CSL 204 OPERATING SYSTEMS LAB

MITS – VISION & MISSION

VISION

To be a centre of excellence for learning and research in engineering and technology, producing intellectually well-equipped and socially committed citizens possessing an ethical value system.

MISSION

*Offer well-balanced programme of instruction, practical exercise and opportunities in technology.

* Foster innovation and ideation of technological solutions on sustainable basis.

* Nurture a value system in students and engender in them a spirit of inquiry.

AI&DS - VISION, MISSION, PEO, PO & PSO

VISION OF AI &DS DEPARTMENT

To be recognized as a socially accountable thought leader in applications of computational technologies and a centre of excellence in solving complex and cross-disciplinary problems.

MISSION OF AI&DS DEPARTMENT

- Facilitate learning and innovative research in the frontier areas of AI & DS
- Drive students for technology development and transfer to solve socially relevant problems
- Create socially responsible professionals

PROGRAM EDUCATIONAL OBJECTIVES (PEO)

PEO1. Graduates will exhibit the ability to adapt well to the professional environment and contribute to the goals of the organization they are associated with.
PEO2. Graduates will exhibit thought leadership in solving complex problems in their respective domains.
PEO3. Graduates will be socially committed lifelong learners who strive for constantly upgrading and sharing their knowledge to nurture a healthy and open eco-system around them.

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

PO1. Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3. Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4. Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5. Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6. Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7. Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10. Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11. Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12. Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

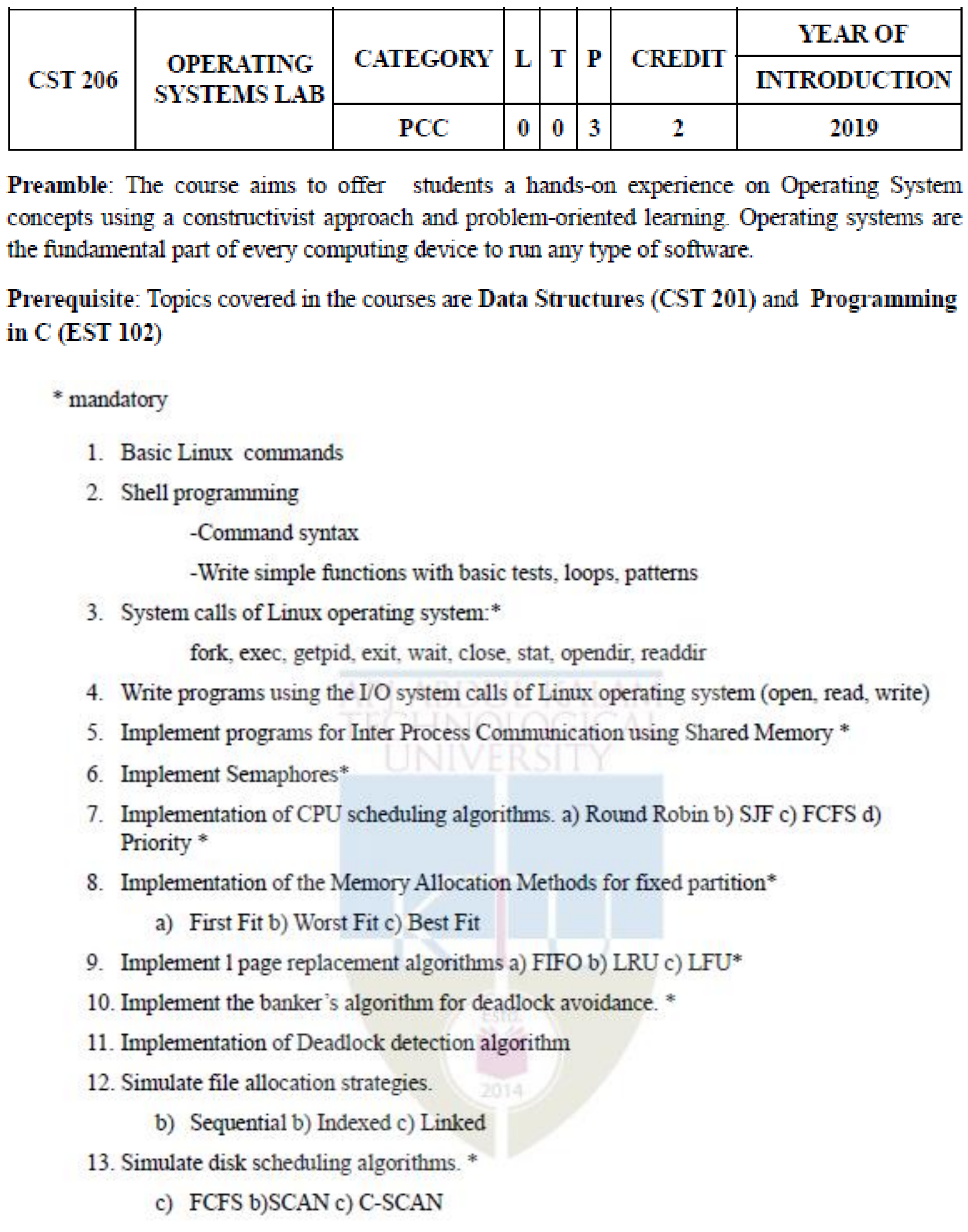
PROGRAM SPECIFIC OUTCOMES (PSOs)

Student of the AI &DS program will:

PSO1.Execute data science tasks by processing data, evaluating models, and interpreting outcomes using intelligent algorithms

PSO2. Apply concepts in hardware architecture and programming to manage high performance systems like cloud, grid, and cluster computing

SYLLABUS– KTU



CST 206	OPERATING SYSTEMS LAB	CATEGORY	L	T	P	CREDIT	YEAR OF INTRODUCTION
		PCC	0	0	3	2	2019

Preamble: The course aims to offer students a hands-on experience on Operating System concepts using a constructivist approach and problem-oriented learning. Operating systems are the fundamental part of every computing device to run any type of software.

Prerequisite: Topics covered in the courses are **Data Structures (CST 201)** and **Programming in C (EST 102)**

* mandatory

1. Basic Linux commands
2. Shell programming

   -Command syntax

   -Write simple functions with basic tests, loops, patterns
3. System calls of Linux operating system:*

   fork, exec, getpid, exit, wait, close, stat, opendir, readdir
4. Write programs using the I/O system calls of Linux operating system (open, read, write)
5. Implement programs for Inter Process Communication using Shared Memory *
6. Implement Semaphores*
7. Implementation of CPU scheduling algorithms. a) Round Robin b) SJF c) FCFS d) Priority *
8. Implementation of the Memory Allocation Methods for fixed partition*

   a) First Fit b) Worst Fit c) Best Fit
9. Implement l page replacement algorithms a) FIFO b) LRU c) LFU*
10. Implement the banker's algorithm for deadlock avoidance. *
11. Implementation of Deadlock detection algorithm
12. Simulate file allocation strategies.

    b) Sequential b) Indexed c) Linked
13. Simulate disk scheduling algorithms. *

    c) FCFS b)SCAN c) C-SCAN

CSL 204 OPERATING SYSTEMS LAB

COURSE OUTCOMES & MAPPING

COURSE OUTCOMES:

At the end of the course the student will be able to:

CO1:Apply Linux commands specific to user requirements	L3
CO2: Develop shell scripts	L3
CO3:Compare the performance of CPU scheduling algorithms.	L4
CO4:Simulate operating system concepts using C programming	L3
CO5: Utilize one's ability as an individual for the development of effective communication, practical skill and document design.	L3

CO-PO MAPPING:

COs	PROGRAM OUTCOMES (PO)											
	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	3	2	-	-	-	-	-	-	-	-	-	3
CO 2	3	-	2	-	-	-	-	-	-	-	-	-
CO 3	3	3	2	1	-	-	-	-	-	-	-	-
CO 4	3	2	2	-	-	-	-	-	-	-	-	-
CO 5	-	-	-	-	-	-	-	-	3	3	-	-

CO-PSO MAPPING:

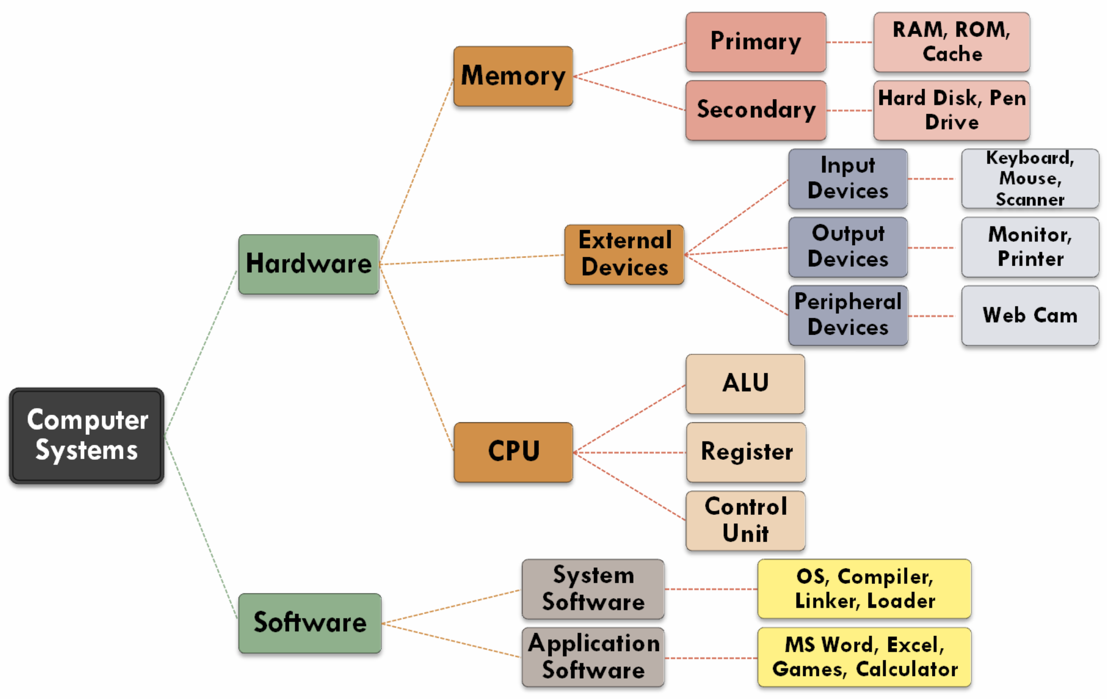
COs	PROGRAM SPECIFIC OUTCOMES (PSO)	
	PSO 1	PSO 2
CO 1	-	3
CO 2	3	3
CO 3	3	2
CO 4	3	2
CO 5	-	3

LIST OF EXPERIMENTS

SL. NO.	EXPERIMENT DETAILS	COs
1	Familiarization of basic Linux commands.	CO1,CO5
2	Shell programming	CO2,CO5
3	Study of Linux System Calls-I i) Implement programs to familiarize the following system calls: fork, exec, getpid, exit, wait, open, close, opendir,readdir.	CO1,CO5
4	Implement programs for Inter Process Communication using Shared Memory.	CO4,CO5
5	Simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time. a) FCFS b) Priority c) Round Robin (pre-emptive) d) SJF	CO3,CO5
6	Implement semaphore operations	CO4,CO5
7	Implement the banker's algorithm for deadlock avoidance.	
8	Simulate the following contiguous memory allocation techniques a)Worst-fit b) Best-fit c) First-fit	CO4,CO5
9	Simulate the following page replacement algorithms a) FIFO b)LRU c) LFU	CO4,CO5
10	Simulate the following disk scheduling algorithms. a) FCFS b)SCAN c) C-SCAN	CO4,CO5
OPEN ENDED QUESTIONS		
11	Implement the producer-consumer problem using semaphores.	CO4,CO5
ADDITIONAL EXPERIMENT		
12	Simulate the following file allocation strategies a) Sequential b) Linked c) Indexed	CO4,CO5

INTRODUCTION TO OPERATING SYSTEM

FUNCTIONAL UNITS OF A COMPUTER:



COMPUTER SOFTWARE:

Software is a set of programs that can accomplish a specific task.Without its software, a computer is just a useless lump of metal. With its software, a computer can store, process, and retrieve information. Computer software can be divided into two kinds:

- System programs: Manage the operation of the computer itself.
- Application programs: Perform the user tasks.

SOFTWARE CATEGORIES:

Application Software:

It is developed to solve a specific problem for the user.

Examples:

- Editors: Notepad, Wordpad etc,
- Spreadsheet packages: Microsoft Excel, Lotus 123
- Database Applications: MS Access, Oracle, MS SQL, MySQL

System Software:

It provides a convenient environment for program development and execution and acts as an interface between the hardware and the applications.

Examples:

- **Operating System, Compiler, Assembler, Interpreter, Linker, Loader**

COMPUTER SYSTEM COMPONENTS:

- **Hardware** – Provides basic computing resources
  - CPU, memory, I/O devices
- **Operating system** – Controls and coordinates the use of the hardware among the various application programs for the various users.
- **Programming & Applications software** – Define the ways in which the system resources are used to solve the computing problems of the users
  - Compilers, database systems, games, business programs etc
- **Users** – People, machines, other computers.

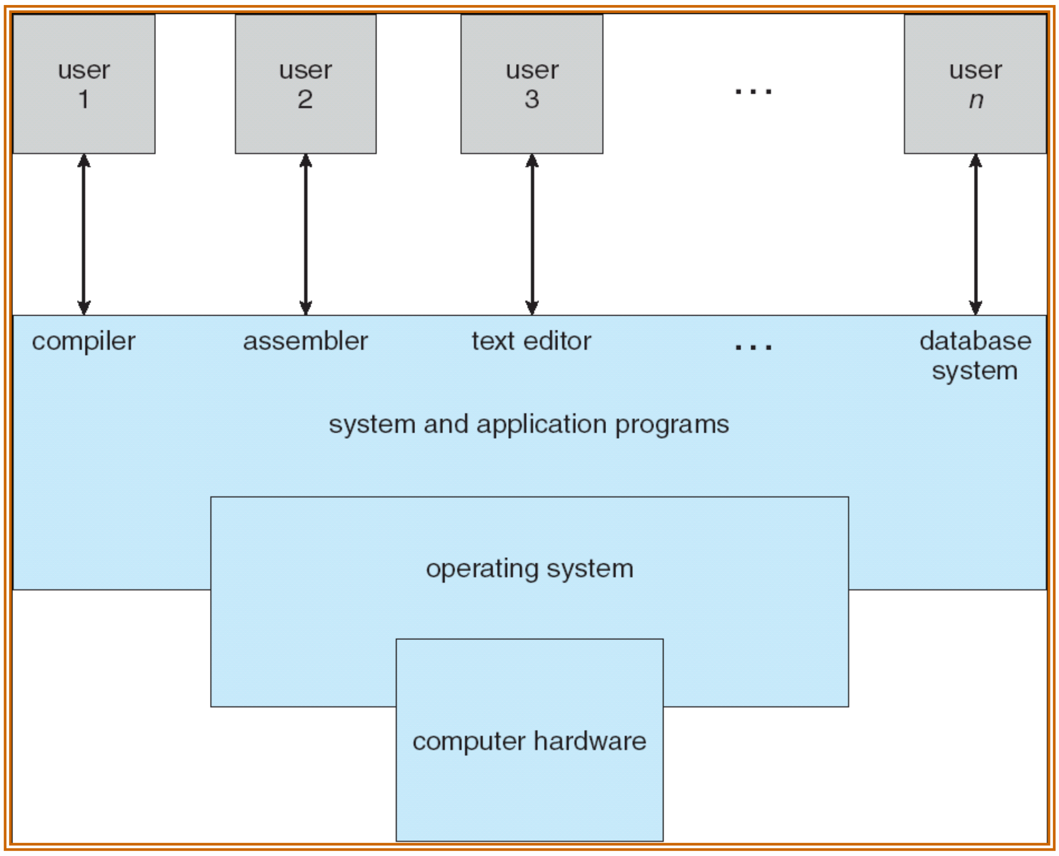


Fig: Components of a Computer System

OPERATING SYSTEM:

An operating system is software that manages the computer hardware.It is a vital component of the system software in a computer system.It serves as a platform for other software to run on a computer and acts as an intermediary between the user of a computer and the computer hardware.

- Shield programmers from the complexity of the hardware.
- Manages all the devices/resources and provide user programs with a simpler interface to different types of hardware.
- Increase Portability by minimizing machine specific code

The purpose of OS is to provide an environment in which a user can execute programs in a convenient and efficient manner.

OS Examples:

Non Proprietary - Unix, Minix

- Linux – Fedora, Debian
- Free BSD

Disk Os

- DOS
- PC-DOS
- MS-DOS

Proprietary

- Windows XP
- Windows NT / Server 2008
- Windows 7 / 8 / 10
- Mac OS
- Mainframes

Embedded OS for PDA and Mobiles

- Palm Os
- Symbian Os
- Windows Ce
- Blackberry Os
- Android
- Windows Mobile
- Apple Ios

Web OS

- Google Chrome Os
- Glide Os

Network OS

- Novel's Netware

FUNCTIONS OF OS:

i) Process Management:

A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.Process needs resources to accomplish its task - CPU, memory, I/O, files and initialization data. Process termination requires reclaim of any reusable resources. Single-threaded process has one program counter specifying location of next instruction to execute. Here, process executes instructions sequentially, one at a time, until completion. Multi-threaded process has one program counter per thread. Typically, system has many processes, some user, some operating system running concurrently on one or more CPUs. Concurrency is obtained by multiplexing the CPUs among the processes / threads.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**.

An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as traffic controller.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

ii) Memory Management:

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory.

An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

iii) Storage Management:

OS provides uniform, logical view of information storage. It abstracts physical properties to logical storage unit - file. Each medium is controlled by device (i.e., disk drive, tape drive). Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

File-System management: Files are usually organized into directories. Access control on most systems to determine who can access what

OS activities include

- Creating and deleting files and directories
- Primitives to manipulate files and directories
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

iv) Mass-Storage Management:

Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time. Proper management is of central importance. Entire speed of computer operation hinges on disk subsystem and its algorithms.

OS activities include

- Free-space management
- Storage allocation
- Disk scheduling

Some storage need not be fast. Tertiary storage includes optical storage, magnetic tape. Still it must be managed – by OS or applications - varies between WORM (write-once, read-many-times) and RW (read-write)

v) I/O Subsystem:

One purpose of OS is to hide peculiarities of hardware devices from the user. I/O subsystem is responsible for:

- Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
- General device-driver interface
- Drivers for specific hardware devices

vi) Protection and Security:

Protection refers to any mechanism for controlling access of processes or users to resources defined by the OS. Security is the defense of the system against internal and external attacks. There are a huge range of examples, including denial-of-service, worms, viruses, identity theft, theft of service.Systems generally first distinguish among users, to determine who can do what

- User identities (user IDs, security IDs) include name and associated number, one per user
- User ID then associated with all files, processes of that user to determine access control
- Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
- Privilege escalation allows user to change to effective ID with more rights

Exp No: 1

BASIC LINUX COMMANDS

PROBLEM DEFINITION:

To familiarize with Linux commands for directory operations, displaying directory structure in tree format etc.

THEORETICAL BACKGROUND:

ABOUT LINUX

Linux refers to the family of Unix-like computer operating systems using the Linux kernel. Linux can be installed on a wide variety of computer hardware, ranging from mobile phones, tablet computers and video game consoles, to mainframes andsupercomputers. Linux is a leading server operating system, and runs the 10 fastest supercomputers in the world. Use of Linux by end-users or consumers has increased in recent years, partly owing to the popular Ubuntu, Fedora, and open SUSE distributions and the emergence of net books with pre- installed Linux systems and smart phones running embedded Linux.

The development of Linux is one of the most prominent examples of free and open source software collaboration; typically all the underlying source code can be used, freely modified, and redistributed, both commercially and non-commercially, by anyone under licenses such as the GNU General Public License. Typically Linux is packaged in a format known as a Linux distribution for desktop and server use. Linux distributions includethe Linux kernel and all of the supporting software required to run a complete system,sucha as utilities and libraries, the X Window System, the GNOME and KDE desktopenvironments, and the Apache HTTP Server. The name "Linux" comes from the Linux kernel,originallywritten In 1991by
Linus Torvalds. The main supporting user space system tools and libraries from the GNU Project are the basis for the Free Software Foundation's preferred name GNU/Linux.

BASIC FEATURES

Following are some of the important features of Linux Operating System.

- **Portable** − Portability means software can works on different types of hardware in same way. Linux kernel and application programs support their installation on any kind of hardware platform.
- **Open Source** − Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** − Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.

- **Multiprogramming** − Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** − Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Linux Distributions List

- There are on an average six hundred Linux distributors providing different features. Here, we'll discuss about some of the popular Linux distros today.

  1) Ubuntu: It came into existence in 2004 by Canonical and quickly became popular. Canonical wants Ubuntu to be used as easy graphical Linux desktop without the use of command line. It is the most well known Linux distribution. Ubuntu is a next version of Debian and easy to use for newbie. It comes with lots of pre-installed apps and easy to use repositories libraries. Earlier, Ubuntu uses GNOME2 desktop environment but now it has developed its own unity desktop environment. It releases every six months and currently working to expand to run on tablets and smart phones.

  2) Linux Mint: Mint is based on Ubuntu and uses its repository software so some packages are common in both.Earlier it was an alternative of Ubuntu because media codecs and proprietary software are included in mint but was absent in Ubuntu. But now it has its own popularity and it uses cinnamon and mate desktop instead of Ubuntu's unity desktop environment.

3) Debian: Debian has its existence since 1993 and releases its versions much slowly then Ubuntu and mint.This makes it one of the most stable Linux distributor.Ubuntu is based on Debian and was founded to improve the core bits of Debian more quickly and make it more user friendly. Every release name of Debian is based on the name of the movie Toy Story.

4) Red Hat Enterprise / CentOS: Red hat is a commercial Linux distributor. These products are red hat enterprise Linux (RHEL) and Fedora which are freely available. RHEL is well tested before release and supported till seven years after the release, whereas, fedora provides faster update and without any support. Red hat uses trademark law to prevent their software from being redistributed. CentOS is a community project that uses red hat enterprise Linux code but removes all its trademark and make it freely available. In other words, it is a free version of RHEL and provides a stable platform for a long time.

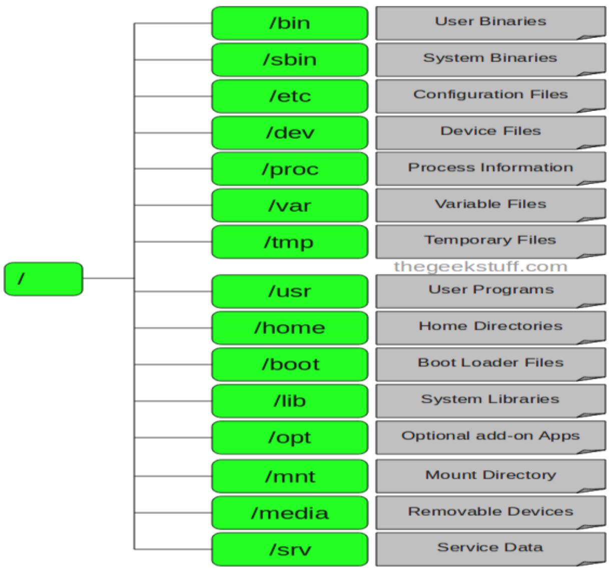
5) Fedora: It is a project that mainly focuses on free software and provides latest version of software. It doesn't make its own desktop environment but used 'upstream' software. By default it has GNOME3 desktop environment. It is less stable but provides the latest stuff.

Choosing a Linux Distribution

Distribution	**When To Use**
Ubuntu	It works like Mac OS and easy to use.
Linux mint	It works like windows and should be use by new comers.
Debian	It provides stability but not recommended to a new user.
Fedora	If you want to use red hat and latest software.
Red hat enterprise	To be used commercially.
CentOS	If you want to use red hat but without its trademark.
OpenSUSE	It works same as Fedora but slightly older and more stable.
Arch Linux	It is not for the beginners because every package has to be installed by yourself

LINUX FILE SYSTEM HIERARCHY

A file system is the way how system contains and stores all your files like your documents, games, programs, music, videos etc.In Linux world, the same theory holds true you have directories andfolders in which you arrange your files .**Linux Directory Structure** is totally different from windows. Linux Directory Structure is as shown in the figure.



1. / – Root

- The Forward Slash (/) called root in Linux directory structure
- Every single file and directory starts from the root directory.
- Only root user has write privilege under this directory.
- Please note that /root is root user's home directory, which is not same as /.

2 . /bin – User Binaries

- Contains binary executables.
- Common linux commands you need to use in single-user modes are located under this directory.
- Commands used by all the users of the system are located here.
- For example: ps, ls, ping, grep, cp,shell

3. /sbin – System Binaries

- Just like /bin, /sbin also contains binary executables.
- But, the linux commands located under this directory are used typically by system administrator, for system maintenance purpose.
- For example: iptables, reboot, fdisk, ifconfig, swapon

4. /etc – Configuration Files

- Contains configuration files required by all programs.

- This also contains startup and shutdown shell scripts used to start/stop individual programs.
- For example: /etc/resolv.conf, /etc/logrotate.conf

5. /dev – Device Files

- Contains device files.
- Devices and hardwares are treated as files that are available to a Linux system.If you attached floppy drive, you will find it under this path /dev/fd0, hard drive will be accessible under this location /dev/had (first IDE hard drive), CD drive located under /dev/cdrom drive, and so on.
- For example: /dev/tty1, /dev/usbmon0

6. /proc – Process Information

- Contains information about system process.
- This is a pseudo filesystem contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid.
- This is a virtual file system with text information about system resources. For example: /proc/uptime

7. /var – Variable Files

- var stands for variable files.
- Content of the files that are expected to grow can be found under this directory.
- System and program log files
- This includes — system log files (/var/log); packages and database files (/var/lib); emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed across reboots (/var/tmp);

8. /tmp – Temporary Files

- Directory that contains temporary files created by system and users.
- Files under this directory are deleted when system is rebooted.

9. /usr – User Programs

- This is one of the most significant directories in the system as it contains all the user binaries, their documentation, libraries, header files, etc.
- Contains binaries, libraries, documentation, and source-code for second level programs.
- /usr/bin contains binary files for user programs. If you can’t find a user binary under /bin, look under /usr/bin. For example: at, awk, cc, less, scp.
- /usr/sbin contains binary files for system administrators. If you can’t find a system binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel.

- /usr/lib contains libraries for /usr/bin and /usr/sbin.
- /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2

10. /home – Home Directories

- This directory contains user’s personal files and folders. Every user has their directory under /home and they allowed saving and deleting files in their appropriate directory. This directory is equivalent to windows â C:\Document and Setting\ folder.
- Home directories for all users to store their personal files.
- For example: /home/john, /home/nikita

11. /boot – Boot Loader Files

- Contains boot loader related files.
- Kernel initrd, vmlinux, grub files are located under /boot
- For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic

12. /lib – System Libraries

- Contains library files that supports the binaries located under /bin and /sbin
- Library filenames are either ld* or lib*.so.*
  For example: ld-2.11.1.so, libncurses.so.5.7

13. /opt – Optional add-on Applications

- opt stands for optional.
- Contains add-on applications from individual vendors.
- add-on applications should be installed under either /opt/ or /opt/ sub-directory.

14. /mnt – Mount Directory

- /mnt directory is contains mount points. Some physical storage hardware and devices like the hard disk drives, floppies, CD-ROM’s must be attached to some directory in the file system tree before they can be accessed. This attaching is called mounting, and the directory where the device is attached is called the mount point. The /mnt directory contains mount point for these devices, like /mnt/cdrom for CD-ROM, /mnt/floppy for floppy drive, and so on. However, you can use other directory as mount point instead of using /mnt.
- Temporary mount directory where sysadmins can mount filesystems.

15. /media – Removable Media Devices

- Temporary mount directory for removable devices.
- For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer

16. /srv – Service Data

- srv stands for service.
- Contains server specific services related data.
- For example, /srv/cvs contains CVS related data

BASIC COMMANDS

Name: date

Syntax: date

Description: To print and set system date and time

Name: time

Syntax: date

Description: Displays current time and date. If you are interested only in time, you can use 'date +%T'

Name: cal

Syntax:cal

Description: Displays the calendar of the current month

Name: whatis

Syntax: whatis <command>

Description: This command gives a one line description about the command. It can be used as a quick reference for any command.

Name: whoami

Syntax: whoami

Description:This command reveals the user who is currently logged in.

Name: clear

Syntax:clear

Description:This command clears the screen

Name: man

Syntax: man<command>

Description: To see a command's manual page, man command is used.

Files and Directory related commands
Directory operations

Name: cd

Syntax: cd [directory]

Description: The current working directory to the directory specified by "directory".

Example: enter the directory / usr / bin /: cd / usr / bin

Name: ls

Syntax: ls [options] [pathname-list]

Description: display the file name within the directory and file name specified in the "pathname-list"

Example: List all names in the current working directory is s at the beginning of the file: ls s *

Name: pwd

Syntax: pwd

Description: Displays the absolute path of the current directory.

Name: mkdir

Syntax: mkdir [options] dirName

Description: create name is dirName subdirectory.

Example: In the working directory, create a subdirectory named AA: mkdir AA

Name: rmdir

Syntax: rmdir [-p] dirName

Description: delete empty directories.

Example: to delete the working directory, subdirectory named AA: rmdir AA 2 file operations

Name: cp

Syntax: cp [options] file1 file2

Description: Copy the file file1 to file2. Common options:-r copy the entire directory Example: aaa copy (existing), and named bbb: cpaaabbb

Name: mv

Syntax: mv [options] source ... directory

Description: Rename the file, or the number of files to another directory. Example: aaa renamed as bbb: mv aaabbb

Name: rm

Syntax: rm [options] name...

Description: delete files and directories. Commonly used options:-f to force delete files

Example: Remove all but the suffix named c file rm *. C

Name: cat

Syntax: cat [options] [file-list]

Description: standard output connection, display a list of files in the file-list file

Example 1: Displays the contents of file1 and file2 - cat file1 file2

Example 2: file1 and file2 merged into file3 - cat file1 file2> file3

Name: more

Syntax: more [options] [file-list]

Description: standard output is connected to the paging file in the file list file-list

Example: paging file AAA more AAA

Name: head

Syntax: head [options] [file-list]

Description: Display the initial part of the file in the list of files in the file-list, the default display 10 lines;

Example: the initial part of the file AAA head AAA

Name: tail

Syntax: tail [options] [file-list]

Description: Displays the tail of the list of files in the file-list file; default display 10 lines;

Example: tail file AAA tail AAA

Name: chmod

Syntax: chmod [option] mode file-list

Description: read, write, or execute permissions change or set the parameters in the file-list

Example: Add file job executable permissions chmod + x job

Name: tar

Syntax: chmod [option] [files]

Description: The backup file. Can be used to create a backup file or restore a backup file.

Example 1: a backup test directory the file named test.tar.gz, executable commands: tar-zcvf test.tar.gz test

Name: echo

Syntax: echo $ variable

Description: Displays the value of the variable variable.

Example 1: Display the current user's PATH value echo $ PATH

Name: ps

Syntax: $ ps [options]

Description: The active process is used to view the current system

Example 1: display all current processes ps-aux

Name: kill

Syntax: $ kill [-signal] pid

Description: terminates the specified process

Example 1: the process of termination of 1511 kills 1511

Name: file

Syntax:file<pathname>

Description: This command is used to find the type of the file.

Name: sort

Syntax: sort [OPTIONS]

Description: The Linux sort command can be used to sort the contents of a file in a number of ways. By default, the Linux sort command sorts the contents in alphabetical order depending on the first letter in each line. For example, the sort /etc/passwd command would sort all users by username.

Important options of the sort are

•-b	(Ignores spaces at beginning of the line)
•-d	(Uses dictionary sort order and ignores the punctuation)
•-f	(Ignores caps)
•-i	(Ignores nonprinting control characters)
•-m	(Merges two or more input files into one sorted output)
•-r	(Sorts in reverse order)
• -u	(If line is duplicated only display once)

Name:wc

Syntax:wc[OPTIONS]

Description: The Linux wc (word count) command, can return the number of lines, words, and characters in a file. Important options of the Linux wc command are

•-c	(Print the byte counts)
•-m	(Print the character counts)
•-l	(Print the new line counts)
• -w	(Print the word counts)

CONCLUSION:

Familiarized with the basic linux commands.

Exp No: 2

SHELL PROGRAMMING

PROBLEM DEFINITION:

To develop shell scripts with simple functions with asic test,loops and patterns.

THEORETICAL BACKGROUND:

A shell is software the gives a user interface to various operating system functions and services. So in other words your interface to operating system is called shell. Shells provide a way for you to communicate with the operating system. This communication is carried out either interactively input from the keyboard is acted upon immediately or as a shell script.

