LAB MANUAL

21CSC202J – OPERATING SYSTEMS

SEMESTER – III

SCHOOL OF COMPUTING

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
(Deemed University u/s 3 of UGC Act 1956)
Kattankulathur, Chengalpattu District, 603 202.

Ex.No.	Name of the Exercise
1.	a. Operating system Installation b. Booting Process of Linux
2.	a. Basic Linux Commands b. Filters and Admin Commands
3.	Shell Programs
4.	Process Creation
5.	Multithreading
6.	Mutual Exclusion-Semaphore and Reader Writer Solution
7.	Dining Philosopher problem
8.	CPU Scheduling Algorithms - FCFS and SJF
9.	CPU Scheduling Algorithms – Priority and Round Robin
10.	Bankers Algorithm - Deadlock Avoidance
11.	Memory Allocation Techniques First-Best-Worst Fit
12.	Page Replacement Algo - FIFO-LRU-LFU
13	Disk Scheduling Algorithms-FCFS-SCAN-C-SCAN
14	File Allocation-Sequential-Indexed

Ex. No. 1a	OPERATING SYSTEM INSTALLATION	Date :

Linux operating system can be installed as either dual OS in your system or you can install through a virtual machine (VM).

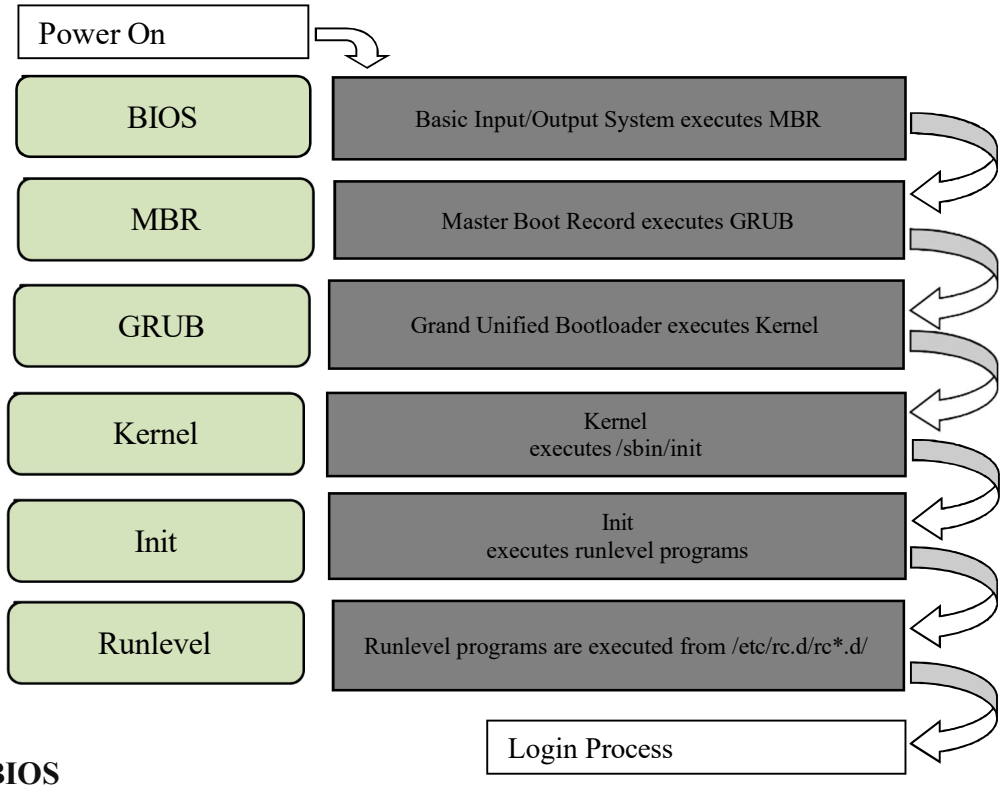
Installation of Ubuntu in your Windows OS through a Virtual machine

Steps

1. Download VMware Player or Workstation recent version.
2. Download Ubuntu LTS recent version.
3. Install VM ware Player in your host machine.
4. Open VMware Workstation and click on "New Virtual Machine".
5. Select "Typical (recommended)" and click "Next".
6. Select "Installer disc image (ISO)", click "Browse" to select the Ubuntu ISO file, click "Open" then "Next".
7. You have to type in "Full name", "User name" that must only consist of lowercase and numbers then you must enter a password. After you finished, click "Next".
8. You can type in a different name in "Virtual machine name" or leave as is and select an appropriate location to store the virtual machine by clicking on "Browse" that is next to "Location" -- you should place it in a drive/partition that has at least 5GB of free space. After you selected the location click "OK" then "Next".
9. In "Maximum disk size" per Ubuntu recommendations you should allocate at least 5GB -- double is recommended to avoid running out of free space.
10. Select "Store virtual disk as a single file" for optimum performance and click "Next".
11. Click on "Customize" and go to "Memory" to allocate more RAM -- 1GB should suffice, but more is always better if you can spare from the installed RAM.
12. Go to "Processors" and select the "Number of processors" that for a normal computer is 1 and "Number of cores per processor" that is 1 for single core, 2 for dual core, 4 for quad core and so on -- this is to insure optimum performance of the virtual machine.
13. Click "Close" then "Finish" to start the Ubuntu install process.
14. On the completion of installation, login to the system

Ex. No. 1b	BOOTING PROCESS OF LINUX	Date :

Press the power button on your system, and after few moments you see the Linux login prompt. From the time you press the power button until the Linux login prompt appears, the following sequence occurs. The following are the 6 high level stages of a typical Linux boot process.



