University Institute of Technology ,

Rajiv Gandhi Proudyogiki Vishwavidyalaya

(Affiliated to RGPV University & Approved by AICTE, New Delhi)

LABORATORY MANUAL

OPERATING SYSTEMS LAB

NAME: _____

ROLL NO:_____

BRANCH:_____ **SEM:**_____

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERNG

University Institute of Technology, RGPV

VISION

To nurture and enlighten the budding talents into socially committed & dexterous Computer Science & Engineering professionals, possessing a robust foundation and adaptable attitude for rapid technological advancements in the field along with sustainable ethical values, to meet global challenges.

MISSION

- To persistently endeavor for the consistent development of students by providing them a robust foundation in knowledge arena by enabling students to imbibe the very essence and philosophy of Computer Science & Engineering domain.

- To continuously strive for the overall development of students by educating them in blooming cutting edge technology.

- To curate a conductive environment for exploring, analyzing and applying knowledge to improve technical & interpersonal skills of students.

- To inculcate ethics, environmental awareness and societal commitment contributing to their professional personality to serve better for a sustainable global society.

- To provide an exposure of Innovative research culture in the demanding arena of Computer Science & Engineering.

University Institue of Technology, RGPV

DEPARTMENT
OF
COMPUTER SCIENCE AND ENGINEERING

LABORATORY MANUAL

OPERATING SYSTEMS LAB

University Institue Of Technology,RGPV

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION & MISSION

VISION

- Strive to be a Centre of Excellence in Computer Science engineering and producing graduate engineers instilled with human values and professional ethics, who will serve as a valuable resource to the nation.

MISSION

- To build strong technical foundation through balanced curriculum, high quality teaching and practical skills.
- To groom the graduating engineers for Industry, Research and Higher Education.
- To inculcate human, ethical and moral values by providing a congenial environment.

University Institue Of Technology,RGPV

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM EDUCATIONAL OBJECTIVES

PEO1: Job Opportunities: To attain professional competency in the fast growing area of Computer Science & Engineering.

PEO2: Research and Development: To be able to apply knowledge of basic sciences and core engineering to pursue higher education, research and also to be able to work in competitive multidisciplinary environment.

PEO3: Entrepreneur: To flourish as a socially committed Computer Science & Engineering grandaunts, possessing state of the art domain
Knowledge, entrepreneurial skill, problem solving & research Attitude, good leadership qualities and high ethical values.

University Institue Of Technology,RGPV

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM OUTCOME

Engineering graduates will be able to:

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

PO2: Problem Analysis - Identify, formulate, review research literature and analyze complex engineering problem reaching substantiated conclusions using first principle of mathematics natural science and engineering science.

PO3: Design/Development of solutions - Design solutions for complex engineering problem and design system components or processes that needs the specified needs with appropriate consideration for the public health and safety and the cultural, societal and environmental consideration.

PO4: Conduct Investigation of Complex Problem - Use research-based knowledge and research methods including design of experiments analysis and interpretation of data, and synthesis of information to provide valid conclusions.

PO5: Modern Tool Usage - 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: The engineer and society - Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

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

PO8: Ethics - Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.

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

PO10: Communication - Communicate effectively on complex engineering activities with the engineering community
and with the 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: Project management and finance - Demonstrate knowledge and understanding
of the engineering and management principles and apply these to one's own work, a s a member and leader in a team to manage projects and in multidisciplinary environments.

PO12: Life-long learning - 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

Computer Science and Engineering graduates at UNIVERSITY INSTITUE OF TECHNOLOGY,RGPV will be able to:

PSO1: Professional Skills-The ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, network security, IoT and networking for efficient design of computer-based systems of varying.

PSO2: Problem-Solving Skills-The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business success.

PSO3: Successful Career and Entrepreneurship- The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, and a zest for higher studies.

Course Code	Course Title						Core / Elective
CS501	**OPERATING SYSTEMS LAB**						**CORE**
Prerequisite	Contact Hours Per Week				CIE	SEE	Credits
	L	T	D	P			

Course Objectives:

- To learn shell programming and the use of filters in the LINUX environment.
- To practice multithreaded programming.
- To implement CPU Scheduling Algorithms and memory management algorithms.

Course Outcomes:

- Evaluate the performance of different types of CPU scheduling algorithms.
- Implement producer-consumer problem, reader-writers problem, Dining philosopher's problem.
- Simulate Banker's algorithm for deadlock avoidance.
- Implement paging replacement and disk scheduling techniques.
- Use different system calls for writing application programs.

I. CASE STUDY

Perform a case study by installing and exploring various types of operating systems on a physical or logical (virtual) machine.

II. List of Experiments (preferred programming language is C)

1. Write a C programs to implement UNIX system calls and file management.
2. Write C programs to demonstrate various process related concepts.
3. Write C programs to demonstrate various thread related concepts.
4. Write C programs to simulate CPU scheduling algorithms: FCFS, SJF, and Round Robin.
5. Write C programs to simulate Intra & Inter – Process Communication (IPC) techniques: Pipes, Messages Queues, and Shared Memory.
6. Write C programs to simulate solutions to Classical Process Synchronization Problems: Dining Philosophers, Producer – Consumer, Readers – Writers.
7. Write a C program to simulate Bankers Algorithm for Deadlock Avoidance.
8. Write C programs to simulate Page Replacement Algorithms: FIFO, LRU.
9. Write C programs to simulate implementation of Disk Scheduling Algorithms: FCFS, SSTF.

University Institue Of Technology,RGPV

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Outcomes (CO's):

SUBJECT NAME: OPERATING SYSTEMS LAB **CODE:CS 501**

SEMESTER: V

CO No.	Course Outcome	Taxonomy Level
501CS.1	Evaluate the performance of different types of CPU scheduling algorithms.	Evaluating
501CS.2	Implement producer – consumer problem, reader – writers problem, Dining philosopher's problem.	Applying
501CS.3	Simulate Banker's algorithm for deadlock avoidance.	Creating
501CS.4	Implement paging replacement and disk scheduling techniques.	Applying
501CS.5	Use different system calls for writing application programs.	Applying
501CS.6	Ability to implement inter process communication between two processes.	Applying

University Institue Of Technology,RGPV

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the program / experiment details.
3. Student should enter into the laboratory with:
   a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
   b. Laboratory Record updated up to the last session experiments.
   c. Formal dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviours with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out. If anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department **Principal**

University Institue Of Technology,RGPV

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CODE OF CONDUCT FOR THE LABORATORY

- All students must observe the dress code while in the laboratory
- Footwear is NOT allowed
- Foods, drinks and smoking are NOT allowed
- All bags must be left at the indicated place
- The lab timetable must be strictly followed
- Be PUNCTUAL for your laboratory session
- All programs must be completed within the given time
- Noise must be kept to a minimum
- Workspace must be kept clean and tidy at all time
- All students are liable for any damage to system due to their own negligence
- Students are strictly PROHIBITED from taking out any items from the laboratory
- Report immediately to the lab programmer if any damages to equipment

BEFORE LEAVING LAB:

- Arrange all the equipment and chairs properly.
- Turn off / shut down the systems before leaving.
- Please check the laboratory notice board regularly for updates.

Lab In – charge

University Institue Of Technology,RGPV

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LIST OF EXPERIMENTS

S. No.	Name of the Experiment	Date of Experiment	Date of Submission	Page No	Faculty Signature
1.	Perform a case study by installing and exploring various types of operating systems on a physical or logical (virtual) machine. (Linux Installation).			1	
2.	Write a C programs to implement UNIX system calls and file management			8	
3.	Write C programs to demonstrate various thread related concepts.			27	
4.	Write C programs to simulate CPU scheduling algorithms: FCFS, SJF, and Round Robin.			31	
5.	Write C programs to simulate Intra & Inter – Process Communication (IPC) techniques: Pipes, Messages, Queues and Shared Memory.			46	

ADDITIONAL EXPERIMENTS

PROGRAM I(CASE STUDY): Perform a case study by installing and exploring various types of operating systems on a physical or logical (virtual) machine. (Linux Installation).

Instructions to Install Ubuntu Linux 12.04 (LTS) along with Windows

Back Up Your Existing Data!

This is highly recommended that you should take backup of your entire data before start with the installation process.

Obtaining System Installation Media

Download latest Desktop version of Ubuntu from this link:

http://www.ubuntu.com/download/desktop

Booting the Installation System

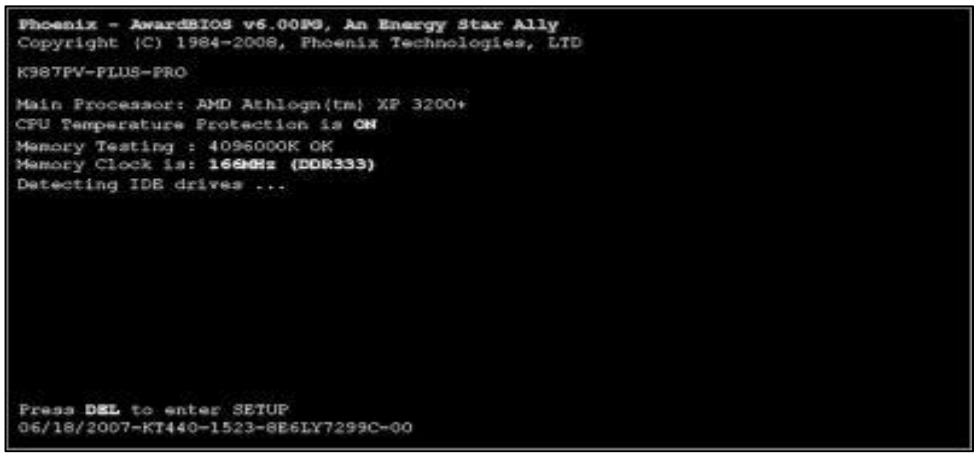
There are several ways to boot the installation system. Some of the very popular ways are , Booting from a CD ROM, Booting from a USB memory stick, and Booting from TFTP.

Here we will learn how to boot installation system using a CD ROM.

Before booting the installation system, one need to change the boot order and set CD-ROM as first boot device.

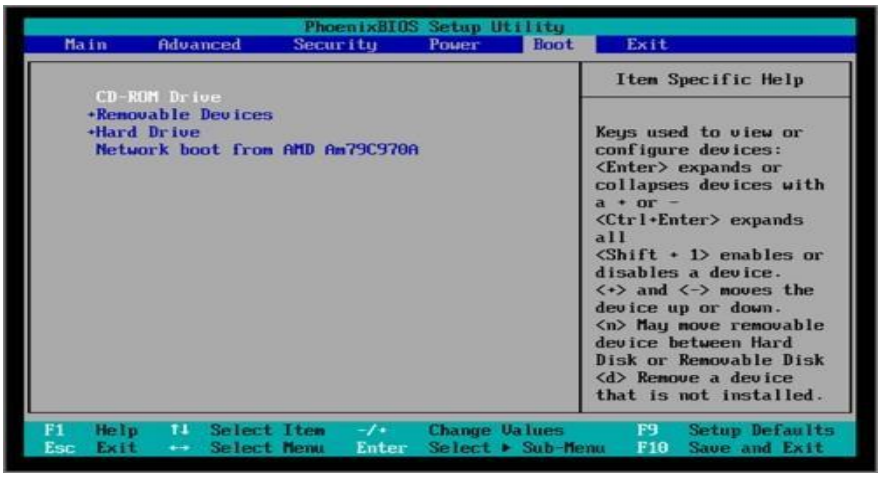
Changing the Boot Order of a Computers

As your computer starts, press the DEL, ESC, F1, F2, F8 or F10 during the initial startup screen. Depending on the BIOS manufacturer, a menu may appear. However, consult the hardware documentation for the exact key strokes. In my machine, its DEL key as shown in following screen-shot.