The UNIX shell program interprets user commands, which are either directly entered by the user, or which can be read from a file called the shell script or shell program. (ieA shell script is a file that contains ASCII text .ie a script is a human-readable text file containing a group of commands that could also be manually executed one-by-one at the LINUX operating system command prompt. They are often kept in a file (script) because they are too numerous to type in at the command prompt each time you want to perform a specific task, or because together they form a complex computer program..)Shell scripts are interpreted, not compiled. The shell reads commands from the script line per line and searches for those commands on the system (see Section 1.2), while a compiler converts a program into machine readable form, an executable file - which may then be used in a shell script.

Apart from passing commands to the kernel, the main task of a shell is providing a user environment, which can be configured individually using shell resource configuration files.

Shell types

Just like people know different languages and dialects, your UNIX system will usually offer a variety of shell types.

In UNIX there are two major types of shells:

- The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
- The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows −

- Bourne shell (sh): the original shell still used on UNIX systems and in UNIX-related environments. This is the basic shell, a small program with few features. While this is not the standard shell, it is still available on every Linux system for compatibility with UNIX programs.
- Korn shell (ksh): sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.
- Bourne again shell (bash): the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, bash is the standard shell for common users. This shell is a so-called superset of the Bourne shell, a set of add-ons and plug-ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that

work in sh, also work in bash. However, the reverse is not always the case. All examples and exercises in this book use bash.

- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh): the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.
- TENEX/TOPS C shell (tcsh): : a superset of the common C shell, enhancing user-friendliness and speed. That is why some also call it the Turbo C shell.

The file /etc/shells gives an overview of known shells on a Linux system:

```
administrator@administrator-H310M-H-2-0:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

Your default shell is set in the /etc/passwd file

```
administrator@administrator-H310M-H-2-0:~$ echo $SHELL
/bin/bash
```

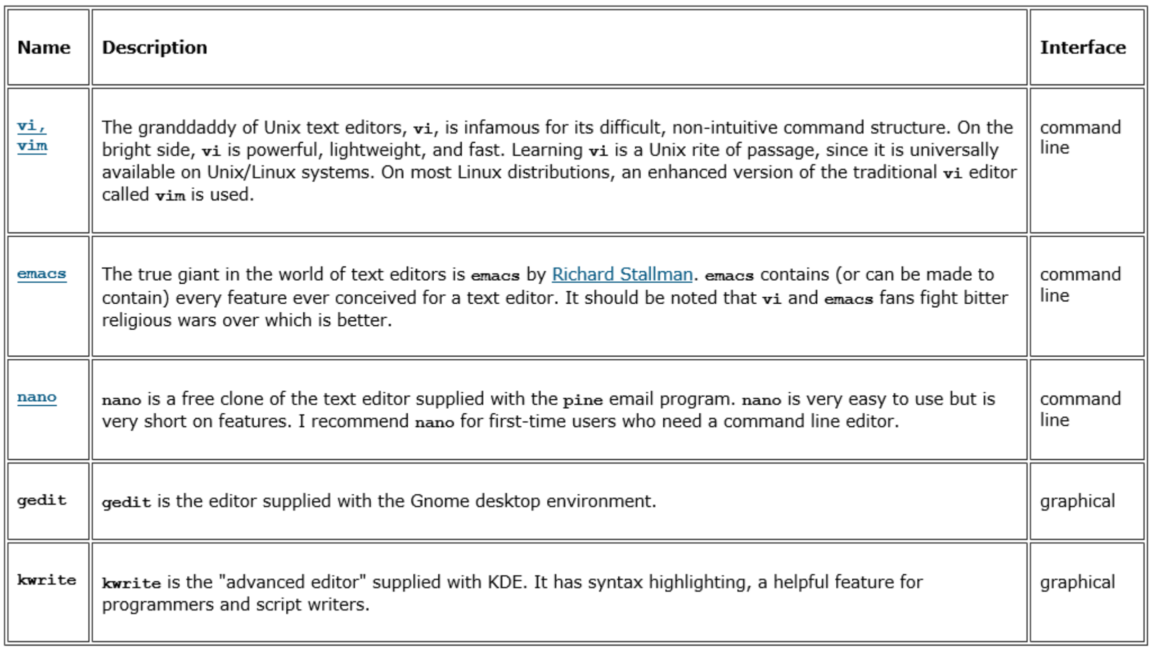
To switch from one shell to another, just enter the name of the new shell in the active terminal. The system finds the directory where the name occurs using the PATH settings, and since a shell is an executable file (program), the current shell activates it and it gets executed. A new prompt is usually shown, because each shell has its typical appearance:

Writing a script

To successfully write a shell script, you have to do three things:

1. Write a script
2. Give the shell permission to execute it
3. Put it somewhere the shell can find it

To create a shell script, you use a *text editor*. A text editor is a program, like a word processor, that reads and writes ASCII text files. There are many, many text editors available



Name	Description	Interface
vi, vim	The granddaddy of Unix text editors, `vi`, is infamous for its difficult, non-intuitive command structure. On the bright side, `vi` is powerful, lightweight, and fast. Learning `vi` is a Unix rite of passage, since it is universally available on Unix/Linux systems. On most Linux distributions, an enhanced version of the traditional `vi` editor called `vim` is used.	command line
emacs	The true giant in the world of text editors is `emacs` by Richard Stallman. `emacs` contains (or can be made to contain) every feature ever conceived for a text editor. It should be noted that `vi` and `emacs` fans fight bitter religious wars over which is better.	command line
nano	`nano` is a free clone of the text editor supplied with the `pine` email program. `nano` is very easy to use but is very short on features. I recommend `nano` for first-time users who need a command line editor.	command line
gedit	`gedit` is the editor supplied with the Gnome desktop environment.	graphical
kwrite	`kwrite` is the "advanced editor" supplied with KDE. It has syntax highlighting, a helpful feature for programmers and script writers.	graphical

for your Linux system, both for the command line environment and the GUI environment. Here is a list of some common ones:

bash shell scripting fundamentals

- **Shell definition line :**The first line in your script must be#!/bin/bash

  ... that is a # (Hash) followed by a ! (bang) followed by the path of the shell. This line lets the environment know the file is a shell script and the location of the shell.
- The name of your shell script must end with a .sh . This lets the user know that the file is a shell script. This is not compulsory but is the norm.
- Before executing your script, you should make the script executable. You do it by using the following command:

  $ chmodugo+x your_shell_script.sh
- **Shell Script Comments:** Comments, or non-command code, in a shell script begin with the # (pound) character and can be used for different purposes. They may be used to document what the overall purpose of the script is, to describe what each line of program code does, to track modifications made to the script and who made them, or for a number of other purposes. The comment character (#) may appear on any line of the script, may appear by itself on a line, or may follow actual program code. Both of these lines from the example shell script are valid program comments:

  #This is a comment on a line by itself

clear # This comment is sharing the line with the clear command

As the text in the comment lines indicate, the first comment does not share the line with anything else, and the second comment shares a line with the UNIX clear command.

- **Displaying Output:**The example shell script above shows two methods for displaying output to standard output:

  echo"TextLine1"

  print "Text Line 2"

Many people are familiar with the echo command, but the print command may be new to them. The print command is the replacement for the echo command. You should use the print command when you are writing shell scripts because it is more powerful than the echo command, and its syntax has been standardized on multiple operating systems. The print and echo commands are often used to display interactive messages to the person running the script, write informational or error messages to a log file, or to write data to a data file.

- **Exiting a Shell Script:** All shell scripts should be terminated with the exit command: **exit 0.**The word "should" is used here because a script will run successfully without including it, but it is good to get in the habit of including exit in your scripts.You will notice that the exit command is passed an argument of 0 (the number zero). Most UNIX commands and programs will return a number, calledthe return code, to the parent process.

  It is common practice to return a 0 (zero) if the command or program completed successfully, and a non-zero number if it did not. The non-zero number will in most cases be a 1 (the number one), but may be another non-zero number to give a more specific indication of why the command or Program failed. **NOTE:** If a number is not specified with the exit command, the return code of the last command executed in the script will be returned to the parent process. If 0 (zero) was not specified in the example script, the return code for the print command would be returned to the parent process.

- **How to Run a Shell Script:** Different methods can be used to run (execute) a shell script. You can run the script in the current shell, or you can spawn (create) a new shell to run the script in. If you run a script containing the exit command in the current shell, you will be logged out of the system when the script executes the exit command. Before you can run the script, you need to make the shell script file an executable by using the

UNIX chmod command. The following command will allow only the user (owner) of the script to execute it:

$ chmodu+x script1.sh

If you wanted to allow everyone (all) to execute the script, you would use this command:

$ chmoda+x script1.sh

After the script file has been made an executable with the chmod command, you can run the script in a new shell by giving the path to the script:

$ ./script1.sh

This (./) would be the path to script1 if you are in the same directory as the script. You can optionally run the shell program and pass it the script name as an argument:

$ /bin/bash script1.sh

This command also indicates that we are in the same directory as script1 because the path to the script is not specified. If you were in a different directory than the script, you could use one of the following commands to run it:

$/home/student1/script1.sh

or

$ /bin/bash /home/student1/script1.sh

Remember that all of the methods used above to run the script will spawn (create) a new shell to run it in. Running this sample script will produce the following output:

TextLine1

Text Line 2

Conditional statements

'if' Statement

The 'if' statement evaluates a condition which accompanies its command line.

Syntax:

```
ifcondition_is_true
then
  //execute commands
else
 //execute commands
fi
```

'if' condition also permits multi-way branching. That is you can evaluate more conditions if the previous condition fails.

```
ifcondition_is_true
then
  //execute commands
elifanother_condition_is_true
then
 //execute commands
else
//execute commands
fi
```

Example :

```
ifgrep "aboutlinux" thisfile.html
then
echo "Found the word in the file"
else
echo "Sorry no luck!"
fi
```

if's companion - test

test is an internal feature of the shell. 'test' evaluates the condition placed on its right, and returns either a true or false exit status. For this purpose, 'test' uses certain operators to evaluate the condition. They are as follows:

Relational Operators

- -eq - Equal to
- -lt - Less than
- -gt - Greater than
- -ge - Greater than or Equal to
- -le - Less than or Equal to

File related tests

- -f file - True if file exists and is a regular file.
- -r file - True if file exists and is readable.
- -w file - True if file exists and is writable.
- -x file - True if file exists and is executable.
- -d file - True if file exists and is a directory.
- -s file - True if file exists and has a size greater than zero.

String tests

- -nstr - True if string str is not a null string.
- -z str - True if string str is a null string.
- str1 == str2 - True if both strings are equal.
- str - True if string str is assigned a value and is not null.
- str1 != str2 - True if both strings are unequal.
- -s file - True if file exists and has a size greater than zero.

Test also permits the checking of more than one expression in the same line.

- -a - Performs the AND function
- -o - Performs the OR function

A few Example snippets of using test

```
test $d -eq 25 && echo $d
```

which means, if the value in the variable d is equal to 25, print the value. Otherwise don't print anything.

```
test $s -lt 50 &&do_something
if [ $d -eq 25 ]
then
echo $d
fi
```

In the above example, square brackets are used instead of the keyword test - which is another way of doing the same thing.

```
if [ $str1 == $str2 ]
then
   //do something
fi
if [ -n "$str1" -a -n "$str2" ]
then
echo 'Both $str1 and $str2 are not null'
fi
```

above, I have checked if both strings are not null then execute the echo command.

Things to remember while using test

1. If you are using square brackets [ ] instead of test, then care should be taken to insert a space after the [and before the].
2. test is confined to integer values only. Decimal values are simply truncated.

3. Do not use wildcards for testing string equality - they are expanded by the shell to match the files in your directory rather than the string.

Case statement

Case statement is the second conditional offered by the shell.

Syntax:

```
case expression in
pattern1) //execute commands ;;
pattern2) //execute commands ;;
...
esac
```

The keywords here are in, case and esac. The ';;' is used as option terminators. The construct also uses ')' to delimit the pattern from the action.

Example:

```
echo "Enter your option : "
read i;
case $i in
1) ls -l ;;
2) ps -aux ;;
3) date ;;
4) who ;;
5) exit
esac
```

The last case option need not have ;; but can be provided if we want.

Here is another example:

```
case `date |cut -d" " -f1` in
Mon) commands ;;
Tue) commands ;;
Wed) commands ;;
...
esac
```

Case can also match more than one pattern with each option.You can also use shell wild-cards for matching patterns.

```
...
echo "Do you wish to continue? (y/n)"
```

```
readans
case $ans in
Y|y) ;;
[Yy][Ee][Ss]) ;;
N|n) exit ;;
[Nn][Oo]) exit ;;
*) echo "Invalid command"
esac
```

In the above case, if you enter YeS, YES,yEs and any of its combinations, it will be matched.

This brings us to the end of conditional statements.

Looping Statements

while loop

while loop syntax -

```
whilecondition_is_true
do
   //execute commands
done
```

Example:

```
while [ $num -gt 100 ]
do
sleep 5
done
while :
do
   //execute some commands
Done
```

The above code implements a infinite loop. You could also write 'while true' instead of 'while :'.

Here I would like to introduce two keywords with respect to looping conditionals. They are *break* and *continue*.

break- This keyword causes control to break out of the loop.

continue - This keyword will suspend the execution of all statements following it and switches control to the top of the loop for the next iteration.

until loop

until complements while construct in the sense that the loop body here is executed repeatedly as long as the condition remains false.

Syntax:

```
until false
do
   //execute commands
Done
```

Example:

```
until [ -r myfile ]
do
sleep 5
done
```

The above code is executed repeatedly until the file myfile can be read.

for loop

for loop syntax :

```
for variable in list
do
   //execute commands
done
```

Example:

```
for x in 1 2 3 4 5
do
echo "The value of x is $x";
done
```

Here the list contains 5 numbers 1 to 5. Here is another example:

```
For var in $PATH $MAIL $HOME
do
echo $var
done
```

Suppose you have a directory full of java files and you want to compile those. You can write a script like this:

```
for file in *.java
do
```

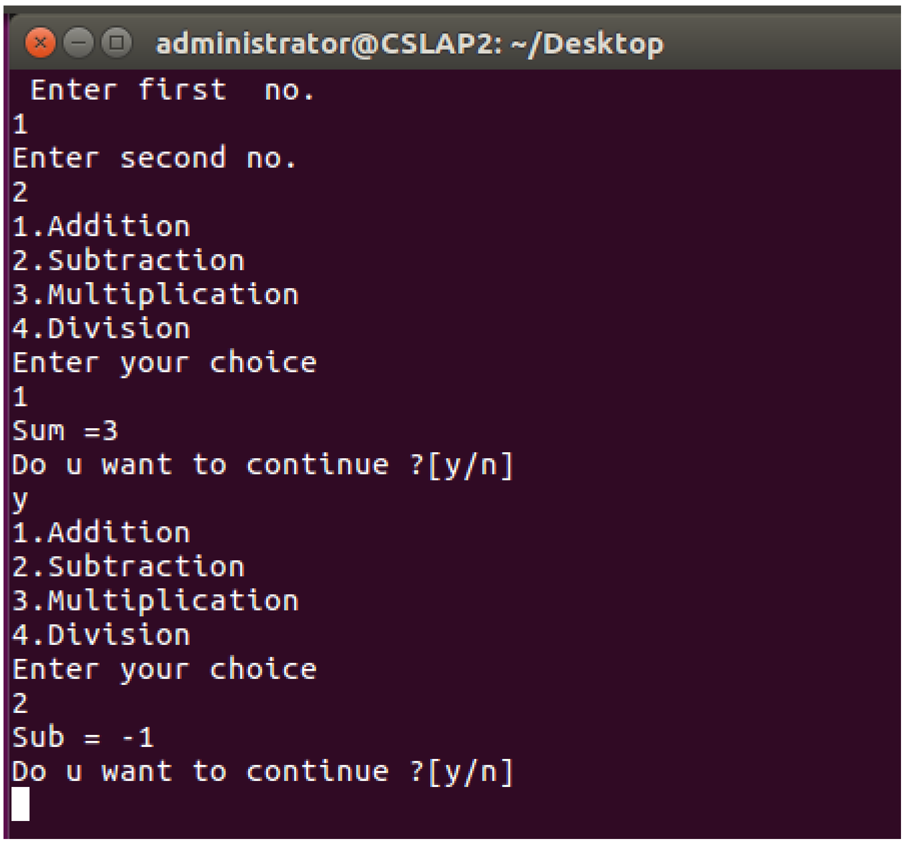
```
javac $file
done
```

The following is an example of a basic shell script:

```
#!/bin/bash      #This is a comment on a line by itself-shell definition line
clear            # This comment is sharing the line with the clear command
echo"Text Line 1"
print "Text Line 2"
exit 0
```

```
//*Shell script to simulate a menu driven calculator
clear
sum=0
i="y"
echo " Enter one no."
read n1
echo "Enter second no."
read n2
while [ $i = "y" ] do
echo "1.Addition"
echo "2.Subtraction"
echo "3.Multiplication"
echo "4.Division"
echo "Enter your choice"
read ch
case $ch in 1)sum=`expr $n1 + $n2`
echo "Sum ="$sum;;
2)sum=`expr $n1 - $n2`
echo "Sub = "$sum;;
3)sum=`expr $n1 \* $n2`
echo "Mul = "$sum;;
4)sum=`expr $n1 / $n2`
echo "Div = "$sum;;
*)echo "Invalid choice";;
esac
echo "Do u want to continue ?"
read i
if [ $i != "y" ] then
exit
fi
done
```

SAMPLE INPUT AND OUTPUT



```
administrator@CSLAP2: ~/Desktop
 Enter first  no.
1
Enter second no.
2
1.Addition
2.Subtraction
3.Multiplication
4.Division
Enter your choice
1
Sum =3
Do u want to continue ?[y/n]
y
1.Addition
2.Subtraction
3.Multiplication
4.Division
Enter your choice
2
Sub = -1
Do u want to continue ?[y/n]
```

CONCLUSION: Familiarized with the fundamentals of shell scripting**.**

Exp No: 3

STUDY OF LINUX SYSTEM CALLS

PROBLEM DEFINITION:

I)Implement a program to create a child process using system call fork() and to demonstrate the system calls getpid() and getppid(). Explore different ways of using fork() call.

II)Implement a program read all text files from a directory

III) Implement a program to list all files in a directory/opening a directory

THEORETICAL BACKGROUND:

UNIX PROCESS:

Process is an entity that executes a given piece of code, has its own execution stack, its own set of memory pages, its own file descriptors table, and a unique process ID. A unique process ID is allocated to each process when it is created . The data type of process id is pid_t. The pid_t data type is a signed integer type which is capable of representing a process ID.

Function: **pid_t getpid (void)** : The getpid function returns the process ID of the current process.

Function: **pid_t getppid (void)**: The getppid function returns the process ID of the parent of the current process. (this is also known as the parent process ID).

Process ID(PID): Every process has a unique process ID or PID. The PID is an integer typically in the range 0 through 30000.The kernel assigns the PID when a new process is created and a process can obtain its PID using the getpid system call.

Syntax: int getpid();

The process with process ID 1 is a special process called the init process. Process ID 0 is also a special kernel process termed either the swapper or the scheduler. On virtual memory implementations of Unix the process with process ID 2 is typically a kernel process termed as page daemon

Parent Process ID(PPID): Every process has a parent process and a corresponding parent process ID. The kernel assigns the aprent process ID when a new process is created and a process can obtain its value using the getppid system call

Syntax: int getppid();

SYSTEM CALL

A system call is a direct entry point through which an active process can obtain services from the kernel. The system calls are specific routines in the operating system kernel that are directly accessible to application programs and are used to manage the file system, control processes, and to provide interprocess communication.

GENERAL CLASS	**SPECIFIC CLASS**	**SYSTEM CALL**
File Structure	Creating a Channel	creat()
Related Calls		open()
		close()
	Input/Output	read()
		write()
	Random Access	lseek()

Process Related Calls	Process Creation and Termination	execve()
		fork()
		wait()

File Structure Related System Calls either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes. The kernel presents and accepts data from the channel as a process reads and writes that channel.

All input and output operations start by opening a file using eitherthe "creat()" or "open()" system calls. These calls return a file descriptor that identifies the I/O channel. File descriptors 0, 1, and 2 refer to standard input, standard output, and standard error files respectively.

creat()

The prototype for the creat() system call is:

```
int creat(file_name, mode)
char *file_name;
int mode;
```

where file_name is pointer to a null terminated character string thatnames the file and mode defines the file's access permissions.

open()

The prototype for the open() system call is:

```
#include <fcntl.h>
int open(file_name, option_flags [, mode])
char *file_name;
int option_flags, mode;
```

where file_name is a pointer to the character string that names thefile, option_flags represent the type of channel, and mode defines thefile's access permissions if the file is being created.

close()

To close a channel, use the close() system call. The prototype for theclose() system call is:

```
int close(file_descriptor)
int file_descriptor;
```

where file_descriptor identifies a currently open channel.

lseek()

The prototype for lseek() is:

```
long lseek(file_descriptor, offset, whence)
```

int file_descriptor;

long offset;

int whence;

where file_descriptor identifies the I/O channel and offset and whencework together to describe how to change the file pointer according tothe following table:

whence	**new position**
0	offset bytes into the file
1	current position in the file plus offset
2	current end-of-file position plus offset

Process Related System Calls

Four system calls are provided for creating a process, ending a process, and waiting for a process to complete. These system calls are fork(), the "exec" family, wait(), and exit(). The UNIX system calls that transform a executable binary file into a process are the "exec" family of system calls.

fork()

To create a new process, you must use the fork() system call. The prototype forthe fork() system call is:

int fork()

fork() causes the UNIX system to create a new process, called the "child process", with a new process ID. The contents of the child process are identical to the contents of the parent process.

wait()

wait() forces the parent to suspend execution until the child is finished. wait() returns the process ID of a child process that finished.If the child finishes before the parent gets around to calling wait(),then when wait() is called by the parent, it will return immediately with the child's process ID. (It is possible to have more than one child process by simply calling fork() more than once.).

The prototype for the wait() system call is:

int wait(status)

int *status;

where status is a pointer to an integer where the UNIX system storesthe value returned by the child process. wait() returns the process ID of theprocess that ended.

execve()

In computing, exec is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable. This act is also referred to as an overlay. As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program. Linux kernel has the corresponding system call named "execve"

The fork() System Call

The fork() system call is the basic way to create a new process. It is declared in the header file unistd.h.

pid_t fork (void)

fork() causes the UNIX system to create a new process, called the 'child process', with a new process ID. The contents of the child process are identical to the contents of the parent process. If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process. If process creation failed, fork returns a value of -1 in the parent process. The child process and the parent process run in separate memory spaces.

execlp( ): Used after the fork() system call by one of the two processes to replace the process memory space with a new program. It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution.The child process overlays its address space with the UNIX command /bin/ls using the execlp system call.

Syntax : execlp( )

wait( ) The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

Syntax : wait(NULL)

exit( ) A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

Syntax: exit(0)

ALGORITHM:

Step 1. Start the process

Step 2. Declare a variable 'p' to be shared by both child and parent.

Step 3. Create a child process using fork system call and get its return value to 'p'.

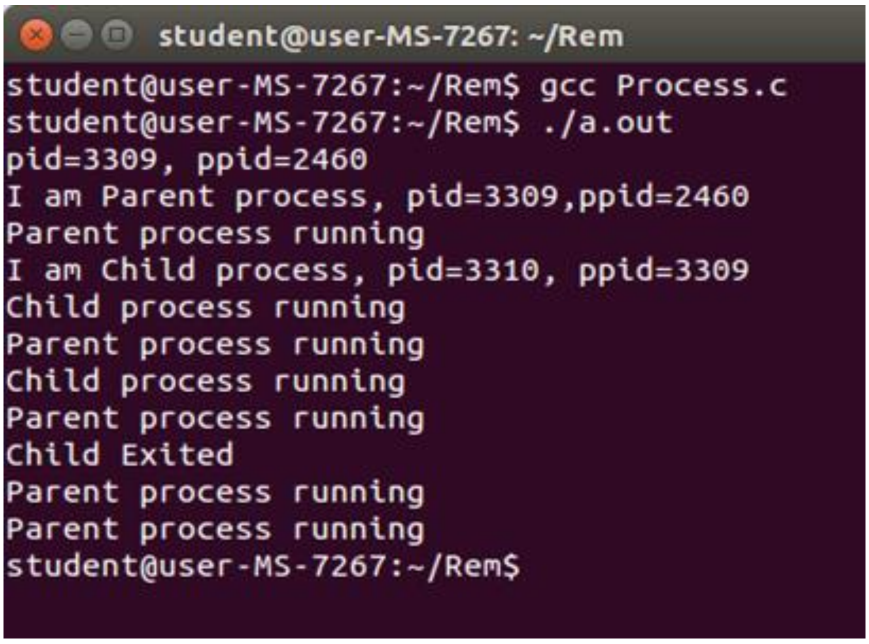
```
Step 4.  If return value is -1 then
           i.   Print "Process creation unsuccessfull"
          ii.   Terminate using exit system call.
Step 5.  If return value is 0 then
           i.   Print "Child process"
          ii.   Print process id of the child using getpid system call
         iii.   Print process id of the parent using getppid system call
Step 6.  Otherwise
           i.   Print "Parent process"
          ii.   Print process id of the parent using getpid system call
         iii.   Print process id of the shell using getppid system call.
Step 7.  Stop the process
```

PROGRAM DEVELOPMENT:

```
//*Program to create child process and to print process ID*/
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
main()
{
   int i, p;
         printf("\n pid=%d, ppid=%d",getpid(),getppid());

   //Creating child process using fork
   p = fork();

   //Inside child, fork returns zero
         if(p == 0)
         {
```

```
    for(i=0;i<3;i++)
            {       printf("\nChild process running, pid=%d, ppid=%d",getpid(),getppid());
                    sleep(2);
            }
            printf("\nChild Exited");
            sleep(2);
      }
      //Inside parent, fork returns the child process id
      else if(p > 0)
      {
    for(i=0;i<6;i++)
            {       printf("\nParent process running, pid=%d,ppid=%d",getpid(),getppid());
                    sleep(2);
            }
      }
      else
        printf("\nError in creating child process");
}
```

SAMPLE INPUT AND OUTPUT:



```
student@user-MS-7267: ~/Rem
student@user-MS-7267:~/Rem$ gcc Process.c
student@user-MS-7267:~/Rem$ ./a.out
pid=3309, ppid=2460
I am Parent process, pid=3309,ppid=2460
Parent process running
I am Child process, pid=3310, ppid=3309
Child process running
Parent process running
Child process running
Parent process running
Child Exited
Parent process running
Parent process running
student@user-MS-7267:~/Rem$
```

ANOTHER IMPLEMENTATION:

PROBLEM DEFINITION:
To load an executable program in a child processes exec system call.

ALGORITHM:

If no. of command line arguments, then stop.

```
Step 1.  Start the process
Step 2.  Create a child process using fork system call.
Step 3.  If return value is -1 then
     i.    Print "Process creation unsuccessfull"
    ii.    Terminate using exit system call.
Step 4.  If return value is > 0 then
     i.    Suspend parent process until child completes using wait system call
    ii.    Print "Child Terminated".
   iii.    Terminate the parent process.
Step 5.  If return value is 0 then
     i.    Print "Child starts"
    ii.    Load the program in the given path into child process using exec system call.
   iii.    If return value of exec is negative then print the exception and stop.
    iv.    Terminate the child process.
Step 6.  Stop
```

PROGRAM DEVELOPMENT:

```
/* Load a program in child process - exec.c */
```

```
/*./a.out <path><cmd>     eg. ./a.out /bin/ls ls  */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
main(int argc, char*argv[])
{
   pid_t pid;
   int i;
   if (argc != 3)
   {
      printf("\nInsufficient arguments to load program");
      printf("\nUsage: ./a.out <path><cmd>\n");
      exit(-1);
   }
   switch(pid = fork())
   {
    case -1:
         printf("Fork failed");
         exit(-1);

    case 0:
         printf("Child process\n");
         i = execl(argv[1], argv[2],0);
         if (i < 0)
         {
            printf("%s program not loaded using exec system call\n", argv[2]);
            exit(-1);
         }
    default:
            wait(NULL);
    printf("Child Terminated\n"); exit(0);
   }
}
```

SAMPLE INPUT AND OUTPUT:

```
$ gcc exec.c
$ ./a.out
Insufficient arguments to load program
Usage: ./a.out <path><cmd>
$ ./a.out /bin/ls ls
```

```
Child process
cmdpipe.c consumer.c
a.out
dirlist.c       ex6a.c       ex6b.c
ex6c.c          ex6d.c       exec.c
fappend.c       fcfs.c        fcreate.c
fork.c          fread.c      hello
list            list.c       pri.c
producer.c      rr.c         simls.c
sjf.c           stat.c       wait.c
Child Terminated
$ ./a.out /bin/who
who Child process
who program not loaded using exec system
call Child Terminated
$ ./a.out /usr/bin/who
who Child process
vijai pts/0 2013-04-24 15:48 (192.168.144.1) Child Terminated.
```

* C program read all text files from a directory */

Different file system types may have different directory entries. The dirent structure defines a file system independent directory entry, which contains information common to directory entries in different file system types. A set of these structures is returned by the getdents(2) system call.

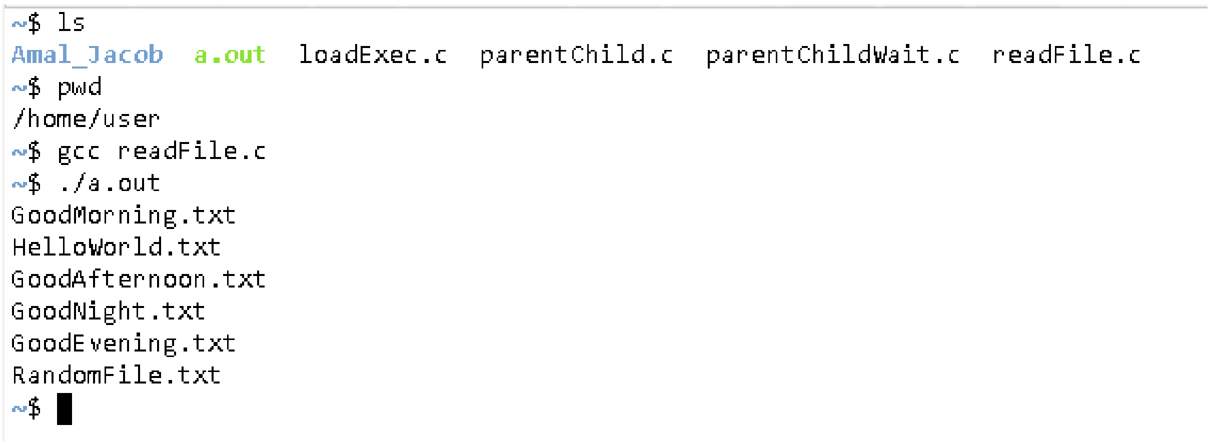
he dirent structure is defined as:

```
        #include<sys/dirent.h>

          struct dirent
          {
             long d_ino; /* inode number */
             off_t d_off; /* offset to the next dirent */
             unsigned short d_reclen; /* length of this record */
            unsigned char d_type; /* type of file */
             char d_name[256];  /* filename*/
           };
```

d_ino	…..	inode number of a file
d_offset	…..	offset of that directory entry in the actual file system directory
d_name	…..	name of the file or directory
d_reclen		record length of this entry
d_type	…..	type of file(text file, block file, directory etc)

```
//program to list all files in a directory/opening a directory
#include<stdio.h>
```

```
#include<dirent.h>
#include<stdlib.h>
void main()
{
   DIR *dirp;
   struct dirent *dp;
   if((dirp=opendir("/root/tist"))==NULL)  /* trying to open kumar directory*/
   {
     printf("\n cannot open");
     exit(1);
   }
   for(dp=readdir(dirp);dp!=NULL;dp=readdir(dirp))
   {
      if(dp->d_type==DT_REG)   /* printing only if file is a text file */
      printf("%s\n",dp->d_name);
   }
   closedir(dirp);
}
```



```
~$ ls
Amal_Jacob  a.out  loadExec.c  parentChild.c  parentChildWait.c  readFile.c
~$ pwd
/home/user
~$ gcc readFile.c
~$ ./a.out
GoodMorning.txt
HelloWorld.txt
GoodAfternoon.txt
GoodNight.txt
GoodEvening.txt
RandomFile.txt
~$ █
```

CONCLUSION:

i) Child process created and parent id and process id displayed. The child process loads a binary executable file into its address space.

ii) read all text files from a directory and displayed

iii) list all files in a directory/opening a directory and displayed

Exp No: 4

STUDY OF I/O SYSTEM CALLS OF LINUX OPERATING SYSTEM

PROBLEM DEFINITION:

To implement a program to read and display the contents of a file using I/O system calls.

THEORETICAL BACKGROUND:

A system call is a direct entry point through which an active process can obtain services from the kernel. The system calls are specific routines in the operating system kernel that are directly accessible to application programs and are used to manage the file system, control processes, and to provide interprocess communication.

GENERAL CLASS	SPECIFIC CLASS	SYSTEM CALL
File Structure Related Calls	Creating a Channel	creat()
		open()
		close()
	Input/Output	***read()***
		write()
	Random Access	lseek()
Process Related Calls	Process Creation and Termination	execve()
		fork()
		wait()

All input and output operations start by opening a file using eitherthe "creat()" or "open()" system calls. These calls return a file descriptor that identifies the I/O channel. File descriptors 0, 1, and 2 refer to standard input, standard output, and standard error files respectively.

read(), write()

prototypes are:

```
int read(file_descriptor, buffer_pointer, transfer_size)
int file_descriptor;
char *buffer_pointer;
unsigned transfer_size;
int write(file_descriptor, buffer_pointer, transfer_size)
int file_descriptor;
char *buffer_pointer;
unsigned transfer_size;
```

where file_descriptor identifies the I/O channel, buffer_pointer pointsto the area in memory where the data is stored for a read() or wherethe data is taken for a write(), and transfer_size defines the maximumnumber of characters transferred between the file and the buffer.

read() and write() return the number of bytes transferred.