Step 1.BIOS

∑ BIOS stands for Basic Input/Output System
∑ Performs some system integrity checks

∑ Searches, loads, and executes the boot loader program.
∑ It looks for boot loader in floppy, CD-ROMs, or hard drive. You can press a key (typically F12 or F2, but it depends on your system) during the BIOS startup to change the boot sequence.
∑ Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
∑ So, in simple terms BIOS loads and executes the MBR boot loader.

Step 2. MBR

∑ MBR stands for Master Boot Record.
∑ It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
∑ MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
∑ It contains information about GRUB (or LILO in old systems).
∑ So, in simple terms MBR loads and executes the GRUB boot loader.

Step 3. GRUB

- GRUB stands for Grand Unified Bootloader.
- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen, waits for few seconds, if you don't enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn't understand filesystem).
- Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.

```
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title CentOS(2.6.18-194.el5PAE)
root(hd0,0)
kernel/boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/
initrd /boot/initrd-2.6.18-194.el5PAE.img
```

- As you notice from the above info, it contains kernel and initrd image.
- So, in simple terms GRUB just loads and executes Kernel and initrd images.

Step 4. Kernel

- Mounts the root file system as specified in the "root=" in grub.conf
- Kernel executes the /sbin/init program
- Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a 'ps -ef | grep init' and check the pid.
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

Step 5. Init

- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
  - 0 – halt
  - 1 – Single user mode
  - 2 – Multiuser, without NFS
  - 3 – Full multiuser mode
  - 4 – unused
  - 5 – X11
  - 6 – reboot
- Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.

- ∑ Execute ‘grep initdefault /etc/inittab’ on your system to identify the default run level
- ∑ If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
- ∑ Typically you would set the default run level to either 3 or 5.

Step 6. Runlevel programs

- ∑ When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail …. OK”. Those are the runlevel programs, executed from the run level directory as defined by your run level.
- ∑ Depending on your default init level setting, the system will execute the programs from one of the following directories.
  - Run level 0 – /etc/rc.d/rc0.d/
  - Run level 1 – /etc/rc.d/rc1.d/
  - Run level 2 – /etc/rc.d/rc2.d/
  - Run level 3 – /etc/rc.d/rc3.d/
  - Run level 4 – /etc/rc.d/rc4.d/
  - Run level 5 – /etc/rc.d/rc5.d/
  - Run level 6 – /etc/rc.d/rc6.d/
- ∑ Please note that there are also symbolic links available for these directory under /etc directly. So, /etc/rc0.d is linked to /etc/rc.d/rc0.d.
- ∑ Under the /etc/rc.d/rc*.d/ directories, you would see programs that start with S and K.
- ∑ Programs starts with S are used during startup. S for startup.
- ∑ Programs starts with K are used during shutdown. K for kill.
- ∑ There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
- ∑ For example, S12syslog is to start the syslog deamon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.

Login Process

1. Users enter their username and password
2. The operating system confirms your name and password.
3. A "shell" is created for you based on your entry in the "/etc/passwd" file
4. You are "placed" in your "home"directory.
5. Start-up information is read from the file named "/etc/profile". This file is known as the system login file. When every user logs in, they read the information in this file.
6. Additional information is read from the file named ".profile" that is located in your "home" directory. This file is known as your personal login file.

Ex. No. 2a	BASIC LINUX COMMANDS	Date :

a) Basics

1. *echo* SRM ➔ to display the string SRM
2. *clear* ➔ to clear the screen
3. *date* ➔ to display the current date and time
4. *cal* 2003 ➔ to display the calendar for the year 2003
   cal 6 2003 ➔ to display the calendar for the June-2003
5. *passwd* ➔ to change password

b) Working with Files

1. *ls* ➔ list files in the present working directory
   ls –l ➔ list files with detailed information (long list)
   ls –a ➔ list all files including the hidden files
2. cat > f1 ➔ to create a file (Press ^d to finish typing)
3. *cat* f1 ➔ display the content of the file f1
4. *wc* f1 ➔ list no. of characters, words & lines of a file f1
   wc –c f1 ➔ list only no. of characters of file f1
   wc –w f1 ➔ list only no. of words of file f1
   wc –l f1 ➔ list only no. of lines of file f1
5. *cp* f1 f2 ➔ copy file f1 into f2
6. *mv* f1 f2 ➔ rename file f1 as f2
7. *rm* f1 ➔ remove the file f1
8. *head* –5 f1 ➔ list first 5 lines of the file f1
   tail –5 f1 ➔ list last 5 lines of the file f1

c) Working with Directories

1. *mkdir* elias ➔ to create the directory elias
2. *cd* elias ➔ to change the directory as elias
3. *rmdir* elias ➔ to remove the directory elias
4. *pwd* ➔ to display the path of the present working directory
5. *cd* ➔ to go to the home directory
   cd .. ➔ to go to the parent directory
   cd - ➔ to go to the previous working directory
   cd / ➔ to go to the root directory

d) File name substitution

1. *ls f?* ➔ list files start with 'f' and followed by any one character
2. *ls* *.c ➔ list files with extension 'c'
3. *ls* [gpy]et ➔ list files whose first letter is any one of the character g, p or y and followed by the word et
4. *ls* [a-d,l-m]ring ➔ list files whose first letter is any one of the character from a to d and l to m and followed by the word ring.