2. Find the Boot option in the setup utility. Its location depends on your BIOS.
   Select the Boot option from the menu, you can now see the options Hard Drive, CD-ROM Drive, Removable Devices Disk etc.

3. Change the boot sequence setting so that the CD-ROM is first. See the list of "Item Specific Help" in right side of the window and find keys which is used to toggle to change the boot sequence.



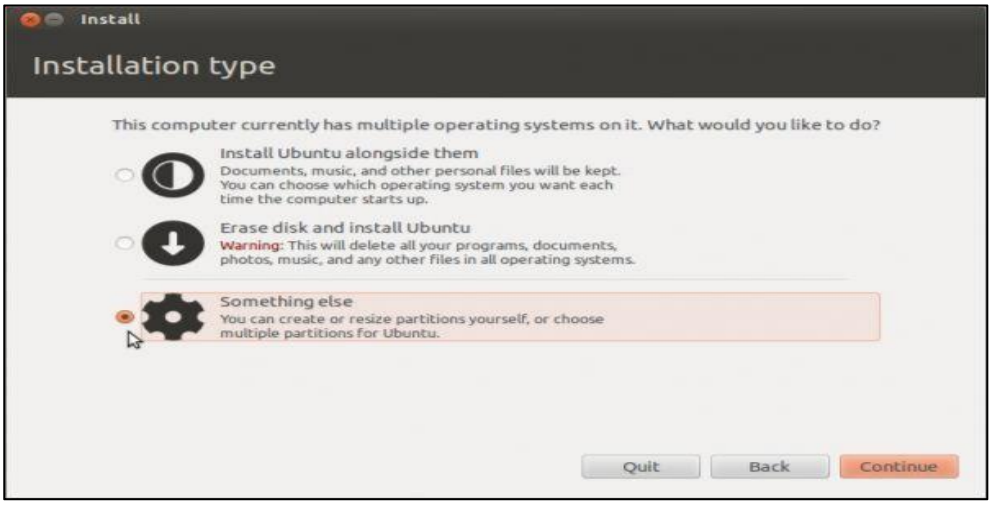
4. Insert the Ubuntu Disk in CD/DVD drive.
5. Save your changes. Instructions on the screen tell you how to save the changes on your computer. The computer will restart with the changed settings.

Machine should boot from CD ROM, Wait for the CD to load...

In a few minutes installation wizard will be started. Select your language and click the "Install Ubuntu" button to continue...

Optionally, you can choose to download updates while installing and/or install third party software, such as MP3 support. Be aware, though, that if you select those options, the entire installation process will be longer!

Since we are going to create partitions manually, select **Something else**, then click **Continue**. Keep in mind that even if you do not want to create partitions manually, it is better to select the same option as indicated here. This would insure that the installer will not overwrite your Windows , which will destroy your data. The assumption here is that **sdb** will be used just for Ubuntu 12.04, and that there are no valuable data on it.



Where are you? Select your location and Click the "Continue" button.

Keyboard layout

Select your keyboard layout and UK (English) and Click on “Continue” button.

Who are you?

Fill in the fields with your real name, the name of the computer (automatically generated, but can be overwritten), username, and the password.

Also at this step, there's an option called "Log in automatically." If you check it, you will automatically be logged in to the Ubuntu desktop without giving the password.

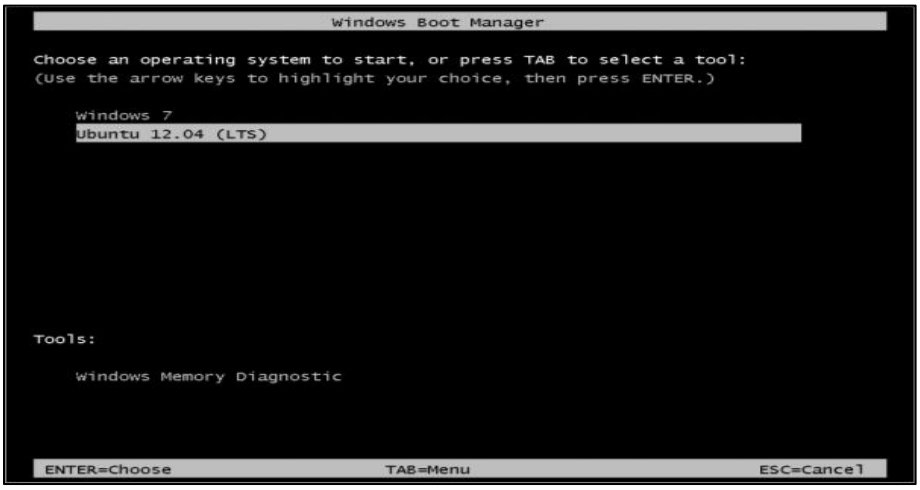
Option "Encrypt my home folder," will encrypt your home folder. Click on the "Continue" button to continue...

Now Ubuntu 12.04 LTS (Precise Pangolin) operating system will be installed.

It will take approximately 10-12 minutes (depending on computer's speed), a pop-up window will appear, notifying you that the installation is complete, and you'll need to restart the computer in order to use the newly installed Ubuntu operating system. Click the "Restart Now" button.

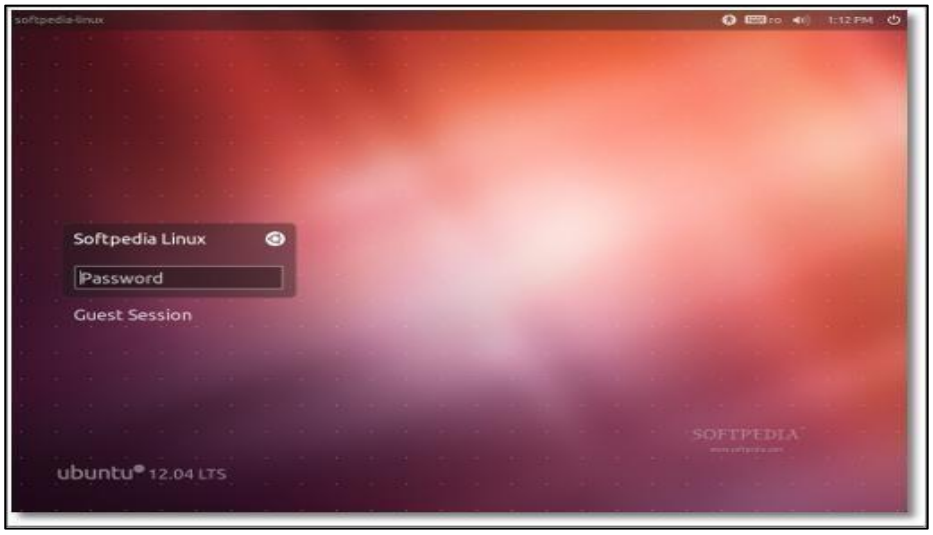
Please remove the CD and press the "Enter" key to reboot. The computer will be restarted. In a few seconds, you should see Windows 7′s boot menu with two entires listed – Windows 7 and Ubuntu 12.04 (LTS).

Then you may choose to boot into Windows 7 or Ubuntu 12.04 using the UP/Down arrow key.



Please select Ubuntu 12.04 (LTS) and press Enter to boot the machine in Ubuntu 12.04 Linux.

Here you can see the users on the machine, Click on the user name and enter the password and press Enter key to login.



We have successfully install and login to Ubuntu 12.04 LTS.

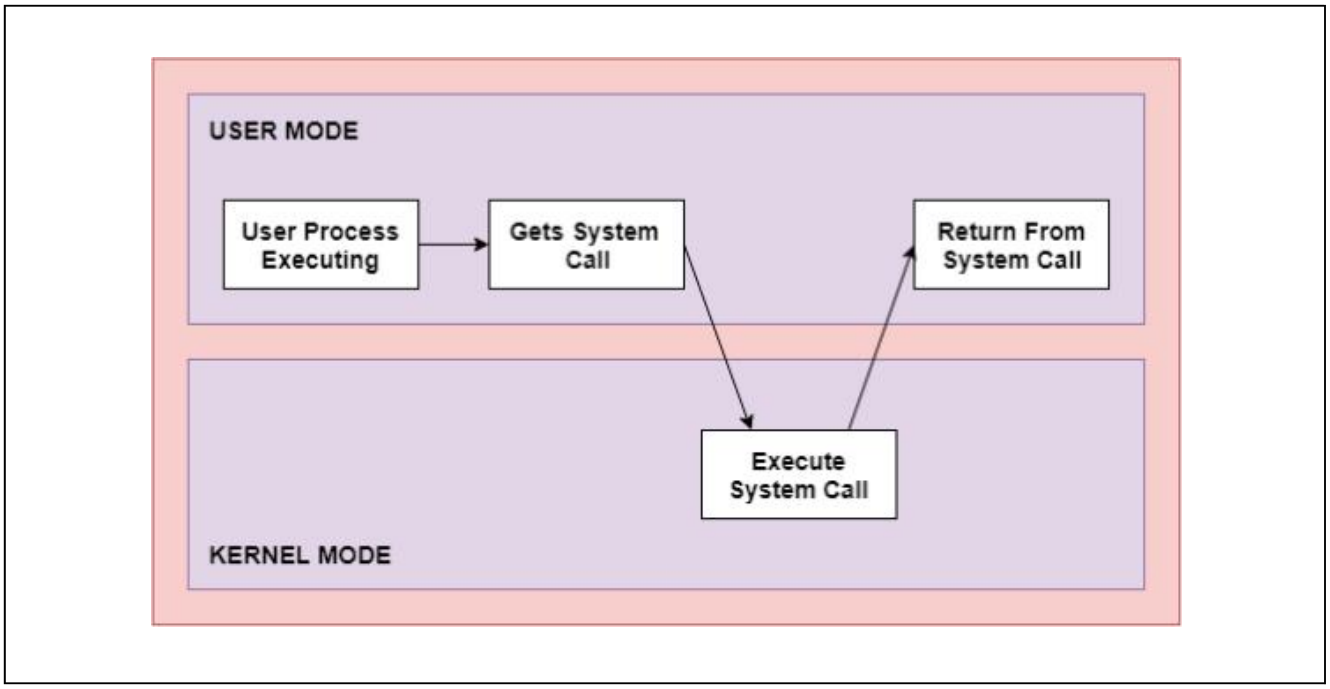
PROGRAM II : List Of Experiments(preferred programming language is C)

1. AIM : Write a C programs to implement UNIX system calls and file management.

System Calls:

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

A figure representing the execution of the system call is given as follows −



As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.

In general, system calls are required in the following situations −

- If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
- Creation and management of new processes.
- Network connections also require system calls. This includes sending and receiving packets.
- Access to a hardware devices such as a printer, scanner etc. requires a system call.

Types of System Calls

There are mainly five types of system calls. These are explained in detail as follows −

Process Control

These system calls deal with processes such as process creation, process termination etc.

File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

Device Management

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

Information Maintenance

These system calls handle information and its transfer between the operating system and the user program.

Communication

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

Some of the examples of all the above types of system calls in Windows and Unix are given as follows −

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Management	SetConsoleMode() ReadConsole()	ioctl() read()

Types of System Calls	Windows	Linux
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

1.A) Fork() System Call:

AIM:To write the program to implement fork () system call.

DESCRIPTION:

Used to create new processes. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax:

Fork ( );

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the variables pid and child id.

Step 3: Get the child id value using system call fork().

Step 4: If child id value is greater than zero then print as “i am in the parent process”.

Step 5: If child id! = 0 then using getpid() system call get the process id.

Step 6: Print “i am in the parent process” and print the process id.

Step 7: If child id! = 0 then using getppid() system call get the parent process id.

Step 8: Print “i am in the parent process” and print the parent process id.

Step 9: Else If child id value is less than zero then print as “i am in the child process”.

Step 10: If child id! = 0 then using getpid() system call get the process id.

Step 11: Print “i am in the child process” and print the process id.

Step 12: If child id! = 0 then using getppid() system call get the parent process id.