```
#include<fcntl.h>
```

```
#include<stdlib.h>
int main()
{
    char c[10],d[10];
    int fd1 = open("sample.txt", O_RDONLY|O_CREAT, 0);
    printf("fd1=%d",fd1);
    read(fd1, &c,3);
    printf("c = %s\n", c);
    close(fd1);
    int fd2 = open("sample.txt", O_RDONLY, 0);
    printf("fd2=%d:",fd2);
     read(fd2, &d, 1);
    printf("d = %s\n", d);
    close(fd2);
    exit(0);
}
```

SAMPLE INPUT AND OUTPUT:

```
administrator@administrator-H310M-H-2-0:~/$ ./a.out
fd1=3c = hel
fd2=3
d = h@
```

CONCLUSION:

Read and displayed the contents of a file using I/O system calls.

Exp No: 5

INTERPROCESS COMMUNICATION USING SHARED MEMORY

PROBLEM DEFINITION:

To make use of shared memory for interprocess communication

THEORETICAL BACKGROUND:

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 process creates a shared memory segment using shmget().Once created, a shared segment can be attached to a process address space using shmat(). It can be detached using shmdt().Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. 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. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypes can be found in <sys/shm.h>.

Accessing a Shared Memory Segment

shmget() is used to obtain access to a shared memory segment. It is prototyped by: int shmget(key_t key, size_t size, int shmflg);

The first argument to shmget() is the key value (returned by a call to ftok()).. The size argument is the size in bytes of the requested shared memory. The shmflg argument specifies the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID.

Attaching and Detaching a Shared Memory Segment

shmat() and shmdt() are used to attach and detach shared memory segments. Prototypes as follows:

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);

shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr.

Algorithm:

Step 1: Start

Step 2: Generate a key using ftok function.

Step 3: Create a shared memory using shmget system call.

Step 4: Print the shared memory id.

Step 5: Attach shared memory to the process address space using shmat system call.

Step 6: Print the starting address of shared memory.

Step 7: Read a string from the keyboard.

Step 8: Copy the string to shared memory.

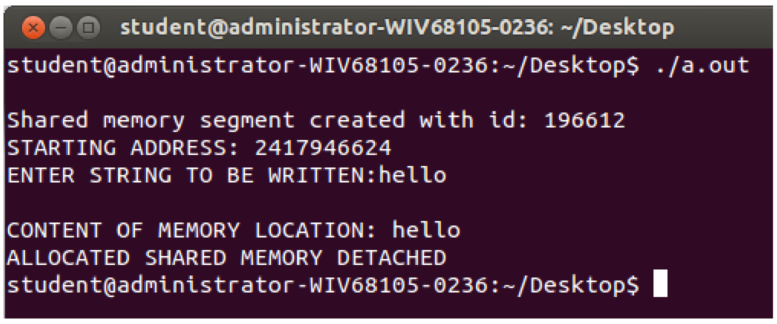
Step 9: Print the content of shared memory.

Step 10: Detach the shared memory using shmdt system call.

Program:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<string.h>
#include<stdlib.h>
main()
{
        int shmid;
        int *val,val1;
        char *bufc[100];
        key_t key=ftok ("pipe.c",'a');
        shmid = shmget(key , 40, IPC_CREAT|0777);
        if (shmid<0)
                        printf ("ERROR");
        printf("\nShared memory segment created with id=%d", shmid);

        if ((val= shmat(shmid,0,0))<0)
        {
                printf ("ERROR");
        }
                else
        {
                printf ("\nSTARTING ADDRESS: %u", val);
                printf ("\nENTER STRING TO BE WRITTEN:");
                scanf ("%s",bufc);
                strncpy (val, bufc, sizeof(bufc));
                printf ("\nCONTENT OF MEMORY LOCATION: %s",val);
                if ((val1 = shmdt (val))<0)
                {
                        printf ("ERROR");
                }
                printf ("\nALLOCATED SHARED MEMORY DETACHED");
        }
}
```

SAMPLE INPUT AND OUTPUT



```
student@administrator-WIV68105-0236:~/Desktop$ ./a.out

Shared memory segment created with id: 196612
STARTING ADDRESS: 2417946624
ENTER STRING TO BE WRITTEN:hello

CONTENT OF MEMORY LOCATION: hello
ALLOCATED SHARED MEMORY DETACHED
student@administrator-WIV68105-0236:~/Desktop$
```

CONCLUSION:

C program to implement shared memory is executed successfully and output is obtained and verified

Exp No: 6

SEMAPHORE OPERATIONS

PROBLEM DEFINITION:

To create semaphores, set its value and perform semaphore operations.

THEORETICAL BACKGROUND:

SEMAPHORE:

Semaphore is a method for Inter-Process Communication (IPC). It may be used to provide exclusive access to resources on the current machine, or to limit the number of processes that may simultaneously use a resource. i.e. it is a synchronization-tool. It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.

A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations: **wait**() and **signal**(). wait() is termed P ("to test") and signal() is termed V ("to increment").
Definition of the **wait**() operation

```
wait(S) {
   while (S <= 0)
      ; // busy wait while someone is already using
   S--;  //decrement and enter the c-s
}
```

Definition of the **signal**() operation

```
signal(S) {
   S++;  //increment and make it free for use
}
```

All modifications to the integer value of S in the wait() and signal() operations must be executed **indivisibly**. When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value. Also, in the case of wait(S), following 2 operations must be executed without interruption:

Testing of S(S<=0) and
Modification of S (S--)

Binary Semaphore: holds integer value of **0 and 1.** Value 0 means semaphore is locked.
1 means it is unlocked.

- Can solve various synchronization problems using this.

Counting Semaphore: Integer value can range over an unrestricted domain.

IMPLEMENTATION USING C:
Semaphores are counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it.

Creating a semaphore - semget():
A semaphore set is created by the **semget** system call, whose synopsis is:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflag);
```

This system call creates semaphore set and **returns an integer id for the semaphore set** or -1 in case of error.

- A key is implemented as an integer (whose size is specified by the key_t type), and a library function called ftok()used by different processes to identify the semaphore set.

  ```
  #include <sys/ipc.h>
  key_t ftok(const char *path, int id);
  ```
- The value of nsems specifies the number of semaphores in newly created semaphore set.
- The value of flag is a bit mask obtained ORing constants that specify:
  - The access permissions for the semaphore set, in the same way as for a file.

    Upon creation, the least significant 9 bits of the argument semflg define the permissions (for owner, group and others) for the semaphore set.
  - The creation mode, can be:
    - IPC_PRIVATE
    - IPC_CREAT – creates a new semaphore set if it no existing semaphore set is associated with key, or point to the semaphore set if such a a set exists.

Data of Semaphore set - semid_ds structure:

For every set of semaphores in the system, the kernel maintains the following structure of information.

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
struct semid_ds{
        struct ipc_prm sem_perm; /*operation permission struct*/
        struct sem *sem_base;    /*ptr to first semaphore in set*/
        ushort sem_nsems;        /*no of semaphores in the set*/
        time_t sem_otime;        /*time of last semop*/
        time_t sem_ctime;        /*time of last change*/
};
```

Data of each semaphore in a set - sem structure:

Internal data structure used by kernel to maintain the set of values for every member of a semaphore set.

```
struct sem
{
        ushort semval; /* semaphore value, non negative*/
        short sempid;  /*pid of last operation*/
        ushort semint;  /* no of awaiting semval>cval*/
        ushort semzcnt; /* no of awaiting semval=0*/
};
```

Setting and getting semaphore value - semctl():

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

The *semctl()* function provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional and depends upon the operation requested. If required, it is of type **union semun.**

```
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short  *array;
} arg;
```

The following semaphore control operations as specified by *cmd* are executed with respect to the semaphore specified by *semid* and *semnum.* The symbolic names for the values of *cmd* are defined by the <*sys/sem.h*> header:

GETVAL: Return the value of *semval*.

SETVAL: Set the value of *semval* to *arg.val*, where *arg* is the value of the fourth argument to *semctl()*.

IPC_RMID: Remove the semaphore-identifier specified by *semid* from the system and destroy the set of semaphores and **semid_ds** data structure associated with it.

Semaphore operations - Locking and unlocking a semaphore - semop():

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);
```

The *semop()* function is used to perform atomically a user-defined array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by the argument *semid*. i.e. *semid* is the id of the semaphore set on which the operations are to be performed. The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The argument *nsops* is the number of such structures in the array. The sops argument points to an array of type sembuf. This structure is declared in linux/sem.h as follows:

```
struct sembuf{
        unsigned short sem_num;  /* semaphore number */
        short sem_op;   /* semaphore operation */
        short sem_flg;  /* operation flags */
};
```

Each of the nsops elements in the array pointed to by sops is a structure that specifies an operation to be performed on a single semaphore. The variable *sem_op* specifies one of three semaphore operations:

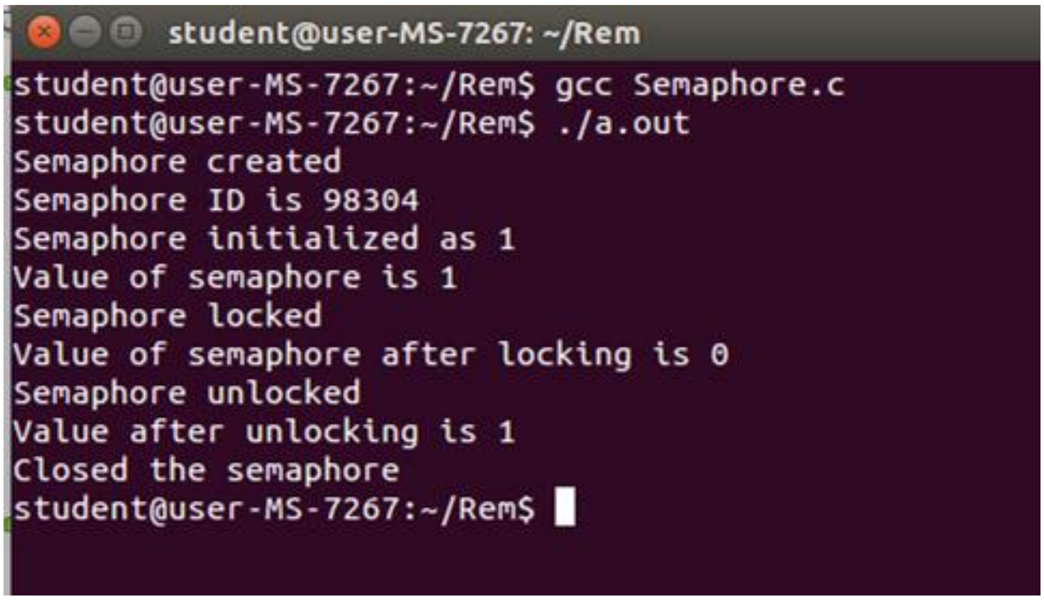
1 If *sem_op* is a positive integer, the operation adds this value to the semaphore value (*semval*).

2. If *sem_op* is less than zero the absolute value of *sem_op* is subtracted from *semval*.
3. If *sem_op* is zero, the process must have read access permission on the semaphore set.

PROGRAM DEVELOPMENT:

```
/*Program to illustrate the use of Semphores*/
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<error.h>
main()
{
        int sid, val, val1;
        //Generating key; eg.txt must exist
        key_t key = ftok("eg.txt", 'a');

   //Create variable of sembuf structure
        struct sembuf s[ ] = {0, -1, 0};

        //Creating semaphore
        if((sid = semget(key, 1, IPC_CREAT | 0666))== -1)
                printf("error");
        else
        {       printf("Semaphore created\n");
                printf("Semaphore ID is %d\n", sid);
        }
   //Setting semaphore value and getting the value
        if(semctl(sid, 0, SETVAL, 1) == 0)
                printf("Semaphore initialized as 1\n");
        if((val1 = semctl(sid, 0, GETVAL)) > 0)
                printf("Value of semaphore is %d\n", val1);

        //Locking the semaphore
        if((val = semop(sid, &s[0], 1)) == 0)
                printf("Semaphore locked\n");
        val1 = semctl(sid,0,GETVAL);
        printf("Value of semaphore after locking is %d\n",val1);

        //Unlocking the semaphore
        s[0].sem_op = 1;
        semop(sid, &s[0], 1);
        printf("Semaphore unlocked\n");
        val1 = semctl(sid, 0, GETVAL);
        printf("Value after unlocking is %d\n",val1);

        //Removing the semaphore
        if(semctl(sid, 0, IPC_RMID) == 0)
        printf("Closed the semaphore\n");
}
```

SAMPLE INPUT AND OUTPUT:



student@user-MS-7267: ~/Rem

```
student@user-MS-7267:~/Rem$ gcc Semaphore.c
student@user-MS-7267:~/Rem$ ./a.out
Semaphore created
Semaphore ID is 98304
Semaphore initialized as 1
Value of semaphore is 1
Semaphore locked
Value of semaphore after locking is 0
Semaphore unlocked
Value after unlocking is 1
Closed the semaphore
student@user-MS-7267:~/Rem$ 
```

CONCLUSION:

Semaphore created and locking-unlocking operations performed.

Exp No: 7

PROCESS SCHEDULING

PROBLEM DEFINITION:

Simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.

a) FCFS   b) Priority   c) Round Robin (pre-emptive)   d) SJF

THEORETICAL BACKGROUND:

MULTIPROGRAMMING:

In a single-processor system, only one process can run at a time, others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, thereby to maximize CPU utilization. CPU scheduling is the basis

of multi-programmed systems. By switching the CPU among processes, the OS can make the computer more productive.
When a process waits for completion of an I/O request, CPU sits idle.All this waiting time is wasted; no useful work is accomplished. With multiprogramming, OS tries to use this time productively.Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.
Maximum CPU utilization is obtained with multiprogramming.

CPU–I/O Burst Cycle:
Process execution consists of a cycle of CPU execution and I/O wait.CPU burst is followed by I/O burst. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

PROCESS SCHEDULING:
The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. Process scheduling is an essential part of a **multiprogramming** operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

TYPES OF SCHEDULING:
These are of two types of scheduling.

1. Preemptive scheduling algorithms
2. Non-Preemptive scheduling algorithms

Preemtive Scheduling algorithm: In this, the CPU can release the process even in the middle of execution. For example: the cpu executes the process p1, in the middle of execution the cpu received a request signal from process p2, then the OS compares the priorities of p1&p2. If the priority p1 is higher than the p2 then the cpu continue the execution of process p1.Otherwise the cpu preempt the process p1 and assigned to process p2.

Non-Preemtive Scheduling algorithm: In this, once the CPU assigned to a process the processor do not release until the completion of that process. The cpu will assign to some other job only after the previous job has finished.

SCHEDULING CRITERIA:
Choice of a scheduling algorithm is based on:

- **CPU Utilization:**Percentage of time that the processor is busy.
  - keep the CPU as busy as possible
  - Range from 0 to 100 percent. (40% to 90% – lightly/heavily loaded)
- **Throughput**: number of processes that complete their execution per unit time.
  - Number of jobs completed per second.

  - Less throughput for longer processes, high throughput for shorter processes.
- **Turnaround time:** amount of time to execute a particular process.
  - Time interval between the submission of process and the time of completion.

  Turn around time = waiting time in ready queue + executing time + waiting time in waiting queue for I/O.
- **Waiting time:** amount of time a process has been waiting in ready queue
  - Sum of periods spent waiting by a process in the ready queue.
  - Algorithm with least average waiting time is said to be the best algorithm.
- **Response time:** amount of time it takes from when a request was submitted until the first response is produced, not output (for interactive time-sharing environment)
  - Time duration between the submission of job and first response.

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Calculations

- **Average turn around time**: (Avg TAT)

  Turn around time = Finished time – Arrival time

  Avg TAT= Total TAT of all processes / No of processes
- **Average waiting time**:(Avg WT)

  Waiting time = Starting time – Arrival time *[Consider all instances of execution]*

  Avg WT = Total WT of all processes / No of processes
- **Average response time**: (Avg RT)

  Response time = First response – Arrival time

  Avg RT= Total RT of all processes / No of processes

SCHEDULING ALGORITHMS:

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

(a) FIRST COME FIRST SERVE (FCFS)

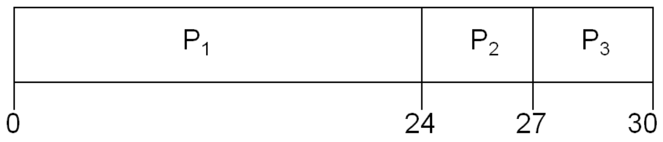
FCFS is the simplest algorithm which is**non preemptive**.Processes **gets CPU in the order they request it**. There is a single **FIFOqueue** of ready processes. When the first job enters the system, it is started immediately and allowed to run as long as it wants to. As other jobs come in, they are put onto the end of the ready queue. When the running process blocks, the next first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.

- **Pros**:
  - Easy to understand and equally easy to program.
  - It is also fair.
  - A single linked list keeps track of all ready processes.
  - Picking a process to run just requires removing one from the front of the ready queue.
  - Adding a new job or unblocked process just requires attaching it to the end of the ready queue.
- **Cons**:
  - Poor CPU performance.
  - Average waiting time may be long.

FCFS Example

Process	CPU Burst Time (milli sec)
P_1	24
P_2	3
P_3	3

Assume that processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



A **Gantt chart** is a graphical representation of the duration of tasks against the progression of time.

- Burst time:Time required by the CPU to execute that job.
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  Average WT: (0 + 24 + 27)/3 = 51/3 = **17 ms**
- Turn around time for P_1 = 24-0=24; P_2 = 27-0=27; P_3 = 30-0=30
  Average TAT: (24 + 27 + 30)/3 = 81/3 = **27 ms**

Performance:

- The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.
- **Convoy effect** or head-of-line blocking
  short process blocked behind long process - lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
- FCFS scheduling algorithm is nonpreemptive.
  troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

(b) PRIORITY SCHEDULING

A **priority number** (integer) is associated with each process.The CPU is allocated to the **process with the highest priority** (smallest integer ≡ highest priority). SJF is priority scheduling where priority is the inverse of predicted next CPU burst time. Equal-priority processes are scheduled in FCFS order. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. We assume that **low numbers represent high priority**.

Two Schemes

Priority scheduling can be either **preemptive** or **nonpreemptive**.

- When a process arrives at the ready queue, its priority is **compared** with the priority of the currently running process.
- **Preemptive**: algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
  Example - SJF: priority scheduling where priority is the predicted next CPU burst time.
- **Nonpreemptive** priority: algorithm will simply put the new process at the head of the ready queue.

Setting Priorities

Priorities can be defined either **internally** or **externally**. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. Time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst etc have been used in computing priorities.

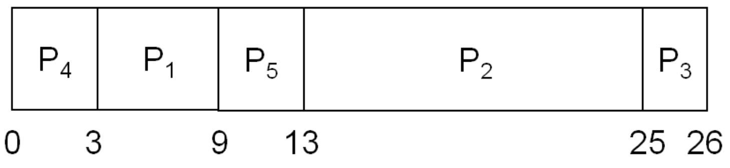
External priorities are set by criteria outside the OS. The importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Example - Priority Scheduling

Process	Burst Time	Priority
P_1	6	2
P_2	12	4
P_3	1	5
P_4	3	1

P_5	4	3

The Gantt chart is:



- Average WT = (3 + 13+ 25 + 0 + 9)/5 = 50/5 = **10 ms**
- Average TAT = (9 + 25 + 26 + 3 + 13)/5 =76/5 = **15.2 ms**
- Average RT = (3 + 13 + 25 + 0 + 9)/5 =50/5 = **10 ms**

Performance:

- Problem: **indefinite blocking** or **Starvation**
  - Low priority processes may be delayed indefinitely or may never execute at all.
- Solution: **Aging** – as time progresses, increase the priority of the process.
  - Priorities range from 127 (low) to 0 (high), gradually increase the priority of a waiting process by 1 every 15 minutes.Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. Takes 32 hours approximately to age into this level.

(C) ROUND ROBIN SCHEDULING

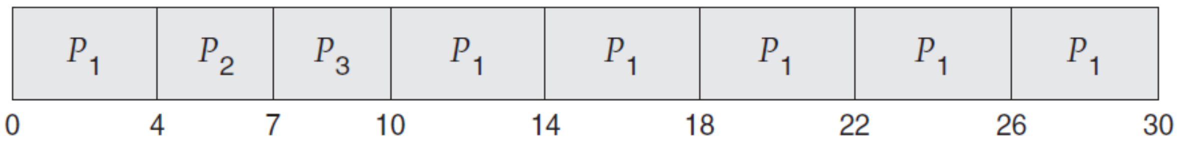
RR scheduling is preemptive, simple, fair and widely used. It is similar to FCFS, but preemption is added to switch between processes. This scheduling is designed for time sharing systems.Each process is assigned a time interval, called its **quantum** or **slice**, generally 10 to 100 milliseconds in length, for which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done.

The average waiting time under the RR policy is often long.Typically, higher average turnaround than SJF, but better response time.

Example of RR with Time Quantum = 4 ms

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:



- P1 waits for 6 milliseconds (10 - 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds.
- Thus, the average waiting time is 17/3 = 5.66 milliseconds.

Performance:

- Performance depends heavily on the size of the time quantum.
  - If the quantum is extremely large, RR policy will be the same as FCFS policy.
  - If it is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches.
- Quantum should be large compared to context switch time.
- Usually 10 milliseconds to 100 milliseconds, considering that context switch time < 10 microseconds.
  - the context-switch time is a small fraction of the quantum.
  - 80 percent of the CPU bursts should be shorter than quantum.

(d) SHORTEST-JOB-FIRST SCHEDULING (SJF)

SJF scheduling ssumes that CPU run times are known in advance. Scheduler picks the shortest job first.

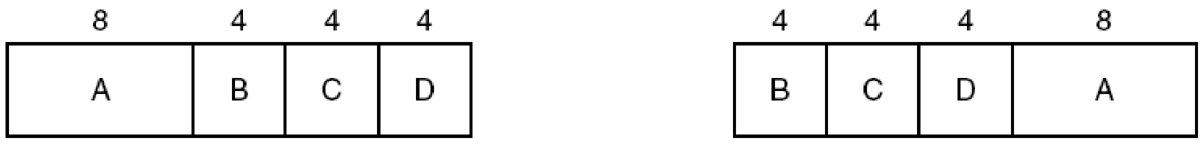
- With each process, associate the length of its next CPU burst
- Use these lengths to schedule the process with the shortest time

If two jobs have same burst time, FCFS is used to break the tie. SJF scheduling is used frequently in long-term scheduling.

Shortest-next-CPU-burst algorithm: Scheduling depends on the length of the next CPU burst of a process, rather than its total length.

Comparison

- Four jobs A, B, C, D with run times of 8, 4, 4, and 4 ms, respectively.
- Using FCFS:
  TAT for A is 8, for B is 12, for C is 16 and for D is 20 minutes for an average of 14 ms.
- Using SJF:
  TATs are now 4, 8, 12, and 20 ms, giving an average of 11 ms.



(a) Running four jobs in the original order. (b) Running them in SJF order.

SJF is optimal – gives minimum average waiting time for a given set of processes. Moving a short process before a long process decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average WT decreases.

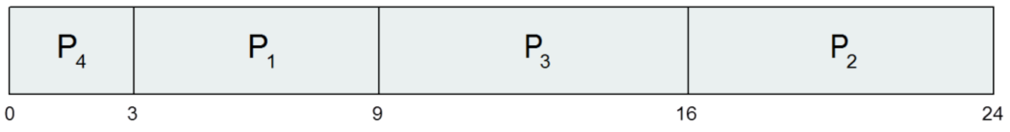
The difficulty is knowing the length of the next CPU request.Could ask the user - users are motivated to estimate the process time limit accurately.It cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst. – calculate approximate length

Example - Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	6

P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

SJF Gantt chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7 ms
- If we were using the FCFS, the average WT would be 10.25 milliseconds.

Performance:

- Pros:
  - Least Avg WT, Avg RT and Avg TAT.
  - Avg WT is reduced by almost 50% than in FCFS.
- Cons:
  - Knowing the CPU burst time is difficult.
  - Aging problem: Big jobs have to wait for so much time for CPU.
  - Optimal algorithm: cannot be implemented at the level of short term CPU scheduling.
  - Optimal only when all the jobs are available simultaneously.

ALGORITHM:

<u>FCFS Scheduling:</u>

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Set the waiting of the first process as ‘0’ and its burst time as its turn around time
Step 5: for each process in the Ready Q calculate
(a) Waiting time for process(n) = waiting time of process(n-1) + Burst time of process(n-1)
(b) Turn around time for Process(n) = waiting time of Process(n) + Burst time for process (n)
Step 6: For each process executed, print Gantt Chart data: process id, burst time, start time and end time of all the processes.
Step 7: Calculate and print the
(a) Average waiting time = Total waiting Time / Number of process
(b) Average Turnaround time = Total Turnaround Time / Number of process Step Step
8: Stop the process

<u>Priority Scheduling:</u>

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time and priority value
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as '0' and its burst time as its turn around time
Step 6: For each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 7: For each process executed, print Gantt Chart data: process id, burst time, priority, start time and end time of all the processes.
Step 8: Calculate and print the

(a) Average waiting time = Total waiting Time / Number of process
(b) Average Turnaround time = Total Turnaround Time / Number of process Step Step

9: Stop the process

<u>Round Robin Scheduling:</u>

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Calculate and save the process details such as total burst time, remaining time and completion flag for each process.
Step 5: Reset waiting time and turnaround time for each process as zero.
Step 6: Consider the ready queue is a circular Q, for each process executed, print Gantt Chart data. Do until the completion flag is set for all processes.

(a) If remaining time is less then quantum, set duration as remaining time, turnaround time as (start time+duration), waiting time as (turnaround time-burst) and the set the completion flag to indicate that the process is finished.
Go to (b) otherwise
(b) Set duration as quantum time.
(c) Print the Gantt chart values in this round: process id, burst time, remaining time, start time and end time.
(d) Reset remaining time and stime time for next turn as (remaining time-duration) and (start time+duration) respectively.
(e) Move to step 6 (a) to continue with the next process in a round robin fashion.

Step 7: Print the waiting times and turn around times for each of the processes and calculate the total waiting time and total average time.
Step 8: Calculate and print the

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 9: Stop the process

SJF Scheduling:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the burst time.
Step 4: Sort the ready queue according to the ascending order of burst times.
Step 5: Set the waiting time of the first process as ‘0’ and its burst time as its turn around time
Step 6: For each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 7: For each process executed, print Gantt Chart data: process id, burst time, priority, start time and end time of all the processes.
Step 8: Calculate and print the

(a) Average waiting time = Total waiting Time / Number of process
(b) Average Turnaround time = Total Turnaround Time / Number of process Step Step

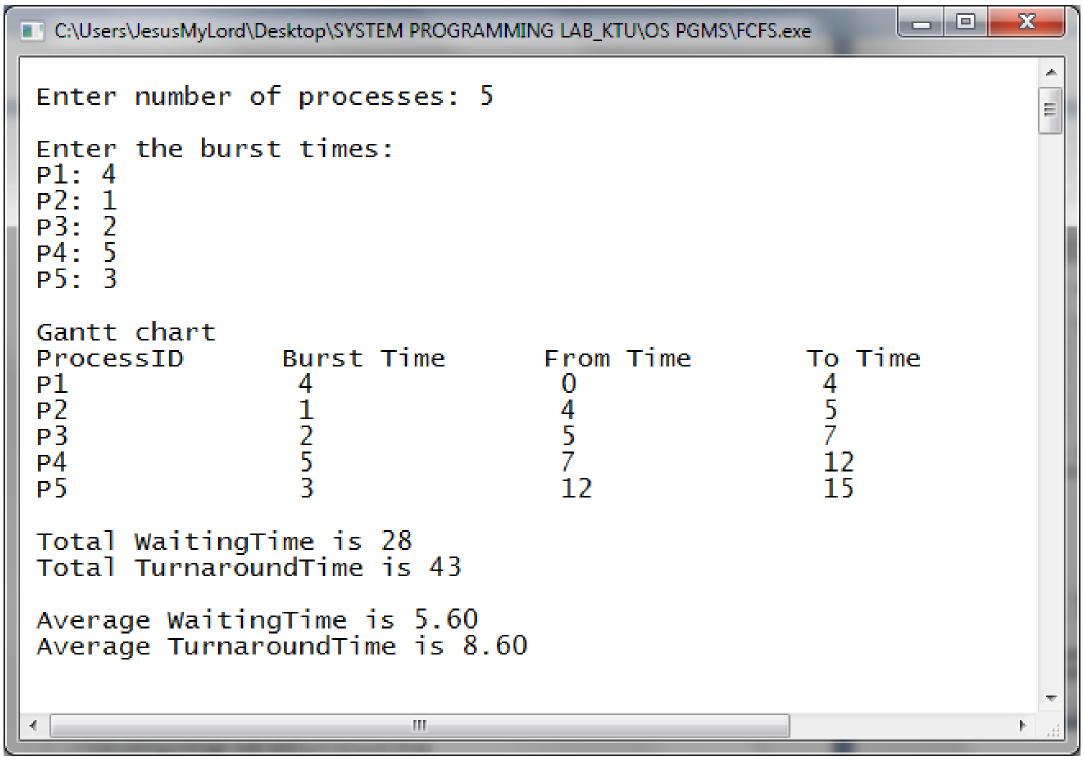
9: Stop the process

PROGRAM DEVELOPMENT:

```
/* Program to implement First Come First Serve Scheduling*/
#include<stdio.h>
int main()
{
   int n, pid[10], burst[10], wt[10], tat[10], twt=0, ttat=0, stime=0, i;
   float avgwt, avgtat;

   //Input process details
   printf("\n Enter number of processes: ");
   scanf("%d", &n);
   printf("\n Enter the burst times: \n");
   for(i=1; i<=n; i++)
   {
     //Saving process id and burst times
     pid[i] = i;
     printf(" P%d: ", i);
     scanf("%d", &burst[i]);
     //Recording the start and end times
     wt[i] = stime;
```

```
        stime = stime + burst[i];
        tat[i] = stime;
    }

    printf("\n Gantt chart ");
    printf("\n ProcessID \tBurst Time \tFrom Time \tTo Time\n");
    for(i=1; i<=n; i++)
    {
        printf(" P%d\t\t %d\t\t %d\t\t %d\t\t\n", pid[i], burst[i], wt[i], tat[i]);
        //Calculating total wait and turn around times
        twt += wt[i];
        ttat += tat[i];
    }
    //Calculating average wait and turn around times
    printf("\n Total WaitingTime is %d", twt);
    printf("\n Total TurnaroundTime is %d\n", ttat);
    avgwt = (float)twt/n;
    avgtat = (float)ttat/n;
    printf("\n Average WaitingTime is %.2f", avgwt);
    printf("\n Average TurnaroundTime is %.2f\n", avgtat);
    getch();
}
```

SAMPLE INPUT AND OUTPUT:

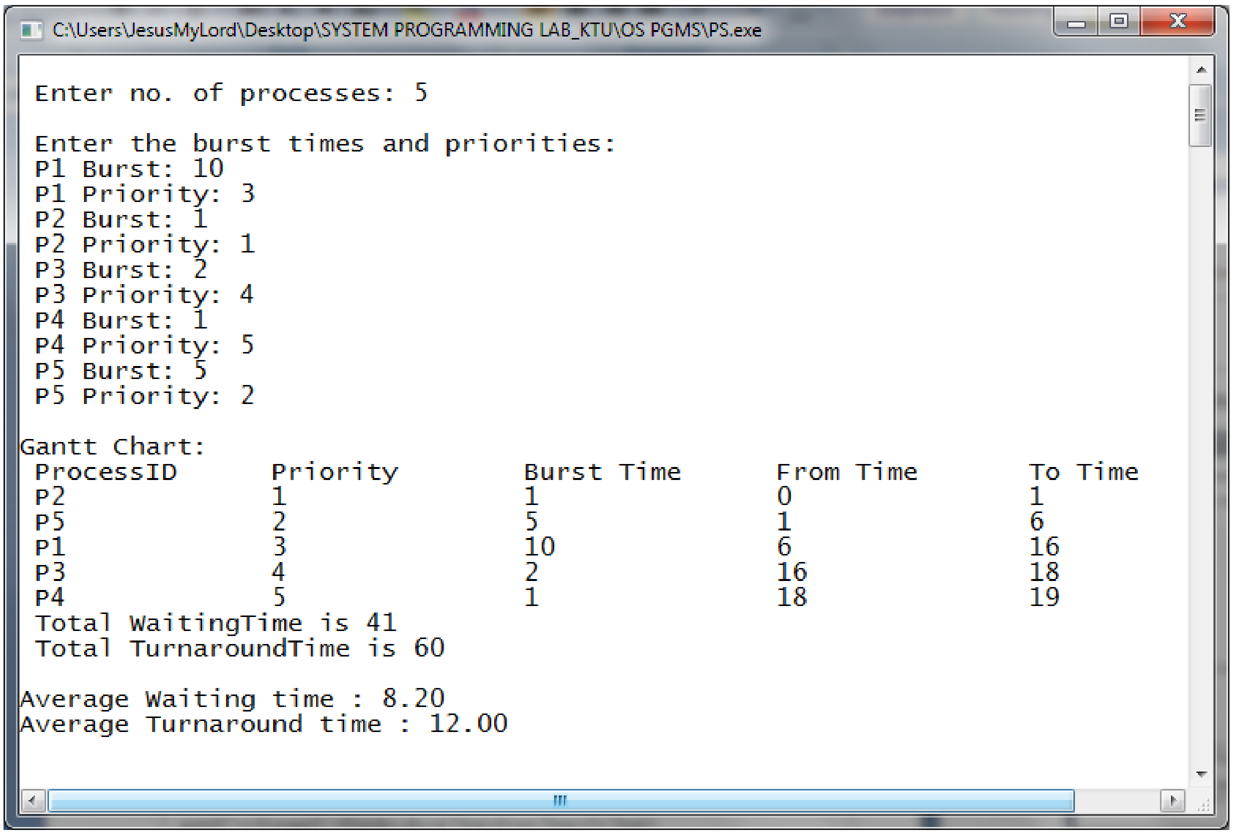

C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\FCFS.exe

