c

```
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
const char* journal_filename = "journal.log";
void write_journal_entry (char* entry){
int fd = open (journal_filename, O_WRONLY | O_CREAT | O_APPEND, 0660);
write (fd, entry, strlen (entry));
write (fd, "\n", 1);
fsync (fd);
close (fd);}
```

Completed – LOCK.c

Explanation DISK.c

Lab Task – 4

Write a C program that limits CPU usage, some of the most useful resource limits that may be changed are listed here, with their codes:

- **RLIMIT_CPU**—The maximum CPU time, in seconds, used by a program. This is the amount of time that the program is actually executing on the CPU, which is not necessarily the same as wall-clock time. If the program exceeds this time limit, it is terminated with a **SIGXCPU** signal.
- **RLIMIT_DATA**—The maximum amount of memory that a program can allocate for its data. Additional allocation beyond this limit will fail.
- **RLIMIT_NPROC**—The maximum number of child processes that can be running for this user. If the process calls fork and too many processes belonging to this user are running on the system, fork fails.
- **RLIMIT_NOFILE**—The maximum number of file descriptors that the process may have open at one time.

Your program must set the limit on CPU time consumed by a program. If it sets a 1-second CPU time limit and then spins in an infinite loop. Linux must kill the process soon afterward, when it exceeds 1 second of CPU time.

Commented CPU.c

Output

Output CPU.c

Lab Task – 5

Write a C program to detect Memory Access using **mprotect**. The program should follow these steps:

- The program installs a signal handler for **SIGSEGV**.
- The program allocates a page of memory by mapping /dev/zero and writing a value to the allocated page to obtain a private copy.
- The program protects the memory by calling **mprotect** with the **PROT_NONE** permission.
- When the program subsequently writes to memory, Linux sends it **SIGSEGV**, which is handled by **segv_handler**. The signal handler unprotects the memory, which allows the memory access to proceed.
- When the signal handler completes, control returns to main, where the program deallocates the memory using **munmap**.

Commented MEMALLOC.c

Output and Reason – MEMALLOC.c

Lab Task – 6

SEND.c

Run the following code and state step by step execution and reason the output.

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[]) {
int read_fd;
int write_fd;
struct stat stat_buf;
off_t offset = 0;
read_fd = open (argv[1], O_RDONLY);
fstat (read_fd, &stat_buf);
write_fd = open (argv[2], O_WRONLY | O_CREAT, stat_buf.st_mode);
sendfile (write_fd, read_fd, &offset, stat_buf.st_size);
close (read_fd);
close (write_fd);
return 0;}
```

Commented SEND.c

Lab Task – 6

Write a C program that displays system time using **gettimeofday**, the program must follow following structure:

- **tm_hour, tm_min, tm_sec** —The time of day, in hours, minutes, and seconds.
- **tm_year, tm_mon, tm_day** —The year, month, and date.
- **tm_wday** —The day of the week. Zero represents Sunday.
- **tm_yday** —The day of the year.
- **tm_isdst** —A flag indicating whether daylight savings time is in effect.

Description

The **strftime** function additionally can produce from the struct tm pointer a customized, formatted string displaying the date and time. The format is specified in a manner similar to **printf**, as a string with embedded codes indicating which time fields to include. For example, below format of string

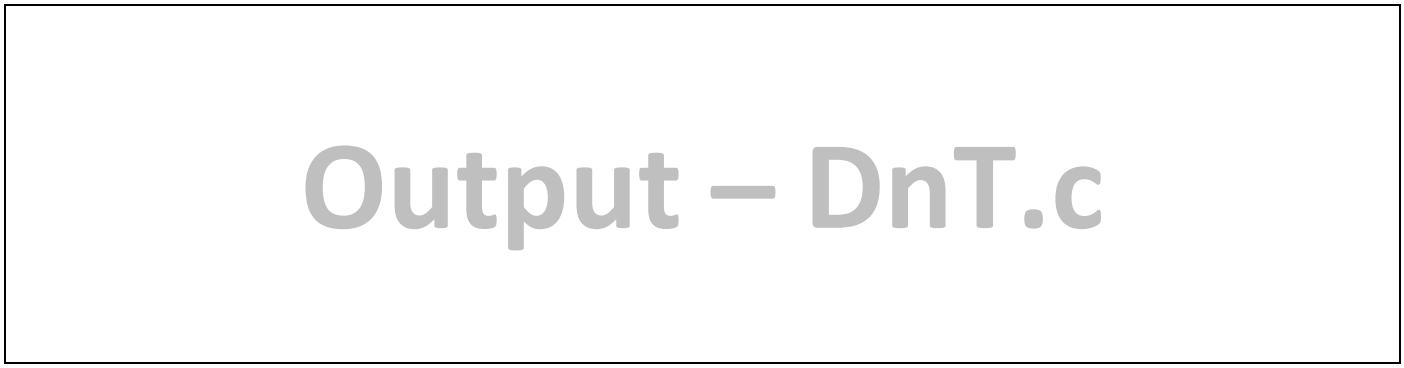
```
"%Y-%m-%d %H:%M:%S"
```

specifies the date and time in this form:

```
2018-11-15 12:04:08.942
```

Pass **strftime** a character buffer to receive the string, the length of that buffer, the format string, and a pointer to a struct tm variable. See the **strftime** man page for a complete list of codes that can be used in the format string. Notice that neither **localtime** nor **strftime** handles the fractional part of the current time more precise than 1 second (the **tv_usec** field of struct **timeval**). If you want this in your formatted time strings, you may include it yourself, **include <time.h>** if you want to call **localtime** or **strftime**.

Output



Lab 15
Disk Scheduling Algorithms

Objective

Write a C program to simulate disk scheduling algorithms

1) FCFS 2) SCAN 3) C-SCAN

Theory

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

Lab Task – 1

DISK-FCFS.c

```
#include<stdio.h>
main() {
int t[20], n, i, I, j, tohm[20], tot=0;
 float avhm;
 system ("clear");
 printf("enter the no.of tracks");
     scanf("%d",&n);
 printf("enter the tracks to be traversed");
 for(i=2;i<n+2;i++)
     scanf("%d",&t[i]);
 for(i=1;i<n+1;i++)  {
     tohm[i]=t[i+1]-t[i];
     if(tohm[i]<0)
     tohm[i]=tohm[i]*(-1); }
 for(i=1;i<n+1;i++)
      tot+=tohm[i];
      avhm=(float)tot/n;
 printf("Tracks traversed\tDifference between tracks\n");
 for(i=1;i<n+1;i++)
```

```
        printf("%d\t\t\t%d\n",t[i],tohm[i]);
    printf("\nAverage header movements:%f",avhm);  }
```

Output – DISK-FCFS.c

Lab Task – 2

SCAN-DISK.c

```
#include<stdio.h>
main() {
int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
systen ("clear");
printf("enter the no of tracks to be traveresed");
scanf("%d'",&n);
printf("enter the position of head");
scanf("%d",&h);
t[0]=0;t[1]=h;
printf("enter the tracks");
for(i=2;i<n+2;i++)
scanf("%d",&t[i]);
for(i=0;i<n+2;i++){
for(j=0;j<(n+2)-i-1;j++)
{ if(t[j]>t[j+1]) {
temp=t[j];
t[j]=t[j+1];
t[j+1]=temp; } } }
for(i=0;i<n+2;i++)
if(t[i]==h)
j=i;k=i;
```

```
p=0;
while(t[j]!=0) {
atr[p]=t[j];
 j--;
 p++; }
 atr[p]=t[j];
 for(p=k+1;p<n+2;p++,k++)
atr[p]=t[k+1];
for(j=0;j<n+1;j++) {
 if(atr[j]>atr[j+1])
 d[j]=atr[j]-atr[j+1];
else
d[j]=atr[j+1]-atr[j];
sum+=d[j];  }
 printf("\nAverage header movements:%f",(float)sum/n); }
```

Output – SCAN-DISK.c

Lab Task – 3

C-SCAN.c

```
#include<stdio.h>
main() {
int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
system("clear");
printf("enter the no of tracks to be traveresed");
scanf("%d'",&n);
printf("enter the position of head");
scanf("%d",&h);
t[0]=0;t[1]=h;
printf("enter total tracks");
scanf("%d",&tot);
t[2]=tot-1;
printf("enter the tracks");
for(i=3;i<=n+2;i++)
scanf("%d",&t[i]);
for(i=0;i<=n+2;i++)
for(j=0;j<=(n+2)-i-1;j++)
if(t[j]>t[j+1]){
temp=t[j];
t[j]=t[j+1];
t[j+1]=temp;}
for(i=0;i<=n+2;i++)
if(t[i]==h)
j=i;
p=0;
while(t[j]!=tot-1) {
 atr[p]=t[j];
 j++;
 p++; }
 atr[p]=t[j];
 p++;
 i=0;
 while(p!=(n+3) && t[i]!=t[h]){
atr[p]=t[i];
i++;
p++;}
for(j=0;j<n+2;j++){
```

```
 if(atr[j]>atr[j+1])
 d[j]=atr[j]-atr[j+1];
else
d[j]=atr[j+1]-atr[j];
sum+=d[j];}
printf("total header movements%d",sum);
printf("avg is %f\n",(float)sum/n); }
```

Output – C-SCAN.c

Lab Task – 4

Write your observations about each simulation program above:

Observation – DISK-FCFS.c
Observation – SCAN-DISK.c
Observation – C-SCAN.c