e) I/O Redirection

1. Input redirection

   wc –l < ex1 ➔ To find the number of lines of the file 'ex1'

2. Output redirection

   who > f2 ➔ the output of 'who' will be redirected to file f2

3. *cat* >> f1 ➔ to append more into the file f1

f) Piping

Syntax : Command1 | command2

Output of the command1 is transferred to the command2 as input. Finally output of the command2 will be displayed on the monitor.

ex. *cat* f1 | *more* ➔ list the contents of file f1 screen by screen

head –6 f1 |*tail* –2 ➔ prints the 5th & 6th lines of the file f1.

g) Environment variables

1. *echo $HOME* ➔ display the path of the home directory
2. *echo $PS1* ➔ display the prompt string $
3. *echo $PS2* ➔ display the second prompt string (> symbol by default)
4. *echo $LOGNAME* ➔ login name
5. *echo $PATH* ➔ list of pathname where the OS searches for an executable file

h) File Permission

-- chmod command is used to change the access permission of a file.

Method-1

Syntax : *chmod* [ugo] [+/-] [rwxa] filename

u : user, g : group, o : others
+ : Add permission - : Remove the permission r : read, w : write, x : execute, a : all permissions

ex. *chmod* ug+rw f1
adding 'read & write' permissions of file f1 to both user and group members.

Method-2

Syntax : *chmod* octnum file1

The 3 digit octal number represents as follows

- first digit -- file permissions for the user
- second digit -- file permissions for the group
- third digit -- file permissions for others

Each digit is specified as the sum of following
4 – read permission, 2 – write permission, 1 –

execute permission ex. *chmod* 754 f1

it change the file permission for the file as follows

- read, write & execute permissions for the user ie; 4+2+1 = 7
- read, & execute permissions for the group members ie; 4+0+1 = 5
- only read permission for others ie; 4+0+0 = 4

Ex. No. 2b	FILTERS and ADMIN COMMANDS	Date :

FILTERS

1. cut

- Used to cut characters or fileds from a file/input

Syntax : **cut** **-c**chars filename
-ffieldnos filename

- By default, tab is the filed separator(delimiter). If the fileds of the files are separated by any other character, we need to specify explicitly by **–d** option

cut **-d**delimitchar **-f**fileds filname

2. grep

- Used to search one or more files for a particular pattern.

Syntax : **grep** pattern filename(s)

- Lines that contain the *pattern* in the file(s) get displayed
- pattern can be any regular expressions
- More than one files can be searched for a pattern

-v option displays the lines that do not contain the *pattern*
-l list only name of the files that contain the *pattern*
-n displays also the line number along with the lines that matches the *pattern*

3. sort

- Used to sort the file in order

Syntax : **sort** filename

- Sorts the data as text by default
- Sorts by the first filed by default

-r option sorts the file in descending order
-u eliminates duplicate lines
-o filename writes sorted data into the file *fname*
-tdchar sorts the file in which fileds are separated by *dchar*
-n sorts the data as number
+1n skip first filed and sort the file by second filed numerically

4. Uniq

- Displays unique lines of a sorted file

Syntax : **uniq** filename

-d option displays only the duplicate lines
-c displays unique lines with no. of occurrences.

5. diff

- Used to differentiate two files

Syntax : **diff** f1 f2
compare two files f1 & f2 and prints all the lines that are differed between f1 & f2.

Q1. Write a command to cut 5 to 8 characters of the file *f1*.
$

Q2. Write a command to display user-id of all the users in your system.
$

Q3. Write a command to check whether the user j*udith* is available in your system or not. (use grep)
$

Q4. Write a command to display the lines of the file *f1* starts with SRM.
$

Q5. Write a command to sort the file /etc/passwd in descending order
$

Q6. Write a command to display the unique lines of the sorted file *f21*. Also display the number of occurrences of each line.
$

Q7. Write a command to display the lines that are common to the files *f1* and *f2*.
$

SYSTEM ADMIN COMMANDS

INSTALLING SOFTWARE

To Update the package repositories

```
sudo apt-get update
```

To update installed software

```
sudo apt-get upgrade
```

To install a package/software

```
sudo apt-get install  <package-name>
```

To remove a package from the system

```
sudo apt-get remove <package-name>
```

To reinstall a package

```
sudo apt-get install  <package-name> --reinstall
```

Q8. Update the package repositories

Q9. Install the package “simplescreenrecorder”

Q10. Remove the package “simplescreenrecorder”

MANAGING USERS

- ∑ Managing users is a critical aspect of server management.
- ∑ In Ubuntu, the root user is disabled for safety.
- ∑ Root access can be completed by using the sudo command by a user who is in the “admin” group.
- ∑ When you create a user during installation, that user is added automatically to the admin group.