Step 13: Print “i am in the child process” and print the parent process id.

Step 14: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* fork system call */
#include<stdio.h>
#include <unistd.h>
#include<sys/types.h>
int main()
{
        int id,childid;
        id=getpid();
        if((childid=fork())>0)
        {
                printf("\n i am in the parent process %d",id);
                printf("\n i am in the parent process %d",getpid());
                printf("\n i am in the parent process %d\n",getppid());
        }
        else
        {
                printf("\n i am in child process %d",id);
                printf("\n i am in the child process %d",getpid());
                printf("\n i am in the child process %d",getppid());
        }
}
```

OUTPUT:

```
$ vi fork.c
$ cc fork.c
$ ./a.out
 i am in child process 3765
 i am in the child process 3766
i am in the child process 3765
i am in the parent process 3765
i am in the parent process 3765
i am in the parent process 3680
```

RESULT:

Thus the program was executed and verified successfully.

1.B) Wait () and Exit () System Calls:

AIM:To write the program to implement the system calls wait ( ) and exit ( ).

DESCRIPTION:

i. fork ( )

Used to create new process. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax: fork ( );

ii. 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);

iii. 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 program.

Step 2: Declare the variables pid and i as integers.

Step 3: Get the child id value using the system call fork ().

Step 4: If child id value is less than zero then print “fork failed”.

Step 5: Else if child id value is equal to zero, it is the id value of the child and then start the child process to execute and perform Steps 7 & 8.

Step 6: Else perform Step 9.

Step 7: Use a for loop for almost five child processes to be called.

Step 8: After execution of the for loop then print “child process ends”.

Step 9: Execute the system call wait ( ) to make the parent to wait for the child process to get over.

Step 10: Once the child processes are terminated, the parent terminates and hence prints “Parent process ends”.

Step 11: After both the parent and the child processes get terminated it execute the wait ( ) system call to permanently get deleted from the OS.

Step 12: Stop the program.

PROGRAM:

1.B.1) SOURCE CODE:

```
#include<stdio.h>
#include<unistd.h>
int main( )
{
        int i, pid;
        pid=fork( );
        if(pid== -1)
        {
                printf("fork failed");
                exit(0);
        }
        else if(pid==0)
        {
                printf("\n Child process starts");
                for(i=0; i<5; i++)
                {
                        printf("\n Child process %d is called", i);
                }
                printf("\n Child process ends");
        }
        else
        {
                wait(0);
                printf("\n Parent process ends");
        }
        exit(0);
}
```

OUTPUT:

$ vi waitexit.c

$ cc waitexit.c

$ ./a.out

Child process starts

Child process 0 is called

Child process 1 is called

Child process 2 is called

Child process 3 is called

Child process 4 is called

Child process ends

Parent process ends

RESULT:

Thus the program was executed and verified successfully

1.B.2) WAIT () AND EXIT () SYSTEM CALLS

PROGRAM:

SOURCE CODE:

```
/* wait system call */
#include <stdlib.h>
#include <errno.h>
#include<stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
        pid_t pid;
        int rv;
        switch(pid=fork())
        {
                case -1:
                                perror("fork");
                                exit(1);
                case 0:
                                printf("\n CHILD: This is the child process!\n");
                                fflush(stdout);
                                printf("\n CHILD: My PID is %d\n", getpid());
                                printf("\n CHILD: My parent's PID is %d\n",getppid());
                                printf("\n CHILD: Enter my exit status (make it small):\n ");
                                printf("\n CHILD: I'm outta here!\n");
                                scanf(" %d", &rv);
                                exit(rv);
                default:
                                printf("\nPARENT: This is the parent process!\n");
                                printf("\nPARENT: My PID is %d\n", getpid());
                                fflush(stdout);
                                wait(&rv);
                                fflush(stdout);
                                printf("\nPARENT: My child's PID is %d\n", pid);
```

```
                printf("\nPARENT: I'm now waiting for my child to exit()...\n");
                fflush(stdout);
                printf("\nPARENT: My child's exit status is:
                %d\n",WEXITSTATUS(rv));
                printf("\nPARENT: I'm outta here!\n");
        }
}
```

OUTPUT:

```
$ vi wait.c
$ cc wait.c
$ ./a.out
CHILD: This is the child process!
CHILD: My PID is 3821
CHILD: My parent's PID is 3820
CHILD: Enter my exit status (make it small):
CHILD: I'm outta here!
PARENT: This is the parent process!
PARENT: My PID is 3820
10
PARENT: My child's PID is 3821
PARENT: I'm now waiting for my child to exit()...
PARENT: My child's exit status is: 10
PARENT: I'm outta here!
```

RESULT:

Thus the program was executed and verified successfully

1. C) EXECL SYSTEM CALL

AIM: To write a program to implement the system call execl ( ).

DESCRIPTION:

In execl () system function takes the path of the executable binary file (i.e. **/bin/ls**) as the first and second argument. Then, the arguments (i.e. **-lh**, **/home**) that you want to pass to the executable followed by **NULL**. Then execl () system function runs the command and prints the output. If any error occurs, then execl () returns -1. Otherwise, it returns nothing.

Syntax:

int execl (const char *path, const char *arg, ..., NULL);

ALGORITHM

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Print execution of exec system call for the ls Unix command.

Step 4: Execute the execl function using the appropriate syntax for the Unix command ls.

Step 5: The list of all files and directories of the system is displayed.

Step 6: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* execl system call */
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
main()
{
        printf("Before execl \n");
        execl("/bin/ls","ls",(char*)0);
        printf("After Execl\n");
}
```

OUTPUT:

$ vi execl.c

$ cc execl.c

$ ./a.out

Before execl

a1 aaa aaa.txt abc a.out b1 b2 comm.c db db1 demo2 dir1 direc.c execl.c f1.txt fflag.c file1 file2 fork.c m1 m2 wait.c xyz

RESULT:

Thus the program was executed and verified successfully

1.D) EXECV SYSTEM CALL

AIM: To write a program to implement the system call execv ( ).

DESCRIPTION:

In execl() function, the parameters of the executable file is passed to the function as different arguments. With execv(), you can pass all the parameters in a NULL terminated array **argv**. The first element of the array should be the path of the executable file. Otherwise, execv() function works just as execl() function.

Syntax:int execv (const char *path, char *const argv[]);

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Print execution of exec system call for the ls Unix command.

Step 4: Execute the execv function using the appropriate syntax for the Unix command ls.

Step 5: The list of all files and directories of the system is displayed.

Step 6: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* execv system call */
#include<stdio.h>
#include<sys/types.h>
main(int argc,char *argv[])
{
        printf("before execv\n");
        execv("/bin/ls",argv);
        printf("after execv\n");
}
```

OUTPUT:

```
$ vi execv.c
$ cc execv.c
$ ./a.out
before execv
a1 aaa aaa.txt abc a.out b1 b2 comm.c db db1  demo2  dir1  direc.c  execl.c  execv.c  f1.txt
fflag.c file1 file2 fork.c  m1 m2 wait.c xyz
```

RESULT:

Thus the program was executed and verified successfully.

1.E)Directory Management (Directory Hierarchy):
AIM:To write the program to implement the system calls opendir ( ), readdir ( ),closedir ().

DESCRIPTION:

UNIX offers a number of system calls to handle a directory. The following are most commonly used system calls.

SYSTEM CALLS USED:

i. **opendir ( )**

Open a directory.

Syntax:

DIR * opendir (const char * dirname);

opendir () takes dirname as the path name and returns a pointer to a DIR structure. On error returns NULL.

ii. **readdir ( )**

Read a directory.

Syntax:

struct dirent * readdir (DIR *dp) ;

A directory maintains the inode number and filename for every file in its fold. This function returns a pointer to a dirent structure consisting of inode number and filename. 'dirent' structure is defined in <dirent.h> to provide at least two members – inode number and directory name.

```
struct dirent
{
ino_t d_ino; // directory inode number
char d_name[]; // directory name
}
```

iii. **closedir ():**

Close a directory.

Syntax:

int closedir (DIR * dp);

Closes a directory pointed by dp. It returns 0 on success and -1 on error.

ALGORITHM:

Step 1: Start the program.

Step 2: In the main function pass the arguments.

Step 3: Create structure as stat buff and the variables as integer.

Step 4: Open the directory.

Step 5: Read the contents of the directory (filenames).

Step 6: Display the contents of the directory.

Step 7: Close the directory.

Step 4: Stop the program.

PROGRAM:

SOURCE CODE:

```
/* Recursively descend a directory hierarchy pointing a file */
#include<stdio.h>
#include<dirent.h>
#include<errno.h>
#include<fcntl.h>
#include<unistd.h>
int main(int argc,char *argv[])
{
        struct dirent *direntp; DIR *dirp; if(argc!=2)
        {
                printf("ussage %s directory name \n",argv[0]);
                return 1;

        }
        if((dirp=opendir(argv[1]))==NULL)
        {
                perror("Failed to open directory \n");
                return 1;
        }
        while((direntp=readdir(dirp))!=NULL)
                printf("%s\n",direntp->d_name);
        while((closedir(dirp)==-1)&&(errno==EINTR));
                return 0;
}
```

OUTPUT:

```
$ vi direc.c
$ cc direc.c
$ ./a.out ./
f1.txt
fflag.c
.
..
b2
comm.c
dir1
demo2
a1
m1
db
a.out
direc.c
db1
file1
b1
xyz
abc
aaa
file2
aaa.txt
m2
```

RESULT:

Thus the program was executed and verified successfully.

1.F) File Management:

AIM:To write the program to implement the system calls open ( ), read ( ), write ( ) & close ().

DESCRIPTION:

The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations 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. To a process then, all input and output operations are synchronous and unbuffered.

SYSTEM CALLS USED:

System calls are functions that a programmer can call to perform the services of the operating system.

Open ():

Open () system call to open a file.

open () returns a file descriptor, an integer specifying the position of this open n file in the table of open files for the current process.

Close ():

Close () system call to close a file.

Read ():

Read () data from a file opened for reading.

Write ():

Write () data to a file opened for writing.

The open () system call:

#include<fcntl.h>

int open (const char *path, int oflag);

The return value is the descriptor of the file. Returns -1 if the file could not be opened. The first parameter is path name of the file to be opened and the second parameter is the opening mode specified by bitwise oring one or more of the following values

Value	Meaning
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_APPEND	Open at end of file for writing
O_CREAT	Create the file if it doesn't already exist
O_EXCL	If set and O_CREAT set will cause open() to fail if the file already exists
O_TRUNC	Truncate file size to zero if it already exists

close () system call:

The close () system call is used to close files.

#include <unistd.h>

int close (int fildes);

It is always good practices to close files when not needed as open files do consume resources and all normal systems impose a limit on the number of files that a process can hold open.

The read () system call:

The read () system call is used to read data from a file or other object identified by a file descriptor. The prototype is

#include<sys/types.h>

size_t read (int fildes, void *buf, size_t nbyte);

fildes is the descriptor, buf is the base address of the memory area into which the data is read and nbyte is the maximum amount of data to read. The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read. An attempt to read beyond the end of a file results in a return value of zero.

The write () system call:

The write () system call is used to write data to a file or other object identified by a file descriptor. The prototype is

#include<sys/types.h>

size_t write (int fildes, const void *buf, size_t nbyte);

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the structure elements.

Step 3: Create a temporary file named temp1.

Step 4: Open the file named “test” in a write mode.

Step 5: Enter the strings for the file.

Step 6: Write those strings in the file named “test”.

Step 7: Create a temporary file named temp2.

Step 8: Open the file named “test” in a read mode.

Step 9: Read those strings present in the file “test” and save it in temp2.

Step 10: Print the strings which are read.

Step 11: Stop the program.

PROGRAM :

SOURCE CODE:

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
int main( )
{
        int fd[2];
        char buf1[25]= ”just a test\n”; char
        buf2[50];
        fd[0]=open(“file1”, O_RDWR);
        fd[1]=open(“file2”, O_RDWR);
        write(fd[0], buf1, strlen(buf1));
        printf(“\n Enter the text now….”);
        gets(buf1);
        write(fd[0], buf1, strlen(buf1));
        lseek(fd[0], SEEK_SET, 0);
        read(fd[0], buf2, sizeof(buf1));
        write(fd[1], buf2, sizeof(buf2));
        close(fd[0]);
        close(fd[1]);
        printf(“\n”);
```

```
    return0;
}
```

OUTPUT:

Enter the text now….progress

Cat file1 Just a

test progress

Cat file2 Just a test progress

RESULT:

Thus the program was executed successfully.