```
Enter number of processes: 5

Enter the burst times:
P1: 4
P2: 1
P3: 2
P4: 5
P5: 3

Gantt chart
ProcessID       Burst Time       From Time       To Time
P1               4                0               4
P2               1                4               5
P3               2                5               7
P4               5                7               12
P5               3                12              15

Total WaitingTime is 28
Total TurnaroundTime is 43

Average WaitingTime is 5.60
Average TurnaroundTime is 8.60
```

PROGRAM DEVELOPMENT:

```
/* Program to implement Priority Scheduling*/
/* Lowest number indicating highest priority */
#include<stdio.h>
int main()
{
  int n, pid[10], burst[10], prio[10], wt[10], tat[10];
  int i, j, temp=0, stime=0, twt=0, ttat=0;
  float avgtat, avgwt;

  //Input process details
  printf("\n Enter no. of processes: ");
  scanf("%d",&n);
  printf("\n Enter the burst times and priorities: \n");
  for(i=1; i<=n; i++)
  {
     //Saving process id, priority and burst times
     pid[i] = i;
     printf(" P%d Burst: ", i);
     scanf("%d", &burst[i]);
     printf(" P%d Priority: ",i);
     scanf("%d", &prio[i]);
  }

  // Sort as per priority order
  for(i=1; i<=n; i++)
    for(j=i+1; j<=n; j++)
       if(prio[i] > prio[j])
       {
         //do swapping of id, burst and priority to sort them
         temp = pid[i];
         pid[i] = pid[j];
         pid[j] = temp;
         temp = burst[i];
         burst[i] = burst[j];
         burst[j] = temp;
         temp = prio[i];
         prio[i] = prio[j];
         prio[j] = temp;
       }

   printf("\nGantt Chart: ");
   printf("\n ProcessID \tPriority \tBurst Time \tFrom Time \tTo Time");
   for(i=1; i<=n; i++)
   {
     //Recording the start and end times
     wt[i] = stime;
     stime = stime + burst[i];
     tat[i] = stime;
     //Calculating total wait and turn around times
     twt += wt[i];
     ttat += tat[i];
     //Printing chart
     printf("\n P%d \t\t%d \t\t%d \t\t%d \t\t%d", pid[i], prio[i], burst[i], wt[i], tat[i]);
   }

   //Calculating average wait and turn around times
   printf("\n Total WaitingTime is %d", twt);
   printf("\n Total TurnaroundTime is %d\n", ttat);
   avgwt = (float)twt/n;
   avgtat = (float)ttat/n;
   printf("\nAverage Waiting time : %.2f", avgwt);
   printf("\nAverage Turnaround time : %.2f", avgtat);
   getch();
}
```

SAMPLE INPUT AND OUTPUT:



C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\PS.exe

```
 Enter no. of processes: 5

 Enter the burst times and priorities:
 P1 Burst: 10
 P1 Priority: 3
 P2 Burst: 1
 P2 Priority: 1
 P3 Burst: 2
 P3 Priority: 4
 P4 Burst: 1
 P4 Priority: 5
 P5 Burst: 5
 P5 Priority: 2

Gantt Chart:
 ProcessID      Priority       Burst Time     From Time      To Time
 P2             1              1              0              1
 P5             2              5              1              6
 P1             3              10             6              16
 P3             4              2              16             18
 P4             5              1              18             19
 Total WaitingTime is 41
 Total TurnaroundTime is 60

Average Waiting time : 8.20
Average Turnaround time : 12.00
```

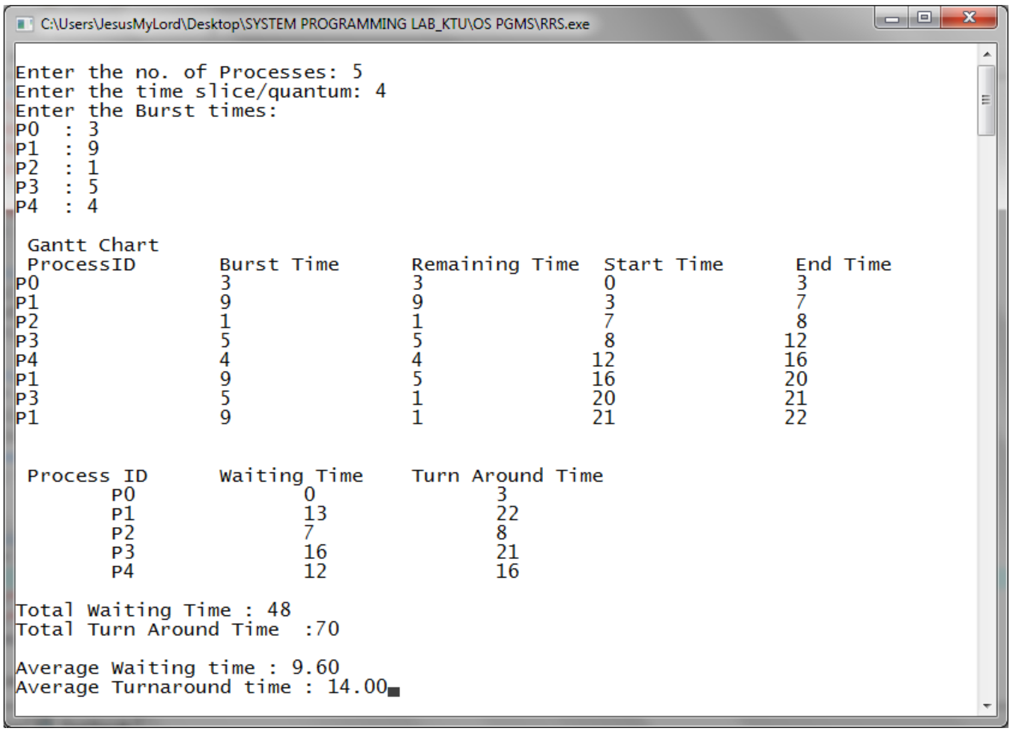
PROGRAM DEVELOPMENT:

```
/* Program to implement Round Robin Scheduling*/
#include<stdio.h>
int main()
{
   int i, n, qt, tbt=0, twt=0, ttat=0;
   int pid[10], burst[10], rem[10], flag[10], wt[10], tat[10];
   int stime, dur;
   float awt,atat;

   //Input process details
   printf("\nEnter the no. of Processes: ");
   scanf("%d", &n);
   printf("Enter the time slice/quantum: ");
   scanf("%d", &qt);

   printf("Enter the Burst times:\n");
   for(i=0; i<n; i++)
   {
     //Saving process id and burst times
     pid[i] = i;
     printf("P%d  : ", i);
```

```
  scanf("%d", &burst[i]);

  //Saving process details - total burst time, remaining time, completion flag
  tbt = tbt + burst[i];
  rem[i] = burst[i];
  flag[i] = 0;
  //Reset waiting time and turnaround time arrays
  wt[i] = tat[i] = 0;
}

//Calculating Gantt Chart entries
stime = 0;
i = 0;
printf("\n Gantt Chart");
printf("\n ProcessID\t Burst Time\t Remaining Time\t Start Time\t End Time\n");
while(stime < tbt)
{
  if(flag[i] == 0) //if the process not completed yet
  {
    if(rem[i] <= qt) //if remaining time is less than quantum
    {
      dur = rem[i];
      tat[i] = dur + stime;
      wt[i] = tat[i] - burst[i];
      flag[i] = 1; //set the process as finished
    }
    else
      dur = qt; //else, the duration of execution is quantum

    printf("P%d %15d %15d %15d %15d\n", pid[i], burst[i], rem[i], stime, stime+dur);

    //Reset remaining time and stime time for next turn
    rem[i] = rem[i] - dur;
    stime = stime + dur;
  }
  //Move to next process in a round robin manner
  i = (i+1) % n;
}
printf("\n\n Process ID \t Waiting Time \t Turn Around Time");
for(i=0; i<n; i++)
{
  printf("\n\tP%d \t\t%d \t\t%d", pid[i], wt[i], tat[i]);
  //Calculating total wait and turn around times
  twt = twt + wt[i];
```

```
       ttat = ttat + tat[i];
     }
     printf("\n\nTotal Waiting Time : %d", twt);
     printf("\nTotal Turn Around Time  :%d", ttat);

     //Calculating average wait and turn around times
     awt = (float)twt/n;
     atat = (float)ttat/n;
     printf("\n\nAverage Waiting time : %.2f", awt);
     printf("\nAverage Turnaround time : %.2f", atat);

     getch();
}
```

SAMPLE INPUT AND OUTPUT:


C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\RRS.exe

```
Enter the no. of Processes: 5
Enter the time slice/quantum: 4
Enter the Burst times:
P0   : 3
P1   : 9
P2   : 1
P3   : 5
P4   : 4

 Gantt Chart
 ProcessID          Burst Time          Remaining Time   Start Time          End Time
P0                  3                   3                   0                   3
P1                  9                   9                   3                   7
P2                  1                   1                   7                   8
P3                  5                   5                   8                  12
P4                  4                   4                  12                  16
P1                  9                   5                  16                  20
P3                  5                   1                  20                  21
P1                  9                   1                  21                  22


 Process ID         Waiting Time        Turn Around Time
        P0                 0                   3
        P1                 13                  22
        P2                 7                   8
        P3                 16                  21
        P4                 12                  16

Total Waiting Time : 48
Total Turn Around Time  :70

Average Waiting time : 9.60
Average Turnaround time : 14.00
```

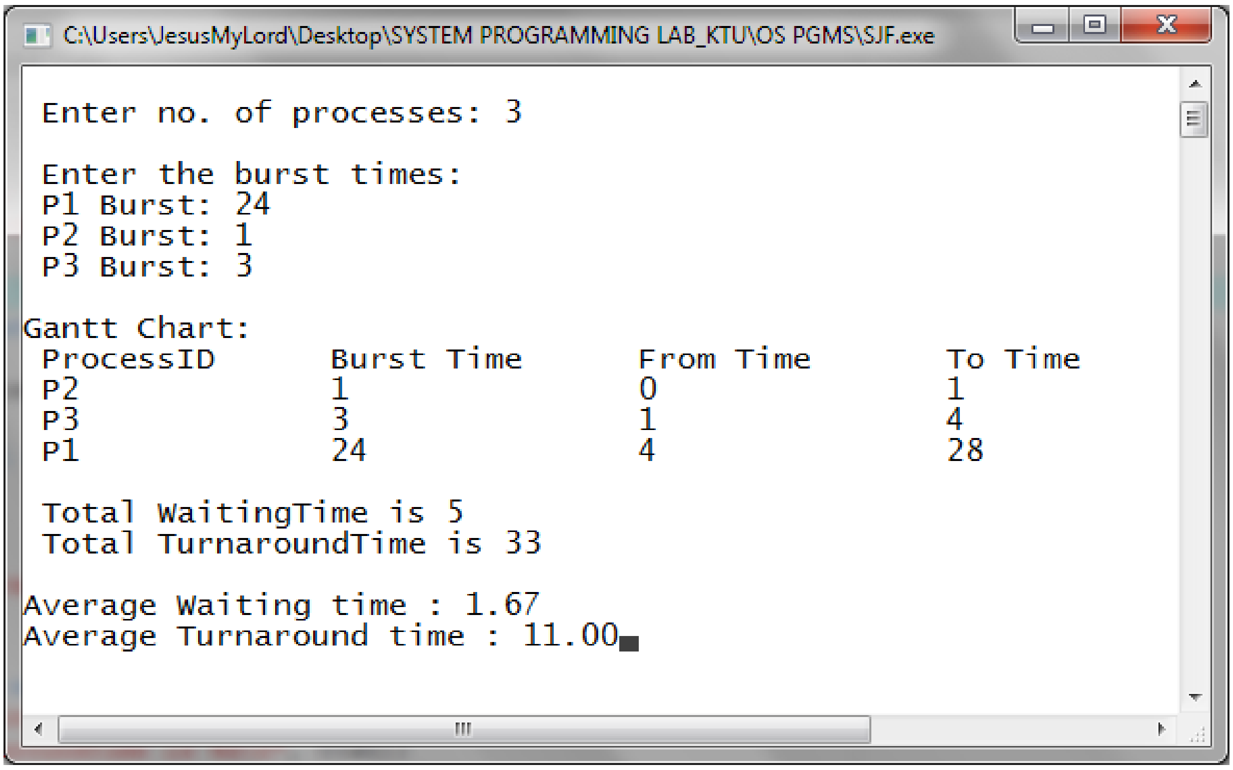
PROGRAM DEVELOPMENT:

```
/* Program to implement Shortest Job First Scheduling*/
#include<stdio.h>
int main()
{
  int n, pid[10], burst[10], wt[10], tat[10];
  int i, j, temp=0, stime=0, twt=0, ttat=0;
  float avgtat, avgwt;
```

```
//Input process details
printf("\n Enter no. of processes: ");
scanf("%d",&n);
printf("\n Enter the burst times: \n");
for(i=1; i<=n; i++)
{
   //Saving process id and burst times
   pid[i] = i;
   printf(" P%d Burst: ", i);
   scanf("%d", &burst[i]);
}

// Sort as per burst time order
for(i=1; i<=n; i++)
  for(j=i+1; j<=n; j++)
     if(burst[i] > burst[j])
     {
       //do swapping of id and burst to sort them
       temp = pid[i];
       pid[i] = pid[j];
       pid[j] = temp;

       temp = burst[i];
       burst[i] = burst[j];
       burst[j] = temp;
     }

 printf("\nGantt Chart: ");
 printf("\n ProcessID \tBurst Time \tFrom Time \tTo Time");
 for(i=1; i<=n; i++)
 {
    //Recording the start and end times
    wt[i] = stime;
    stime = stime + burst[i];
    tat[i] = stime;
    //Calculating total wait and turn around times
    twt += wt[i];
    ttat += tat[i];
    //Printing chart
    printf("\n P%d \t\t%d \t\t%d \t\t%d", pid[i], burst[i], wt[i], tat[i]);
 }
 //Calculating average wait and turn around times
 printf("\n\n Total WaitingTime is %d", twt);
 printf("\n Total TurnaroundTime is %d\n", ttat);
```

```
  avgwt = (float)twt/n;
  avgtat = (float)ttat/n;
  printf("\nAverage Waiting time : %.2f", avgwt);
  printf("\nAverage Turnaround time : %.2f", avgtat);
  getch();
}
```

SAMPLE INPUT AND OUTPUT:



C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\SJF.exe

```
 Enter no. of processes: 3

 Enter the burst times:
 P1 Burst: 24
 P2 Burst: 1
 P3 Burst: 3

Gantt Chart:
 ProcessID       Burst Time       From Time        To Time
 P2              1                0                1
 P3              3                1                4
 P1              24               4                28

 Total WaitingTime is 5
 Total TurnaroundTime is 33

Average Waiting time : 1.67
Average Turnaround time : 11.00
```

CONCLUSION:

Programs to simulate process scheduling algorithms is implemented successfully and the performance comparison is done successfully.

Exp No: 8

IMPLEMENTATION OF MEMORY ALLOCATION METHODS

FOR FIXED PARTITION

PROBLEM DEFINITION:

. To Simulate the following memory allocation Techniques.

a) Worst-fit
b) Best-fit
c) First-fit

THEORETICAL BACKGROUND:

Memory management

Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Need for memory management:

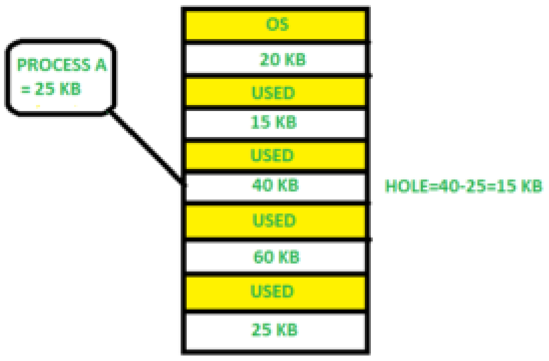
- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

Memory allocation:

To gain proper memory utilization, memory allocation must be allocated efficient manner. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions and each partition contains exactly one process. Thus, the degree of multiprogramming is obtained by the number of partitions.

Multiple partition allocation: In this method, a process is selected from the input queue and loaded into the free partition. When the process terminates, the partition becomes available for other processes.
Fixed partition allocation: In this method, the operating system maintains a table that indicates which parts of memory are available and which are occupied by processes. Initially, all memory is available for user processes and is considered one large block of available memory. This available memory is known as “Hole”. When the process arrives and needs memory, we search for a hole that is large enough to store this process. If the requirement fulfills then we allocate memory to process, otherwise keeping the rest available to satisfy future requests. While allocating a memory sometimes dynamic storage allocation problems occur, which concerns how to satisfy a request of size n from a list of free holes. There are some solutions to this problem:

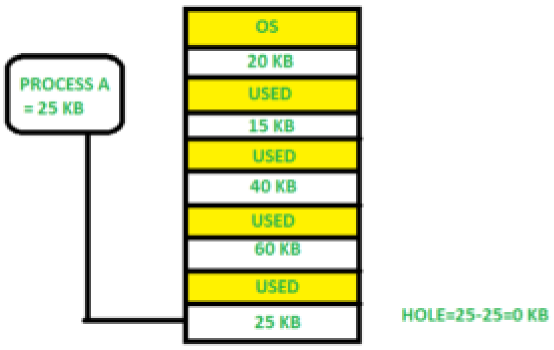
First fit:-
In the first fit, the first available free hole fulfills the requirement of the process allocated.



Here, in this diagram 40 KB memory block is the first available free hole that can store process A (size of 25 KB), because the first two blocks did not have sufficient memory space.

Best fit:-

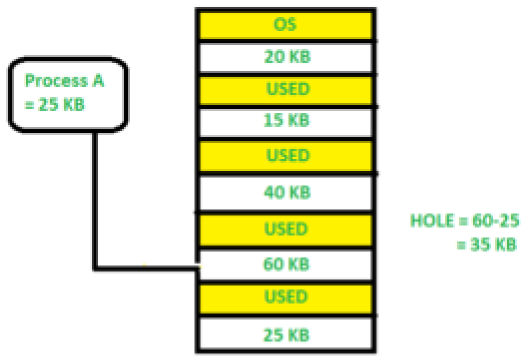
In the best fit, allocate the smallest hole that is big enough to process requirements. For this, we search the entire list, unless the list is ordered by size.



Here in this example, first, we traverse the complete list and find the last hole 25KB is the best suitable hole for Process A(size 25KB).

In this method memory utilization is maximum as compared to other memory allocation techniques.

Worst fit:-In the worst fit, allocate the largest available hole to process. This method produces the largest leftover hole.



Here in this example, Process A (Size 25 KB) is allocated to the largest available memory block which is 60KB. Inefficient memory utilization is a major issue in the worst fit.

PROGRAM DEVELOPMENT:

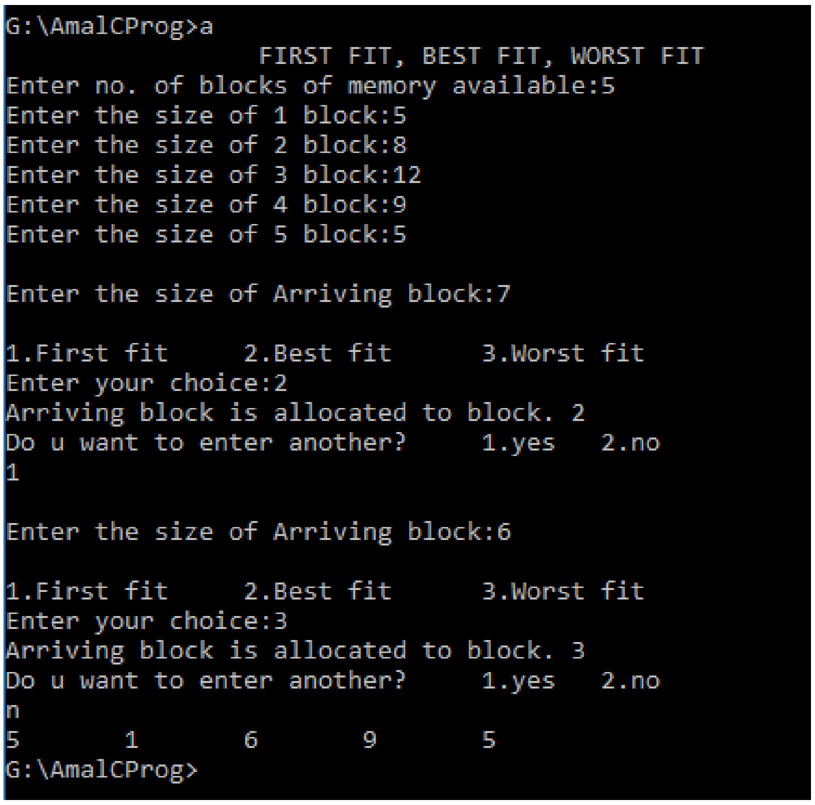
```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,temp,b[10],c[10],arr,n,ch,a;
printf("\t\t FIRST FIT, BEST FIT, WORST FIT\n");
printf("Enter no. of blocks of memory available:");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
printf("Enter the size of %d block:",i);
scanf("%d",&b[i]);
c[i]=b[i];
}
inp:
ch=-1;
printf("\nEnter the size of Arriving block:");
scanf("%d",&arr);
opt:
printf("\n1.First fit\t2.Best fit\t3.Worst fit\nEnter your choice:");
scanf("%d",&ch);
a=-1;
switch(ch)
{
case 1:
for(i=1;i<=n;i++)
{
if(c[i]>=arr)
{
```

```
printf("Arriving block is allocated to block %d.\n",i);
c[i]-=arr;
a=0;
break;
}
else
continue;
}
if(a==-1)
printf("Cannot allocate memory for arriving block\n");
break;
case 2:
for(i=1;i<=n;i++)
{
for(j=1;j<=n-i;j++)
{
if(b[i]>=b[i+1])
{
temp=b[i];
b[i]=b[i+1];
b[i+1]=temp;
}
}
}
for(i=1;i<=n;i++)
{
if(b[i]>=arr)
{
a=b[i];
break;
}
else
continue;
}
if(a!=-1)
{
for(i=1;i<=n;i++)
{
if(c[i]==a)
{
printf("Arriving block is allocated to block. %d\n",i);
c[i]-=arr;
}
}
```

```
}
else
printf("Cannot allocate memory for arriving block\n");
break;
case 3:
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(b[i]>=b[i+1])
{
temp=b[i];
b[i]=b[i+1];
b[i+1]=temp;
}
}
}
for(i=n;i>=1;i--)
{
if(b[i]>=arr)
{
a=b[i];
break;
}
else
continue;
}
if(a!=-1)
{
for(i=1;i<=n;i++)
{
if(c[i]==a)
{
printf("Arriving block is allocated to block. %d\n",i);
c[i]-=arr;
}
}
}
else
printf("Cannot allocate memory for arriving block\n");
break;
default:
printf("Enter the valid choice:");
goto opt;
```

```
}
printf("Do u want to enter another? \t1.yes\t2.no\n");
scanf("%d",&ch);
if(ch==1)
goto inp;
else
{
for (i = 1; i <= n; ++i)
printf("%d\t",c[i]);
}
}
```

SAMPLE INPUT AND OUTPUT:



```
G:\AmalCProg>a
                  FIRST FIT, BEST FIT, WORST FIT
Enter no. of blocks of memory available:5
Enter the size of 1 block:5
Enter the size of 2 block:8
Enter the size of 3 block:12
Enter the size of 4 block:9
Enter the size of 5 block:5

Enter the size of Arriving block:7

1.First fit     2.Best fit      3.Worst fit
Enter your choice:2
Arriving block is allocated to block. 2
Do u want to enter another?     1.yes   2.no
1

Enter the size of Arriving block:6

1.First fit     2.Best fit      3.Worst fit
Enter your choice:3
Arriving block is allocated to block. 3
Do u want to enter another?     1.yes   2.no
n
5       1       6       9       5
G:\AmalCProg>
```

CONCLUSION:

The program was executed and output obtained successfully and was verified

Exp No: 9

PAGE REPLACEMENT ALGORITHMS

PROBLEM DEFINITION:

To develop a C program to simulate page replacement algorithms

a) FIFO b) LRU c) LFU

THEORETICAL BACKGROUND:

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme.

A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

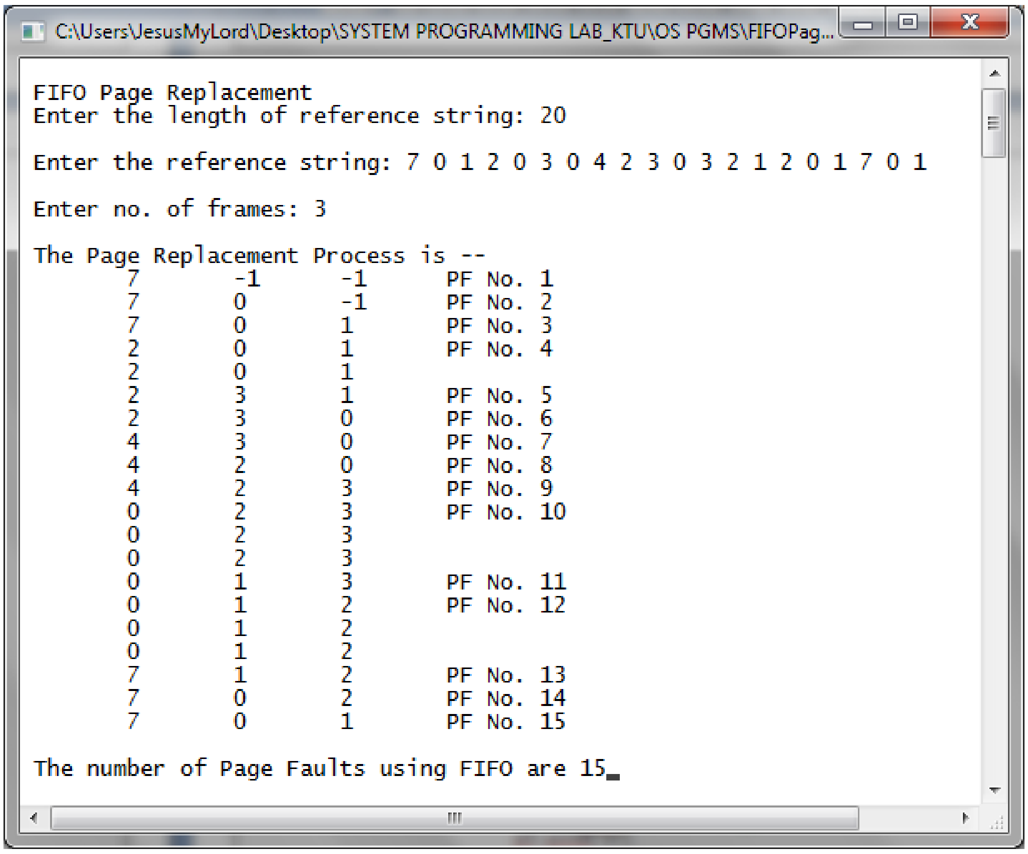
PROGRAM DEVELOPMENT:

```
/*FIFO PAGE REPLACEMENT*/
#include<stdio.h>
#include<conio.h>
int main()
{
   int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
   printf("\n FIFO Page Replacement");
   printf("\n Enter the length of reference string: ");
   scanf("%d",&n);
   printf("\n Enter the reference string: ");
   for(i=0;i<n;i++)
      scanf("%d",&rs[i]);
   printf("\n Enter no. of frames: ");
   scanf("%d",&f);
   for(i=0;i<f;i++)
      m[i]=-1;
   printf("\n The Page Replacement Process is -- \n");
   for(i=0;i<n;i++)
   {
      for(k=0;k<f;k++)
      {
         if(m[k]==rs[i])
```

```
            break;
        }
        if(k==f)
        {
            m[count++]=rs[i];
            pf++;
        }
        for(j=0;j<f;j++)
            printf("\t%d",m[j]);
        if(k==f)
        printf("\tPF No. %d",pf);
        printf("\n");
        if(count==f)
            count=0;
    }
    printf("\n The number of Page Faults using FIFO are %d",pf);
    getch();
}
```

SAMPLE INPUT AND OUTPUT:

//



C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\FIFOPag...