To add a user:

```
sudo adduser username
```

To disable a user:

```
sudo passwd -l username
```

To enable a user:

```
sudo passwd -u username
```

To delete a user:

```
sudo userdel –r username
```

To create a group:

```
sudo addgroup groupname
```

To delete a group:

```
sudo delgroup groupname
```

To create a user with group:

```
sudo adduser username groupname
```

To see the password expiry value for a user,

```
sudo chage -l username
```

To make changes:

```
sudo chage username
```

GUI Tool for user management

If you do not want to run the commands in terminal to manage users and groups, then you can install a GUI add-on .

```
sudo apt install gnome-system-tools
```

Once done, type

```
users-admin
```

Q11. Create a user ‘elias’. Login to the newly created user and exit.

Q12. Disable the user ‘elias’, try to login and enable again.

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 3	SHELL PROGRAMS	Date :

How to run a Shell Script

- Edit and save your program using editor
- Add execute permission by *chmod* command
- Run your program using the name of your program

```
./program-name
```

Important Hints

∑ No space before and after the assignment operator Ex. `sum=0`

∑ *Single quote* ignores all special characters. Dollar sign, Back quote and Back slash are not ignored inside *Double quote*. *Back quote* is used as command substitution. *Back slash* is used to remove the special meaning of a character.

∑ Arithmetic expression can be written as follows : `i=$((i+1))` or `i=$(expr $i + 1)`

∑ Command line arguments are referred inside the programme as `$1, $2`, ..and so on

∑ `$*` represents all arguments, `$#` specifies the number of arguments

∑ `read` statement is used to get input from input device. Ex. `read a b`

Syntax for if statement

```
if [ condition ]
then
      ...
elif [ condition ]
then
      ...
else
      ...
fi
```

Syntax for case structure

```
case value in
pat1)  ...
                statement;;
pat2)   ...
                Statement;;
*)      ...
                Statement;;
esac
```

Syntax for for-loop

```
for var in list-of-values
do
          ...
          ...
done
```

Syntax for While loop

```
while command_t
do
          ...
          ...
done
```

Syntax for printf statement

```
printf  "string and format"  arg1  arg2 … …
```

- ∑ Break and continue statements functions similar to C programming
- ∑ Relational operators are –lt, -le, -gt, -ge, -eq,-ne
- ∑ Ex. (i>= 10) is written as [$i -ge 10]
- ∑ Logical operators (and, or, not) are -o, -a, !
- ∑ Ex. (a>b) && (a>c) is written as [$a –gt $b –a $a –gt $c]
- ∑ Two strings can be compared using = operator

Q1. Given the following values

num=10, x=*, y=`date` a="Hello, 'he said'"

Execute and write the output of the following commands

Command	Output
echo num	
echo $num	
echo $x	
echo '$x'	
echo "$x"	
echo $y	
echo $(date)	
echo $a	
echo \$num	
echo \$$num	

Q1. Find the output of the following shell scripts

```
$ vi  ex31
     echo Enter value for n
     read n
     sum=0
     i=1
     while [ $i -le $n ]
     do
            sum=$((sum+i))
            i=$((i+2))
     done
     echo Sum is $sum
```

Output :

Q2. Write a program to check whether the file has execute permission or not. If not, add the permission.

```
$ vi ex32
```

Q3. Write a shell script to list only the name of sub directories in the present working directory

```
$ vi ex33
```

Q4. Write a program to check all the files in the present working directory for a pattern (passed through command line) and display the name of the file followed by a message stating that the pattern is available or not available.

```
$ vi ex34
```

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 4	PROCESS CREATION using fork() and Usage of getpid(), getppid(), wait() functions	Date :

Compilation of C Program

Step 1 : Open the terminal and edit your program and save with extension ".c"

Ex. nano test.c

Step 2 : Compile your program using gcc compiler

Ex. gcc test.c ➔ Output file will be "a.out"

(or)

gcc –o test text.c ➔ Output file will be "test"

Step 3 : Correct the errors if any and run the program

Ex. ./a.out (or) ./test

Syntax for process creation

int fork();

Returns 0 in child process and child process ID in parent process.

Other Related Functions

int getpid() ➔ returns the current process ID

int getppid() ➔ returns the parent process ID

wait() ➔ makes a process wait for other process to complete

Virtual fork

vfork() is similar to fork but both processes shares the same address space.

Q1. Find the output of the following program

```
#include <stdio.h>
#include<unistd.h>
int main()
{
  int a=5,b=10,pid;
  printf("Before fork a=%d b=%d \n",a,b);
  pid=fork();

  if(pid==0)
  {
    a=a+1; b=b+1;
    printf("In child a=%d b=%d \n",a,b);
  }
  else
  {
    sleep(1);
    a=a-1; b=b-1;
    printf("In Parent a=%d b=%d \n",a,b);
  }
  return 0;
}
```

Output :

Q2. Rewrite the program in Q1 using vfork() and write the output

Q3. Calculate the number of times the text “SRMIST” is printed.