2. AIM :Write C programs to demonstrate various thread related concepts.

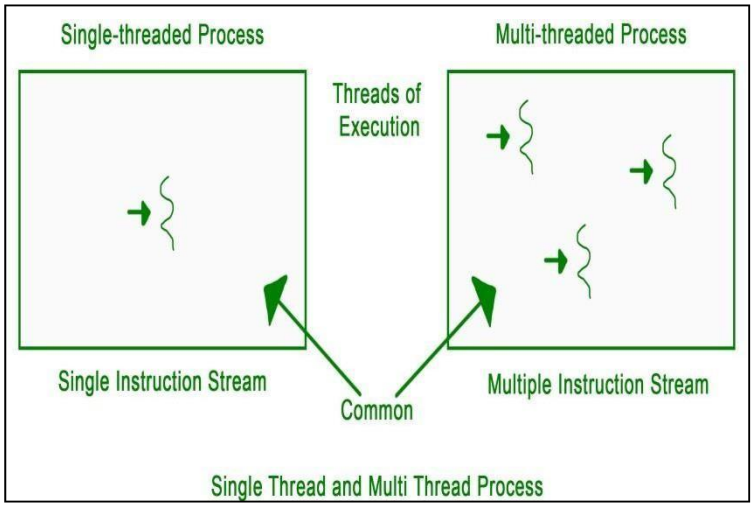
Threads:

A **thread** is a path which is followed during a program's execution. Majority of programs written now a days run as a single thread. Lets say, for example a program is not capable of reading keystrokes while making drawings. These tasks cannot be executed by the program at the same time. This problem can be solved through multitasking so that two or more tasks can be executed simultaneously.

Multitasking is of two types: Processor based and thread based. Processor based multitasking is totally managed by the OS, however multitasking through multithreading can be controlled by the programmer to some extent.

The concept of **multi-threading** needs proper understanding of these two terms – **a process and a thread**. A process is a program being executed. A process can be further divided into independent units known as threads.

A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.



Applications –

Threading is used widely in almost every field.

- Most widely it is seen over the internet now days where we are using transaction processing of every type like recharges, online transfer, banking etc.
- Threading is a segment which divide the code into small parts that are of very light weight and has less burden on CPU memory so that it can be easily worked out and can achieve goal in desired field.
- The concept of threading is designed due to the problem of fast and regular changes in technology and less the work in different areas due to less application

PROGRAM :

SOURCE CODE:

```
/* Implementing a program using thread */
#include<pthread.h>
#include<stdio.h>
#define NUM_THREADS 3
int je,jo,evensum=0,sumn=0,oddsum=0,evenarr[50],oddarr[50];
void *Even(void *threadid)
{
        int i,n;
        je=0;
        n=(int)threadid;
        for(i=1;i<=n;i++)
        {
                if(i%2==0)
                {
                        evenarr[je]=i;
                        evensum=evensum+i;
                        je++;
                }
        }
}
void *Odd(void *threadid)
{
        int i,n;
        jo=0;
        n=(int)threadid;
        for(i=0;i<=n;i++)
        {
                if(i%2!=0)
                {
                        oddarr[jo]=i;
                        oddsum=oddsum+i;
                        jo++;
                }
```

```
        }
}
void *SumN(void *threadid)
{
        int i,n;
        n=(int)threadid;
        for(i=1;i<=n;i++)
        {
                sumn=sumn+i;
        }
}
int main()
{
        pthread_t threads[NUM_THREADS];
        int i,t;
        printf("Enter a number\n");
        scanf("%d",&t);
        pthread_create(&threads[0],NULL,Even,(void *)t);
        pthread_create(&threads[1],NULL,Odd,(void *)t);
        pthread_create(&threads[2],NULL,SumN,(void *)t);
        for(i=0;i<NUM_THREADS;i++)
        {
                pthread_join(threads[i],NULL);
        }
        printf("The sum of first N natural nos is %d\n",sumn);
        printf("The sum of first N even natural nos is %d\n",evensum);
        printf("The sum of first N odd natural nos is %d\n",oddsum);
        printf("The first N even natural nos is --- \n");
        for(i=0;i<je;i++)
                printf("%d\n",evenarr[i]);
        printf("The first N odd natural nos is --- \n");
        for(i=0;i<jo;i++)
                printf("%d\n",oddarr[i]);
        pthread_exit(NULL);
}
```

OUTPUT:

```
$ vi threadf.c
$ cc threadf.c -pthread
$ ./a.out
Enter a number
12
The sum of first N natural nos is 78
The sum of first N even natural nos is 42
The sum of first N odd natural nos is 36
The first N even natural nos is----
2
4
6
8
10
12
The first N odd natural nos is----
1
3
5
7
9
11
```

RESULT:

Thus the program was executed and verified successfully.

3. AIM :Write C programs to simulate CPU scheduling algorithms: FCFS, SJF, and Round Robin.

CPU Scheduling Algorithms:

Scheduling of processes/work is done to finish the work on time.

Below are different times with respect to a process.

Arrival Time : Time at which the process arrives in the ready queue.

Completion Time : Time at which process completes its execution.

Burst Time : Time required by a process for CPU execution.

Turn Around Time : Time Difference between completion time and arrival time.

Turn Around Time = Completion Time - Arrival Time

Waiting Time(W.T) : Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time - Burst Time

Why do we need scheduling?

A typical process involves both I/O time and CPU time. In a uniprogramming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time. In multiprogramming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

Objectives of Process Scheduling Algorithm

- Max CPU utilization [Keep CPU as busy as possible]
- Fair allocation of CPU.
- Max throughput [Number of processes that complete their execution per time unit]
- Min turnaround time [Time taken by a process to finish execution]
- Min waiting time [Time a process waits in ready queue]
- Min response time [Time when a process produces first response]

Different Scheduling Algorithms:

First Come First Serve (FCFS): Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non – preemptive scheduling algorithm.

Note: First come first serve suffers from convoy effect.

Shortest Job First (SJF): Process which has the shortest burst time is scheduled first. If two processes have the same bust time then FCFS is used to break the tie. It is a non-preemptive scheduling algorithm.

Longest Job First (LJF): It is similar to SJF scheduling algorithm. But, in this scheduling algorithm, we give priority to the process having the longest burst time. This is non – preemptive in nature i.e., when any process starts executing, can't be interrupted before complete execution.

Shortest Remaining Time First (SRTF): It is preemptive mode of SJF algorithm in which jobs are schedule according to shortest remaining time.

Longest Remaining Time First (LRTF): It is preemptive mode of LJF algorithm in which we give priority to the process having largest burst time remaining.

Round Robin Scheduling: Each process is assigned a fixed time (Time Quantum/Time Slice) in cyclic way. It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a FIFO queue o£ processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail o£ the ready queue. The CPU scheduler will then select the next process in the ready queue.

Priority Based scheduling (Non - Preemptive): In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is scheduled first. If priorities of two processes match, then schedule according to arrival time. Here starvation of process is possible.

Highest Response Ratio Next (HRRN) In this scheduling, processes with highest response ratio is scheduled. This algorithm avoids starvation.

Response Ratio = (Waiting Time + Burst time) / Burst time

Multilevel Queue Scheduling: According to the priority of process, processes are placed in the different queues. Generally high priority process is placed in the top level queue. Only after completion of processes from top level queue, lower level queued processes are scheduled. It can suffer from starvation.

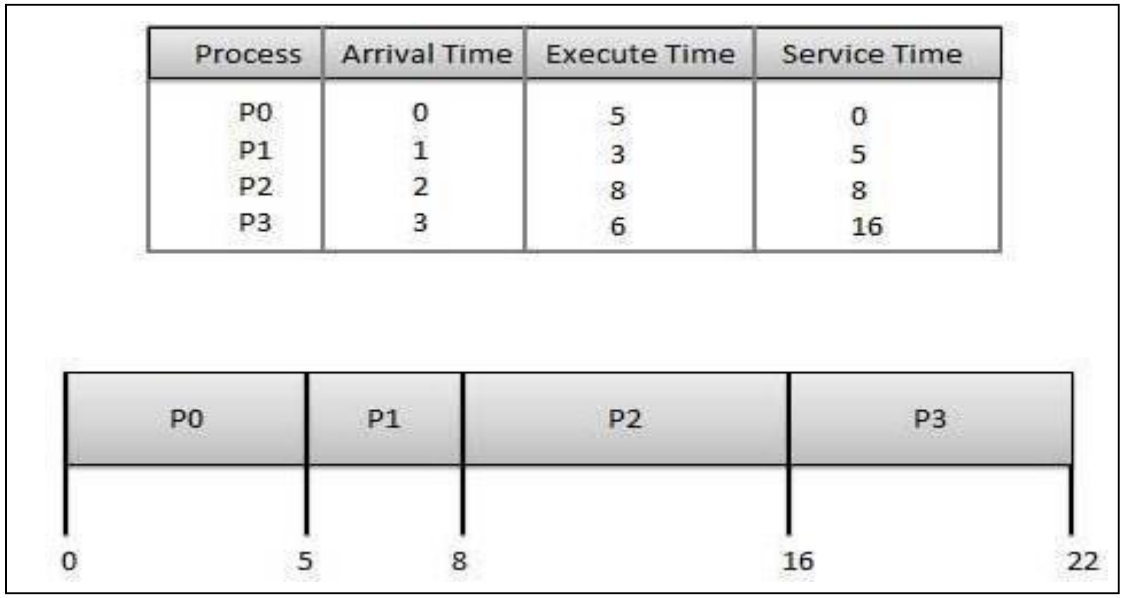
Multi level Feedback Queue Scheduling: It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue.

3.A) Write C program to simulate FCFS CPU scheduling algorithm.

AIM:To write a C program to implement FCFS CPU scheduling algorithm.

DESCRIPTION:

- Jobs are executed on FCFS (First Come, First Serve) basis.
- It is a non-preemptive, preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO (First In First Out) queue.
- Poor in performance as average wait time is high.



Wait time of each process is as follows –

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

ALGORITHM:

Step 1: Start the program.

Step 2: Create the number of process.

Step 3: Get the ID and Service time for each process.

Step 4: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.

Step 5: Calculate the Total time and Processing time for the remaining processes.

Step 6: Waiting time of one process is the Total time of the previous process.

Step 7: Total time of process is calculated by adding Waiting time and Service time.

Step 8: Total waiting time is calculated by adding the waiting time for lack process.

Step 9: Total turn around time is calculated by adding all total time of each process.

Step 10: Calculate Average waiting time by dividing the total waiting time by total number of process.

Step 11: Calculate Average turn around time by dividing the total time by the number of process.

Step 12: Display the result.

Step 13: Stop the program.