```
FIFO Page Replacement
Enter the length of reference string: 20

Enter the reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Enter no. of frames: 3

The Page Replacement Process is --
        7       -1      -1      PF No. 1
        7       0       -1      PF No. 2
        7       0       1       PF No. 3
        2       0       1       PF No. 4
        2       0       1
        2       3       1       PF No. 5
        2       3       0       PF No. 6
        4       3       0       PF No. 7
        4       2       0       PF No. 8
        4       2       3       PF No. 9
        0       2       3       PF No. 10
        0       2       3
        0       2       3
        0       1       3       PF No. 11
        0       1       2       PF No. 12
        0       1       2
        0       1       2
        7       1       2       PF No. 13
        7       0       2       PF No. 14
        7       0       1       PF No. 15

The number of Page Faults using FIFO are 15
```

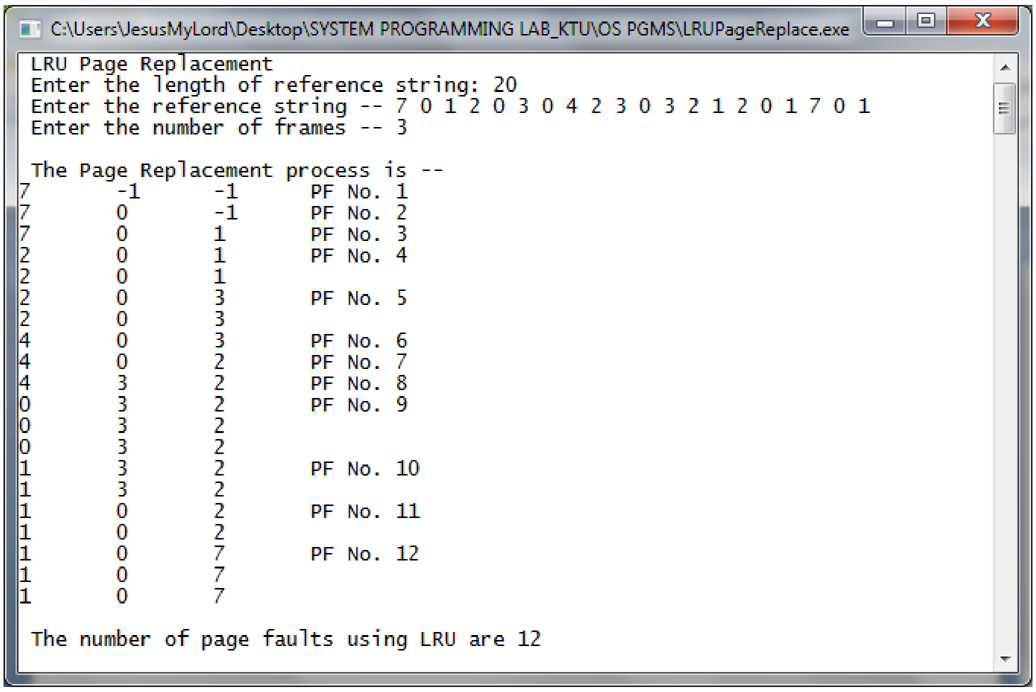
//////////////////

PROGRAM DEVELOPMENT:

```
/*LRU PAGE REPLACEMENT*/
#include<stdio.h>
#include<conio.h>
int main()
{
  int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
  printf("\n LRU Page Replacement");
  printf("\n 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("\n The 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("\n The number of page faults using LRU are %d",pf);
  getch();
}
```

SAMPLE INPUT AND OUTPUT:



C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\LRUPageReplace.exe

```
 LRU Page Replacement
 Enter the length of reference string: 20
 Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
 Enter the number of frames -- 3

 The Page Replacement process is --
7       -1      -1      PF No. 1
7       0       -1      PF No. 2
7       0       1       PF No. 3
2       0       1       PF No. 4
2       0       1
2       0       3       PF No. 5
2       0       3
4       0       3       PF No. 6
4       0       2       PF No. 7
4       3       2       PF No. 8
0       3       2       PF No. 9
0       3       2
0       3       2
1       3       2       PF No. 10
1       3       2
1       0       2       PF No. 11
1       0       2
1       0       7       PF No. 12
1       0       7
1       0       7

 The number of page faults using LRU are 12
```

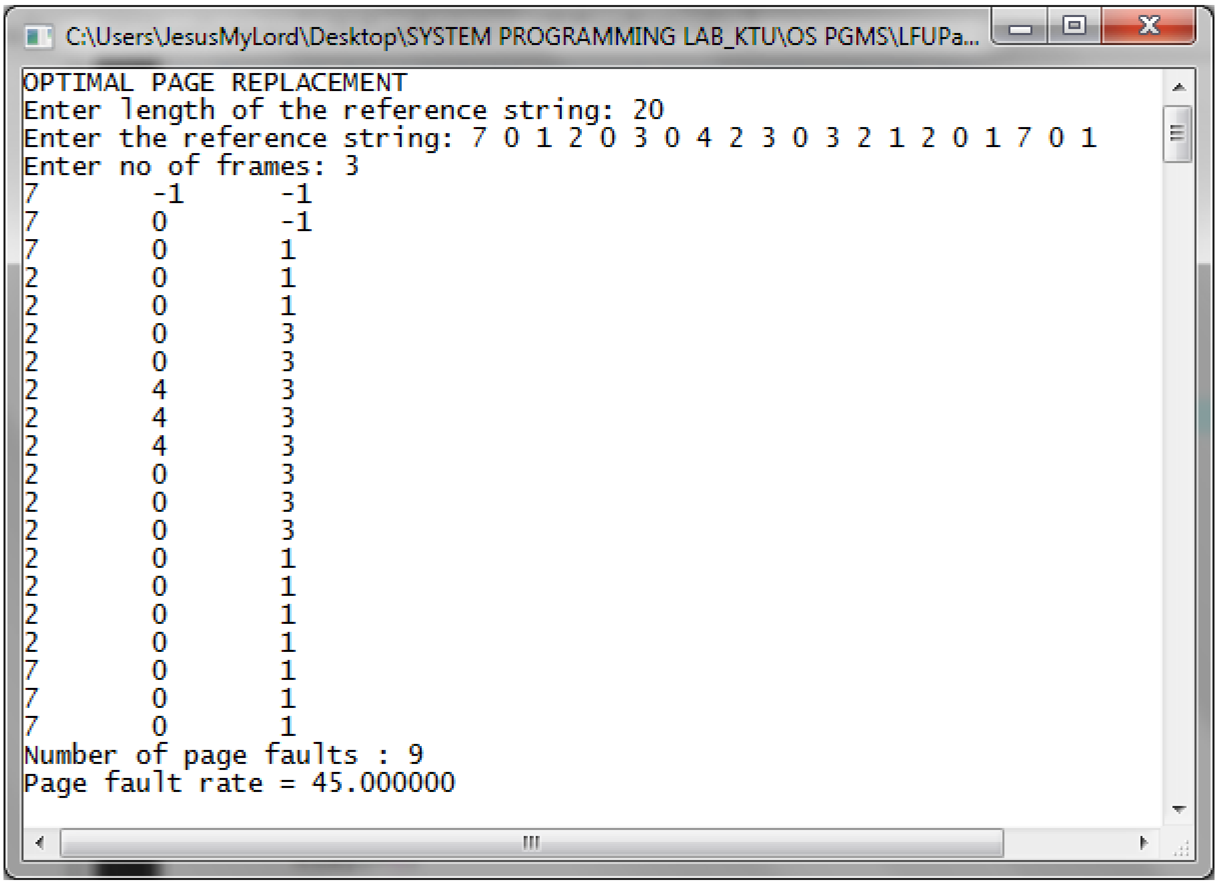
PROGRAM DEVELOPMENT:

```
/* Program to simulate optimal page replacement */
#include<stdio.h>
#include<conio.h>
int fr[3], n, m;
void display();
int main()
{
   int i,j,page[20],fs[10];
   int max,found=0,lg[3],indx,k,l,flag1=0,flag2=0,pf=0;
   float pr;
   printf("OPTIMAL PAGE REPLACEMENT\n");
   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;
   pf=m;
   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[j];
               flag2=1;
               break;
            }
```

```
        }
    }
    if(flag2==0)
    {
        for(i=0;i<m;i++)
            lg[i]=0;
        for(i=0;i<m;i++)
        {
            for(k=j+1;k<=n;k++)
            {
                if(fr[i]==page[k])
                {
                     lg[i]=k-j; break;
                }
            }
        }
        found=0;
        for(i=0;i<m;i++)
        {
            if(lg[i]==0)
            {
                indx=i;
                found = 1;
                break;
            }
        }
        if(found==0)
        {
            max=lg[0];
            indx=0;
            for(i=0;i<m;i++)
            {
                if(max<lg[i])
                {
                    max=lg[i];
                    indx=i;
                }
            }
        }
        fr[indx]=page[j];
        pf++;
    }
    display();
}
```

```
    printf("Number of page faults : %d\n", pf); pr=(float)pf/n*100;
    printf("Page fault rate = %f \n", pr); getch();
}

void display()
{
    int i;
    for(i=0;i<m;i++)
        printf("%d\t",fr[i]);
    printf("\n");
}
```

SAMPLE INPUT AND OUTPUT:



C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\LFUPa...

```
OPTIMAL PAGE REPLACEMENT
Enter length of the reference string: 20
Enter the reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter no of frames: 3
7       -1      -1
7       0       -1
7       0       1
2       0       1
2       0       1
2       0       3
2       0       3
2       4       3
2       4       3
2       4       3
2       0       3
2       0       3
2       0       3
2       0       1
2       0       1
2       0       1
2       0       1
7       0       1
7       0       1
7       0       1
Number of page faults : 9
Page fault rate = 45.000000
```

CONCLUSION:

Implemented programs to simulate page replacement algorithms and verified.

Exp No: 10

BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

PROBLEM DEFINITION:

To implement a C program to simulate the banker's algorithm for deadlock avoidance to find whether the system is in safe state or not.

THEORETICAL BACKGROUND:

DEADLOCK: 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.

Necessary Conditions: A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait: A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Methods for Handling Deadlocks

The deadlock problem can be dealt with in one of threeways:

- Use a protocol to prevent or avoid deadlocks, ensuring that thesystem will *never* enter a deadlocked state.
- Allow the system to enter a deadlocked state, detect it, and recover.
- Ignore the problem altogether and pretend that deadlocks neveroccur in the system.

Deadlock Avoidance: Avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system 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.

- Simplest and most useful model
- Requires that the system has some additional a priori information available regarding the resource usage.
- Requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

- Resource-allocation state is defined by the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Deadlock Avoidance – Example: In a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

Safe State: When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a **safe sequence**.

Safe Sequence: A sequence of processes <P1, P2, …, Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i.That is:If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.When P_j is finished, P_i can obtain all the needed resources, execute, return allocated resources, and terminate.When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

Safe State vs Deadlock: A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock. As long as the state is safe, the OS can avoid unsafe (and deadlocked) states. Deadlock avoidance try to ensure that the system never enters an unsafe state.

Avoidance Algorithms: The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state.Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, **resource utilization may be lower** than it would otherwise be.

- Single instance of a resource type: Use a **resource-allocation graph**
- Multiple instances of a resource type: Use the **Banker's algorithm**

BANKER'S ALGORITHM:

For a system with multiple instances of each resource type, use Banker's Algorithm for deadlock avoidance. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers. It is less efficient than the resource-allocation graph scheme.When a new process enters the system, it must **declare the maximum no. of instances of each resource type** that it may need. This may not exceed the total number of resources in the system. When a **userrequests** a set of resources, the system must determine whether the **allocation of these resources will leave the system in a safe state**. If it will, the resources are **allocated**; otherwise, the process must **wait** until some other process releases enough resources.

Banker's algorithm implementation needs several **data structures**to encode the state of the resource-allocation system.

- Let **n** is the **number of processes** in the system and **m** is the **number of resource** types:
- **Available**: A **vector of length m** indicates the **number of available resources** of each type.
  - If Available[j] equals k, then k instances of resource type Rj are available.
- **Max**: An **n × m matrix** defines the **maximum demand** of each process.
  - If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj.
- **Allocation**: An **n × m matrix** defines the number of **resources** of each type **currentlyallocated** to each process.
  - If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj .
- **Need**: An **n × m matrix** indicates the **remaining resource need** of each process.
  - If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task.
- **Need[i][j] = Max[i][j] - Allocation[i][j]**

These data structures vary over time in both size and value.

To simplify the presentation of the algorithm, establish some notations.

- Let X and Y be vectors of length n.
  - We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all i = 1, 2, ..., n.
- ie., if X = (1,7,3,2) and Y = (0,3,2,1), then $Y \leq X$.
  - In addition, $Y < X$ if $Y \leq X$ and Y <> X.
- Treat each row in the matrices Allocation and Need as vectors and refer to them as **Allocation**$_i$ and **Need**$_i$.
  - **Allocation**$_i$ specifies the resources **currently allocated** to process **Pi**.
  - **Need**$_i$ specifies the **additional resources** that process **Pi** may still request to complete its task.

a. Safety Algorithm:

Algorithm for finding out whether or not a system is in a safe state.

```
1. Let Work and Finish be vectors of length m and n, respectively.
        Initialize
        Work = Available      (work vector is set as the number of resources available)
        Finish[i] = false for i = 0, 1, ..., n − 1.  (process finished or not, set to false)
2. Find an index i such that both
        a. Finish[i] == false          (find any process not finished, and
        b. Need_i ≤ Work               whose need is less than the available resources)
        If no such i exists, go to step 4.
3. Work = Work + Allocation_i     (add the freed up resources of Pi to work vector)
```

Finish[i] = true *(mark process as finished)*

Go to step 2. *(iterate and find such processes one by one & allocate resources)*

4. If **Finish[i] == true** for all i, then the system is in a safe state.

This algorithm require an order of $\mathbf{m \times n^2}$operations to determine whether a state is safe.

b. Resource-Request Algorithm:

Algorithm for determining whether requests can be safely granted.

Let **Request$_i$** be the request vector for process Pi, i.e. the **resources requested** by the process. If Request$_i$ [j] == k, then process Pi wants k instances of resource type Rj.

When a resource request is made by process Pi, the following actions taken:

1. If **Request$_i$ ≤ Need$_i$** , go to step 2.

Otherwise, raise an error condition (requesting for mare than its need), since the process has exceeded its maximum claim.

2. If **Request$_i$ ≤ Available**, go to step 3.

Otherwise*, Pi must* ***wait****, since the resources are not available.*

3. Have the system **pretend** to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available – Request$_i$;

Allocation$_i$ = Allocation$_i$ + Request$_i$;

Need$_i$ = Need$_i$ – Request$_i$;

If the resulting resource-allocation state is **safe**, *(check using safety algorithm*, the transaction is completed, and process Pi is **allocated its resources**.

However, if the new state is **unsafe**, then Pi must **wait** for Request$_i$, and the **old resource-allocation state is restored**.

EXAMPLE OF BANKER'S ALGORITHM:

Consider a system with 5 processes P0 through P4 and3 resource types A, B, and C; Resource type A – 10 instances, Resource B – 5 instances, Resource C – 7 instances

Max – A(10), B(5), C(7)

Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	***A B C***	***A B C***
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- The matrix **Need** is defined to be (**Max − Allocation**)

Need Matrix Contents:

A B C

P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a **safe state** since the sequence **<P_1, P_3, P_4, P_2, P_0>**satisfies safety criteria.

Suppose now that process P1 requests 1additional instance of resource type A and 2 instances of resource type C. **Request$_1$ = (1,0,2)**
To decide whether this request can be immediately granted, check that
Request$_1$ ≤ Available, (1,0,2) ≤ (3,3,2), which is true.

We must determine **whether this new system state is safe?**

- **Execute safety algorithm** and find that the sequence **<P1, P3, P4, P0, P2>**satisfies the safety requirement. Hence, we can immediately grant the request of process P1.
- Pretend that this request has been fulfilled, and we arrive at the following new state:

	<u>*Allocation*</u>	<u>*Need*</u>	<u>*Available*</u>
	A B C	***A B C***	***A B C***
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <***P_1*, *P_3*, *P_4*, *P_0*, *P_2***> satisfies safety requirement. Hence it can be allocated.

ALGORITHM:

```
Step1: Start the process
Step 2: Get the number of processes and number of  resources
Step 3: Read the available number of instances for each resource
Step 4: Read the maximun resource count for each processes
Step 5: Read the allocated resource count for each processes
Step 6: Calculate  for each process and resource, need = max – allocated
Step 7: Calculate the current available instances for each resource,
                    available – total allocated
Step 8: For each process Pi to Pn, check whether need < available, if yes do steps 8.1,
        else increment i and go to step 8 to continue with the next process
 Step 8.1: Allocate resources to Pi as per need and complete the execution of Pi
Step 8.2: Release all the resources allocated to Pi
Step 8.3: Mark the process as completed
Step 8.4: Update current available count as available+max for Pi
Step 9: If all process are completed successfully, print that the system is in safe state, else,
        print that the system is not safe
Step 10: Stop the process
```

PROGRAM DEVELOPMENT:

```
#include<stdio.h>
#include<conio.h>
struct da
{
      int max[10],al[10],need[10],before[10],after[10];
}p[10];

main()
{
   int n,r,i,j,k,l;
   int tot[10],av[10];
   int cn=0,cz=0,temp=0,c=0;

   printf("\nEnter the no of processes:");
   scanf("%d",&n);
   printf("\nEnter the no of resources:");
   scanf("%d",&r);

   //Read the total number of instances for each resource type
   for(i=0; i<r; i++)
   {
      printf("Enter total no. of instances of resource %d:", i+1);
      scanf("%d", &tot[i]);
   }

   //Read the max and allocated resources for each process
   for(i=0; i<n; i++)
   {
      printf("\n*** PROCESS %d *** \n",i+1);
      //Read the maximum claim for each resources
      for(j=0; j<r; j++)
      {
         printf("Max claim for resource %d:", j+1);
         scanf("%d", &p[i].max[j]);
      }
      //Read the allocated instances for each resources
      for(j=0; j<r; j++)
      {
         printf("Allocated instances of resource %d:", j+1);
         scanf("%d", &p[i].al[j]);

         //Calculating need value for each process as max-allocation
         p[i].need[j] = p[i].max[j]-p[i].al[j];
```

```
    }
}

//Calculating the available resource instances
for(i=0; i<r; i++)
{
    for(j=0; j<n; j++)
    {
        temp = temp + p[j].al[i]; //find sum of resources for each process
    }
    av[i] = tot[i]-temp; //calculate available
    temp=0;
}
//
printf("\n\tMAX \tALLOCATION \tNEEDED");
for(i=0; i<n; i++)
{
    printf("\n P%d \t",i+1);
    //Print max
    for(j=0; j<r; j++)
        printf("%d ", p[i].max[j]);
    printf("\t");
    //Print allocated
    for(j=0; j<r; j++)
        printf("%d ", p[i].al[j]);
    printf("\t\t");
    //Print need
    for(j=0; j<r;j ++)
        printf("%d ", p[i].need[j]);
    printf("\t\t");
}
printf("\n\n\tTOTAL \tAVAILABLE\n");
    //Calculate and print total
    printf("\t");
    for(j=0; j<r; j++)
    {
        printf("%d ",tot[j]);
    }
    printf("\t");
    //Calculate and print available
    for(j=0;j<r;j++)
    {
        printf("%d ",av[j]);
    }
    printf("\n\n Checking for safe sequence...");
    printf("\n\n\tAVAIL-BEFORE \tAVAIL-AFTER");
    for(l=0; l<n; l++)
    {
        for(i=0; i<n; i++)
        {
            for(j=0; j<r; j++)
            {
                if(p[i].need[j] > av[j])
                    cn++; //increment cn if any process need is greater than available, cannot execute
                if(p[i].max[j]==0) //if no more resource needed
                    cz++;
            }
            if(cn>0)
                printf("\n Process %d has to wait as enough resources not available.", i+1);
            if(cn==0 && cz!=r) //if more resource needed and enough resource avaiable
            {
                for(j=0; j<r; j++)
                {
                    p[i].before[j] = av[j] - p[i].need[j]; //allocate resources
                    p[i].after[j] = p[i].before[j] + p[i].max[j]; //free the resources after executing
                    //p[i].after[j] = p[i].before[j] + p[i].need[j] + p[i].al[j]; //free the resources after
executing
                    av[j] = p[i].after[j]; //update the available resource count
                    p[i].max[j] = 0; //no more resource needed
                }
                printf("\n P%d \t", i+1);
                for(j=0; j<r; j++)
                    printf("%d ", p[i].before[j]);

                printf("\t\t");
                for(j=0; j<r; j++)
                    printf("%d ", p[i].after[j]);
                cn=0;
                cz=0;
                c++;
                break; //restart from first process and check for each process waiting
            }
            else
            {
                 cn=0;
                 cz=0;
            }
        }
    }
    if(c==n)
       printf("\n\n THE ABOVE SEQUENCE IS A SAFE SEQUENCE.");
    else
       printf("\n\n Deadlock Occured");

    getch();
}
```

SAMPLE INPUT AND OUTPUT:

OUTPUT SAMPLE 1:

```
ENTER THE NO OF PROCESSES:5
ENTER THE NO OF RESOURCES:3
ENTER TOTAL NO. OF INSTANCES OF RESOURCE 1:10
ENTER TOTAL NO. OF INSTANCES OF RESOURCE 2:5
ENTER TOTAL NO. OF INSTANCES OF RESOURCE 3:7
*** PROCESS 1 ***
MAX CLAIM FOR RESOURCE 1:7
MAX CLAIM FOR RESOURCE 2:5
MAX CLAIM FOR RESOURCE 3:3
ALLOCATED INSTANCES OF RESOURCE 1:0
ALLOCATED INSTANCES OF RESOURCE 2:1
ALLOCATED INSTANCES OF RESOURCE 3:0
*** PROCESS 2 ***
MAX CLAIM FOR RESOURCE 1:3
MAX CLAIM FOR RESOURCE 2:2
MAX CLAIM FOR RESOURCE 3:2
ALLOCATED INSTANCES OF RESOURCE 1:2
ALLOCATED INSTANCES OF RESOURCE 2:0
ALLOCATED INSTANCES OF RESOURCE 3:0
*** PROCESS 3 ***
MAX CLAIM FOR RESOURCE 1:9
MAX CLAIM FOR RESOURCE 2:0
MAX CLAIM FOR RESOURCE 3:2
ALLOCATED INSTANCES OF RESOURCE 1:3
ALLOCATED INSTANCES OF RESOURCE 2:0
ALLOCATED INSTANCES OF RESOURCE 3:2
*** PROCESS 4 ***
MAX CLAIM FOR RESOURCE 1:2
MAX CLAIM FOR RESOURCE 2:2
MAX CLAIM FOR RESOURCE 3:2
ALLOCATED INSTANCES OF RESOURCE 1:2
ALLOCATED INSTANCES OF RESOURCE 2:1
ALLOCATED INSTANCES OF RESOURCE 3:1
*** PROCESS 5 ***
MAX CLAIM FOR RESOURCE 1:4
MAX CLAIM FOR RESOURCE 2:3
MAX CLAIM FOR RESOURCE 3:3
ALLOCATED INSTANCES OF RESOURCE 1:0
ALLOCATED INSTANCES OF RESOURCE 2:0
ALLOCATED INSTANCES OF RESOURCE 3:2
        MAX         ALLOCATION      NEEDED
 P1     7 5 3       0 1 0           7 4 3
 P2     3 2 2       2 0 0           1 2 2
 P3     9 0 2       3 0 2           6 0 0
 P4     2 2 2       2 1 1           0 1 1
 P5      4 3 3      0 0 2           4 3 1
        TOTAL            AVAILABLE
         10 5 7               3 3 2
 CHECKING FOR SAFE SEQUENCE...
        AVAIL-BEFORE    AVAIL-AFTER
 PROCESS 1 HAS TO WAIT AS ENOUGH RESOURCES NOT AVAILABLE.
 P2     2 1 0           5 3 2
 PROCESS 1 HAS TO WAIT AS ENOUGH RESOURCES NOT AVAILABLE.
 PROCESS 3 HAS TO WAIT AS ENOUGH RESOURCES NOT AVAILABLE.
 P4     5 2 1           7 4 3
 P1     0 0 0           7 5 3
 P3     1 5 3           10 5 5
 P5     6 2 4           10 5 7
 THE ABOVE SEQUENCE IS A SAFE SEQUENCE.
```

OUTPUT SAMPLE 2:

```
ENTER THE NO OF PROCESSES:5
ENTER THE NO OF RESOURCES:4
ENTER TOTAL NO. OF INSTANCES OF RESOURCE 1:3
ENTER TOTAL NO. OF INSTANCES OF RESOURCE 2:14
ENTER TOTAL NO. OF INSTANCES OF RESOURCE 3:12
ENTER TOTAL NO. OF INSTANCES OF RESOURCE 4:12

*** PROCESS 1 ***
MAX CLAIM FOR RESOURCE 1:0
MAX CLAIM FOR RESOURCE 2:0
MAX CLAIM FOR RESOURCE 3:1
MAX CLAIM FOR RESOURCE 4:2
ALLOCATED INSTANCES OF RESOURCE 1:0
ALLOCATED INSTANCES OF RESOURCE 2:0
ALLOCATED INSTANCES OF RESOURCE 3:1
ALLOCATED INSTANCES OF RESOURCE 4:2
```

```
*** PROCESS 2 ***
MAX CLAIM FOR RESOURCE 1:1
MAX CLAIM FOR RESOURCE 2:7
MAX CLAIM FOR RESOURCE 3:5
MAX CLAIM FOR RESOURCE 4:0
ALLOCATED INSTANCES OF RESOURCE 1:1
ALLOCATED INSTANCES OF RESOURCE 2:0
ALLOCATED INSTANCES OF RESOURCE 3:0
ALLOCATED INSTANCES OF RESOURCE 4:0
*** PROCESS 3 ***
MAX CLAIM FOR RESOURCE 1:2
MAX CLAIM FOR RESOURCE 2:3
MAX CLAIM FOR RESOURCE 3:5
MAX CLAIM FOR RESOURCE 4:6
ALLOCATED INSTANCES OF RESOURCE 1:1
ALLOCATED INSTANCES OF RESOURCE 2:3
ALLOCATED INSTANCES OF RESOURCE 3:5
ALLOCATED INSTANCES OF RESOURCE 4:4
*** PROCESS 4 ***
MAX CLAIM FOR RESOURCE 1:0
MAX CLAIM FOR RESOURCE 2:6
MAX CLAIM FOR RESOURCE 3:5
MAX CLAIM FOR RESOURCE 4:2
ALLOCATED INSTANCES OF RESOURCE 1:0
ALLOCATED INSTANCES OF RESOURCE 2:6
ALLOCATED INSTANCES OF RESOURCE 3:3
ALLOCATED INSTANCES OF RESOURCE 4:2
*** PROCESS 5 ***
MAX CLAIM FOR RESOURCE 1:0
MAX CLAIM FOR RESOURCE 2:6
MAX CLAIM FOR RESOURCE 3:5
MAX CLAIM FOR RESOURCE 4:6
ALLOCATED INSTANCES OF RESOURCE 1:0
ALLOCATED INSTANCES OF RESOURCE 2:0
ALLOCATED INSTANCES OF RESOURCE 3:1
ALLOCATED INSTANCES OF RESOURCE 4:4

        MAX         ALLOCATION      NEEDED
P1      0 0 1 2     0 0 1 2         0 0 0 0
P2      1 7 5 0     1 0 0 0         0 7 5 0
P3      2 3 5 6     1 3 5 4         1 0 0 2
P4      0 6 5 2     0 6 3 2         0 0 2 0
P5      0 6 5 6     0 0 1 4         0 6 4 2
```

```
     TOTAL          AVAILABLE
     3 14 12 12          1 5 2 0

CHECKING FOR SAFE SEQUENCE...

         AVAIL-BEFORE    AVAIL-AFTER
P1       1 5 2 0         1 5 3 2
PROCESS 2 HAS TO WAIT AS ENOUGH RESOURCES NOT AVAILABLE.
P3       0 5 3 0         2 8 8 6
P2       2 1 3 6         3 8 8 6
P4       3 8 6 6         3 14 11 8
P5       3 8 7 6         3 14 12 12

THE ABOVE SEQUENCE IS A SAFE SEQUENCE.
```

CONCLUSION:
The program to simulate Banker's algorithm is executed and safe sequence is verified.

Exp No: 11

IMPLEMENTATION OF DEADLOCK DETECTION ALGORITHM

PROBLEM DEFINITION:
/* Banker's algorithm
This is a deadlock avoidance/prevention algorithm means avoid/prevent the happening of deadlock.
THEORETICAL BACKGROUND:
This is a resource allocation algorithm means allocate the resources in way in which deadlock should not occur. This is a deadlock detection algorithm means if there is no way to prevent deadlock, then say deadlock occurred. This algorithm was developed by Edsger Dijkstra. */
PROGRAM DEVELOPMENT:

```
#include <stdio.h>;
#include <conio.h>;
void main()
```

```
{
 int found,flag,l,p[4][5],tp,tr,c[4][5],i,j,k=1,m[5],r[5],a[5],temp[5],sum=0;
 clrscr();
 printf("Enter total no of processes");
 scanf("%d",&tp);
 printf("Enter total no of resources");
 scanf("%d",&tr);
 printf("Enter claim (Max. Need) matrix\n");
 for(i=1;i<=tp;i++)
 {
  printf("process %d:\n",i);
  for(j=1;j<=tr;j++)
  scanf("%d",&c[i][j]);
 }
 printf("Enter allocation matrix\n");
 for(i=1;i<=tp;i++)
 {
  printf("process %d:\n",i);
  for(j=1;j<=tr;j++)
  scanf("%d",&p[i][j]);
 }
 printf("Enter resource vector (Total resources):\n");
 for(i=1;i<=tr;i++)
 {
  scanf("%d",&r[i]);
 }
 printf("Enter availability vector (available resources):\n");
 for(i=1;i<=tr;i++)
 {
  scanf("%d",&a[i]);
  temp[i]=a[i];
 }

 for(i=1;i<=tp;i++)
 {
  sum=0;
  for(j=1;j<=tr;j++)
  {
   sum+=p[i][j];
  }
  if(sum==0)
  {
   m[k]=i;
   k++;
  }
 }
 for(i=1;i<=tp;i++)
 {
  for(l=1;l<k;l++)
  if(i!=m[l])
  {
   flag=1;
   for(j=1;j<=tr;j++)
   if(c[i][j]<temp[j])
    {
     flag=0;
     break;
    }
  }
  if(flag==1)
  {
   m[k]=i;
   k++;
   for(j=1;j<=tr;j++)
   temp[j]+=p[i][j];
  }
 }
 printf("deadlock causing processes are:");
 for(j=1;j<=tp;j++)
 {
  found=0;
  for(i=1;i<k;i++)
  {
   if(j==m[i])
   found=1;
  }
  if(found==0)
  printf("%d\t",j);
 }
 getch();
}.
```

SAMPLE INPUT & OUTPUT :

```
administrator@administrator-H310M-H-2-0:~/Desktop$ gcc test.c
administrator@administrator-H310M-H-2-0:~/Desktop$ ./a.out
Enter total no of processes 4
Enter total no of resources 5
Enter claim (Max. Need) matrix
process 1:
0 1 0 0 1
process 2:
0 0 1 0 1
process 3:
0 0 0 0 1
process 4:
1 0 1 0 1
Enter allocation matrix
process 1:
1 0 1 1 0
```

```
process 2:
1 1 0 0 0
process 3:
0 0 0 1 0
process 4:
0 0 0 0 0
Enter resource vector (Total resources):
2 1 1 2 1
Enter availability vector (available resources):
0 0 0 0 1
deadlock causing processes are:2 3
```

Exp No: 12

FILE ALLOCATION STRATEGIES

PROBLEM DEFINITION:

Simulate the following file allocation strategies.

a) Sequential b) Indexed c) Linked

THEORETICAL BACKGROUND:

A file is a collection of data, usually stored on disk. As a logical entity, a file enables to divide data into meaningful groups. As a physical entity, a file should be considered in terms of its organization. The term "file organization" refers to the way in which data is stored in a file and, consequently, the method(s) by which it can be accessed.

SEQUENTIAL FILE ALLOCATION

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record. A record of a sequential file can only be accessed by reading all the previous records.

LINKED FILE ALLOCATION

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.

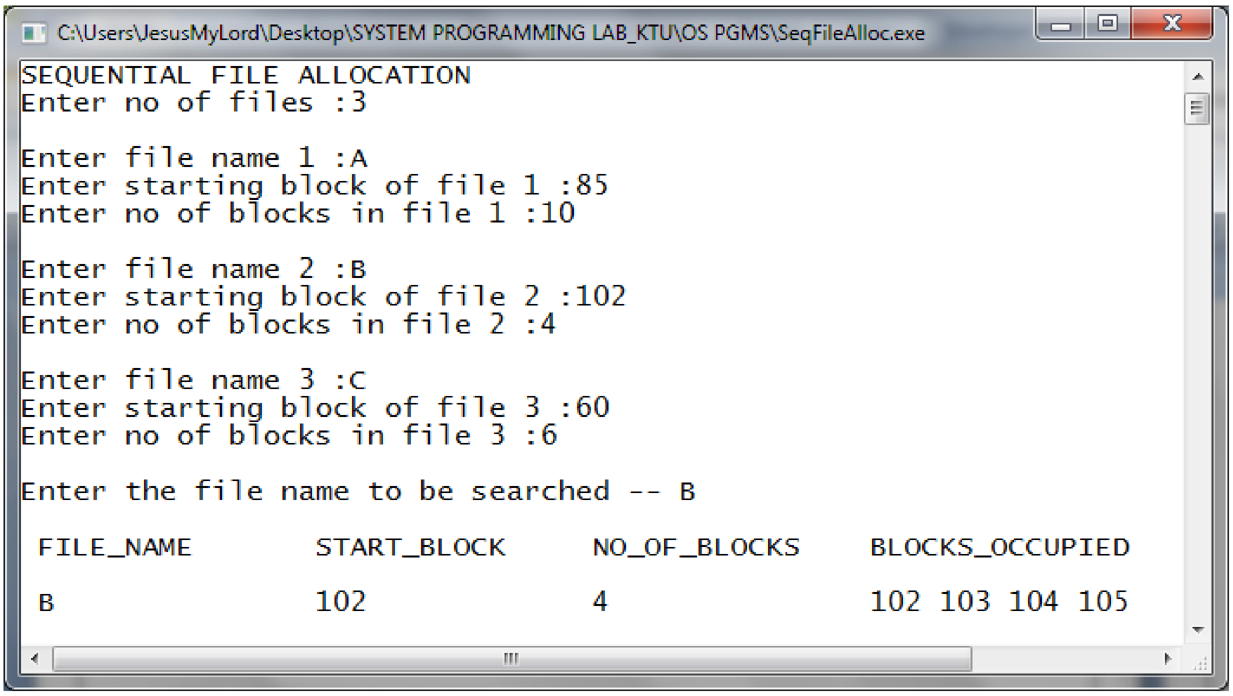
INDEXED FILE ALLOCATION

Indexed file allocation strategy brings all the pointers together into one location: an index block. Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block. To find and read the i^{th} block, the pointer in the i^{th} index-block entry is used.

PROGRAM DEVELOPMENT:

```
/*SEQUENTIAL FILE ALLOCATION*/
#include<stdio.h>
#include<conio.h>

struct fileTable
{
  char name[20]; int sb, nob;
}ft[30];

int main()
{
  int i, j, n;
  char s[20];
  printf("SEQUENTIAL FILE ALLOCATION\n");
  printf("Enter no of files :");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("\nEnter file name %d :",i+1);
    scanf("%s",ft[i].name);
    printf("Enter starting block of file %d :",i+1);
    scanf("%d",&ft[i].sb);
    printf("Enter no of blocks in file %d :",i+1);
    scanf("%d",&ft[i].nob);
  }
  printf("\nEnter the file name to be searched -- ");
  scanf("%s",s);
  for(i=0;i<n;i++)
  {
    if(strcmp(s, ft[i].name)==0)
      break;
  }
  if(i==n)
    printf("\nFile Not Found");
  else
  {
    printf("\n FILE_NAME\t START_BLOCK\t NO_OF_BLOCKS\t BOCKS_OCCUPIED\n");
    printf("\n %s\t\t %d\t\t %d\t\t ",ft[i].name,ft[i].sb,ft[i].nob);
    for(j=0;j<ft[i].nob;j++)
      printf("%d ",ft[i].sb+j);
  }
  getch();
}
```

SAMPLE INPUT AND OUTPUT:

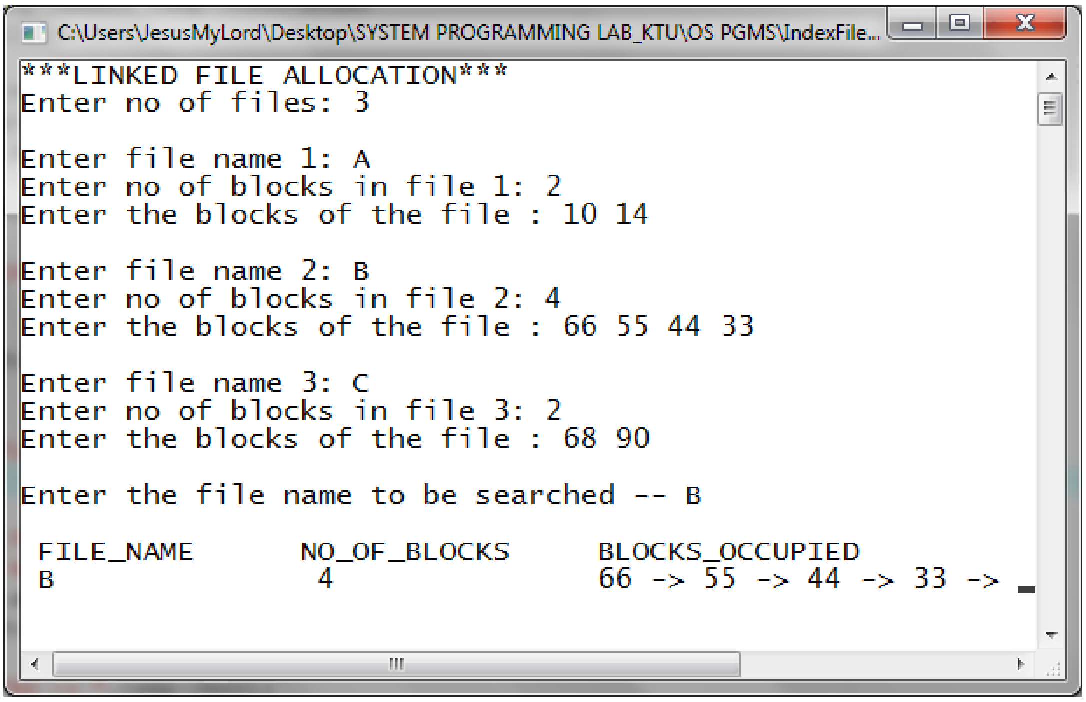


```
C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\SeqFileAlloc.exe
SEQUENTIAL FILE ALLOCATION
Enter no of files :3

Enter file name 1 :A
Enter starting block of file 1 :85
Enter no of blocks in file 1 :10

Enter file name 2 :B
Enter starting block of file 2 :102
Enter no of blocks in file 2 :4

Enter file name 3 :C
Enter starting block of file 3 :60
Enter no of blocks in file 3 :6

Enter the file name to be searched -- B

 FILE_NAME          START_BLOCK         NO_OF_BLOCKS        BLOCKS_OCCUPIED

 B                  102                 4                   102 103 104 105
```

PROGRAM DEVELOPMENT:

```
/*LINKED FILE ALLOCATION*/
#include<stdio.h>
#include<conio.h>

struct fileTable
{
  char name[20];
  int nob;
  struct block *sb;
}ft[30];

struct block
{
  int bno;
  struct block *next;
};

int main()
{
  int i, j, n;
  char s[20];
  struct block *temp;
  printf("***LINKED FILE ALLOCATION***\n");
  printf("Enter no of files: ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("\nEnter file name %d: ",i+1);
    scanf("%s",ft[i].name);
    printf("Enter no of blocks in file %d: ",i+1);
    scanf("%d",&ft[i].nob);
    ft[i].sb=(struct block*)malloc(sizeof(struct block));
    temp = ft[i].sb;
    printf("Enter the blocks of the file : ");
    scanf("%d",&temp->bno); temp->next=NULL;
    for(j=1;j<ft[i].nob;j++)
    {
      temp->next = (struct block*)malloc(sizeof(struct block));
      temp = temp->next;
      scanf("%d",&temp->bno);
    }
    temp->next = NULL;
  }
  printf("\nEnter the file name to be searched -- ");
  scanf("%s",s);
  for(i=0;i<n;i++)
    if(strcmp(s, ft[i].name)==0)
      break;
    if(i==n)
      printf("\nFile Not Found");
    else
    {
      printf("\n FILE_NAME\tNO_OF_BLOCKS\t BLOCKS_OCCUPIED");
      printf("\n %s\t\t %d\t\t ",ft[i].name,ft[i].nob);
      temp=ft[i].sb;
      for(j=0;j<ft[i].nob;j++)
      {
        printf("%d -> ",temp->bno);
        temp = temp->next;
      }
    }
  getch();
}
```

SAMPLE INPUT AND OUTPUT:

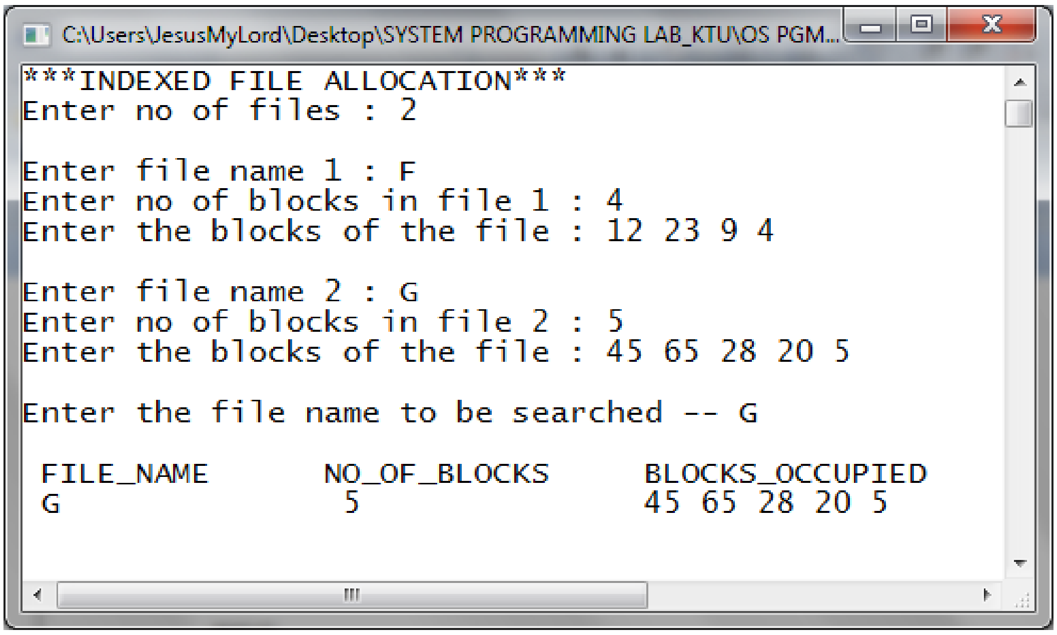

C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\IndexFile...

```
***LINKED FILE ALLOCATION***
Enter no of files: 3

Enter file name 1: A
Enter no of blocks in file 1: 2
Enter the blocks of the file : 10 14

Enter file name 2: B
Enter no of blocks in file 2: 4
Enter the blocks of the file : 66 55 44 33

Enter file name 3: C
Enter no of blocks in file 3: 2
Enter the blocks of the file : 68 90

Enter the file name to be searched -- B

 FILE_NAME       NO_OF_BLOCKS      BLOCKS_OCCUPIED
 B                4                66 -> 55 -> 44 -> 33 -> _
```

PROGRAM DEVELOPMENT:

```
/*INDEXED FILE ALLOCATION*/
#include<stdio.h>
#include<conio.h>

struct fileTable
{
  char name[20];
  int nob, blocks[30];
}ft[30];
int main()
{
   int i, j, n;
   char s[20];
   printf("***INDEXED FILE ALLOCATION***\n");
   printf("Enter no of files : ");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      printf("\nEnter file name %d : ",i+1);
      scanf("%s",ft[i].name);
      printf("Enter no of blocks in file %d : ",i+1);
      scanf("%d",&ft[i].nob);
      printf("Enter the blocks of the file : ");
      for(j=0;j<ft[i].nob;j++)
        scanf("%d",&ft[i].blocks[j]);
    }
    printf("\nEnter the file name to be searched -- ");
    scanf("%s",s);
    for(i=0;i<n;i++)
      if(strcmp(s, ft[i].name)==0)
        break;
      if(i==n)
        printf("\nFile Not Found");
      else
      {
        printf("\n FILE_NAME\tNO_OF_BLOCKS\t BLOCKS_OCCUPIED");
        printf("\n %s\t\t %d\t\t ",ft[i].name,ft[i].nob);
        for(j=0;j<ft[i].nob;j++)
          printf("%d ",ft[i].blocks[j]);
      }
    getch();
}
```

SAMPLE INPUT AND OUTPUT:



C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGM...

```
***INDEXED FILE ALLOCATION***
Enter no of files : 2

Enter file name 1 : F
Enter no of blocks in file 1 : 4
Enter the blocks of the file : 12 23 9 4

Enter file name 2 : G
Enter no of blocks in file 2 : 5
Enter the blocks of the file : 45 65 28 20 5

Enter the file name to be searched -- G

 FILE_NAME       NO_OF_BLOCKS      BLOCKS_OCCUPIED
 G                5                45 65 28 20 5
```

CONCLUSION:

The file allocation methods, sequential, linked and indexed, are simulated and verified.

Exp No: 13

DISK SCHEDULING

PROBLEM DEFINITION:

Implement programs to simulate the following disk scheduling algorithms.

a) FCFS b)SCAN c) C-SCAN

THEORETICAL BACKGROUND:

Disks provide a bilk of secondary storage. Disks come in various sizes, speed and information can be stored optically or magnetically.

Disk scheduling algorithms schedule the order of disk I/Os to maximize performance.Disk Scheduling is the process of deciding which of the cylinder request is in the ready queue is to be accessed next. The access time and the bandwidth can be improved by scheduling the servicing of disk I/O requests in good order.

Access Time:The access time has two major components: Seek time and Rotational Latency.

Seek Time: Seek time is the time for disk arm to move the heads to the cylinder containing the desired sector.

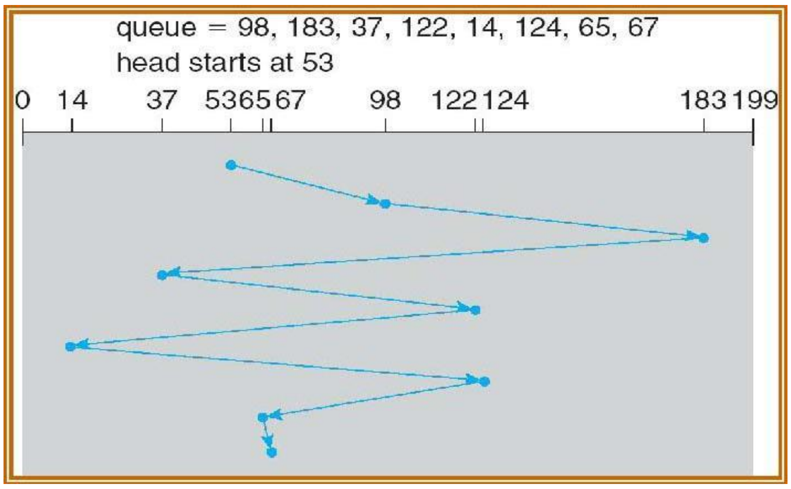
Rotational Latency:Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.

Bandwidth:The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Different types of scheduling algorithms are as follows:

FCFS scheduling algorithm:This is the simplest form of disk scheduling algorithm. Thisservices the request in the order they are received. This algorithm is fair but do not provide fastest service. It takes no special time to minimize the overall seek time.

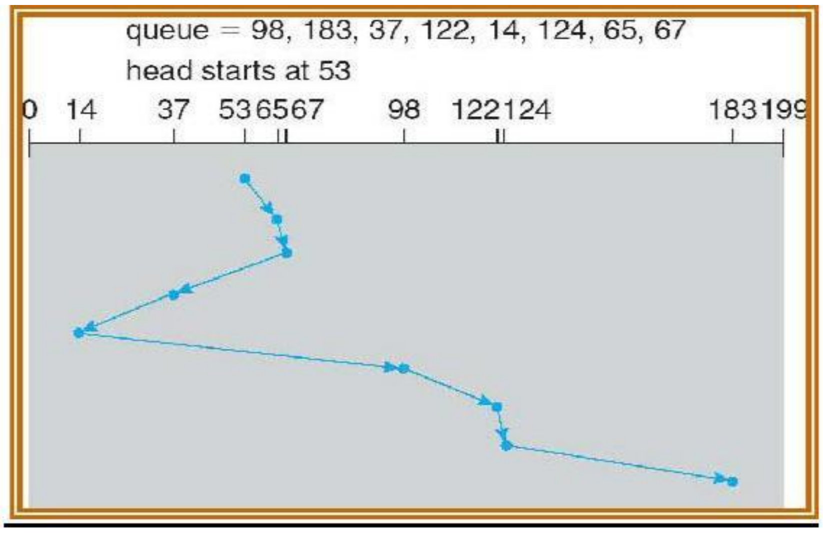
Eg:-consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



If the disk head is initially at 53, it will first move from 53 to 98 then to 183 and then to 37, 122, 14, 124, 65, 67 for a total head movement of 640 cylinders. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders

37 and 14 could be serviced together before or after 122 and 124 the total head movement could be decreased substantially and performance could be improved.

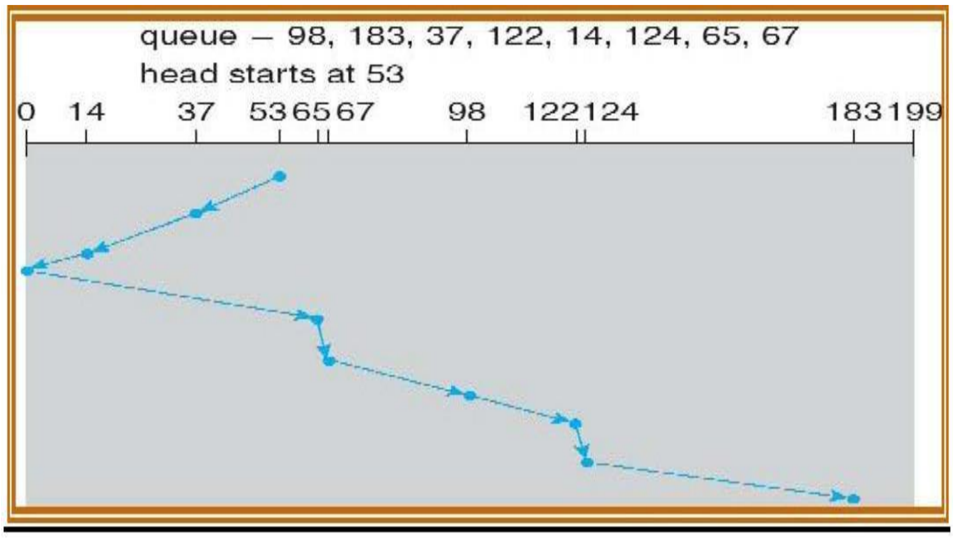
SSTF (Shortest seek time first) **algorithm**:This selects the request with minimum seek timefrom the current head position. Since seek time increases with the number of cylinders traversed by head, SSTF chooses the pending request closest to the current head position. *Eg*:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124,65, 67



If the disk head is initially at 53, the closest is at cylinder 65, then 67, then 37 is closer then 98 to 67. So it services 37, continuing we service 14, 98, 122, 124 and finally 183. The total head movement is only 236 cylinders. SSTF is essentially a form of SJF and it may cause starvation of some requests. SSTF is a substantial improvement over FCFS, it is not optimal.

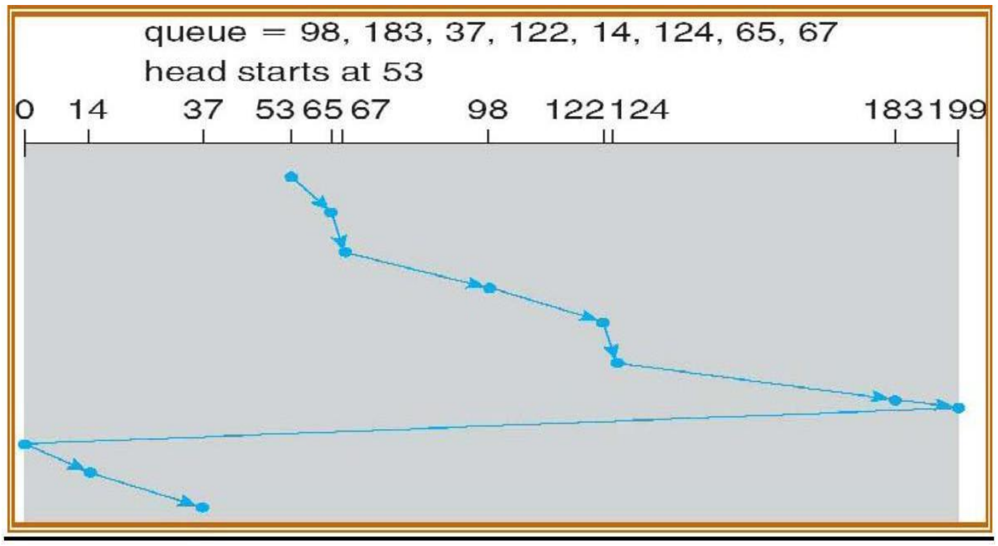
SCAN algorithm:In this the disk arm starts at one end of the disk and moves towards theother end, servicing the request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of the head movement is reversed and servicing continues. *Eg*:-:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14,124, 65, 67

If the disk head is initially at 53 and if the head is moving towards 0, it services 37 and then 14. At cylinder 0 the arm will reverse and will move towards the other end of the disk servicing 65, 67, 98, 122, 124 and 183. If a request arrives just in from of head, it will be serviced immediately and the request just behind the head will have to wait until the arms reach other end and reverses direction. The SCAN is also called as elevator algorithm.



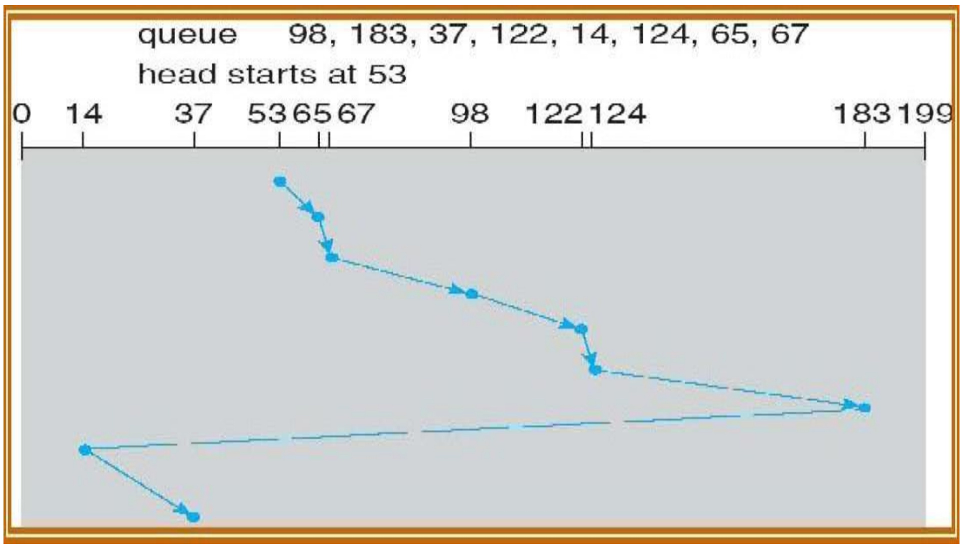
C-SCAN (Circular scan) **algorithm**:

C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from end of the disk to the other servicing the request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return. The C-SCAN treats the cylinders as circular list that wraps around from the final cylinder to the first one. *Eg*:-



LOOK Scheduling algorithm:

Both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice neither of the algorithms is implemented in this way. The arm goes only as far as the final request in each direction. Then it reverses, without going all the way to the end of the disk. These versions of SCAN and CSCAN are called Look and C-Look scheduling because they look for a request before continuing to move in a given direction. Eg:



Selection of Disk Scheduling Algorithm:

SSTF is common and it increases performance over FCFS.

SCAN and C-SCAN algorithm is better for a heavy load on disk.

SCAN and C-SCAN have less starvation problem.

SSTF or Look is a reasonable choice for a default algorithm.

ALGORITHM:

Step 1. Start the process

Step 2. Input the maximum number of cylinders and work queue

Step 3. Imput the disk head starting position and previous position

Step 4. **FCFS Scheduling**: The operations are performed in order requested.

Step 5. Scan from the first request till the last request

Step 6. Compute the seek time as the sum of absolute disk movements

Step 7. **SCAN Scheduling**:

Step 8. Sort the request work queue in order

Step 9. Check the previous position to find out right scan or leaft scan to be performed

Step 10. Accordingly, start at the current position and move the disk head to the last request in the direction, servicing each of them

Step 11. Then move to the farther end and start back servising the requests towards other direction

Step 12. Compute seek time by adding up the head movements

Step 13. **C-SCAN Scheduling**:

Step 14. Sort the request work queue in order

Step 15. Check the previous position to find out right scan or leaft scan to be performed

Step 16. Accordingly, start at the current position and move the disk head to the last request in the direction

Step 17. Then move to the farther end in the current direction and move back to the farther end in the opposite direction

Step 18. Start back with the requests towards the same direction

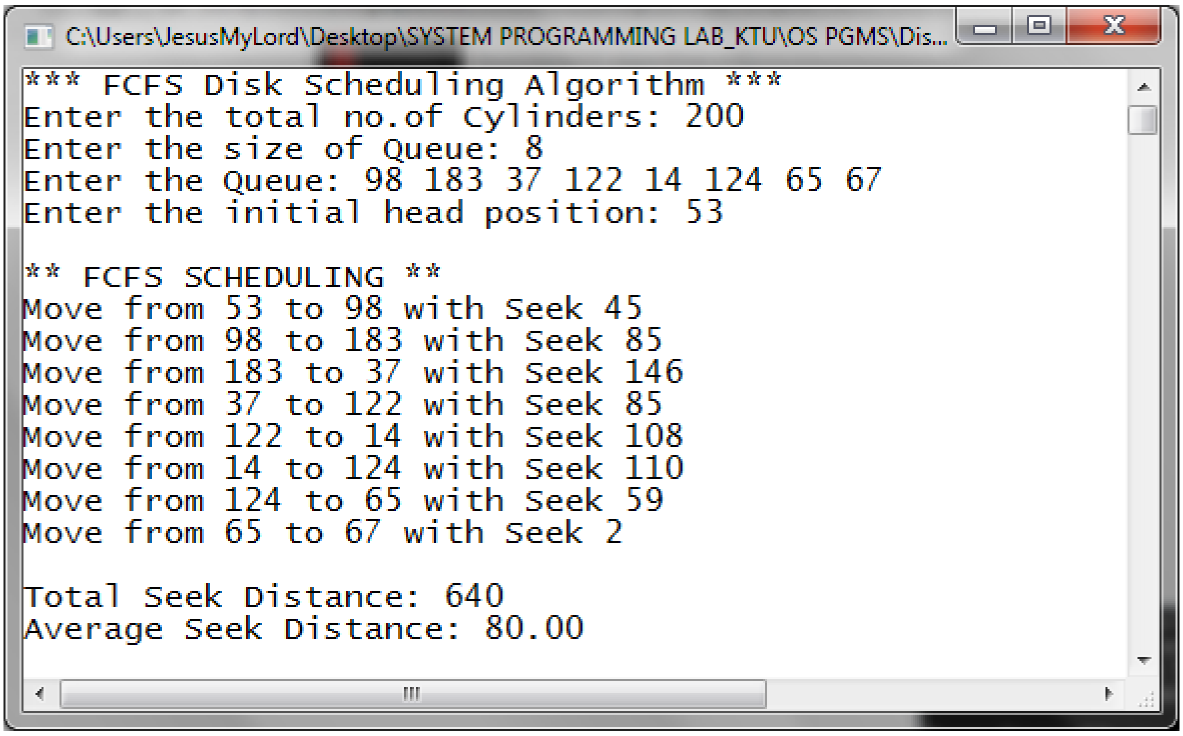
Step 19. Compute seek time by adding up the head movements

Step 20. Stop the process

PROGRAM DEVELOPMENT:

```
/*FCFS Disk Scheduling Mechanism*/
#include<stdio.h>
#include<conio.h>
int main()
{
   int n, cy, queue[100], head, seek=0, diff, I, j;
   float avg;

   printf("*** FCFS Disk Scheduling Algorithm ***\n");
   printf("Enter the total no.of Cylinders: ");
   scanf("%d",&cy);
   printf("Enter the size of Queue: ");
   scanf("%d",&n);
   printf("Enter the Queue: ");
   for(i=1; i<=n; i++)
   {
      scanf("%d",&queue[i]); //Read the seek positions to array
   }
   printf("Enter the initial head position: ");
   scanf("%d",&head);
   queue[0]=head; //Set the first seek location as head position
   //Scheduling
   printf("\n** FCFS SCHEDULING **\n");
   for(j=0; j<=n-1; j++)
   {
      diff = abs(queue[j+1] - queue[j]); //Calculate seek distance
      seek+=diff;
      printf("Move from %d to %d with Seek %d\n", queue[j], queue[j+1], diff);
   }
   printf("\nTotal Seek Distance: %d\t",seek);
   //Calculate average seek distance
   avg = seek/(float)n;
   printf("\nAverage Seek Distance: %.2f\t",avg);
   getch();
}
```

SAMPLE INPUT AND OUTPUT:


C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\Dis...

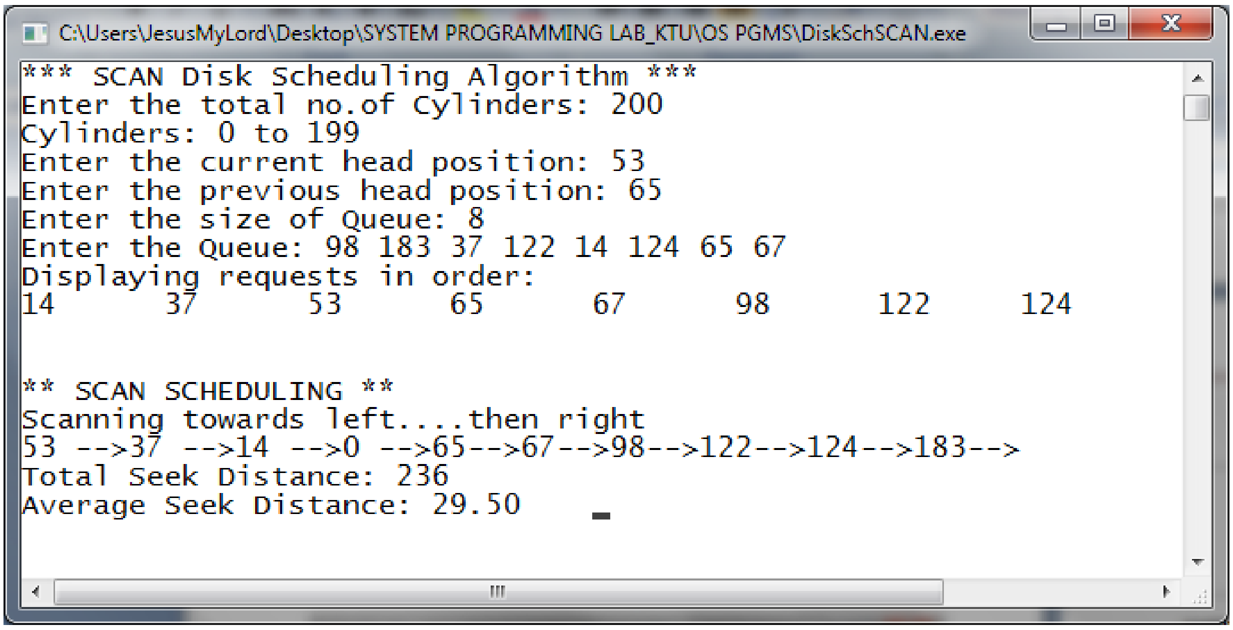
```
*** FCFS Disk Scheduling Algorithm ***
Enter the total no.of Cylinders: 200
Enter the size of Queue: 8
Enter the Queue: 98 183 37 122 14 124 65 67
Enter the initial head position: 53

** FCFS SCHEDULING **
Move from 53 to 98 with Seek 45
Move from 98 to 183 with Seek 85
Move from 183 to 37 with Seek 146
Move from 37 to 122 with Seek 85
Move from 122 to 14 with Seek 108
Move from 14 to 124 with Seek 110
Move from 124 to 65 with Seek 59
Move from 65 to 67 with Seek 2

Total Seek Distance: 640
Average Seek Distance: 80.00
```

PROGRAM DEVELOPMENT:

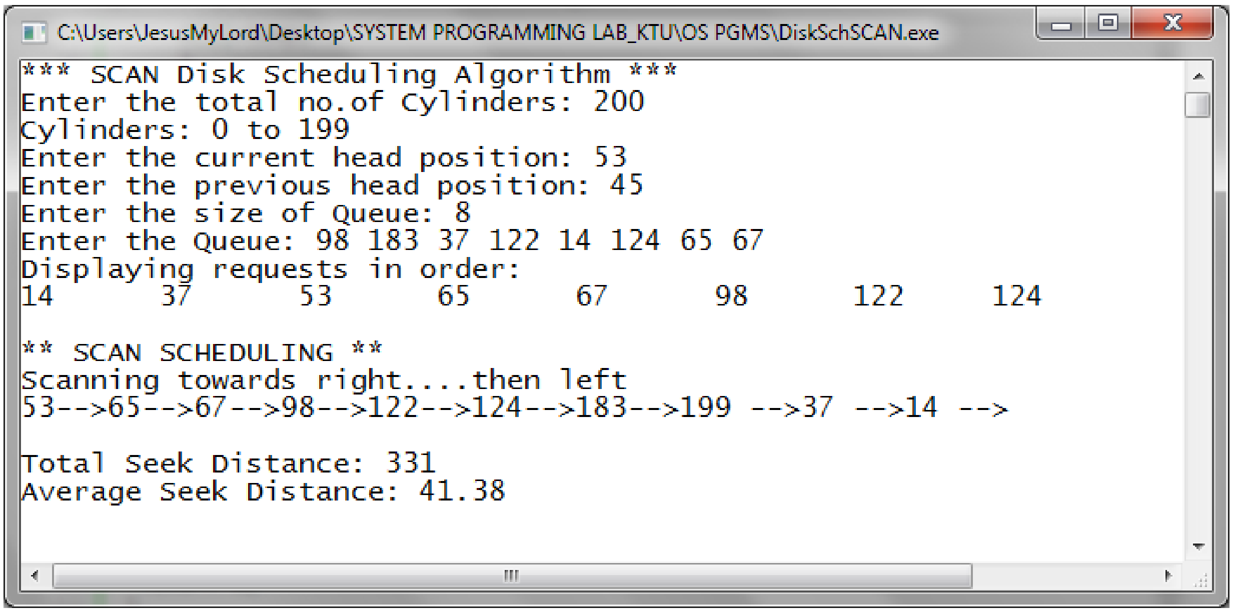
```
/*SCAN Disk Scheduling Mechanism*/
#include<stdio.h>
#include<conio.h>
int main()
{
   int n, cy, queue[100], head, prev, seek=0, diff;
   int i, j, temp, max, loc;
   float avg;
   int s1, s2;

   printf("*** SCAN Disk Scheduling Algorithm ***\n");
   printf("Enter the total no.of Cylinders: ");
   scanf("%d",&cy);
   max = cy-1;  //cylinders numbered 0 to max
   printf("Cylinders: 0 to %d\n", max);
   printf("Enter the current head position: ");
   scanf("%d",&head);
   queue[0]=head; //Set the first seek location as head position
   printf("Enter the previous head position: ");
   scanf("%d",&prev);
   printf("Enter the size of Queue: ");
   scanf("%d",&n);
   printf("Enter the Queue: ");
```

```
for(i=1; i<=n; i++)
{
    scanf("%d",&queue[i]); //Read the seek positions to array
}