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

int main()
{
    fork();
    fork();
    fork();
    printf(“SRMIST\n”);
    return 0;
}
```

Output :

Q4. Complete the following program as described below :
The child process calculates the sum of odd numbers and the parent process calculate the sum of even numbers up to the number ‘n’. Ensure the Parent process waits for the child process to finish.

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

int main()
{
  int pid,n,oddsum=0,evensum=0;

  printf("Enter the value of n : ",a);
  scanf("%d",&n);
  pid=fork();
  // Complete the program









  return 0;
}
```

Sample Output :

Enter the value of n 10
Sum of odd numbers 25
Sum of even numbers : 30

Q5. How many child processes are created for the following code?
Hint : Check with small values of ‘n’.

```
for (i=0; i<n; i++)
        fork();
```

Output :

Q6. Write a program to print the Child process ID and Parent process ID in both Child and Parent processes

```
#include <stdio.h>
#include<unistd.h>
int main()
{









return 0;
}
```

Sample Output:

```
In Child Process
Parent Process ID       :     18
Child Process ID        :     20

In Parent Process
Parent Process ID       :     18
Child Process ID        :     20
```

Q7. How many child processes are created for the following code?

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

int main()
{
    fork();
    fork()&&fork()||fork();
    fork();
    printf("Yes ");
    return 0;
}
```

Output :

Web Reference:

https://www.geeksforgeeks.org/fork-system-call/
https://www.geeksforgeeks.org/getppid-getpid-linux/
https://www.geeksforgeeks.org/wait-system-call-c/

Ex.No.5 **MULTI-THREADING**

AIM:

Write a C program to demonstrate various thread related concepts.

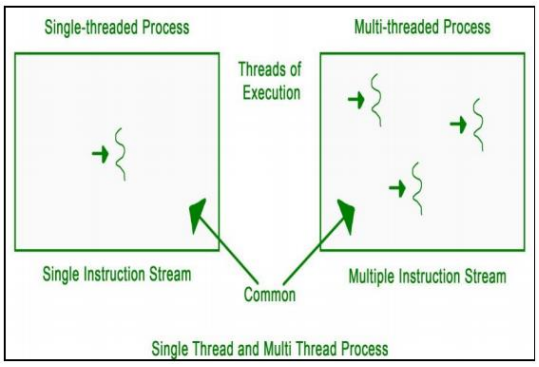
Threads:

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. Let's 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.



Why Multithreading? Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:
1) Thread creation is much faster.
2) Context switching between threads is much faster.
3) Threads can be terminated easily
4) Communication between threads is faster.

Unlike Java, multithreading is not supported by the language standard. POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

We must include the pthread.h header file at the beginning of the script to use all the functions of the pthreads library.

The **functions** defined in the **pthreads library** include:

a. ***pthread_create:*** used to create a new thread

Syntax:

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

Parameters:

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

b. ***pthread_exit:*** used to terminate a thread

Syntax:

```
void pthread_exit(void *retval);
```

Parameters: This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

c. ***pthread_join:*** used to wait for the termination of a thread.

Syntax:

```
int pthread_join(pthread_t th,
                 void **thread_return);
```

Parameter: This method accepts following parameters:

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

d. ***pthread_self:*** used to get the thread id of the current thread.

Syntax:

```
pthread_t pthread_self(void);
```

e. ***pthread_equal:*** compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.

Syntax:

```
int pthread_equal(pthread_t t1,
                  pthread_t t2);
```

Parameters: This method accepts following parameters:

- t1: the thread id of the first thread
- t2: the thread id of the second thread

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.

/* Implementing a program using thread */

Program Description:

Create 3 threads, first one to find the sum of odd numbers; second one to find the sum of even numbers; third one to find the sum of natural numbers;
This program also displays the list of odd/even numbers.

Complete the code snippet wherever applicable in the below program **highlighted in red colour font**.

```
#include /* include suitable header file */
#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(__________)
{
        int i,n; jo=0;
        n=(int)threadid;

        for(i=0;i<=n;i++)
        {
                if(logic to allow only odd numbers)
                {Calculate sum of odd numbers only}
        }
}

void *SumN(__________)
{
        int i,n;
        n=(int)threadid;

        for(i=1;i<=n;i++)
        { Calculate sum of natural numbers only}
}
```

```
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);
        create a thread to call Odd function
        create a thread to call SumN function

        for(i=0;i<NUM_THREADS;i++)
        {
                pthread_join(threads[i],NULL);
        }
        printf("The sum of first N natural numbers is %d\n", display the sum of natural numbers);
        printf("The sum of first N even numbers is %d\n",display the sum of even numbers);
        printf("The sum of first N odd numbers is %d\n",oddsum);
        printf("The first N Even numbers are----\n");
        Print all the Even numbers
        printf("The first N Odd numbers are ----\n");
        Print all the ODD numbers
        pthread_exit(NULL);
}
```

Result:

Thus, the program has been executed successfully by creating three threads.

Ex. No. 6	Mutual Exclusion-Semaphore and Reader Writer Solution	Date :

Semaphore

Semaphore is used to implement process synchronization. This is to protect critical region shared among multiples processes.

SYSTEM V SEMAPHORE SYSTEM CALLS

Include the following header files for System V semaphore