PROGRAM:

SOURCE CODE:

```
/* A program to simulate the FCFS CPU scheduling algorithm */
#include<stdio.h>
int main()
{
        char pn[10][10];
        int arr[10],bur[10],star[10],finish[10],tat[10],wt[10],i,n;
        int totwt=0,tottat=0;
        printf("Enter the number of processes:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter the Process Name, Arrival Time & Burst Time:");
                scanf("%s%d%d",&pn[i],&arr[i],&bur[i]);
        }
        for(i=0;i<n;i++)
        {
```

```
            if(i==0)
            {
                star[i]=arr[i];
                wt[i]=star[i]-arr[i];
                finish[i]=star[i]+bur[i];
                tat[i]=finish[i]-arr[i];
            }
            else
            {
                star[i]=finish[i-1];
                wt[i]=star[i]-arr[i];
                finish[i]=star[i]+bur[i];
                tat[i]=finish[i]-arr[i];
            }
        }
        printf("\nPName        Arrtime Burtime Start TAT Finish");
        for(i=0;i<n;i++)
        {
            printf("\n%s\t%6d\t\t%6d\t%6d\t%6d\t%6d",pn[i],arr[i],bur[i],star[i],tat[i],finish[i]);
            totwt+=wt[i];
            tottat+=tat[i];
        }
        printf("\nAverage Waiting time:%f", (float)totwt);
        printf("\nAverage Turn Around Time:%f", (float)tottat);
}
```

OUTPUT:

```
$ vi fcfs.c
$ cc fcfs.c
$ ./a.out
Enter the number of processes: 3
Enter the Process Name, Arrival Time & Burst Time: 1 2 3
Enter the Process Name, Arrival Time & Burst Time: 2 5 6
Enter the Process Name, Arrival Time & Burst Time: 3 6 7
```

PName	Arrtime	Burtime	Start	TAT	Finish
1	2	3	2	3	5
2	5	6	5	6	11
3	6	7	11	12	18

```
Average Waiting time: 1.666667
Average Turn Around Time: 7.000000
```

RESULT:

Thus the program was executed and verified successfully.

3.B) Write C program to simulate SJF CPU scheduling algorithm.

AIM:To write a C program to implement SJF CPU scheduling algorithm.

DESCRIPTION:

- This is also known as shortest job first, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time

Process	Arrival Time	Execution Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	14
P3	3	6	8

Waiting time of each process is as follows –

Process	Waiting Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	14 - 2 = 12
P3	8 - 3 = 5

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of process.

Step 3: Get the id and service time for each process.

Step 4: Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.

Step 5: Calculate the total time and waiting time of remaining process.

Step 6: Waiting time of one process is the total time of the previous process.

Step 7: Total time of process is calculated by adding the waiting time and service time of each process.

Step 8: Total waiting time calculated by adding the waiting time of each process.

Step 9: Total turn around time calculated by adding all total time of each process.

Step 10: Calculate average waiting time by dividing the total waiting time by total number of process.

Step 11: Calculate average turn around time by dividing the total waiting time by total number of process.

Step 12: Display the result.

Step 13: Stop the program.

PROGRAM:

SOURCE CODE:

```
/* A program to simulate the SJF CPU scheduling algorithm */
#include<stdio.h>
#include<string.h>
main()
{
        int i=0,pno[10],bt[10],n,wt[10],temp=0,j,tt[10];
        float sum,at;
        printf("\n Enter the no of process ");
        scanf("\n %d",&n);
        printf("\n Enter the burst time of each process");
        for(i=0;i<n;i++)
        {
                printf("\n p%d",i);
                scanf("%d",&bt[i]);
```

```
        }
        for(i=0;i<n-1;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(bt[i]>bt[j])
                        {
                                temp=bt[i];
                                bt[i]=bt[j];
                                bt[j]=temp;
                                temp=pno[i];
                                pno[i]=pno[j];
                                pno[j]=temp;
                        }
                }
        }
        wt[0]=0;
        for(i=1;i<n;i++)
        {
                wt[i]=bt[i-1]+wt[i-1];
                sum=sum+wt[i];
        }
        printf("\n process no \t burst time\t waiting time \t turn around time\n");
        for(i=0;i<n;i++)
        {
                tt[i]=bt[i]+wt[i];
                at+=tt[i];
                printf("\n p%d\t\t%d\t\t%d\t\t%d",i,bt[i],wt[i],tt[i]);
        }
        printf("\n\n\t Average waiting time%f\n\t Average turn around time%f", sum, at);
}
```

OUTPUT:

```
$ vi sjf.c
$ cc sjf.c
$ ./a.out
Enter the no of process 5
Enter the burst time of each process
 p0      1
 p1      5
 p2      2
 p3      3
 p4      4

process no      burst time      waiting time    turn around time
 p0              1               0               1
 p1              2               1               3
 p2              3               3               6
 p3              4               6               10
 p4              5               10              15

Average waiting time 4.000000
Average turn around time 7.000000
```

RESULT:

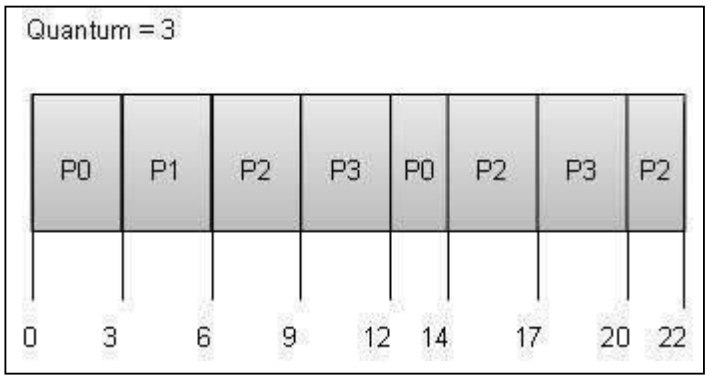
Thus the program was executed and verified successfully.

3.C) Write C program to simulate Round Robin CPU scheduling algorithm.

AIM:**To write a C program to implement Round Robin CPU scheduling algorithm.**

DESCRIPTION

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.



Wait time of each process is as follows –

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

ALGORITHM:

Step 1: Start the program.

Step 2: Initialize all the structure elements.

Step 3: Receive inputs from the user to fill process id, burst time and arrival time.

Step 4: Calculate the waiting time for all the process id.

i. The waiting time for first instance of a process is calculated as: a[i].waittime=count + a[i].arrivt.

ii. The waiting time for the rest of the instances of the process is calculated as:

a) If the time quantum is greater than the remaining burst time then waiting time is calculated as: a[i].waittime=count + tq.

b) Else if the time quantum is greater than the remaining burst time then waiting time is calculated as: a[i].waittime=count - remaining burst time

Step 5: Calculate the average waiting time and average turnaround time

Step 6: Print the results of the step 4.

Step 7: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* A program to simulate the Round Robin CPU scheduling algorithm */
#include<stdio.h>
struct process
{
        int burst,wait,comp,f;
}p[20]={0,0};
int main()
{
        int n,i,j,totalwait=0,totalturn=0,quantum,flag=1,time=0;
        printf("\nEnter The No Of Process     :");
        scanf("%d",&n);
        printf("\nEnter The Quantum time (in ms) :");
        scanf("%d",&quantum);
        for(i=0;i<n;i++)
        {
                printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);
                scanf("%d",&p[i].burst);
```

```
        p[i].f=1;
}
printf("\nOrder Of Execution \n");
printf("\nProcess Starting Ending Remaining");
printf("\n\t\tTime \tTime \t Time");
while(flag==1)
{
        flag=0;
        for(i=0;i<n;i++)
        {
                if(p[i].f==1)
                {
                        flag=1;
                        j=quantum;
                        if((p[i].burst-p[i].comp)>quantum)
                        {
                                p[i].comp+=quantum;
                        }
                        else
                        {
                                p[i].wait=time-p[i].comp;
                                j=p[i].burst-p[i].comp;
                                p[i].comp=p[i].burst;
                                p[i].f=0;
                        }
                        printf("\nprocess # %-3d %-10d %-10d %-10d", i+1, time, time+j,
                p[i].burst-p[i].comp);
                        time+=j;
                }
        }
}
printf("\n\n----------");
printf("\nProcess \t Waiting Time TurnAround Time ");
for(i=0;i<n;i++)
{
```

```
            printf("\nProcess # %-12d%-15d%-15d",i+1,p[i].wait,p[i].wait+p[i].burst);
            totalwait=totalwait+p[i].wait;
            totalturn=totalturn+p[i].wait+p[i].burst;
        }
        printf("\n\nAverage\n----------------          ");
        printf("\nWaiting Time: %fms",totalwait/(float)n);
        printf("\nTurnAround Time : %fms\n\n",totalturn/(float)n);
        return 0;
}
```

OUTPUT:

```
$ vi rr.c
$ cc rr.c
$ ./a.out
Enter The No Of Process: 3
Enter The Quantum time (in ms): 5
Enter The Burst Time (in ms) For Process # 1: 25
Enter The Burst Time (in ms) For Process # 2: 30
Enter The Burst Time (in ms) For Process # 3: 54
Order Of Execution
```

Process	Starting Time	Ending Time	Remaining Time
process # 1	0	5	20
process # 2	5	10	25
process # 3	10	15	49
process # 1	15	20	15
process # 2	20	25	20
process # 3	25	30	44
process # 1	30	35	10
process # 2	35	40	15
process # 3	40	45	39
process # 1	45	50	5
process # 2	50	55	10
process # 3	55	60	34
process # 1	60	65	0

process # 2	65	70	5
process # 3	70	75	29
process # 2	75	80	0
process # 3	80	85	24
process # 3	85	90	19
process # 3	90	95	14
process # 3	95	100	9
process # 3	100	105	4
process # 3	105	109	0

Process	Waiting Time	TurnAround Time
Process # 1	40	65
Process # 2	50	80
Process # 3	55	109

Average

Waiting Time: 48.333333ms

TurnAround Time: 84.666667ms

RESULT:

Thus the program was executed and verified successfully.

4. AIM: Write C programs to simulate Intra & Inter – Process Communication (IPC) techniques: Pipes, Messages Queues, and Shared Memory.

DESCRIPTION:

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

Communication can be of two types:

Between related processes initiating from only one process, such as parent and child processes.

Between unrelated processes, or two or more different processes.

Following are some important terms that we need to know before proceeding further on this topic.

Pipes − Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

FIFO − Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

Message Queues − Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.

Shared Memory − Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.

Semaphores − Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.

Signals − Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

Note − Almost all the programs in this tutorial are based on system calls under Linux Operating System (executed in Ubuntu).

PROGRAM 4.A) ECHOSERVER USING PIPES

SOURCE CODE:

```
/* Implementation of echoserver using Pipes */
#include<stdio.h>
#include<string.h>
#define msgsize 29
int main()
{
        int ser[2],cli[2],pid;
        char inbuff[msgsize];
        char *msg="Thank you";
        system("clear"); pipe(ser);
        pipe(cli);
        printf("\n server read id =%d,write id=%d",ser[0],ser[1]);
        printf("\n client read id =%d,write id=%d",cli[0],cli[1]);
        pid=fork();
        if(pid==0)
        {
                printf("\n i am in child process !");
                close(cli[0]);
                close(ser[1]);
                write(cli[1],msg,msgsize);
                printf("\n message written to pipe..!");
                sleep(2);
                read(ser[0],inbuff,msgsize);
                printf("\n echo message received from server");
                printf("\n %s",inbuff);
        }
        else
        {
                close(cli[1]);
                close(ser[0]);
                printf("\n parent process");
                read(cli[0],inbuff,msgsize);
```

```
                write(ser[1],inbuff,msgsize);
                printf("\n parent ended!");
        }
}
```

OUTPUT:

```
$ vi echopipe.c
$ cc echopipe.c
$ ./a.out

 server read id =3,write id=4
 client read id =5,write id=6
 i am in child process !
 client read id =5,write id=6
 parent process
 parent ended!