//Sort the queue in ascending order
for(i=0; i<=n; i++)
{
   for(j=i+1; j<=n; j++)
   {
      if(queue[i] > queue[j])
      {
         temp = queue[i];
         queue[i] = queue[j];
         queue[j] = temp;
      }
   }
}
printf("Displaying requests in order:\n");
for(i=0; i<n; i++)
{
    printf("%d \t",queue[i]);
}
printf("\n\n");

printf("\n** SCAN SCHEDULING **\n");
// Find the location of head in array
for(i=0; i<=n; i++)
{
    if(head==queue[i])
    {
        loc = i;
        break;
    }
}
//Scan left - as previous location is greater than current head
if((head-prev) < 0)
{
    printf("Scanning towards left....then right\n");
    for(i=loc; i>=0; i--)
    {
        printf("%d -->", queue[i]);
        seek =  seek + abs(head - queue[i]);
        head = queue[i];
        }
        seek =  seek + abs(head - 0);
        head = 0;
        printf("0 -->");
        for(i=loc+1; i<=n; i++)
        {
            printf("%d-->",queue[i]);
            seek =  seek + abs(head - queue[i]);
            head = queue[i];
        }
        //or do this only
        //seek = queue[loc] + queue[n];
    }
    //Scan right - as previous location is less than current head
    else
    {
        printf("Scanning towards right....then left\n");
        for(i=loc; i<=n; i++)
        {
            printf("%d-->",queue[i]);
            seek =  seek + abs(head - queue[i]);
            head = queue[i];
        }
        seek =  seek + abs(head - max);
        head = max;
        printf("%d -->", max);
        for(i=loc-1; i>=0; i--)
        {
            printf("%d -->", queue[i]);
            seek =  seek + abs(head - queue[i]);
            head = queue[i];
        }
        //or do this only
        //seek = abs(queue[loc]-max) + abs(max-queue[0]);
    }
    printf("\n\nTotal Seek Distance: %d\t",seek);

    //Calculate average seek distance
    avg = seek/(float)n;
    printf("\nAverage Seek Distance: %.2f\t",avg);
    getch();
}
```

Sample Input and Output:


C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\DiskSchSCAN.exe

```
*** SCAN Disk Scheduling Algorithm ***
Enter the total no.of Cylinders: 200
Cylinders: 0 to 199
Enter the current head position: 53
Enter the previous head position: 65
Enter the size of Queue: 8
Enter the Queue: 98 183 37 122 14 124 65 67
Displaying requests in order:
14      37      53      65      67      98      122     124


** SCAN SCHEDULING **
Scanning towards left....then right
53 -->37 -->14 -->0 -->65-->67-->98-->122-->124-->183-->
Total Seek Distance: 236
Average Seek Distance: 29.50
```


C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\DiskSchSCAN.exe

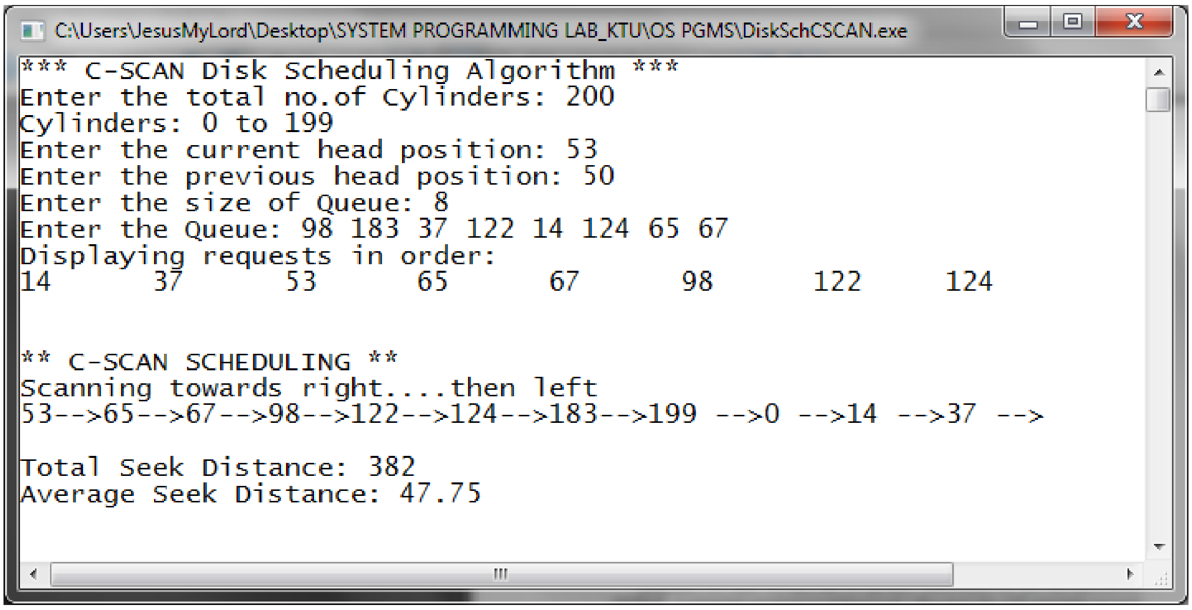
```
*** SCAN Disk Scheduling Algorithm ***
Enter the total no.of Cylinders: 200
Cylinders: 0 to 199
Enter the current head position: 53
Enter the previous head position: 45
Enter the size of Queue: 8
Enter the Queue: 98 183 37 122 14 124 65 67
Displaying requests in order:
14      37      53      65      67      98      122     124

** SCAN SCHEDULING **
Scanning towards right....then left
53-->65-->67-->98-->122-->124-->183-->199 -->37 -->14 -->

Total Seek Distance: 331
Average Seek Distance: 41.38
```

PROGRAM DEVELOPMENT:

```
/*C-SCAN Disk Scheduling Mechanism – CIRCULAR SCAN */
#include<stdio.h>
#include<conio.h>
int main()
{
   int n, cy, queue[100], head, prev, seek=0, diff;
   int i, j, temp, max, loc;
   float avg;
   int s1, s2;

   printf("*** C-SCAN Disk Scheduling Algorithm ***\n");
   printf("Enter the total no.of Cylinders: ");
   scanf("%d",&cy);
   max = cy-1;  //cylinders numbered 0 to max
   printf("Cylinders: 0 to %d\n", max);
   printf("Enter the current head position: ");
   scanf("%d",&head);
   queue[0]=head; //Set the first seek location as head position
   printf("Enter the previous head position: ");
   scanf("%d",&prev);
   printf("Enter the size of Queue: ");
   scanf("%d",&n);
   printf("Enter the Queue: ");
   for(i=1; i<=n; i++)
   {
      scanf("%d",&queue[i]); //Read the seek positions to array
   }
   //Sort the queue in ascending order
   for(i=0; i<=n; i++)
   {
     for(j=i+1; j<=n; j++)
     {
       if(queue[i] > queue[j])
       {
         temp = queue[i];
         queue[i] = queue[j];
         queue[j] = temp;
       }
     }
   }
   printf("Displaying requests in order:\n");
   for(i=0; i<n; i++)
   {
      printf("%d \t",queue[i]);
   }
   printf("\n\n");

   printf("\n** C-SCAN SCHEDULING **\n");
   // Find the location of head in array
   for(i=0; i<=n; i++)
   {
      if(head==queue[i])
      {
         loc = i;
         break;
       }
   }
   //Scan left - as previous location is greater than current head
   if((head-prev) < 0)
   {
       printf("Scanning towards left end....then restart at right end\n");
       for(i=loc; i>=0; i--)
       {
           printf("%d -->", queue[i]);
           seek =  seek + abs(head - queue[i]);
           head = queue[i];
       }
       //moving to 0
       seek =  seek + abs(head - 0);
       head = 0;
       printf("0 -->");
       //moving to max
       seek =  seek + max;
       head = max;
       printf("%d -->", max);
       for(i=n; i>=loc+1; i--)
       {
           printf("%d-->",queue[i]);
           seek =  seek + abs(head - queue[i]);
           head = queue[i];
       }
   }
   //Scan right - as previous location is less than current head
   else
   {
       printf("Scanning towards right end....then restart at left end\n");
       for(i=loc; i<=n; i++)
       {
           printf("%d-->",queue[i]);
           seek =  seek + abs(head - queue[i]);
           head = queue[i];
       }
       //moving to max
       seek =  seek + abs(head - max);
       head = max;
       printf("%d -->", max);
       //moving to 0
       seek =  seek + abs(head - 0);
       head = 0;
```

```
        printf("%d -->", 0);
        for(i=0; i<=loc-1; i++)
        {
            printf("%d -->", queue[i]);
            seek =  seek + abs(head - queue[i]);
            head = queue[i];
        }
    }
    printf("\n\nTotal Seek Distance: %d\t",seek);

    //Calculate average seek distance
    avg = seek/(float)n;
    printf("\nAverage Seek Distance: %.2f\t",avg);
    getch();
}
```

SAMPLE INPUT AND OUTPUT:



```
*** C-SCAN Disk Scheduling Algorithm ***
Enter the total no.of Cylinders: 200
Cylinders: 0 to 199
Enter the current head position: 53
Enter the previous head position: 50
Enter the size of Queue: 8
Enter the Queue: 98 183 37 122 14 124 65 67
Displaying requests in order:
14      37      53      65      67      98      122     124


** C-SCAN SCHEDULING **
Scanning towards right....then left
53-->65-->67-->98-->122-->124-->183-->199 -->0 -->14 -->37 -->

Total Seek Distance: 382
Average Seek Distance: 47.75
```

CONCLUSION:

The disk scheduling algorithems, FCFS, SCAN and C-SCAN are simulated and the seek times are verified.

ADDITIONAL
&
OPEN ENDED EXPERIMENTS

Exp No: 14

PRODUCER CONSUMER PROBLEM USING SEMAPHORES

PROBLEM DEFINITION:

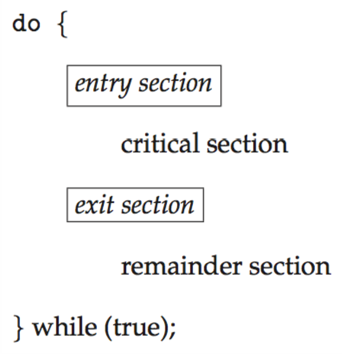
To implement a C program to simulate producer-consumer problem using semaphores.

THEORETICAL BACKGROUND:

CRITICAL SECTION PROBLEM:

Consider system of *n* processes {p_0, p_1, ... p_{n-1}}. Each process has critical section segment of code. Such a segment is a region of code where the shared resources are accessed – eg. changing common variables, updating table, writing file, etc. When one process in critical section, no other may be there; No two processes are executing in their c-s at the same time.*Critical section problem* is to design protocol to solve this. Each process must ask permission to enter critical section in entry section, critical section may follow with exit section, then remainder section.

General structure of process P_i



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to Critical-Section Problem

A solution to the c-s problem must satisfy the three requirements: mutual exclusion, progress and bounded wait.

SEMAPHORE:

Semaphore is a method for Inter-Process Communication (IPC). It may be used to provide exclusive access to resources on the current machine, or to limit the number of processes that may simultaneously use a resource. i.e. it is a synchronization-tool. It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.

A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations: **wait**() and **signal**(). wait() is termed P ("to test") and signal() is termed V ("to increment").

Definition of the **wait**() operation

```
wait(S) {
   while (S <= 0)
     ; // busy wait while someone is already using
   S--;  //decrement and enter the c-s
}
```

Definition of the **signal**() operation

```
signal(S) {
   S++;  //increment and make it free for use
}
```

All modifications to the integer value of S in the wait() and signal() operations must be executed **indivisibly**. When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value. Also, in the case of wait(S), following 2 operations must be executed without interruption:

Testing of S(S<=0) and
Modification of S (S--)

Binary Semaphore: holds integer value of **0 and 1.** Value 0 means semaphore is locked. 1 means it is unlocked.

- Can solve various synchronization problems using this.

Counting Semaphore: Integer value can range over an unrestricted domain.

IMPLEMENTATION USING C:
Refer 'Semaphore Operations' in Open Ended Section

THE BOUNDED-BUFFER PROBLEM (OR) PRODUCER CONSUMER PROBLEM:
The bounded-buffer problem is also known as the producer consumer problem. Two processes, Producer and Consumer are executing concurrently in a system. There is a pool of n buffers, each capable of holding one item. Producer and consumer share a common, *fixed-size buffer (bounded buffer)* to store and retrieve information.

- Producer *puts (produces) information* into the buffer and consumer *takes it out (consumes)*.

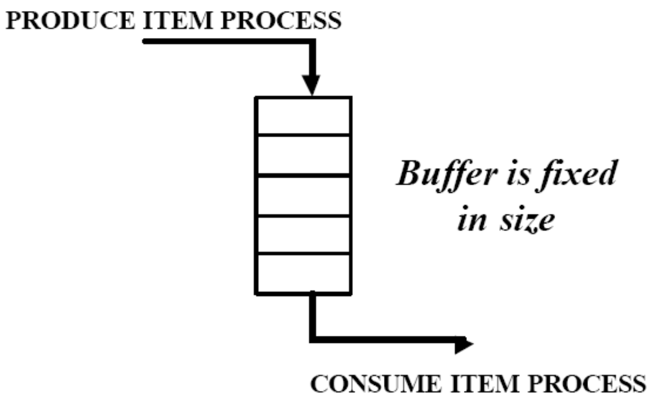


Fig: Bounded Buffer

- The operations of producer and consumer **must be synchronized** so that producerand consumer **should not be allowed to access the buffer at the same time.**

- Must ensure that the buffer does not over or underflow.
  (1) Overflow: Producer wants to put a new item in buffer, but it is already full.
  Producer goes to *sleep* and is *awakened* when the consumer has removed one or more items.
  (2) Underflow: Consumer wants to remove an item from the buffer but the buffer is empty.

Consumer goes to *sleep* until the producer puts something in the buffer and *wakes* it up.

SEMAPHORE USAGE AND INITIALIZATION:

Three semaphores namely **mutex**, **empty** and **full** can be used to solve the bounded buffer problem.

The producer and consumer processes share the following data structures:

- A pool of n buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1,
  provides mutual exclusion for accesses to the buffer pool
- Semaphore **full** initialized to the value 0
  count the number of full buffers; initially no items.
- Semaphore **empty**initialized to the value n
  count the number of empty buffers; initially, all are empty.

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

The case of producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer is as described below.

The structure of the producer process:

```
      do {
     ...
      /* produce an item in next_produced */
     ...
   wait(empty);   //decrement empty count when producing
   wait(mutex);  //lock the c-s before producing to n
     ...
      /* add next produced to the buffer */
     ...
   signal(mutex); //release lock when exiting from c-s
   signal(full); //increment full count when produced
 } while (true);
```

The structure of the consumer process:

```
      do {
   wait(full); //decrement full count when consuming
```

```
    wait(mutex); //lock the c-s before producing to n
      ...
        /* remove an item from buffer to next_consumed */
      ...
    signal(mutex); //release lock when exiting from c-s
    signal(empty); //increment empty count when consumed
      ...
        /* consume the item in next consumed */
      ...
      } while (true);
```

ALGORITHM:

Step 1: Start the process

Step 2: Generate keys using ftok() and create semaphores mutex, empty and full using semget()

Step 3: Initialize the semaphores; mutex as 1, full as 0 and empty as N using semctl()

Step 4: Create a child process using fork()

Step 5: If the process executing is parent, for i=1 to N, repeat steps 5.1 to 5.5 to produce items. Else go to Step 6.

Step 5.1: Decrement the empty count

Step 5.2: Decrement and lock mutex

Step 5.3: Produce ith item to buffer

Step 5.4: Increment and unlock mutex

Step 5.5: Increment full count

Step 6: In child process, for i=1 to N, repeat steps 6.1 to 6.5 to consume items.

Step 6.1: Decrement the full count

Step 6.2: Decrement and lock mutex

Step 6.3: Consume ith item from buffer

Step 6.4: Increment and unlock mutex

Step 6.5: Increment empty count

Step 7: Stop the process

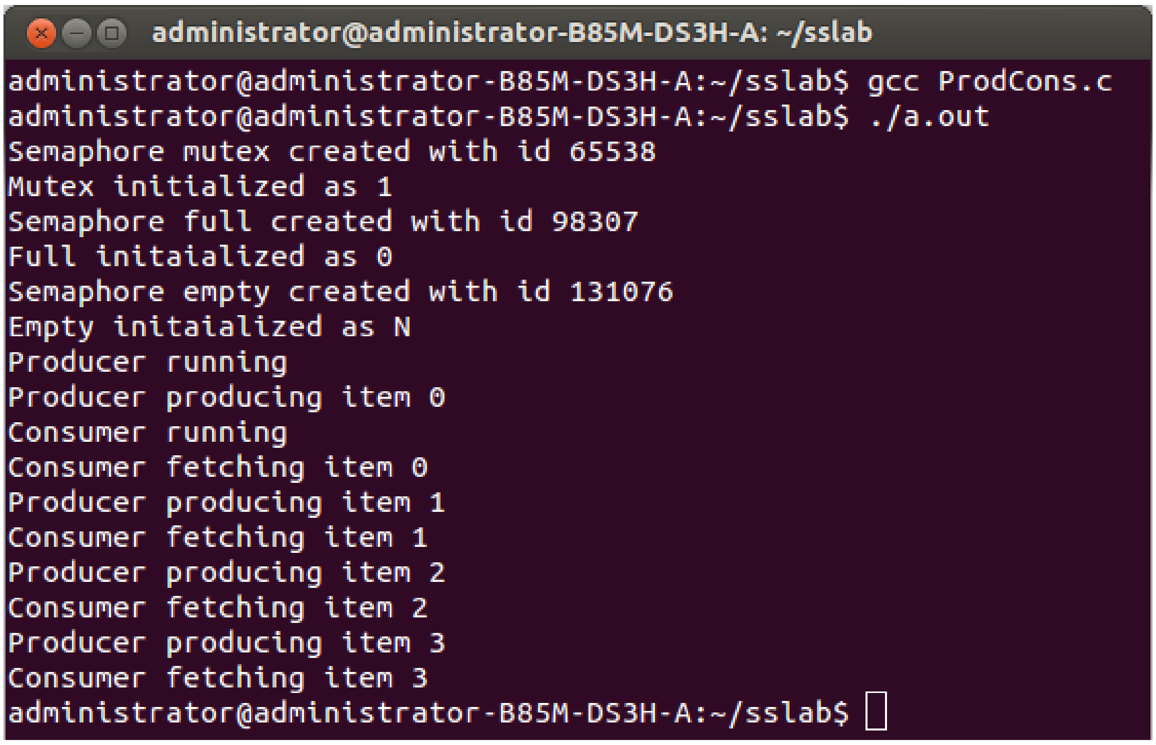
PROGRAM DEVELOPMENT:

```
/*Program to illustrate Producer Consumer Problem*/
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<error.h>
#define N 4
main( )
{
        int mutex, full, empty, pid, i;
```

```
key_t key;

//Declaring functions to increment and decrement semaphores
void down(int );
void up(int );

//Creating multex semaphore for synchronization
key = ftok("eg.txt", 'b');
mutex = semget(key, 1, IPC_CREAT | 0666);
printf("Semaphore mutex created with id %d\n",mutex);
semctl(mutex ,0, SETVAL, 1);
printf("Mutex initialized as 1\n");

//Creating full semaphore for counting full buffers
key = ftok("eg.txt", 'd');
full = semget(key, 1, IPC_CREAT | 0666);
printf("Semaphore full created with id %d\n",full);
semctl(full, 0, SETVAL, 0);
printf("Full initaialized as 0\n");

//Creating empty semaphore for counting empty buffers
key = ftok("eg.txt", 'e');
empty = semget(key, 1, IPC_CREAT | 0666);
printf("Semaphore empty created with id %d\n",empty);
semctl(empty, 0, SETVAL, N);
printf("Empty initaialized as N\n");
fflush(stdout);

//Creating new process; parent as producer and child as consumer
pid = fork( );

//Producer process
if(pid > 0)
{
printf("Producer running\n");
        for(i=0;i<N;i++)
        {
down(empty); //decrement empty count
down(mutex); //lock mutex
printf("Producer producing item %d\n", i);
up(mutex); //unlock mutex
up(full); //increment full count
                sleep(2);
        }
```

```
            }
            //Consumer process
            else if(pid == 0)
            {
            printf("Consumer running\n");
                        for(i=0;i<N;i++)
                        {
            down(full);//decrement full count
            down(mutex);//lock mutex
            printf("Consumer fetching item %d\n", i);
            up(mutex); //unlock mutex
            up(empty); //increment empty count
                                    sleep(2);
                        }
            }
            else
               printf("Process creation error");
}

//Decrement semphore
void down(int id)
{           struct sembuf sb[1] = {0,-1,0};
            if(semop(id, &sb[0], 1) == -1)
            {           printf("Error\n");
            }
}

//Increment semphore
void up(int id)
{           struct sembuf sb[1] = {0,1,0};
            if(semop(id, &sb[0], 1) == -1)
            {           printf("Error\n");
            }
}
```

SAMPLE INPUT AND OUTPUT:



administrator@administrator-B85M-DS3H-A: ~/sslab

```
administrator@administrator-B85M-DS3H-A:~/sslab$ gcc ProdCons.c
administrator@administrator-B85M-DS3H-A:~/sslab$ ./a.out
Semaphore mutex created with id 65538
Mutex initialized as 1
Semaphore full created with id 98307
Full initaialized as 0
Semaphore empty created with id 131076
Empty initaialized as N
Producer running
Producer producing item 0
Consumer running
Consumer fetching item 0
Producer producing item 1
Consumer fetching item 1
Producer producing item 2
Consumer fetching item 2
Producer producing item 3
Consumer fetching item 3
administrator@administrator-B85M-DS3H-A:~/sslab$ 
```

CONCLUSION:

The program to solve producer consumer problem is implemented using semaphores.

Exp No: 15

DINING PHILOSPHERS PROBLEM USING SEMAPHORES

PROBLEM DEFINITION:
To implement a C program that simulates the solution for Dining philosopher's problem.

THEORETICAL BACKGROUND:
The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes in a deadlock-free and starvation-free manner.

Dining Philospher's Problem: Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.

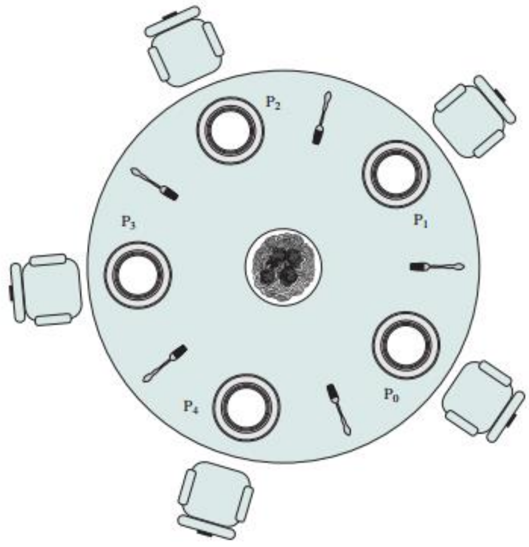


Fig: Representation of Dining Philosophers

At any instant, a philosopher is either eating or thinking. They do not interact with their neighbors while thinking, occasionally gets hungry and try to pick up 2 chopsticks (one chopstick at a time) to eat from bowl. Thus, an eating philosopher uses two chopsticks - one from their left and one from their right. When a philosopher finishes eating, he keeps down both chopsticks at their original place.

- Need both to eat, then release both only when done.
- Cannot pick up a chopstick that is already in the hand of a neighbour.

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating. This problem is the representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

Solution: Represent each chopstick with a semaphore.

A philosopher tries to grab a chopstick by executing a wait() operation on the left and right chopstick semaphores. He releases chopsticks by executing the signal() operation on the appropriate semaphores. In the case of 5 philosophers, the shared data are

- Bowl of rice (data set)
- Semaphore chopstick [5] , each initialized to 1

The code for each philosopher looks like:

```
semaphore chopstick[5];
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
        // mod is used because if i=5, next chopstick
                is 1 (dining table is circular)
        . . .
        /* eat for awhile */
        . . .
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        . . .
        /* think for awhile */
        . . .
} while (true);
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down. This solution uses only boolean semaphores. There is one global semaphore to provide mutual exclusion for exectution of critical protocols. There is one semaphore for each chopstick. In addition, a local two-phase prioritization scheme is used, under which philosophers defer to their neighbors who have declared themselves "hungry." All arithmetic is modulo 5.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Chopsticks must be picked up only in a critical section.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

ALGORITHM:

```
Step 1: Start the process
Step 2: Define the number of philosophers, N as 5
Step 3: Define states THINKING as 0, HUNGRY as 1 and EATING as 2
Step 4: Define LEFT as (i+4)%N and RIGHT as (i+1)%N
Step 5: Generate key and create mutex semaphore for synchronization and set value to 1
Step 7: For i=0 to N-1, repeat steps 7.1 To 7.3
    Step 7.1: Set philosopher i to THINKING state.
    Step 7.2: Generate keys and create semaphore array representing each chopstick
    Step 7.3: Set value of each semaphore to unlocked state 1
Step 8: For i=0 to N-1, repeat steps 8.1 To 8.2
    Step 8.1: When a philosopher is hungry, check if chopsticks on both sides are free. If
    free, goto step 8.1.1, else goto step 8.2
            Step 8.1.1: Change state to HUNGRY
            Step 8.1.2: Decrement and lock mutex
            Step 8.1.3: Acquire and lock both ith and LEFT chopsticks
            Step 8.1.4: Change state to EATING
            Step 8.1.5: Unlock and restore the ith and LEFT chopsticks
            Step 8.1.6: Increment and unlock mutex
            Step 8.1.7: Change state back to THINKING
    Step 8.2: If chopsticks aren't free, change state to HUNGRY and wait till the sticks are
    available
Step 9: Stop the process
```

PROGRAM DEVELOPMENT:

```
/*Program to illustrate dining philosophers problem and solution*/
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<error.h>

#define N 5
#define LEFT (i+4)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int mutex, s[N], state[N];
char ch='a';
void up(int);
void down(int);
void testit(int);
void take_forks(int i);
void put_forks(int i);
void main()
{
   int i;
   struct sembuf sb[1]={0,-1,0};
   //Generate key
   key_t key=ftok("eg.txt",ch);
   if(key==-1)
      printf("\n Error in ftok");
   //Creating multex semaphore for synchronization and set value to 1 - unlocked
   if((mutex=semget(key,1,IPC_CREAT|0666))==-1)
      printf("\n Error in semget");
   else
      printf("\n CREATED A SEMAPHORE MUTEX WITH ID = %d", mutex);
   if(semctl(mutex,0,SETVAL,1)==-1)
      printf("\n Error in semctl");
   //Creating semaphore array for each chopstick and set value to 1 - unlocked
   for(i=0;i<N;i++)
   {
         //Initializing all philosophers to thinking state
         state[i]=THINKING;
         ch++;
         if(key=ftok("eg.txt",ch)==-1)
                  printf("\n Error in ftok");
         if((s[i]=semget(key,1,IPC_CREAT|0666))==-1)
         printf("\n Error in semget");
         else
        printf("\n CREATED A SEMAPHORE s[%d] WITH ID = %d", i, s[i]);
         if(semctl(s[i],0,SETVAL,1)==-1)
                  printf("\n ERROR IN SEMCTL");
   }
   //Switching between thinking and eating state for each philosopher
   for(i=0;i<N;i++)
   {
         printf("\n\n PHILOSOPHER %d THINKS",i);
         take_forks(i);
         printf("\n PHILOSOPHER %d FINISHED EATING",i);
         put_forks(i);
         printf("\n HE HAS PUT THE FORKS DOWN\n");
   }
}
//Going to eat, use mutex semaphore for synchronization
void take_forks(int i)
{
  down(mutex);
  state[i]=HUNGRY;
  printf("\n PHILOSOPHER %d IS HUNGRY",i);
  printf("\n HE THEN TAKES THE FORK");
  testit(i);  //lock the chopstick semaphores
  up(mutex);
}
//Check and lock the semaphores for eating
void testit(int i)
{
  if(state[i]==HUNGRY && state[LEFT]!=EATING && state[i]!=EATING)
  {
        state[i]=EATING;
        down(s[i]);
        printf(" \n Locking the semaphore s[%d]",i);
        down(s[LEFT]);
        printf(" \n Locking the semaphore s[%d]",LEFT);
        printf("\n PHILOSOPHER %d EATS",i);
  }
}
//Finished eating, back to thinking, use mutex semaphore for synchronization
void put_forks(int i)
{
  down(mutex);
  state[i]=THINKING;
  printf("\n PHILOSOPHER %d IS BACK TO THINKING",i);
  //Unlock the chopstick semaphores
  up(s[i]);
  printf(" \n Unlocking the semaphore s[%d]",i);
  up(s[LEFT]);
  printf(" \n Unlocking the semaphore s[%d]",LEFT);
  up(mutex);
}
//Decrement semphore
void down(int id)
{
   struct sembuf sb[1]={0,-1,0};
   if(semop(id,&sb[0], 1)==1)
   {
      printf("\n ERROR IN SEMOP DOWN");
   }
}
//Increment semphore
void up(int id)
```

```
{
   struct sembuf sb[1]={0,1,0};
   if(semop(id,&sb[0],1)==1)
   {
       printf("\n ERROR IN SEMOP UP");
   }
}
```

SAMPLE INPUT AND OUTPUT:

```
administrator@administrator-B85M-DS3H-A:~/sslab$ gcc DiningPhilosopher.c
administrator@administrator-B85M-DS3H-A:~/sslab$ ./a.out
 CREATED A SEMAPHORE MUTEX WITH ID = 196613
 CREATED A SEMAPHORE s[0] WITH ID = 393227
 CREATED A SEMAPHORE s[1] WITH ID = 425996
 CREATED A SEMAPHORE s[2] WITH ID = 458765
 CREATED A SEMAPHORE s[3] WITH ID = 491534
 CREATED A SEMAPHORE s[4] WITH ID = 524303

 PHILOSOPHER 0 THINKS
 PHILOSOPHER 0 IS HUNGRY
 HE THEN TAKES THE FORK
 Locking the semaphore s[0]
 Locking the semaphore s[4]
 PHILOSOPHER 0 EATS
 PHILOSOPHER 0 FINISHED EATING
 PHILOSOPHER 0 IS BACK TO THINKING
 Unlocking the semaphore s[0]
 Unlocking the semaphore s[4]
 HE HAS PUT THE FORKS DOWN

 PHILOSOPHER 1 THINKS
 PHILOSOPHER 1 IS HUNGRY
 HE THEN TAKES THE FORK
 Locking the semaphore s[1]
 Locking the semaphore s[0]
 PHILOSOPHER 1 EATS
 PHILOSOPHER 1 FINISHED EATING
 PHILOSOPHER 1 IS BACK TO THINKING
 Unlocking the semaphore s[1]
 Unlocking the semaphore s[0]
 HE HAS PUT THE FORKS DOWN

 PHILOSOPHER 2 THINKS
 PHILOSOPHER 2 IS HUNGRY
 HE THEN TAKES THE FORK
 Locking the semaphore s[2]
 Locking the semaphore s[1]
 PHILOSOPHER 2 EATS
 PHILOSOPHER 2 FINISHED EATING
 PHILOSOPHER 2 IS BACK TO THINKING
 Unlocking the semaphore s[2]
 Unlocking the semaphore s[1]
 HE HAS PUT THE FORKS DOWN

 PHILOSOPHER 3 THINKS
 PHILOSOPHER 3 IS HUNGRY
 HE THEN TAKES THE FORK
 Locking the semaphore s[3]
 Locking the semaphore s[2]
 PHILOSOPHER 3 EATS
 PHILOSOPHER 3 FINISHED EATING
 PHILOSOPHER 3 IS BACK TO THINKING
 Unlocking the semaphore s[3]
 Unlocking the semaphore s[2]
 HE HAS PUT THE FORKS DOWN

 PHILOSOPHER 4 THINKS
 PHILOSOPHER 4 IS HUNGRY
 HE THEN TAKES THE FORK
 Locking the semaphore s[4]
 Locking the semaphore s[3]
 PHILOSOPHER 4 EATS
 PHILOSOPHER 4 FINISHED EATING
 PHILOSOPHER 4 IS BACK TO THINKING
 Unlocking the semaphore s[4]
 Unlocking the semaphore s[3]
 HE HAS PUT THE FORKS DOWN
administrator@administrator-B85M-DS3H-A:~/sslab$
```

CONCLUSION:

The program to simulate Dining Philosophers problem is developed and demonstrated successfully.

Exp No: 16

DIRECTORY STRUCTURES

PROBLEM DEFINITION:

Simulate the following file organization techniques *
a) Single level directory b) Two level directory c) Hierarchical

THEORETICAL BACKGROUND:

The directory structure is the organization of files into a hierarchy of folders. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

DIECRTORY OPERATIONS:

Operations that are to be performed on a directory:

• **Search for a file**. User must be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, system may want to be able to find all files whose names match a particular pattern.

• **Create a file**. Newfiles need to be created and added to the directory.