```
<sys/ipc.h>,  <sys/sem.h>, <sys/types.h>
```

To create a semaphore array,

```
int semget(key_t key, int nsems, int semflg)
```

key ➔ semaphore id
nsems ➔ no. of semaphores in the semaphore array
semflg ➔ `IPC_CREATE|0664` : to create a new semaphore
`IPC_EXCL|IPC_CREAT|0664` : to create new semaphore and the call fails if the semaphore already exists

To perform operations on the semaphore sets viz., allocating resources, waiting for the resources or freeing the resources,

```
int semop(int semid, struct sembuf *semops, size_t nsemops)
```

semid ➔ semaphore id returned by semget()
nsemops ➔ the number of operations in that array
semops ➔ The pointer to an array of operations to be performed on the semaphore set. The structure is as follows

```
struct sembuf {
        unsigned short sem_num;   /* Semaphore set num */
        short sem_op;             /* Semaphore operation */
        short sem_flg;  /*Operation flags, IPC_NOWAIT, SEM_UNDO */
};
```

Element, sem_op, in the above structure, indicates the operation that needs to be performed −

If sem_op is –ve, allocate or obtain resources. Blocks the calling process until enough resources have been freed by other processes, so that this process can allocate.
If sem_op is zero, the calling process waits or sleeps until semaphore value reaches 0.
If sem_op is +ve, release resources.

To perform control operation on semaphore,

```
int semctl(int semid, int semnum, int cmd,…);
```

semid ➔ identifier of the semaphore returned by semget()
semnum ➔ semaphore number
cmd ➔ the command to perform on the semaphore. Ex. GETVAL, SETVAL
semun ➔ value depends on the cmd. For few cases, this is not applicable.

Q1. Execute and write the output of the following program for *mutual exclusion* using system V semaphore

```
#include<sys/ipc.h>
#include<sys/sem.h>
int main()
{
 int pid,semid,val;
 struct sembuf sop;

 semid=semget((key_t)6,1,IPC_CREAT|0666);

 pid=fork();

 sop.sem_num=0;
 sop.sem_op=0;
 sop.sem_flg=0;

 if (pid!=0)
 {
   sleep(1);
   printf("The Parent waits for WAIT signal\n");
   semop(semid,&sop,1);
   printf("The Parent WAKED UP & doing her job\n");
   sleep(10);
   printf("Parent Over\n");
 }
 else
 {
   printf("The Child sets WAIT signal & doing her job\n");
   semctl(semid,0,SETVAL,1);
   sleep(10);
   printf("The Child sets WAKE signal & finished her job\n");
   semctl(semid,0,SETVAL,0);
   printf("Child Over\n");
 }
 return 0;
}
```

Output :

POSIX SEMAPHORE

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to include `semaphore.h` and compile the code by linking with `-lpthread - lrt`

To lock a semaphore or wait

```
int sem_wait(sem_t *sem);
```

To release or signal a semaphore

```
int sem_post(sem_t *sem);
```

To initialize a semaphore

```
sem_init(sem_t *sem, int pshared, unsigned int
value);
```

Where,

sem : Specifies the semaphore to be initialized.

pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes/threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.

value : Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore

```
sem_destroy(sem_t *mutex);
```

Q2. Program creates two threads: one to increment the value of a shared variable and second to decrement the value of the shared variable. Both the threads make use of semaphore variable so that only one of the threads is executing in its critical section. Execute and write the output.

```
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
 void *fun2();
 int shared=1; //shared variable
 sem_t s; //semaphore variable
 int main()
 {
 sem_init(&s,0,1); //initialize semaphore variable - 1st argument is
//address of variable, 2nd is number of processes sharing semaphore,
//3rd argument is the initial value of semaphore variable
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 sleep(1);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
```

```
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last
//updated value of shared variable
 }
void *fun1()
{
    int x;
    sem_wait(&s); //executes wait operation on s
    x=shared;//thread1 reads value of shared variable
    printf("Thread1 reads the value as %d\n",x);
    x++;  //thread1 increments its value
    printf("Local updation by Thread1: %d\n",x);
    sleep(1); //thread1 is preempted by thread 2
    shared=x; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread1 is:
%d\n",shared);
    sem_post(&s);
}
void *fun2()
{
    int y;
    sem_wait(&s);
    y=shared;//thread2 reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
    y--;  //thread2 increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread2 is preempted by thread 1
    shared=y; //thread2 updates the value of shared variable
    printf("Value of shared variable updated by Thread2 is:
%d\n",shared);
    sem_post(&s);
}
```

The final value of the variable *shared* will be 1. When any one of the threads executes the wait operation the value of "s" becomes zero. Hence the other thread (even if it preempts the running thread) is not able to successfully execute the wait operation on "*s*". Thus, not able to read the inconsistent value of the shared variable. This ensures that only one of the threads is running in its critical section at any given time.

Output:

Reader Writer Solution

In a reader-writer problem, multiple readers can read data from a shared resource, while only one writer can write data to the resource. The challenge is to ensure that the readers do not read data while the writer is writing, and the writer does not write data while the readers are reading.

For example, consider a scenario where a database is being accessed by multiple users. The users can read data from the database, but only one user can write data to the database at a time. The challenge is to ensure that the users do not read data while the database is being written to, and the writer does not write data while the users are reading.

Q3.

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

/*
This program provides a possible solution for first readers writers
problem using mutex and semaphore.
I have used 10 readers and 5 producers to demonstrate the solution.
You can always play with these values.
*/

sem_t wrt;
pthread_mutex_t mutex;
int cnt = 1;
int numreader = 0;

void *writer(void *wno)
{
    sem_wait(&wrt);
    cnt = cnt*2;
    printf("Writer %d modified cnt to %d\n",(*((int *)wno)),cnt);
    sem_post(&wrt);

}
void *reader(void *rno)
{
    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader++;
    if(numreader == 1) {
        sem_wait(&wrt); // If this id the first reader, then it will
block the writer
    }
    pthread_mutex_unlock(&mutex);
    // Reading Section
    printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);

    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
```

```
    numreader--;
    if(numreader == 0) {
        sem_post(&wrt); // If this is the last reader, it will wake
up the writer.
    }
    pthread_mutex_unlock(&mutex);
}

int main()
{

    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);

    int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering
the producer and consumer

    for(int i = 0; i < 10; i++) {
        pthread_create(&read[i], NULL, (void *)reader, (void
*)&a[i]);
    }
    for(int i = 0; i < 5; i++) {
        pthread_create(&write[i], NULL, (void *)writer, (void
*)&a[i]);
    }

    for(int i = 0; i < 10; i++) {
        pthread_join(read[i], NULL);
    }
    for(int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;

}
```