$  message written to pipe..!
 echo message received from server
```

RESULT:

Thus the program was executed and verified successfully.

PROGRAM 4.B) ECHO SERVER USING MESSAGES

SOURCE CODE:

```
/* Implementation of echo server using messages */
#include<sys/ipc.h>
#include<stdio.h>
#include<string.h>
#include<sys/msg.h>
#include<stdlib.h>
struct
{
        long mtype;
        char mtext[20];
}send,recv;
main()
{
        int qid,pid,len;
        qid=msgget((key_t)0X2000,IPC_CREAT|0666);
        if(qid==-1)
        {
                perror("\n message failed");
                exit(1);
        }
        send.mtype=1;
        strcpy(send.mtext,"\n hello i am parent");
        len=strlen(send.mtext);
        pid=fork();
        if(pid>0)
        {
                if(msgsnd(qid,&send,len,0)==-1)
                {
                        perror("\n message sending failed");
                        exit(1);
                }
                printf("\n message has been posted");
                sleep(2);
```

```
                if(msgrcv(qid,&recv,100,2,0)==-1)
                {
                        perror("\n msgrcv error:");
                        exit(1);
                }
                printf("\n message received from child - %s\n",recv.mtext);
        }
        else
        {
                send.mtype=2;
                strcpy(send.mtext,"\n hi i am child");
                len=strlen(send.mtext);
                if(msgrcv(qid,&recv,100,1,0)==-1)
                {
                        perror("\n child message received failed");
                        exit(1);
                }
                if(msgsnd(qid,&send,len,0)==-1)
                {
                        perror("\n child message send failed");
                }
                printf("\n received from parent - %s",recv.mtext);
        }
}
```

OUTPUT:

```
$ vi echomsg.c
$ cc echomsg.c
$ ./a.out
 received from parent -
 hello i am parent message has been posted
 message received from child -
 hi i am child
```

RESULT:

Thus the program was executed and verified successfully.

5. AIM: Write C programs to simulate solutions to Classical Process Synchronization Problems: Dining Philosophers, Producer – Consumer, Readers – Writers.

Process Synchronization

The basic concepts of synchronization have been presented, and several case studies have been implemented as simulation models. The need for synchronization originates when processes need to execute concurrently. The main purpose of synchronization is the sharing of resources without interference using mutual exclusion. The other purpose is the coordination of the process interactions in an operating system. Semaphores and monitors are the most powerful and most commonly used mechanisms to solve synchronization problems. Most operating systems provide semaphores. The producer-consumer problem (also known as the bounded-buffer problem), and the readers-writers problem are two classical case studies considered to describe and test synchronization mechanisms. These two problems are implemented in various simulation models, each with a different solution or different workload parameters to study the process synchronization mechanisms that can be used with interacting processes.

5.A) Dining Philosophers Problem.

AIM:To write C programs to simulate solutions to Dining Philosophers Problem.

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the program.

Step 2: Define the number of philosophers.

Step 3: Declare one thread per philosopher.

Step 4: Declare one chopsticks per philosopher.

Step 5: When a philosopher is hungry.

i. See if chopsticks on both sides are free.
ii. Acquire both chopsticks or.
iii. Eat.
iv. restore the chopsticks.
v. If chopsticks aren't free.

Step 6: Wait till they are available.

Step 7: Stop the program.

PROGRAM:

SOURCE CODE:

```
int tph, philname[20], status[20], howhung, hu[20], cho;
int main()
{
        int i;
        clrscr();
        printf("\n\nDINING PHILOSOPHER PROBLEM");
        printf("\nEnter the total no. of philosophers: ");
        scanf("%d",&tph);
        for(i=0;i<tph;i++)
        {
                philname[i] = (i+1);
                status[i]=1;
        }
        printf("How many are hungry : ");
        scanf("%d", &howhung);
        if(howhung==tph)
        {
                printf("\nAll are hungry..\nDead lock stage will occur");
                printf("\nExiting..");
        }
        else
        {
                for(i=0;i<howhung;i++)
                {
```

```
                printf("Enter philosopher %d position: ",(i+1));
                scanf("%d", &hu[i]);
                status[hu[i]]=2;
        }
        do
        {
                printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your
        choice:");
                scanf("%d", &cho);
                switch (cho)
                {
                        case 1: one();
                                break();
                        case 2: two();
                                break;
                        case 3: exit(0);
                        default: printf("\nInvalid option..");
                }
        }
        while(1);
    }
}
one()
{
    int pos=0, x, i;
    printf("\nAllow one philosopher to eat at any time\n");
    for(i=0;i<howhung; i++, pos++)
    {
        printf("\nP %d is granted to eat", philname[hu[pos]]);
        for(x=pos;x<howhung;x++)
        printf("\nP %d is waiting", philname[hu[x]]);
    }
}
two()
{
```

```
        int i, j, s=0, t, r, x;
        printf("\n Allow two philosophers to eat at same time\n");
        for(i=0;i<howhung;i++)
        {
                for(j=i+1;j<howhung;j++)
                {
                        if(abs(hu[i]-hu[j])>=1&&  abs(hu[i]-hu[j])!=4)
                        {
                                printf("\n\ncombination %d \n", (s+1));
                                t=hu[i];
                                r=hu[j];
                                s++;
                                printf("\nP %d and P %d are granted to eat", philname[hu[i]],
                                philname[hu[j]]);
                                for(x=0;x<howhung;x++)
                                {
                                        if((hu[x]!=t)&&(hu[x]!=r))
                                        printf("\nP %d is waiting", philname[hu[x]]);
                                }
                        }
                }
        }
}
```

OUTPUT:

DINING PHILOSOPHER PROBLEM

Enter the total no. of philosophers: 5

How many are hungry : 3

Enter philosopher 1 position: 2

Enter philosopher 2 position: 4

Enter philosopher 3 position: 5

1.One can eat at a time 2.Two can eat at a time 3.Exit

Enter your choice: 1

Allow one philosopher to eat at any time

P 3 is granted to eat

P 3 is waiting

P 5 is waiting

P 0 is waiting

P 5 is granted to eat

P 5 is waiting

P 0 is waiting

P 0 is granted to eat

P 0 is waiting

1.One can eat at a time 2.Two can eat at a time 3.Exit

Enter your choice: 2

Allow two philosophers to eat at same time

combination 1

P 3 and P 5 are granted to eat

P 0 is waiting

combination 2

P 3 and P 0 are granted to eat

P 5 is waiting

combination 3

P 5 and P 0 are granted to eat

P 3 is waiting

1.One can eat at a time 2.Two can eat at a time 3.Exit

Enter your choice: 3

RESULT:

Thus the program was executed and verified successfully.

5)) Producer – Consumer Problem.

AIM:To write C programs to simulate Producer – Consumer Problem.

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the program.

Step 2: Declare and initialize the necessary variables.

Step 3: Create a Producer.

Step 4: Producer (Child Process) performs a down operation and writes a message.

Step 5: Producer performs an up operation for the consumer to consume.

Step 6: Consumer (Parent Process) performs a down operation and reads or consumes the data (message).

Step 7: Consumer then performs an up operation.

Step 8: Stop the program.

PROGRAM :

SOURCE CODE:

```
#include<stdio.h>
void main()
{
        int buffer[10], bufsize, in, out, produce, consume, choice=0;
        in = 0;
        out = 0;
        bufsize = 10;
        while (choice !=3)
        {
                printf("\n1. Produce \t 2. Consume \t3. Exit");
                printf("\nEnter your choice: ");
```

```
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: if((in+1)%bufsize==out)
                    printf("\nBuffer is Full");
                    else
                    {
                        printf("\nEnter the value: ");
                        scanf("%d", &produce); buffer[in]
                        = produce;
                        in = (in+1)%bufsize;
                    }
                    break;
                    case 2: if(in == out)
                        printf("\nBuffer is Empty");
                        else
                        {
                            consume = buffer[out];
                            printf("\nThe consumed value is %d", consume);
                            out = (out+1)%bufsize;
                        }
                        break;
            }
        }
}
```

OUTPUT:

Produce 2. Consume 3. Exit

Enter your choice: 2

Buffer is Empty

Produce 2. Consume 3. Exit

Enter your choice: 2

Enter the value: 100

Produce 2. Consume 3. Exit

Enter your choice: 1

The consumed values is 100

Produce 2. Consume 3. Exit

Enter your choice: 3

RESULT:

Thus the program was executed and verified successfully.

6) **C)Readers – Writers Problem.**

AIM:To write C programs to simulate Readers – Writers Problem.

DESCRIPTION:

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue. To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the readers – writer's problem.

ALGORITHM:

Step 1: Start the program.

Step 2: Declare and initialize the necessary variables.

Step 3: Create a Reader.

Step 4: Reader performs a down operation and read a message in the database.

Step 5: Reader performs an up operation for the Writer to access a message from the database.

Step 6: Write performs a down operation and writes or access the data (message) from the database.

Step 7: Writer then performs an up operation.

Step 8: Stop the program.

PROGRAM:

SOURCE CODE:

//File: mesg.h

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<stdio.h>
#include<stdlib.h>
#define MKEY1 5543L
#define MKEY2 4354L
#define PERMS 0666
typedef struct
{
        long mtype;
        char mdata[50];
```

```
}mesg;
```

//File: sender1.c

```
#include "mesg.h"
mesg msg;
int main()
{
        int mq_id;
        int n;
        if((mq_id=msgget(MKEY1,PERMS|IPC_CREAT))<0)
        {
                printf("Sender: Error creating message");
                exit(1);
        }
        msg.mtype=1111L;
        n=read(0,msg.mdata,50);
        msg.mdata[n]='\0';
        msgsnd(mq_id,&msg,50,0);
}
```

//File: receiver1.c

```
#include "mesg.h"
mesg msg;
main()
{
        int mq_id;
        int n;
        if( ( mq_id=msgget(MKEY1, PERMS|IPC_CREAT ) ) < 0)
        {
                printf("receiver: Error opening message");
                exit(1);
        }
        msgrcv(mq_id,&msg,50,1111L,0);
        write(1,msg.mdata,50);
        msgctl(mq_id,IPC_RMID,NULL);
}
```

OUTPUT:

$ vi mesg.h

$ vi sender1.c

$ vi receiver1.c

$ cc sender1.c

$ ./a.out

UIT-

RGPV

$ cc receiver1.c

$ ./a.out

UIT-

RGPV

RESULT:

Thus the program was executed and verified successfully.

6) Write a C program to simulate Bankers Algorithm for Deadlock Avoidance.

AIM:To write a C program to implement bankers algorithm for dead lock avoidance

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the Program.

Step 2: Obtain the required data through char and int datatypes.

Step 3: Enter the filename, index block.

Step 4: Print the file name index loop.

Step 5: File is allocated to the unused index blocks.

Step 6: This is allocated to the unused linked allocation.

Step 7: Stop the execution.

PROGRAM:

SOURCE CODE:

```
/* Bankers algorithm for Deadlock Avoidance */
#include<stdio.h>
struct process
{
        int allocation[3];
        int max[3];
        int need[3];
        int finish;
}p[10];
int main()
```

```
{
    int n,i,I,j,avail[3],work[3],flag,count=0,sequence[10],k=0;
    printf("\nEnter the number of process:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter the %dth process allocated resources:",i);
        scanf("%d%d%d",&p[i].allocation[0],&p[i].allocation[1],&p[i].allocation[2]);
        printf("\nEnter the %dth process maximum resources:",i);
        scanf("%d%d%d",&p[i].max[0],&p[i].max[1],&p[i].max[2]);
        p[i].finish=0;
        p[i].need[0]=p[i].max[0]-p[i].allocation[0];
        p[i].need[1]=p[i].max[1]-p[i].allocation[1];
        p[i].need[2]=p[i].max[2]-p[i].allocation[2];
    }
    printf("\nEnter the available vector:");
    scanf("%d%d%d",&avail[0],&avail[1],&avail[2]);
    for(i=0;i<3;i++)
        work[i]=avail[i];
    while(count!=n)
    {
        count=0;
        for(i=0;i<n;i++)
        {
            flag=1;
            if(p[i].finish==0)
                if(p[i].need[0]<=work[0])
                    if(p[i].need[1]<=work[1])
                        if(p[i].need[2]<=work[2])
                        {
                            for(j=0;j<3;j++)
                                work[j]+=p[i].allocation[j];
                            p[i].finish=1;
                            sequence[k++]=i;
                            flag=0;
```

```
                                        }
                    if(flag==1)
                            count++;
                }
        }
        count=0;
        for(i=0;i<n;i++)
                if(p[i].finish==1)
                        count++;
        printf("\n The safe sequence is:\t");
        if(count++==n)
                for(i=0;i<k;i++)
                        printf("%d\n",sequence[i]);
        else
                printf("SYSTEM IS NOT IN A SAFE STATE \n\n");
        return 0;
}
```

OUTPUT

```
$ vi bankersavoidance.c
$ cc bankersavoidance.c
$ ./a.out
Enter the number of process: 3
Enter the 0th process allocated resources: 1  2  3
Enter the 0th process maximum resources: 4  5  6
Enter the 1th process allocated resources: 3  4  5
Enter the 1th process maximum resources: 6  7  8
Enter the 2th process allocated resources: 1  2  3
Enter the 2th process maximum resources: 3  4  5
Enter the available vector: 10  12  11
The safe sequence is:  0  1  2
```

RESULT:

Thus the program was executed and verified successfully.