• **Delete a file**.When a file is no longer needed,user must be able to remove it from the directory.

• **List a directory**. User must be able to list the files in a directory and the contents of the directory entry for each file in the list.

• **Rename a file**. Because the name of a file represents its contents to its users, they must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

• **Traverse the file system**.User may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, this is done by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

SINGLE LEVEL DIRECTORY:

In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory.

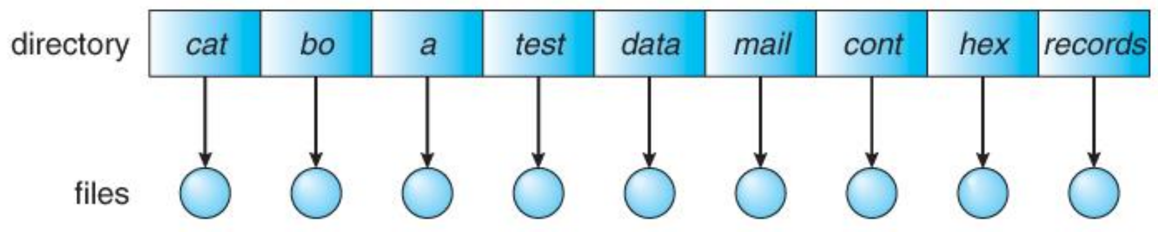


Fig. Single Level Directory

TWO LEVEL DIRECTORY:

In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched. This effectively solves the name collision problem and isolates users from one another.

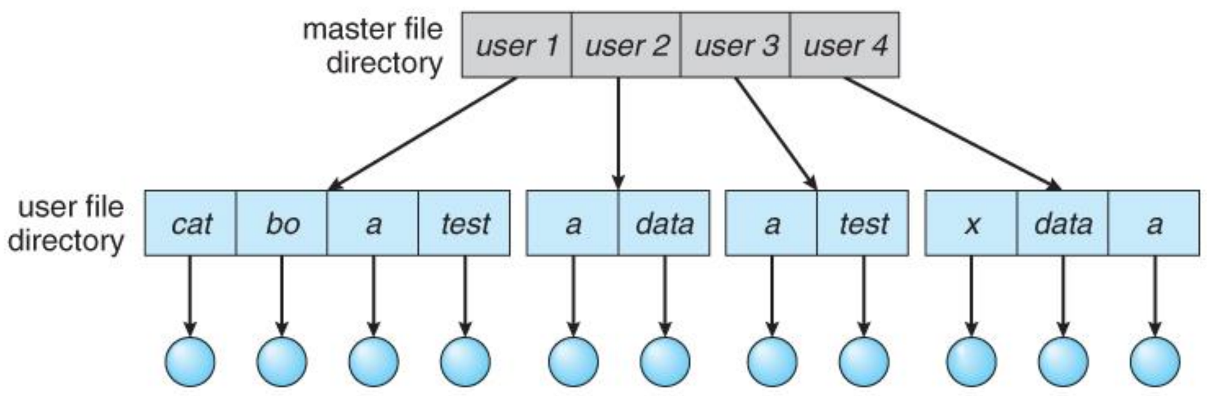


Fig. Two Level Directory

HIERARCHICAL STRUCTURE:

Hierarchical directory structure allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A directory (or subdirectory) contains a set of files or subdirectories.

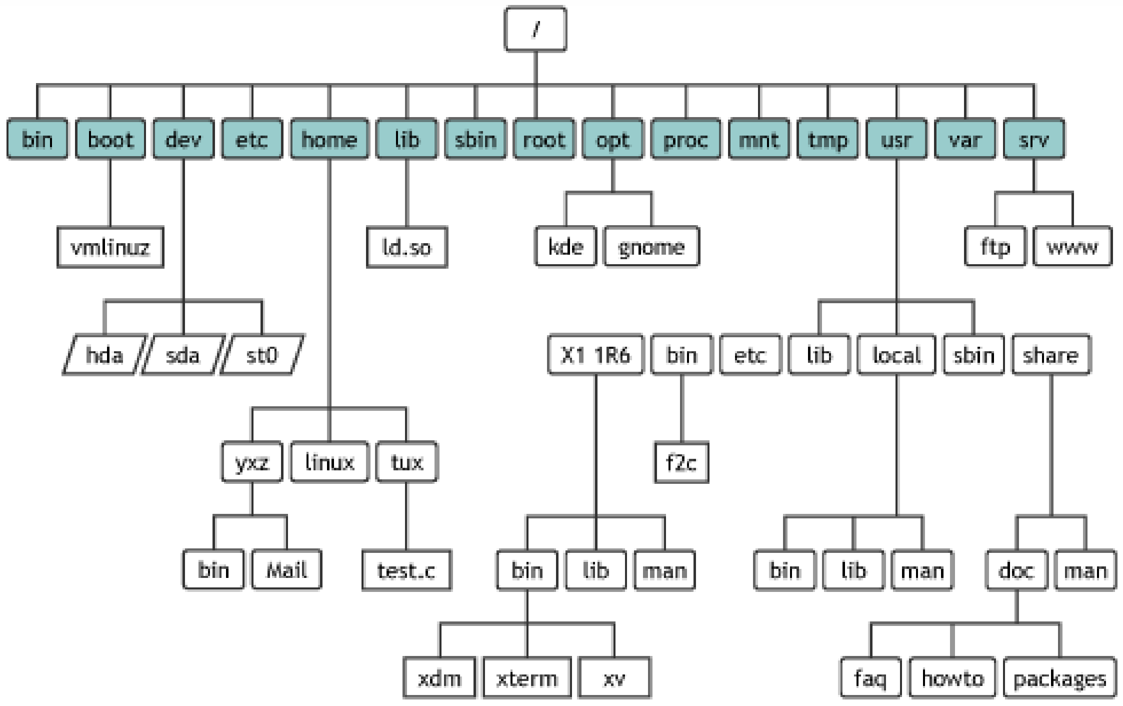
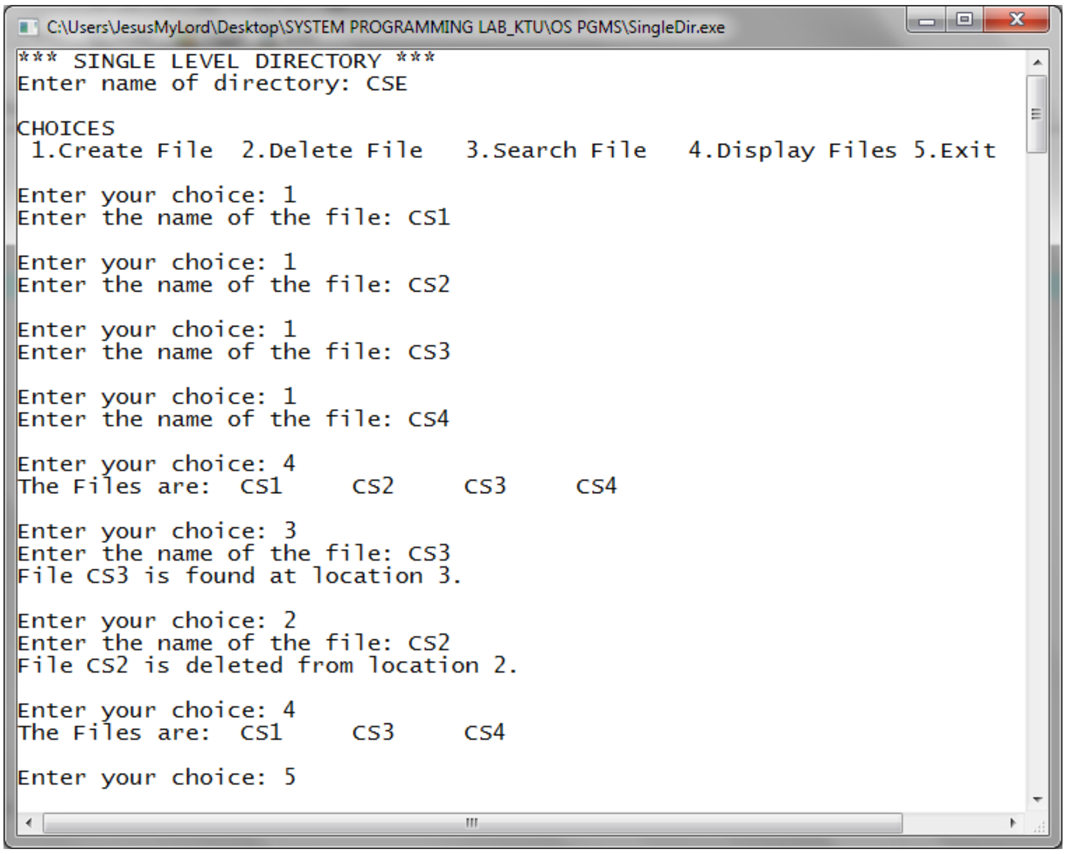


Fig. Hierarchical Structure

PROGRAM DEVELOPMENT:

```
/*Single Level Directory*/
#include<stdio.h>
struct
{
   char dname[10],fname[10][10];
   int fcnt;
}dir;

int main()
{
   int i,j,ch;
   char f[30];
   dir.fcnt = 0;
   printf("*** SINGLE LEVEL DIRECTORY ***");
   printf("\nEnter name of directory: ");
   scanf("%s", dir.dname);
   printf("\nCHOICES\n 1.Create File\t2.Delete File\t3.Search File\t4.Display Files\t5.Exit\n");
   while(1)
   {
      printf("\nEnter your choice: ");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
              printf("Enter the name of the file: ");
              scanf("%s",dir.fname[dir.fcnt]);
              dir.fcnt++;
              break;

         case 2:
              printf("Enter the name of the file: ");
              scanf("%s",f);
              for(i=0;i<dir.fcnt;i++)
              {
                if(strcmp(f, dir.fname[i])==0)
                {
                   printf("File %s is deleted from location %d.\n",f,i+1);
                   //Rearrange remaining directories
                   for(j=i;j<dir.fcnt;j++)
                   {
                      if(j<dir.fcnt-1)
                          strcpy(dir.fname[j],dir.fname[j+1]);
                   }
                   break;
                }
              }
              if(i==dir.fcnt)
                printf("File %s not found.\n",f);
              else
                dir.fcnt--;
              break;

         case 3:
              printf("Enter the name of the file: ");
              scanf("%s",f);
              for(i=0;i<dir.fcnt;i++)
              {
                if(strcmp(f, dir.fname[i])==0)
                {
                   printf("File %s is found at location %d.\n", f, i+1);
                   break;
                }
              }
              if(i==dir.fcnt)
                printf("File %s not found.\n",f);
              break;
         case 4:
              if(dir.fcnt==0)
                printf("Directory Empty.\n");
              else
              {
                printf("The Files are: ");
                for(i=0;i<dir.fcnt;i++)
                printf("\t%s",dir.fname[i]);
              }
              printf("\n");
              break;
         default:
              exit(0);
      }
   }
   getch();
}
```

SAMPLE INPUT AND OUTPUT:


C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\SingleDir.exe

```
*** SINGLE LEVEL DIRECTORY ***
Enter name of directory: CSE

CHOICES
 1.Create File  2.Delete File   3.Search File   4.Display Files 5.Exit

Enter your choice: 1
Enter the name of the file: CS1

Enter your choice: 1
Enter the name of the file: CS2

Enter your choice: 1
Enter the name of the file: CS3

Enter your choice: 1
Enter the name of the file: CS4

Enter your choice: 4
The Files are:  CS1     CS2     CS3     CS4

Enter your choice: 3
Enter the name of the file: CS3
File CS3 is found at location 3.

Enter your choice: 2
Enter the name of the file: CS2
File CS2 is deleted from location 2.

Enter your choice: 4
The Files are:  CS1     CS3     CS4

Enter your choice: 5
```

PROGRAM DEVELOPMENT:

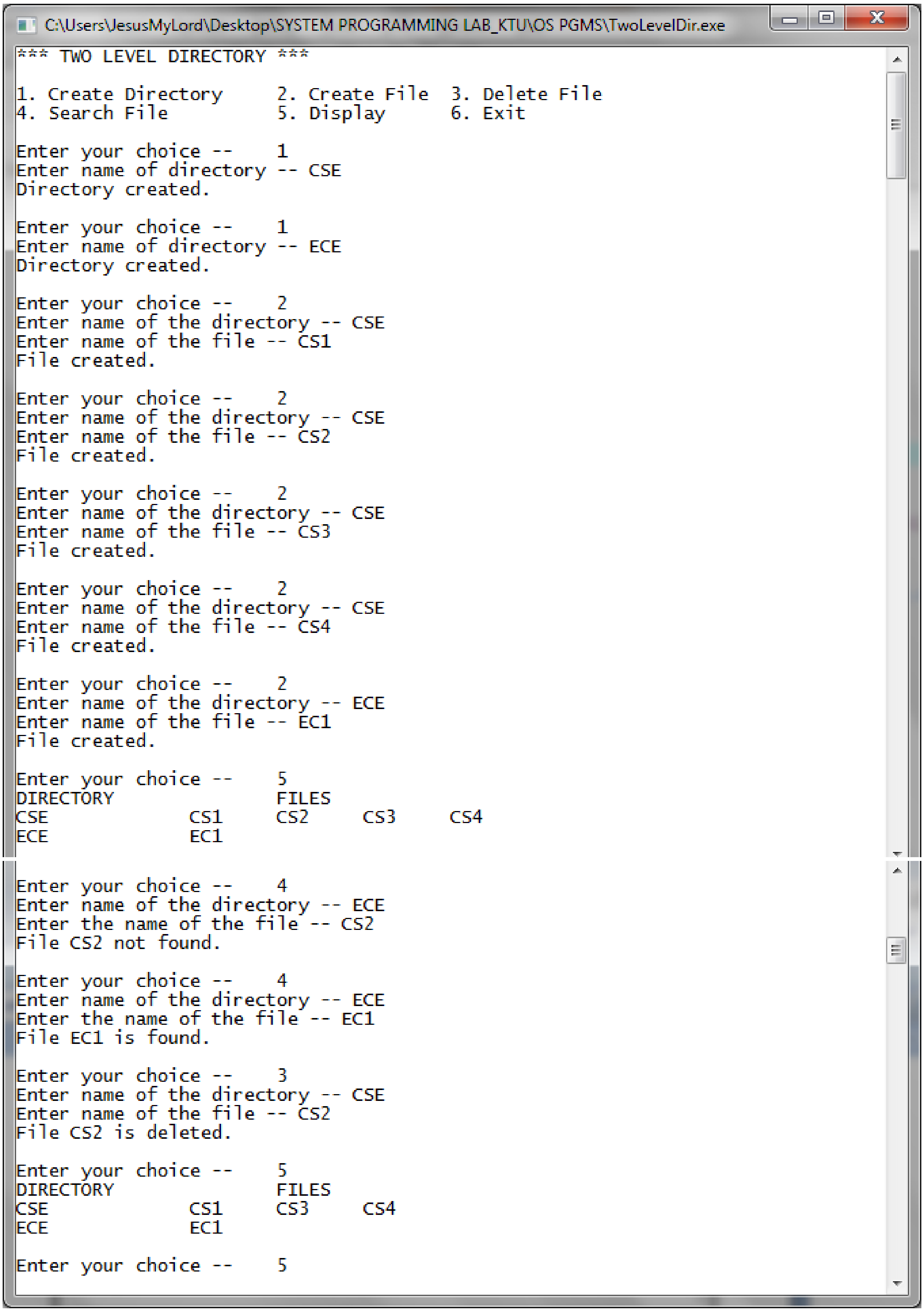
```
/*Two Level Directory*/
#include<stdio.h>
struct
{
    char dname[10],fname[10][10];
    int fcnt;
}dir[10];

int main()
{
   int i,j,ch,dcnt,k;
   char f[30], d[30];
   dcnt=0;
   printf("*** TWO LEVEL DIRECTORY ***");
   printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");
   printf("\n4. Search File\t\t5. Display\t6. Exit\t");
   while(1)
   {
      printf("\n\nEnter your choice --    ");
```

```
scanf("%d",&ch);
switch(ch)
{
  case 1:
       printf("Enter name of directory -- ");
       scanf("%s", dir[dcnt].dname);
       dir[dcnt].fcnt=0;
       dcnt++;
       printf("Directory created.");
       break;

  case 2:
       printf("Enter name of the directory -- ");
       scanf("%s",d);
       for(i=0;i<dcnt;i++)
       {
         if(strcmp(d,dir[i].dname)==0)
         {
             printf("Enter name of the file -- ");
             scanf("%s",dir[i].fname[dir[i].fcnt]);
             dir[i].fcnt++;
             printf("File created.");
             break;
         }
       }
       if(i==dcnt)
         printf("Directory %s not found.",d);
         break;
  case 3:
       printf("Enter name of the directory -- ");
       scanf("%s",d);
       for(i=0;i<dcnt;i++)
       {
         if(strcmp(d,dir[i].dname)==0)
         {
           printf("Enter name of the file -- ");
           scanf("%s",f);
           for(k=0;k<dir[i].fcnt;k++)
           {
             if(strcmp(f, dir[i].fname[k])==0)
             {
               printf("File %s is deleted.",f);
               //Rearrange remaining files
               for(j=k;j<dir[i].fcnt;j++)
```

```
                {
                  if(j < dir[i].fcnt-1)
                      strcpy(dir[i].fname[j],dir[i].fname[j+1]);
                }
                dir[i].fcnt--;
                //strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
                goto jmp;
              }
            }
            printf("File %s not found.",f);
            goto jmp;
          }
        }
        printf("Directory %s not found.",d);
        jmp :
        break;

    case 4:
        printf("Enter name of the directory -- ");
        scanf("%s",d);
        for(i=0;i<dcnt;i++)
        {
          if(strcmp(d,dir[i].dname)==0)
          {
            printf("Enter the name of the file -- ");
            scanf("%s",f);
            for(k=0;k<dir[i].fcnt;k++)
            {
              if(strcmp(f, dir[i].fname[k])==0)
              {
                printf("File %s is found.",f);
                goto jmp1;
              }
            }
            printf("File %s not found.",f);
            goto jmp1;
          }
        }
        printf("Directory %s not found.",d);
        jmp1:
        break;

    case 5:
        if(dcnt==0)
```

```
                printf("No Directory's.");
              else
              {
                printf("DIRECTORY\t\tFILES");
                for(i=0;i<dcnt;i++)
                {
                  printf("\n%s\t",dir[i].dname);
                  for(k=0;k<dir[i].fcnt;k++)
                    printf("\t%s",dir[i].fname[k]);
                }
              }
              break;
          default:
                exit(0);
          }
    }
    getch();
}
```

SAMPLE INPUT AND OUTPUT:


C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\TwoLevelDir.exe

```
*** TWO LEVEL DIRECTORY ***

1. Create Directory      2. Create File   3. Delete File
4. Search File           5. Display       6. Exit

Enter your choice --     1
Enter name of directory -- CSE
Directory created.

Enter your choice --     1
Enter name of directory -- ECE
Directory created.

Enter your choice --     2
Enter name of the directory -- CSE
Enter name of the file -- CS1
File created.

Enter your choice --     2
Enter name of the directory -- CSE
Enter name of the file -- CS2
File created.

Enter your choice --     2
Enter name of the directory -- CSE
Enter name of the file -- CS3
File created.

Enter your choice --     2
Enter name of the directory -- CSE
Enter name of the file -- CS4
File created.

Enter your choice --     2
Enter name of the directory -- ECE
Enter name of the file -- EC1
File created.

Enter your choice --     5
DIRECTORY                FILES
CSE             CS1      CS2      CS3      CS4
ECE             EC1

Enter your choice --     4
Enter name of the directory -- ECE
Enter the name of the file -- CS2
File CS2 not found.

Enter your choice --     4
Enter name of the directory -- ECE
Enter the name of the file -- EC1
File EC1 is found.

Enter your choice --     3
Enter name of the directory -- CSE
Enter name of the file -- CS2
File CS2 is deleted.

Enter your choice --     5
DIRECTORY                FILES
CSE             CS1      CS3      CS4
ECE             EC1

Enter your choice --     5
```

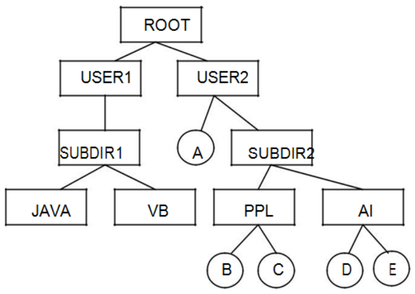
PROGRAM DEVELOPMENT:

```
/*Hierarchical Directory*/
#include<stdio.h>
#include<graphics.h>
struct tree_element
{
  char name[20];
  int x, y, ftype, lx, rx, nc, level;
  struct tree_element *link[5];
};
typedef struct tree_element node;

void main()
{
  int gd=DETECT,gm;
  node *root;
  root=NULL;
  create(&root,0,"root",0,639,320);
  clrscr();
  initgraph(&gd,&gm,"c:\tc\BGI");
  display(root);
  getch();
  closegraph();
}
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
  int i, gap;
  if(*root==NULL)
  {
    (*root)=(node *)malloc(sizeof(node));
    printf("Enter name of dir/file(under %s) : ",dname);
    fflush(stdin);
    gets((*root)->name);
    printf("enter 1 for Dir/2 for file :");
    scanf("%d",&(*root)->ftype);
    (*root)->level=lev;
    (*root)->y=50+lev*50;
    (*root)->x=x;
    (*root)->lx=lx;
    (*root)->rx=rx;
    for(i=0;i<5;i++)
    (*root)->link[i]=NULL;
    if((*root)->ftype==1)
    {
      printf("No of sub directories/files(for %s):",(*root)->name);
      scanf("%d",&(*root)>nc);
      if((*root)->nc==0)
        gap=rx-lx;
      else
        gap=(rx-lx)/(*root)->nc;
        for(i=0;i<(*root)->nc;i++)
create(&((*root)>link[i]),lev+1,(*root)>name,lx+gap*i,lx+gap*i+gap,
        lx+gap*i+gap/2);
    }
    else
      (*root)->nc=0;
  }
}
display(node *root)
{
  int i;
  settextstyle(2,0,4);
  settextjustify(1,1);
  setfillstyle(1,BLUE);
  setcolor(14);
  if(root !=NULL)
  {
    for(i=0;i<root->nc;i++)
      line(root->x,root->y,root->link[i]->x,root->link[i]->y);
    if(root->ftype==1)
      bar3d(root->x-20,root->y-10,root->x+20,root>y+10,0,0);
    else
      fillellipse(root->x,root->y,20,20);
    outtextxy(root->x,root->y,root->name);
    for(i=0;i<root->nc;i++)
      display(root->link[i]);
  }
}
```

SAMPLE INPUT AND OUTPUT:

```
Enter Name of dir/file(under root): ROOT
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for ROOT): 2
Enter Name of dir/file(under ROOT): USER1
 Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for USER1): 1
Enter Name of dir/file(under USER1): SUBDIR1
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for SUBDIR1): 2
```

```
Enter Name of dir/file(under USER1): JAVA
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for JAVA): 0
Enter Name of dir/file(under SUBDIR1): VB
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for VB): 0
Enter Name of dir/file(under ROOT): USER2
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for USER2): 2
Enter Name of dir/file(under ROOT): A
Enter 1 for Dir/2 for File: 2
Enter Name of dir/file(under USER2): SUBDIR2
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for SUBDIR2): 2
Enter Name of dir/file(under SUBDIR2): PPL
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for PPL): 2
Enter Name of dir/file(under PPL): B
Enter 1 for Dir/2 for File: 2
Enter Name of dir/file(under PPL): C
Enter 1 for Dir/2 for File: 2
Enter Name of dir/file(under SUBDIR): AI
Enter 1 for Dir/2 for File: 1
No of subdirectories/files(for AI): 2
Enter Name of dir/file(under AI): D
Enter 1 for Dir/2 for File: 2
Enter Name of dir/file(under AI): E
Enter 1 for Dir/2 for File: 2
```



CONCLUSION:

The programs to simulate directory structures is simulated and output is verified.

Exp No: 17

PAGING TECHNIQUES IN MEMORY MANAGEMENT

PROBLEM DEFINITION:

To simulate paging techniqueS of memory management.

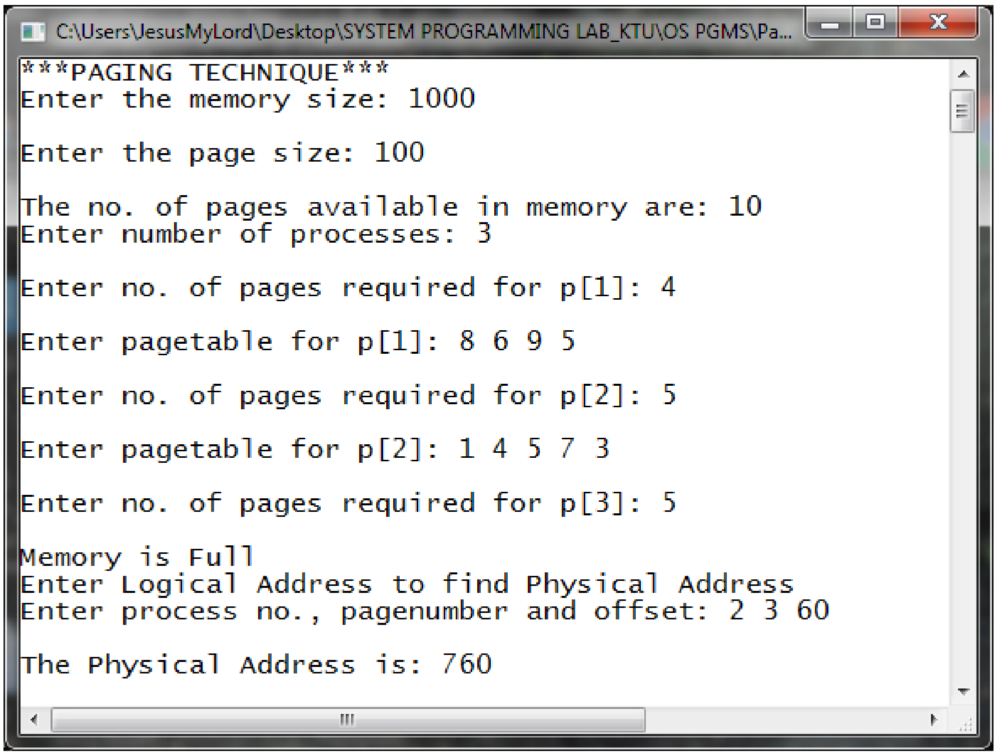
THEORETICAL BACKGROUND:

In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

PROGRAM DEVELOPMENT:

```
/*PAGING TECHNIQUES*/
#include<stdio.h>
#include<conio.h>
int main()
{
  int ms, ps, nop, np, rempages, i, j, x, y, pa, offset; int s[10], fno[10][20];
  printf("***PAGING TECHNIQUE***");
  printf("\nEnter the memory size: ");
  scanf("%d",&ms);
  printf("\nEnter the page size: ");
  scanf("%d",&ps);
  nop = ms/ps;
  printf("\nThe no. of pages available in memory are: %d ",nop);
  printf("\nEnter number of processes: ");
  scanf("%d",&np);
  rempages = nop;
  for(i=1;i<=np;i++)
  {
    printf("\nEnter no. of pages required for p[%d]: ",i);
    scanf("%d",&s[i]);
    if(s[i] > rempages)
    {
      printf("\nMemory is Full");
      break;
    }
    rempages = rempages - s[i];
    printf("\nEnter pagetable for p[%d]: ",i);
    for(j=0;j<s[i];j++)
    scanf("%d",&fno[i][j]);
  }
  printf("\nEnter Logical Address to find Physical Address");
  printf("\nEnter process no., pagenumber and offset: ");
  scanf("%d %d %d",&x,&y, &offset);
  if(x>np || y>=s[i] || offset>=ps)
     printf("\nInvalid Process or Page Number or offset");
  else
  {
     pa=fno[x][y]*ps+offset;
     printf("\nThe Physical Address is: %d",pa);
  }
  getch();
}
```

SAMPLE INPUT AND OUTPUT:



C:\Users\JesusMyLord\Desktop\SYSTEM PROGRAMMING LAB_KTU\OS PGMS\Pa...

```
***PAGING TECHNIQUE***
Enter the memory size: 1000

Enter the page size: 100

The no. of pages available in memory are: 10
Enter number of processes: 3

Enter no. of pages required for p[1]: 4

Enter pagetable for p[1]: 8 6 9 5

Enter no. of pages required for p[2]: 5

Enter pagetable for p[2]: 1 4 5 7 3

Enter no. of pages required for p[3]: 5

Memory is Full
Enter Logical Address to find Physical Address
Enter process no., pagenumber and offset: 2 3 60

The Physical Address is: 760
```

CONCLUSION:

The program to illustrate paging concept is developed and verified.

APPENDIX I : VIVA QUESTIONS

1. Define operating system?
2. What are the different types of operating systems?
3. Define a process?
4. What are the contents of PCB?
5. What is CPU Scheduling?
6. Define arrival time, burst time, waiting time, turnaround time?
7. What are the different CPU scheduling criteria?
8. What is the advantage of round robin CPU scheduling algorithm?
9. Which CPU scheduling algorithm is for real-time operating system?
10. In general, which CPU scheduling algorithm works with highest waiting time?
11. Is it possible to use optimal CPU scheduling algorithm in practice?
12. What is the real difficulty with the SJF CPU scheduling algorithm?
13. Differentiate between the general CPU scheduling algorithms like FCFS, SJF etc and multi-level queue CPU Scheduling?
14. What are CPU-bound and I/O-bound processes?
15. What is the need for process synchronization?
16. What is a critical section?
17. Define a semaphore?
18. Define producer-consumer problem?
19. Discuss the consequences of considering bounded and unbounded buffers in producer-consumer problem?
20. Can producer and consumer processes access the shared memory concurrently? If not which technique provides such a benefit?
21. Differentiate between a monitor, semaphore and a binary semaphore?
22. Define clearly the dining-philosophers problem?
23. Identify the scenarios in the dining-philosophers problem that leads to the deadlock situations?
24. Define file?
25. What are the different kinds of files?
26. What is the purpose of file allocation strategies?
27. Identify ideal scenarios where sequential, indexed and linked file allocation strategies are most appropriate?
28. What are the disadvantages of sequential file allocation strategy?
29. What is an index block?
30. What is the file allocation strategy used in UNIX?
31. What is dynamic memory allocation?
32. What is external fragmentation?
33. Which of the dynamic contiguous memory allocation strategies suffer with external fragmentation?
34. What are the possible solutions for the problem of external fragmentation?
35. What is 50-percent rule?
36. What is compaction?

37. Which of the memory allocation techniques first-fit, best-fit, worst-fit is efficient? Why?
38. What are the advantages of noncontiguous memory allocation schemes?
39. What is the process of mapping a logical address to physical address with respect to the paging memory management technique?
40. Define the terms – base address, offset?
41. Differentiate between paging and segmentation memory allocation techniques?
42. What is the purpose of page table?
43. Whether the paging memory management technique suffers with internal or external fragmentation problem. Why?
44. What is the effect of paging on the overall context-switching time?
45. Define directory?
46. Describe the general directory structure?
47. List the different types of directory structures?
48. Which of the directory structures is efficient? Why?
49. Which directory structure does not provide user-level isolation and protection?
50. What is the advantage of hierarchical directory structure?
51. Define resource. Give examples.
52. What is deadlock?
53. What are the conditions to be satisfied for the deadlock to occur?
54. How can be the resource allocation graph used to identify a deadlock situation?
55. How is Banker's algorithm useful over resource allocation graph technique?
56. Differentiate between deadlock avoidance and deadlock prevention?
57. What is disk scheduling?
58. List the different disk scheduling algorithms?
59. Define the terms – disk seek time, disk access time and rotational latency?
60. What is the advantage of C-SCAN algorithm over SCAN algorithm?
61. Which disk scheduling algorithm has highest rotational latency? Why?
62. Define the concept of virtual memory?
63. What is the purpose of page replacement?
64. Define the general process of page replacement?
65. List out the various page replacement techniques?
66. What is page fault?
67. Which page replacement algorithm suffers with the problem of Belady's anomaly?
68. Define the concept of thrashing? What is the scenario that leads to the situation of thrashing?
69. What are the benefits of optimal page replacement algorithm over other page replacement algorithms?
70. Why can't the optimal page replacement technique be used in practice?

REFERENCES

TEXT BOOKS:

1. Operating System Principles, Abraham Silberchatz, Peter B. Galvin, Greg Gagne, 9th Edition, Wiley Student Edition.

2. Operating Systems – Internals and Design Principles, W.Stallings, 6th Edition, Pearson.

3. UNIX Network Programming – W.Richard Stevens,Prentice Hall India

REFERENCE BOOKS:

1. Modern Operating Systems, Andrew S Tanenbaum, 3rd Edition,PH.

2. Operating Systems A concept-based Approach, 2nd Edition,D.M.Dhamdhere, TMH.

3. Principles of Operating Systems, B.L.Stuart, Cengage learning, India Edition**.**