Output:

Verified by

Faculty In-charge Sign :	**Date :**

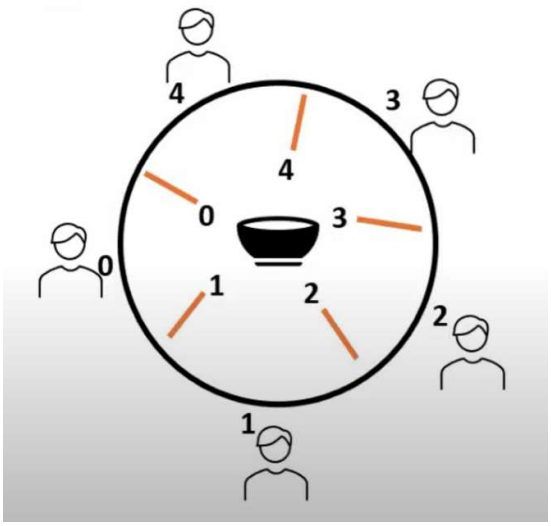
Ex. No. 7	Dining Philosopher problem	Date :

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.



Program: [Complete the code]

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>

sem_t room;
sem_t chopstick[5];
```

```
void * philosopher(void * num)
{
```
//Write coding for Main Philosopher thread

```
}

void eat(int phil)
{
      printf("\nPhilosopher %d is eating",phil);


}


int main()
{
      int i,a[5]; pthread_t tid[5];

      sem_init(&room,0,4);

      for(i=0;i<5;i++)
            sem_init(&chopstick[i],0,1);

      for(i=0;i<5;i++)
      { a[i]=i;
        pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
      }
      for(i=0;i<5;i++)
            pthread_join(tid[i],NULL);
}
```

Output:

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 8	SCHEDULING ALGORITHMS FCFS-SJF	Date :

1. FCFS Scheduling Algorithm

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm. First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.

In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed. Here we are considering that arrival time for all processes is 0.

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

Algorithm:

Step 1. Input the processes along with their burst time (bt).

Step 2. Find waiting time (wt) for all processes.

Step 3. As first process that comes need not to wait so
waiting time for process 1 will be 0 i.e. wt[0] = 0.

Step 4. Find waiting time for all other processes i.e. for
process i ->
wt[i] = bt[i-1] + wt[i-1]

Step 5. Find *turnaround time = waiting_time + burst_time*
for all processes.

Step 6. Find *average waiting time = total_waiting_time / no_of_processes*

Step 7. Similarly, find *average turnaround time =*
total_turn_around_time / no_of_processes.

Input : Processes Numbers and their burst times

Output : Process-wise burst-time, waiting-time and turnaround-time
Also display Average-waiting time and Average-turnaround-time

Q1. Write a program to implement FCFS Scheduling algorithm

#include <stdio.h>
//Write the program here

2. SHORTEST JOB FIRST (SJF) : (Non- Preemption)

DESCRIPTION:

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as _0' and its turnaround time as its burst time.
Step 6: Sort the processes names based on their Burt time Step 7: For each process in the ready queue, calculate
a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n) Step 8: Calculate
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process Step 9: Stop the process

Q2. Write a program to implement SJF Scheduling algorithm

#include <stdio.h>
//Write the program here

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 9	SCHEDULING ALGORITHMS Priority and Round Robin	Date :

1. PRIORITY SCHEDULING ALGORITHMS:

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

DESCRIPTION:

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround time

Step 6: Arrange the processes based on process priority

Step 7: For each process in the Ready Q calculate Step 8:

for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 9: Calculate

c) Average waiting time = Total waiting Time / Number of process

d) Average Turnaround time = Total Turnaround Time / Number of process Print the results

in an order.

Step10: Stop

Q1. Write a program to implement Priority Scheduling algorithm