7. Write a C program to simulate Bankers Algorithm for Deadlock Detection(Additional).

AIM:To write a C program to implement Deadlock Detection algorithm

DESCRIPTION:

Deadlock Detection is an important task of OS. As the OS doesn't take many precautionary means to avoid it. The OS periodically checks if there is any existing deadlock in the system and take measures to remove the deadlocks.

Detection

There are 2 different cases in case of Deadlock detection –

- If resource has single Instance
  - We make a Wait
- If resources have multiple instances
  - We make a different algorithm.

Deadlock Recovery

Deadlock can be recovered by

1) **Kill the Process –** One way is to kill all the process in deadlock or the second way kill the process one by one, and check after each if still deadlock exists and do the same till the deadlock is removed.
2) **Pre – emption –** The resources that are allocated to the processes involved in deadlock are taken away (pre – empted) and are transferred to other processes. In this way, system may recover from deadlock as we may change system state.
3) **Rollback –** The OS maintains a database of all different states of system, a state when the system is not in deadlock is called safe state. A rollback to previous 'n' number of safe states in iterations can help in the recover.

ALGORITHM:

Step 1: Start the Program.

Step 2: Obtain the required data through char and in data types.

Step 3: Enter the filename, index block.

Step 4: Print the file name index loop.

Step 5: File is allocated to the unused index blocks.

Step 6: This is allocated to the unused linked allocation.

Step 7: Stop the execution.

PROGRAM:

SOURCE CODE:

```
/* Bankers algorithm for Deadlock detection */
#include<stdio.h>
void main()
{
	int found,flag,l,p[4][5],tp,c[4][5],i,j,k=1,m[5],r[5],a[5],temp[5],sum=0;
	printf("enter total no of processes: \n");
	scanf("%d",&tp);
	printf("enter clain matrix: \n");
	for(i=0;i<4;i++)
	{
		for(j=0;j<5;j++)
		{
			scanf("%d",&c[i][j]);
		}
	}
	printf("enter allocation matrix: \n");
	for(i=0;i<4;i++)
	{
		for(j=0;j<5;j++)
		{
			scanf("%d",&p[i][j]);
		}
	}
	printf("enter resource vector: \n");
	for(i=0;i<5;i++)
	{
		scanf("%d",&r[i]);
	}
	printf("enter availability vector: \n");
	for(i=0;i<5;i++)
	{
		scanf("%d",&a[i]);
```

```
        temp[i]=a[i];
}
for(i=0;i<4;i++)
{
        sum=0;
        for(j=0;j<5;j++)
        {
                sum+=p[i][j];
        }
        if(sum==0)
        {
                 m[k]=i;
                k++;
        }
}
for(i=0;i<4;i++)
{
        for(l=1;l<k;l++)
        {
                if(i!=m[l])
                {
                        flag=1;
                        for(j=0;j<5;j++)
                        {
                                if(c[i][j]>temp[j])
                                {
                                        flag=0;
                                        break;
                                }
                        }
                }
        }
        if(flag==1)
        {
                m[k]=i;
```

```
                        k++;
                        for(j=0;j<5;j++)
                                temp[j]+=p[i][j];
                }
        }
        printf("deadlock causing processes are: \n");
        for(j=0;j<tp;j++)
        {
                found=0;
                for(i=1;i<k;i++)
                {
                        if(j==m[i])
                                found=1;
                }
                if(found==0)
                        printf("%d\t",j);
        }
}
```

OUTPUT:

```
$ vi bankersdetection.c
$ cc bankersdetection.c
$ ./a.out
enter total no of processes:
4
enter clain matrix:
0  1  0  0  1
0  0  1  0  1
0  0  0  0  1
1  0  1  0  1
enter allocation matrix:
1  0  1  1  0
1  1  0  0  0
0  0  0  1  0
0  0  0  0  0
enter resource vect r:
2  1  1  2  1
enter availability vector:
0  0  0  0  1
deadlock causing processes are:
0  1
```

RESULT:

Thus the program was executed and verified successfully.

8. Write C programs to simulate Page Replacement Algorithms: FIFO, LRU.

Page Replacement Algorithms:

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.

8.A) FIFO.

AIM:To Simulate FIFO page replacement algorithms.

ALGORITHM:

Step 1: Start the program

Step 2: Read the number of frames

Step 3: Read the number of pages

Step 4: Read the page numbers

Step 5: Initialize the values in frames to -1

Step 6: Allocate the pages in to frames in First in first out order.

Step 7: Display the number of page faults.

Step 8: Stop the program

PROGRAM:

SOURCE CODE:

```
/* A program to simulate FIFO Page Replacement Algorithm */
#include<stdio.h>
main()
{
        int a[5],b[20],n,p=0,q=0,m=0,h,k,i,q1=1;
        char f='F';
```

```
printf("Enter the Number of Pages:");
scanf("%d",&n);
printf("Enter %d Page Numbers:",n);
for(i=0;i<n;i++)
        scanf("%d",&b[i]);
for(i=0;i<n;i++)
{
        if(p==0)
        {
                if(q>=3)
                        q=0;
                a[q]=b[i];
                q++;
                if(q1<3)
                {
                        q1=q;
                }
        }
        printf("\n%d",b[i]);
        printf("\t");
        for(h=0;h<q1;h++)
                printf("%d",a[h]);
        if((p==0)&&(q<=3))
        {
                printf("-->%c",f);
                m++;
        }
        p=0;
        for(k=0;k<q1;k++)
        {
                if(b[i+1]==a[k])
                        p=1;
        }
}
printf("\nNo of faults:%d",m);
```

```
}
```

OUTPUT:

```
$ vi fifo.c
$ cc fifo.c
$ ./a.out

Enter the Number of Pages: 12
Enter 12 Page Numbers:
2       3       2       1       5       2       4       5       3       2       5       2

2       2-->F
3       23-->F
2       23
1       231-->F
5       531-->F
2       521-->F
4       524-->F
5       524
3       324-->F
2       324
5       354-->F
2       352-->F

No of faults: 9
```

RESULT:

Thus the program was executed and verified successfully.

8.B) LRU

AIM:To Simulate LRU page replacement algorithms.

ALGORITHM:

Step 1: Start the program.

Step 2: Read the number of frames.

Step 3: Read the number of pages.

Step 4: Read the page numbers.

Step 5: Initialize the values in frames to -1.

Step 6: Allocate the pages in to frames by selecting the page that has not been used for the longest period of time.

Step 7: Display the number of page faults.

Step 8: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* A program to simulate LRU Page Replacement Algorithm */
#include<stdio.h>
main()
{
        int a[5],b[20],p=0,q=0,m=0,h,k,i,q1=1,j,u,n;
        char f='F';
        printf("Enter the number of pages:");
        scanf("%d",&n);
        printf("Enter %d Page Numbers:",n);
        for(i=0;i<n;i++)
                scanf("%d",&b[i]);
        for(i=0;i<n;i++)
        {
                if(p==0)
                {
                        if(q>=3)
                                q=0;
                        a[q]=b[i];
                        q++;
```

```
        if(q1<3)
        {
                q1=q;
        }
}
printf("\n%d",b[i]);
printf("\t");
for(h=0;h<q1;h++)
        printf("%d",a[h]);
if((p==0)&&(q<=3))
{
        printf("-->%c",f);
        m++;
}
p=0;
if(q1==3)
{
        for(k=0;k<q1;k++)
        {
                if(b[i+1]==a[k])
                        p=1;
        }
        for(j=0;j<q1;j++)
        {
                u=0;
                k=i;
                while(k>=(i-1)&&(k>=0))
                {
                        if(b[k]==a[j])
                                u++;
                        k--;
                }
                if(u==0)
                        q=j;
        }
```

```
                }
                else
                {
                        for(k=0;k<q;k++)
                        {
                                if(b[i+1]==a[k])
                                        p=1;
                        }
                }
        }
        printf("\nNo of faults:%d",m);
}
```

OUTPUT:

```
$ vi lru.c
$ cc lru.c
$ ./a.out
Enter the number of pages: 12
Enter 12 Page Numbers:
2       3       2       1       5       2       4       5       3       2       5       2
2       2-->F
3       23-->F
2       23
1       231-->F
5       251-->F
2       251
4       254-->F
5       254
3       354-->F
2       352-->F
5       352
2       352
No of faults: 7
```

RESULT:

Thus the program was executed and verified successfully.

9. Write C programs to simulate implementation of Disk Scheduling Algorithms: FCFS, SSTF.

A] Aim : Write C programs to simulate implementation FCFS Disk Scheduling Algorithm

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37, 122, 14, 124, 65, 67. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

```
#include<stdio.h>
int main()
{
        int queue[20],n,head,i,j,k,seek=0,max,diff;
        float avg;
        printf("Enter the max range of disk\n");
        scanf("%d",&max);
        printf("Enter the size of queue request\n");
        scanf("%d",&n);
        printf("Enter the queue of disk positions to be read\n");
        for(i=1;i<=n;i++)
        scanf("%d",&queue[i]);
        printf("Enter the initial head position\n");
        scanf("%d",&head);
        queue[0]=head;
        for(j=0;j<=n-1;j++)
        {
                diff=abs(queue[j+1]-queue[j]);
                seek+=diff;
                printf("Disk head moves from %d to %d with
seek                                                    %d\n",queue[j],queue[j+1],diff);
        }
        printf("Total seek time is %d\n",seek);
        avg=seek/(float)n;
        printf("Average seek time is %f\n",avg);
        return 0;
}
```

OUTPUT

Enter the max range of disk

200

Enter the size of queue request

8

Enter the queue of disk positions to be read

90 120 35 122 38 128 65 68

Enter the initial head position

50

Disk head moves from 50 to 90 with seek

40

Disk head moves from 90 to 120 with seek

30

Disk head moves from 120 to 35 with seek

85

Disk head moves from 35 to 122 with seek

87

Disk head moves from 122 to 38 with seek

84

Disk head moves from 38 to 128 with seek

90

Disk head moves from 128 to 65 with seek

63

Disk head moves from 65 to 68 with seek

3

Total seek time is 482

Average seek time is 60.250000

RESULT:

Thus the program was executed and verified successfully.

B] Aim : Write C programs to simulate implementation SSTF Disk Scheduling Algorithm

SSTF stands for Shortest Time First which very uses full of learning about how the disk drive manages the data having the shortest seek time.

Algorithm :

1. Let Request array represents an array storing indexes of tracks that have been requested. 'head' is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int main()
{
int queue[100],t[100],head,seek=0,n,i,j,temp;
float avg;
// clrscr();
printf("*** SSTF Disk Scheduling Algorithm ***\n");
printf("Enter the size of Queue\t");
scanf("%d",&n);
printf("Enter the Queue\t");
for(i=0;i<n;i++)
{
scanf("%d",&queue[i]);
}
printf("Enter the initial head position\t");
scanf("%d",&head);
for(i=1;i<n;i++)
t[i]=abs(head-queue[i]);
for(i=0;i<n;i++)
```

```
{
for(j=i+1;j<n;j++)
{
if(t[i]>t[j])
{
temp=t[i];  t[i]=t[j];
t[j]=temp;
temp=queue[i];
queue[i]=queue[j];
queue[j]=temp;
}
}
}
for(i=1;i<n-1;i++)
{
seek=seek+abs(head-queue[i]);
head=queue[i];
}
printf("\nTotal Seek Time is%d\t",seek);
avg=seek/(float)n;
printf("\nAverage Seek Time is %f\t",avg);
return 0;
}
```

OUTPUT:

*** SSTF Disk Scheduling Algorithm ***

Enter the size of Queue 5
Enter the Queue 10 17 2 15 4
Enter the initial head position 3
Total Seek Time is14
Average Seek Time is 2.800000

RESULT:

Thus the program was executed and verified successfully.

ADDITIONAL PROGRAMS

11. SHELL PROGRAMMING

1. Shell or the Command interpreter is the mediator which interprets the commands and then conveys them to the kernel which ultimately executes them.
2. Kernel is usually stored in a file called 'UNIX' where as the shell program in a file called 'sh'.
3. Types of shells:-.
    i. Bourne shell (sh) or Bourne again shell (bash)
    ii. C shell (csh)
    iii. Korn shell (ksh)
4. A shell program is nothing but a series of unix commands.
5. Instead of specifying one job at a time, the shell is given a to-do-list – a program – that carries out an entire procedure.
6. Such programs are known as shell scripts.

Shell programming language incorporates most of the features that most modern day programming languages offer.

Shell variables –

Rules for building shell variables are as follows:

1) A variable name is any combination of alphabets, digits and an underscore ('_').
2) No commas or blanks are allowed within a variable name.
3) The first character of a variable name must either be an alphabet or an underscore.
4) Variable names should be of any reasonable length.
5) Variable names are case sensitive.

Keywords for accepting input – read

Displaying output - echo

Assigning value to variables –

Values can be assigned to variables through read statement or also by using a simple assignment operator. For ex: age=30

Note : While assigning values to variables using assignment operator, no spaces to be given on either side of it. If the variable doesn't exist it will be created and value assigned

1. Unix-defined variables or System variables or Environment variables

2. User- defined variables

Note : To print or access value of a variable use '$' .

For ex: To print value of variable 'flag' write - echo $flag

Arithmetic in Shell script -

1. All shell variables are string variables, hence to carry out arithmetic operations use expr

command which evaluates arithmetic expressions.

2. More than one assignment can be done in a single statement.

3. Before and at the end of expr keyword use ` (back quote) sign not the (single quote i.e. ') sign which is generally above TAB key.

4. Terms of the expression provided to expr must be separated by blanks. Thus expression expr 10+20 is invalid.

5. The '*" symbol must be preceded by a \ ,otherwise the shell treats it as a wildcard character for all files in the current directory

OPERATORS USED IN SHELL SCRIPT –

OPERATOR MEANING

–gt Greater than

–lt Less than

–ge Greater than or equal to

–le Less than or equal to

–ne Not equal to

–eq Equal to

–a Logical AND

–o Logical OR

! Logical NOT

CONTROL INSTRUCTIONS IN SHELLS -

There are four types of control instructions in shell :

• Sequence Control Instruction.

• Selection or Decision control Instruction

• Repetition or Loop control Instruction

• Case Control Instruction

Decision statements –

If-then-else-fi statements:

if condition then Commands else Commands fi

1) For statements :

for control variable in value1 value 2 value3 do

Command list done

2) While statements : while control command do

Command list Done

3) Until statements:

until control command do

Command list done

Case statements:-

case value in

choice 1) commands;

choice 2) commands;

esac.

Steps to write and execute a script

1. Open the terminal. Go to the directory where you want to create your script.
2. Create a file with . sh extension.
3. Write the script in the file using an editor.
4. Make the script executable with command chmod +x <fileName>.
5. Run the script using ./<fileName>.

11.A) Concatenate two strings.

AIM:To write a shell program to concatenate two strings.

ALGORITHM:

Step 1: Start the program.

Step 2: Enter into the vi editor and go to the insert mode for entering the code.

Step 3: Read the first string.

Step 4: Read the second string.

Step 5: Concatenate the two strings.

Step 6: Enter into the escape mode for the execution of the result and verify the output.

Step 7: Stop the program.

SOURCE CODE

```
echo “enter the first string”

read str1

echo “enter the second string”

read str2

echo “the concatenated string is” $str1$str2
```

OUTPUT:

```
$ vi concat.sh
$ sh concat.sh

Enter first string: Hello

Enter first string: World

The concatenated string is HelloWorld
```

RESULT:

Thus the shell program to concatenate two strings is executed and output is verified successfully.

11.B) Comparison of Two Strings

AIM:To write a shell program to compare the two strings.

ALGORITHM:

Step 1: Start the program.

Step 2: Enter into the vi editor and go to the insert mode for entering the code.

Step 3: Read the first string.

Step 4: Read the second string.

Step 5: Compare the two strings using the if loop.

Step 6: If the condition satisfies then print that two strings are equal else print two strings are not equal.

Step 7: Enter into the escape mode for the execution of the result and verify the output.

Step 8: Stop the program.

SOURCE CODE:

```
echo "enter a string 1"
read first
echo "enter a string 2"
read second
if [ $first = $second ]          # this " =" for other than digits equals
then
        echo "strings are equal"
else
        echo "strings are unequal"
fi
```

OUTPUT:

```
$ vi compare1.sh
$ sh compare1.sh
enter a string 1
abcxyz
enter a string 2
abcxyz
```

strings are equal

OUTPUT:

$ vi compare1.sh

$ sh compare1.sh

enter a string 1

hai

enter a string 2

cse

strings are unequal

RESULT:

Thus the shell program to compare the two strings is executed and output is verified successfully.

11.C) Comparison Of Two Strings From Command Lines

AIM:To write a shell program to compare of two strings from command lines.

ALGORITHM:

Step 1: Start the program.

Step 2: Enter into the vi editor and go to the insert mode for entering the code.

Step 3: Read the first string.

Step 4: Read the second string.

Step 5: Compare the two strings using the if loop.

Step 6: If the condition satisfies then print that two strings are equal else print two strings are not equal.

Step 7: Enter into the escape mode for the execution of the result and verify the output.

Step 8: Stop the program.

SOURCE CODE:

```
if [ $1 = $2 ]
then
        echo "Strings are equal….."
else
        echo "Strings are not equal….."
fi
```

OUTPUT:

```
$ vi compare2.sh
$ sh compare2.sh UIT-RGPV UIT-
RGPVStrings are equal…..
$ sh compare2.sh UIT-RGPV
tsidohtemStrings are not equal…..
```

RESULT:

Thus the shell program to compare the two strings from command lines is executed and output is verified successfully.

11.D) Generate Fibonacci Series

AIM:To write a shell program to generate fibonacci series.

ALGORITHM:

Step 1: Start the program.

Step 2: Initialise a to 0 and b to 1.

Step 3: Print the values of ‘a’ and ‘b’.

Step 4: Add the values of ‘a'’ and ‘b’. Store the added value in variable 'c'.

Step 5: Print the value of ‘c’.

Step 6: Initialise ‘a’ to ‘b’ and ‘b’ to ‘c’.

Step 7: Repeat the steps 3, 4, 5 till the value of ‘a’ is less than 10.

Step 8: Stop the program.

SOURCE CODE:

```
echo “enter the no. of numbers in the series”
read n
a=0
b=1
d=2
echo “$a”
echo “$b”
while [ $d -lt $n ]
do
c=`expr $a + $b`
                echo “$c”
                a=$b
        b=$c
                d=`expr $d + 1`
done
```

OUTPUT:

```
$ vi fibonacci.sh
$ sh fibonacci.sh
enter the no. of numbers in the series
10
0
1
1
2
3
5
8
13
21
34
```

RESULT:

Thus the shell program to find the fibonacci series is executed and output is verified successfully.

11.E) Even or Odd Number

AIM:To write a shell program to find even or odd number.

ALGORITHM:

Step 1: Start the program.

Step 2: Enter into the vi editor and go to the insert mode for entering the code.

Step 3: Read the number.

Step 4: Evaluate whether the given number Even or Odd.

Step 5: Print the Result and verify the output.

Step 6: Stop the program.

SOURCE CODE:

```
echo “enter the number”
read n
r=1
r=`expr $n % 2`
if [ $r -eq 0 ]              # “-eq” for digits or numbers
then
        echo “even”
else
        echo “odd”
 fi
```

OUTPUT:

```
$ vi evenodd.sh
$ sh evenodd.sh
enter the number
5
odd
```

RESULT:

Thus the shell program to find whether the given number is even or odd number is executed and output is verified successfully.

11.F) List of Even Numbers in a Given Limit

AIM:To write a shell program to list of even numbers in a given limit.

ALGORITHM:

Step 1: Start the program.

Step 2: Enter into the vi editor and go to the insert mode for entering the code.

Step 3: Read the limit.

Step 4: Read the second string.

Step 5: Evaluate all Even in the given limit.

Step 6: Print the Result and verify the output.

Step 7: Stop the program.

SOURCE CODE:

```
echo -n "enter the limit:"
x=2
read num
while [ $x -lt $num ]
do
        echo -n "$x \t"
        x=`expr $x + 2`
done
```

OUTPUT:

```
$ vi evennums.sh
$ sh evennums.sh
enter the limit:20
2       4       6       8       10      12      14      16      18
```

RESULT:

Thus the shell program to find list of even numbers in a given limit is executed and output is verified successfully.

VIVA – VOCE QUESTIONS

1. **What is an operating system?**
2. **What are short, long and medium-term scheduling?**
3. **Explain the concept of the batched operating systems?**
4. **What is purpose of different operating systems?**
5. **What is virtual memory?**
6. **What is Throughput, Turnaround time, waiting time and Response time?**
7. **What are the various components of a computer system?**
8. **What is a Real-Time System?**
9. **Explain the concept of the Distributed systems?**
10. **What are the different operating systems?**
11. **What is busy waiting?**
12. **What are system calls?**
13. **What are various scheduling queues?**
14. **What are the states of a process?**
15. **What is a job queue?**
16. **What is a ready queue?**
17. **What are turnaround time and response time?**
18. **What are the operating system components?**
19. **Why thread is called as a lightweight process?**
20. **What are operating system services?**
21. **What is a process?**
22. **What are the different job scheduling in operating systems?**
23. **What is dual-mode operation?**
24. **What is a device queue?**
25. **What are the different types of Real-Time Scheduling?**
26. **What is starvation?**
27. **What is a long term scheduler & short term schedulers?**
28. **What is fragmentation?**
29. **What is the state of the processor, when a process is waiting for some event to occur?**
30. **What is the difference between Primary storage and secondary storage?**
31. **What is the difference between Complier and Interpreter?**
32. **What are the different functions of Scheduler**
33. **What are local and global page replacements?**
34. **What are the different operating systems?**

35. What are the basic functions of an operating system?

36. What is kernel?

37. What is a process?

38. What are the states of a process?

40. What is starvation and aging?

41. What is context switching?

42. What is a thread?

43. What is process synchronization?

44. What is virtual memory?

45. What is fragmentation? Tell about different types of fragmentation?

46. What is logical and physical addresses space?

47. What is Memory-Management Unit (MMU)?

48. What is a Real-Time System?

49. What is process migration?

50. Difference between Primary storage and secondary storage?

51. Define compactions.

52. What are the disadvantages of context switching?

53. What is Dispatcher?

54. What is the Translation Lookaside Buffer (TLB)?