```
#include <stdio.h>
//Write the program here
```

2. ROUND ROBIN:

AIM:

To simulate the CPU scheduling algorithm round-robin.

DESCRIPTION:

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not, it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice
Step 5: If the burst time is less than the time slice then the no. of time slices =1.
Step 6: Consider the ready queue is a circular Q, calculate
a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).
Step 7: Calculate
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

Q2. Write a program to implement Round Robin Scheduling algorithm

```
#include <stdio.h>
//Write the program here
```

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 10	BANKERS ALGORITHM-DEAD LOCK AVOIDANCE	Date :

DEAD LOCK AVOIDANCE

AIM:
To Simulate bankers algorithm for Dead Lock Avoidance (Banker's Algorithm)

DESCRIPTION:

Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

n-Number of process,
m-number of resource types.
Available: Available[j]=k, k – instance of resource type Rj is available.
Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj
Need: If
Need[I, j]=k, Pi may need k more instances of resource type Rj, Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both
Finish[i] =False
Need<=Work If no such I
exists go to step 4.
3. work= work + Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state.

Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;
Available=Available-Request I;
Allocation I=Allocation +Request I;
Need i=Need i- Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop the program.
11. end

Q1. Write a program to simulate Bankers algorithm

```
#include <stdio.h>
//Write the program here
```

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 11	MEMORY ALLOCATION TECHNIQUES-FIRST-BEST-WORST FIT	Date :

AIM:
To Write a C program to simulate the following contiguous memory allocation techniques
a) Worst-fit b) Best-fit c) First-fit

DESCRIPTION

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

Q1. Write a program to simulate First, Best and Worst fit memory allocation techniques algorithm

```
#include <stdio.h>
//Write the program here
```

Verified by

Faculty In-charge Sign :	Date :

Ex. No. 12	PAGE REPLACEMENT ALGORITHMS FIFO- LRU - LFU	Date :

AIM: To implement different page replacement technique.
a) FIFO b) LRU c) LFU

DESCRIPTION:
Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out making space for the currently needed page. However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.

FIFO-This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

LRU-The LRU stands for the **Least Recently Used**. It keeps track of page usage in the memory over a short period of time. It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future. It removes the page that has not been utilized in the memory for the longest time. LRU is the most widely used algorithm because it provides fewer page faults than the other methods.

The LFU page replacement algorithm stands for the **Least Frequently Used**. In the LFU page replacement algorithm, the page with the least visits in a given period of time is removed. It replaces the least frequently used pages. If the frequency of pages remains constant, the page that comes first is replaced first.

Q1. Write a program to simulate FIFO, LRU and LFU page replacement techniques algorithm

```
#include <stdio.h>
//Write the program here
```

Verified by

Faculty In-charge Sign :	Date :

Ex. No. 13	DISK SCHEDULING ALGORITHMS FCFS-SCAN-C-SCAN	Date :

AIM:

To Write a C program to simulate disk scheduling algorithms

a) FCFS b) SCAN c) C-SCAN

DESCRIPTION

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

Q1. Write a program to simulate disk scheduling algorithms FCFS-SCAN and C-SCAN

```
#include <stdio.h>
//Write the program here
```

Verified by

Faculty In-charge Sign :	Date :

Ex. No. 14	FILE ALLOCATION STRATEGIES SEQUENTIAL and INDEXED	Date :

A) SEQUENTIAL:

AIM: To write a C program for implementing sequential file allocation method

DESCRIPTION:

The most common form of file structure is the sequential file in this type of file, a fixed format is used for records. All records (of the system) have the same length, consisting of the same number of fixed length fields in a particular order because the length and position of each field are known, only the values of fields need to be stored, the field name and length for each field are attributes of the file structure.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations to each in sequential order a).

Randomly select a location from available location s1= random(100);

a) Check whether the required locations are free from the selected

location.

```
if(b[s1].flag==0){
for
(j=s1;j<s1+p[i];j++){
if((b[j].flag)==0)count++;
}
if(count==p[i]) break;
}
```

b) Allocate and set flag=1 to the allocated locations. for(s=s1;s<(s1+p[i]);s++)

```
{
k[i][j]=s; j=j+1; b[s].bno=s;
b[s].flag=1;
}
```

Step 5: Print the results file no, length, Blocks allocated. Step 6: Stop the program

Q1. Write a program to implement sequential file allocation method

```
#include <stdio.h>
//Write the program here
```

B) INDEXED:

AIM:

To implement allocation method using chained method

DESCRIPTION:

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation.

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of files.
Step 3: Get the memory requirement of each file.
Step 4: Allocate the required locations by selecting a location randomly q= random(100);
a) Check whether the selected location is free .
b) If the location is free allocate and set flag=1 to the allocated locations.

```
q=random(100);
{
if(b[q].flag==0)
b[q].flag=1;
b[q].fno=j;
r[i][j]=q;
```

Step 5: Print the results file no, length ,Blocks
allocated.
Step 6: Stop the program

Q2. Write a program to implement indexed file allocation method

```
#include <stdio.h>
//Write the program here
```

Verified by

Faculty In-charge Sign :	**Date :**